Lecture Notes in Computer Science 2716
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Michael J. Voss (Ed.)

OpenMP Shared Memory Parallel Programming

International Workshop on OpenMP Applications and Tools, WOMPAT 2003
Toronto, Canada, June 26-27, 2003
Proceedings

Springer

Volume Editor

Michael J. Voss
University of Toronto
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
10 King's College Road, Toronto, Ontario, M5S 3G4 Canada
E-mail: voss@eecg.toronto.edu

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): C.1-4, D.1-4, F.1-3, G.1-2

ISSN 0302-9743
ISBN 3-540-40435-X Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by Christian Grosche, Hamburg
Printed on acid-free paper SPIN: 10928813 06/3142 5 4 3 2 1 0

Preface

This book contains the proceedings of the Workshop on OpenMP Applications and Tools, WOMPAT 2003. WOMPAT 2003 was held on June 26 and 27, 2003 in Toronto, Canada. The workshop brought together the users and developers of the OpenMP API to meet, share ideas and experiences, and to discuss the latest developments in OpenMP and its applications.

The OpenMP API is now a widely accepted standard for high-level shared-memory parallel programming. Since its introduction in 1997, OpenMP has gained support from the majority of high-performance compiler and hardware vendors. WOMPAT 2003 was the latest in a series of OpenMP-related workshops, which have included the annual offerings of the Workshop on OpenMP Applications and Tools (WOMPAT), the European Workshop on OpenMP (EWOMP) and the Workshop on OpenMP: Experiences and Implementations (WOMPEI).

The WOMPAT 2003 program committee formally solicited papers. Extended abstracts were submitted by authors and all abstracts were reviewed by three members of the program committee. Of the 17 submitted abstracts, 15 were selected for presentation at the workshop. This book was published in time to be available at WOMPAT 2003, and therefore we hope that the papers contained herein are timely and useful for current developers and researchers.

This book also contains selected papers from WOMPAT 2002. No formal proceedings had been created for this previous offering of the workshop, and so presenters were invited to prepare their papers for inclusion in this volume. WOMPAT 2002 was held on August 5–7, 2002 at the Arctic Region Supercomputing Center at the University of Alaska, Fairbanks.

Sponsors

WOMPAT 2003 was cosponsored by the OpenMP Architecture Review Board (ARB), the OpenMP users group cOMPunity, the Intel Corporation, and the Edward S. Rogers Sr. Department of Electrical and Computer Engineering at the University of Toronto.

WOMPAT 2002 was cosponsored by the OpenMP Architecture Review Board (ARB), the OpenMP users group cOMPunity, and the Arctic Region Supercomputing Center at the University of Alaska, Fairbanks.

June 2003 Michael J. Voss

Organization

WOMPAT 2003 Program Committee

Michael J. Voss, *University of Toronto, Canada* (*Workshop Chair*)

Eduard Ayguade, *Univ. Politecnica de Catalunya, Spain*
Bob Blainey, *IBM Toronto Laboratory, Canada*
Mark Bull, *University of Edinburgh, UK*
Barbara Chapman, *University of Houston, USA*
Rudolf Eigenmann, *Purdue University, USA*
Timothy Mattson, *Intel Corporation, USA*
Mitsuhisa Sato, *University of Tsukuba, Japan*
Sanjiv Shah, *KSL, Intel Corporation, USA*

WOMPAT 2002 Program Committee

Guy Robinson, *University of Fairbanks, USA* (*Workshop Chair*)

Barbara Chapman, *University of Houston, USA*
Daniel Duffy, *US Army Engineer Research and Development Center, USA*
Rudolf Eigenmann, *Purdue University, USA*
Timothy Mattson, *Intel Corporation, USA*
Sanjiv Shah, *KSL, Intel Corporation, USA*

Table of Contents

Tools and Tool Technology I

OpenMP Implementations

OpenMP Experience

Tools and Tool Technology II

OpenMP on Clusters

Selected Papers from WOMPAT 2002

OpenMP Support in the Intel® Thread Checker

Paul Petersen and Sanjiv Shah

Intel Corporation, 1906 Fox Drive, Champaign IL, USA
{paul.petersen,sanjiv.shah}@intel.com

Abstract. The Intel® Thread Checker is the second incarnation of projection based dynamic analysis technology first introduced with Assure that greatly simplifies application development with OpenMP. The ability to dynamically analyze multiple sibling OpenMP teams enhances the previous Assure support and complements previous work on static analysis. In addition, binary instrumentation capabilities allow detection of thread-safety violations in system and third party libraries that most applications use.

1 Introduction

It is easy to write bad programs in any language, and OpenMP is no exception. Users are faced with many challenges, some similar to other languages, and some unique to OpenMP. When using OpenMP, many users start with a sequential algorithm encoded in a sequential application that expresses the method by which the computation will be performed. From this sequential specification, OpenMP directives are added to relax the sequential execution restrictions. Relaxing these restrictions allows the algorithm to be executed concurrently. The manner in which these directives are incorporated into the serial program is crucial to determining the correctness and the performance of the resulting parallel program. The correctness of the OpenMP program is not just a function of which directives are added to the sequential program, but also a function of the computation performed by the program as modified by the directives.

2 Correctness Analysis

It is possible to analyze the semantics of an OpenMP application either statically or dynamically. Static analysis is usually limited by a number of significant factors. Chief among them are the lack of availability of the entire program, increasing use of third party libraries in applications, program flow dependencies on input data and failure of alias analysis. Dynamic analysis is limited by the data sets available for testing. However software programmers already tackle this problem in the validation of their sequential program and have extensive tests available. These same tests can be used to gain the same level of comfort with correctness of the OpenMP application that the programmer already uses for the sequential application. While no dynamic

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 1–12, 2003.

analysis technique can prove arbitrary programs to be correct, dynamic analysis techniques are very powerful at locating instances of defects in applications.

2.1 Correctness: Kinds of Threaded Errors

A threading error is defined as any defect in the program that would not have been a defect if only a single thread of execution were used in the application. We classify all threading errors into the following categories:

Data Races: Data races occur when a thread modifies a data object in an application at a time when another thread is also accessing (modifying or using) the same data object. Some parallel algorithms have deliberate data races and can tolerate them correctly (although they may pay a performance penalty), but in a majority of applications, data races indicate correctness errors. Mutual exclusion constructs like OpenMP CRITICAL or ORDERED constructs, or the OpenMP locks API are usually necessary to prevent data races. Mutual exclusion constructs are a part of what the OpenMP specifications define as synchronization constructs.

Deadlocks: Deadlocks occur when threads are unable to make forward progress because they are waiting for resources that cannot become available. With OpenMP, these typically occur when using mutual exclusion or BARRIER constructs. The OpenMP specifications define this set of constructs as synchronization constructs. The classical example is with two threads, one holding lock A and waiting for lock B, while the other thread is holding lock B and waiting for lock A.

Stalls: Stalls are temporary conditions that occur when a thread is not able to acquire a mutual exclusion resource. Stalls may be purposely introduced into applications, or may be the side effects of missed signals, or logic errors.

Undefined Behaviors: The OpenMP specifications have many instances of undefined behaviors. In many instances, these behaviors cannot be validated by an OpenMP implementation due to their dynamic nature. Examples of undefined behaviors are objects in PRIVATE clauses that are not defined before use, and lock objects that are not initialized before locking. Data races are also undefined behavior, but they are pervasive and important enough to justify a separate category.

Live Locks: Live locks occur when threads execute portions of the application repeatedly and are not able to make forward progress out of this portion of the application. Live locks are similar to stalls, and are usually caused by logic errors. Any application that uses polling mechanisms may possibly miss the reception of a signal. If the signal is not repeatedly sent the application may continue to check for the missed signal indefinitely.

Logic Errors: Logic errors occur when some implicit assumptions of the application are accidentally violated. For example, the application might assume that certain events B must always be preceded by certain other events A, but the presence of logic errors in the threading allows this to be violated and unexpected events C sometimes precede B.

Certain classes of these errors have well researched solutions. Logic errors, however, are usually detected as a side effect of the error, and not directly. Logic errors usually cause the execution path of the program to diverge from the intended path.

2.2 Pros and Cons of Static Analysis for Correctness

Static analysis is a powerful analysis mode that theoretically has the potential to prove the correctness of a program. It also has the power to do the analysis based simply on the size of the input program and not dependent on the size of the input data. The size of the programs and the use of 3rd party binary libraries often prevent static analysis from fully analyzing programs. The combinations of the size of the program, the number of different paths through the program together with a dependence on the input data makes static analysis extremely conservative and difficult to use for correctness checking of parallel programs.

2.3 Pros and Cons of Dynamic Analysis for Correctness

Dynamic analysis provides meaningful answers to programmers even when applications are too large or complex for static analysis. Instead of trying to provide answers which are valid for all execution paths and all data sets, dynamic analysis instead focuses on "common" execution paths as exhibited by test data sets. Observing the program as it runs helps generate examples that may prove the program incorrect. Once such an example is detected, details that aid in understanding and fixing the problem can be collected and presented to the programmer. The biggest drawback to dynamic analysis for correctness is its cost: the increase in memory usage during analysis can be more than an order of magnitude; the increase in execution time can be one or two orders of magnitude. These drawbacks are significant, but are easily offset in practice by carefully planned use of dynamic analysis tools and by the increase in programmer productivity and efficiency (and reduction in development time) that is attained when using such tools. Ideally, tools would combine both static and dynamic analyses to offer the best of both worlds: static analysis for the kinds of errors that can be diagnosed very quickly at compile time and dynamic analysis for the data dependent errors, and for errors in which the entire program is not available.

3 Dynamic Correctness Analysis of OpenMP Applications

Two approaches can be used to perform dynamic correctness analysis of OpenMP applications. The first approach treats the OpenMP application as a specific instance of the more general class of threaded applications. The relationships between threads can be calculated using techniques pioneered by Lamport [1] and used in Eraser [2], and RecPlay [3] to find instances of resources that are accessed by two or more threads where the order of access is not guaranteed. This technique was used by Assure for Java and Assure for Threads [4], and is a part of the analysis performed by Intel® Thread Checker [5] for explicitly threaded applications. This approach is called "simulation". The second approach is to treat the OpenMP program as an annotation to a sequential application. By treating the sequential application as a specification for the correct behavior of the OpenMP program, any differences between the two programs can be identified as likely errors in the OpenMP program. Using OpenMP in this way requires some restrictions to the general OpenMP language: that the OpenMP program has a sequential counterpart, and that the OpenMP program does not associate identities with particular threads and depend on the number of threads. OpenMP programs that exhibit these properties are called "relaxed sequential programs". Obviously, not all OpenMP programs have sequential counterparts and exhibit these properties; programs that do not exhibit these properties cannot be analyzed by this latter method. The technology behind this second approach to dynamic analysis of OpenMP applications is called "projection technology" and is the basis for the current version of Intel® Thread Checker for OpenMP.

3.1 Relaxed Sequential Programming and Benefits

Parallel machines can be programmed in many different ways to exploit the available hardware resources. A common way is to assign a unique identity to each thread (or processor) and then using the identity of that thread to generate the work to be assigned to that thread. While this is a common way of creating a parallel program, it diverts a lot of the programmer's attention to the problem of determining the number of threads or processors to use, remembering each threads or processors unique identity and calculating the mapping of the available data and computation onto the hardware and software environment.

Relaxed sequential programming allows the programmer to ignore this mapping problem and the details of the particular hardware or software environment that the program is running on and instead focus on only the simpler job of specifying the parallelism in the data and computation. The mapping problem is delegated to the OpenMP implementation and other software and hardware system components. The OpenMP specifications have features like work-sharing constructs and orphaning that allow a diligent programmer to avoid thread identities for the most part, as is natural in sequential programming where there is no such concept.

The Intel® Thread Checker for OpenMP uses projection technology which exploits relaxed sequential programming to find errors in OpenMP programs. Relaxed sequential programs are really two programs in one set of sources: the original sequential program obtained by ignoring the OpenMP directives, and the OpenMP program. Projection technology executes the sequential version of the application and treats it as the specification for the OpenMP application. This approach can detect many different kinds of errors, far more than possible with the simulation approach above. Besides data-races, projection technology can also detect access patterns in the parallel program that would cause values to be computed which are impossible in the sequential version of the application. Sometimes these differences are due to the OpenMP directives performing an algorithm substitution (such as a reduction), but in general these differences are indicative of correctness problems.

3.2 Projection Technology

The best way to understand the sequential specification is to run the sequential version of the application. From the sequential execution an annotated memory trace is extracted which defines the sequential behavior of the application. This sequential annotated memory trace can be transformed to generate a projection of the memory trace that a parallel execution would generate. The sequential trace is partitioned at the places where OpenMP annotations are present. These partitioned sections of the trace are analyzed as if the trace were executed concurrently to determine if the parallel program would violate the behavior specified by the serial program. This approach has several consequences. First, since the serial program is generating the trace, it is not possible to present multiple thread identities to the executing program. Each API call, which asks for a thread identity, needs to return exactly the same value. Because of this, applications that require unique thread identities to be present cannot be analyzed. The second consequence is that any application, which depends on a specific number of threads, also cannot be analyzed. This behavior is most notable in the execution of work-shared constructs. The analysis performed via projection technology assigns a logical thread to each independent unit of work in the work-shared construct. By assuming that maximal parallelism is exposed the maximal amount of data can be checked for data races.

The combination of the relaxed sequential program and projection technology lets a single projection automatically find errors in the OpenMP application for all numbers of threads and for all possible schedules. This extremely powerful test is a solid reason to use OpenMP for relaxed sequential programming.

4 Intel® Tools for OpenMP

Intel® provides several tools for OpenMP. These tools are:

- **Intel® C++ and Fortran compilers** [6]: implementation of OpenMP specifications,
- **Thread profiler** [5]: plug-in to the VTune™ Performance Environment [7]. The current version of the Thread profiler is designed only for OpenMP performance analysis, and
- **Intel® Thread Checker:** the second incarnation of the technology pioneered in the KAI Assure [4] line of products.

The Intel® Thread Checker represents a major redesign of the original Assure products. The following breakdown shows the redesigned components:

Explicit Threading: The functionality of Assure for Threads and Assure for OpenMP are now combined in a single product, which allows the analysis of explicitly threaded applications, whereby OpenMP has been added to one or more threads. This kind of usage is common in the GUI world, which was a target for this redesign. To accommodate this use, Intel® Thread Checker switches between simulation technology and the projection technology for OpenMP analysis as necessary in different parts of the application.

Binary Instrumentation: To examine libraries and executables from third parties, Intel® Thread Checker now has binary instrumentation capabilities. Modern software uses many libraries from third parties, including the Operating System. Often threading related errors in applications are hidden in such third party component, making source instrumentation alone impractical.

Source Instrumentation: For OpenMP applications we have added instrumentation capability to the Intel® C++ and Fortran compilers to analyze applications which contain OpenMP directives.

User Interface: While building on the core concepts that were field proven in the legacy products, the user interfaces have been redesigned to plug into the VTune™ Performance Environment and allow the user more flexibility in categorizing, and displaying information.

4.1 Creating Parallel Applications

The Intel® Thread Checker can be used for two different usage models. The obvious one is in debugging or quality assurance of parallel applications. But, a more fundamental mode of usage is even more valuable. Using a serial execution profiler the significant sections of an application can be located. If these sections appear to involve iterative application of functions to disjoint data, then an OpenMP directive can

be added to the application to specify a proposal for this section of code to be executed in parallel.

The Intel® Thread Checker when used on this version of the application with the proposed parallelism annotations causes the serial application to be run, but the analysis to be performed as if the OpenMP construct was mapped to multiple threads. If few or no diagnostics are generated from the introduction of this OpenMP construct then you can assume that it may be a good candidate, but if many diagnostics are generated this may indicate that significant work needs to occur before this section of the application can be executed concurrently (or it may not have been a good candidate to execute concurrently).

4.2 Debugging Parallel Applications

The obvious usage of the Intel® Thread Checker is as an intelligent debugger. Traditional debuggers (like dbx, or the Microsoft Visual Studio debugger) allow the execution of a program to be controlled via the use of breakpoints, and also to allow the execution state of the program to be displayed at arbitrary points in the program's execution. These features allow a sophisticated software developer to examine what went wrong, and to attempt to reconstruct the sequence of events that caused the application to arrive at an incorrect conclusion. In contrast, the Thread Checker automatically performs the consistency analysis while every instruction in the application is executed. As soon as a violation of the OpenMP specification is detected, or a violation of common thread-safety rules is detected a diagnostic is generated. These thread safety rules include data-races, deadlocks, stalls, and API violations.

5 Relaxed Sequential Programming: A Concrete Example

Consider the following simple computation:

```
void
histogram( double D[], int N,
      double(*F)(double), int S ) {
  /* D[N] accumulates the distribution   */
  /* F() is the function being measured  */
  /* S  is the number of samples to take */
  for (int i=0; i<S; ++i) {
    int k=(F(i/(double)S)*N);
    D[ k ] += 1;
  }
}
```

In this example you are given a function `F`, and asked to take a number `S` of samples of the function, and to classify these samples into discrete bins to record the distribution `D` of the values of the function. The algorithm as stated says to sequen-

tially enumerate the domain `[0...1)` and for each of the values calculate the value of the function `F`. To create a parallel version of this algorithm a good question to ask is "what serial constraints can I relax". It is useful to pose this question for each of the components of this algorithm in turn as a thought experiment.

The first question you may ask is; can I evaluate two instances of the function `F` concurrently? If the function calculates a recurrence through the use of internal state it may be difficult to produce the same set of answers using concurrent evaluation. If the function does use internal state then does the function also use internal locking to protect this state and make the thread safe to use in a threaded environment?

The next question is; can I relax the sequential constraints on calculating the number of samples in each bin of the histogram? Here the answer is simple. Since addition is commutative and associative it makes no difference if the samples are accumulated in an order different from the sequential order.

The final question to ask is; can the domain of the function be calculated in non-sequential order. From a mathematical standpoint the domain of a function is a set, and sets do not have a predetermined order, so the fact that the domain was enumerated in sequential order can be removed and a different concurrent order substituted. The OpenMP "parallel for" pragma is useful to remove the serial dependence on the increment of the variable `i` which is used to specify the domain of the function.

6 Sample Use of Intel® Thread Checker

The results of our thought experiment on relaxing the sequential constraints on the original serial algorithm produce the following proposal for a parallel program. Annotating the loop that creates a number of samples with a "parallel for" pragma gives us this parallel algorithm.

```
void
histogram( double D[], int N,
      double (*F)(double), int S ) {
  /* D[N] accumulates the distribution   */
  /* F() is the function being measured  */
  /* S  is the number of samples to take */
#pragma omp parallel for private(i) shared(D,F,N,S)
  for (int i=0; i<S; ++i) {
    int k=(F(i/(double)S)*N);
    D[ k ] += 1;
  }
}
```

This example can be compiled with the Intel 7.0 compilers using the command line "icl /Qopenmp/Qtcheck h1.cpp" to prepare the program for analysis by the Intel® Thread Checker. When this application is run inside the Intel® Thread Checker GUI, the results shown in Fig. 1 are generated.

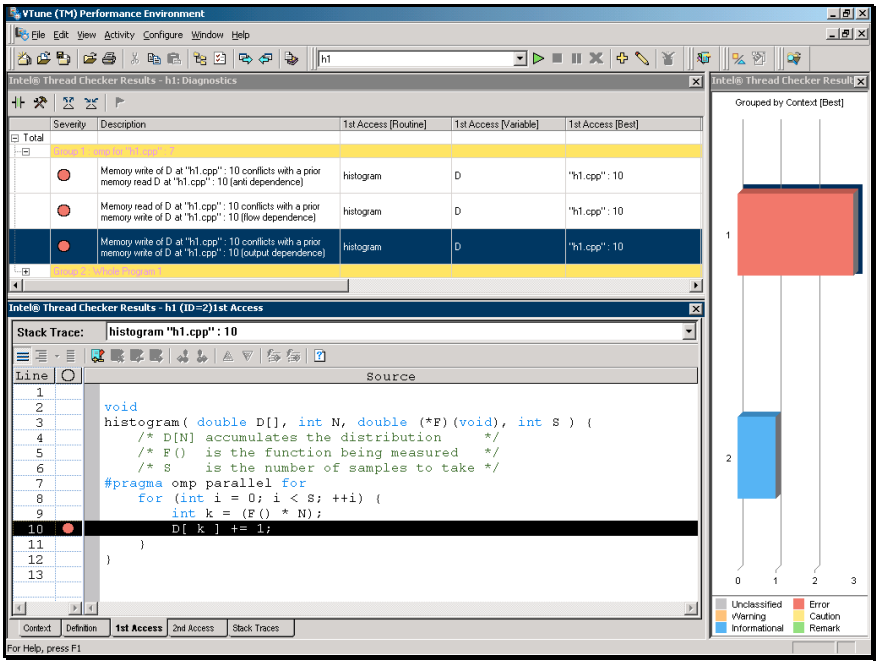


Fig. 1. A screen capture of the Intel® Thread Checker user interface showing the original attempt at parallelization of the algorithm. Notice the Diagnostics display shows that several access violations exist in this example

In Fig. 1 we see three significant user interface components. The first is the Diagnostics list. This control displays all of the diagnostics collected during the execution of the instrumented application. The user is allows to customize this display, and to organize the information into groups. The groups are displayed in the second user interface component. The second component is the Graphical Summary. This bar chart graphically displays the contents of each group in the Diagnostics list. Each bar is color coded to display the severity level of the diagnostics in each group. The length of the bars is proportional the number of diagnostics in each group. The final significant component is the Source View. The source view consists of a component comprised of five tabs. The first four tabs (when possible) display different source code locations (and stack traces) associated with this diagnostic.

The four types of source code locations are:

1) The “Context”: For OpenMP programs the context is the parallel region that created the threads involved in the diagnostic.
2) The “Definition”: This is the allocation point of the object or memory locations under conflict in the diagnostic.

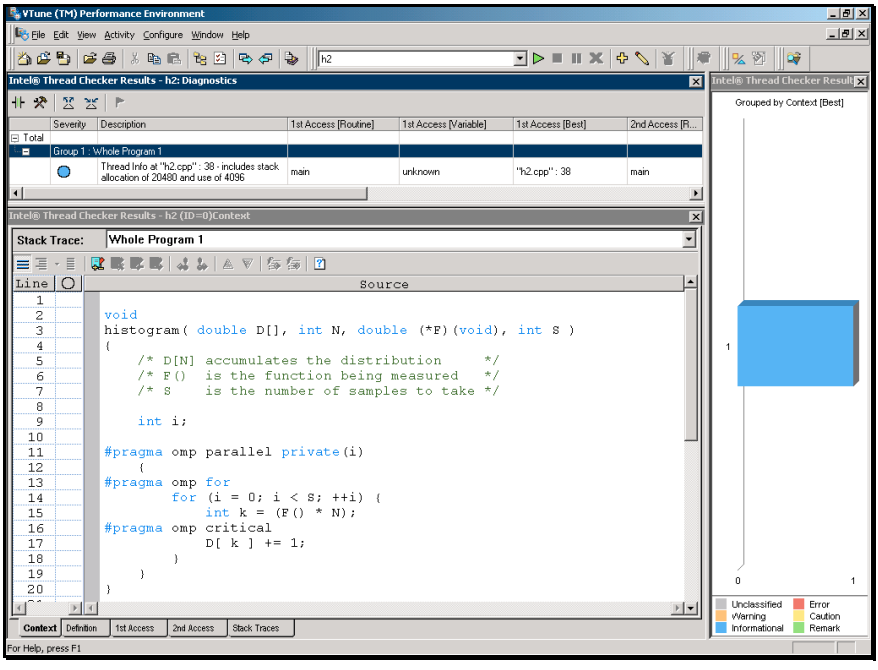


Fig. 2. A screen capture from the Intel® Thread Checker user interface. In this example the memory access errors have been removed by the introduction of the "critical" section

3) The "1st Access": This is the state of the 1st thread when it accessed the object or memory location in conflict
4) The "2nd Access": This is the state of the 2nd thread when it accessed the object or memory location in conflict.

The final tab in the Source View component displays a graphical view of the stack traces for each of the 4 source code locations. If multiple dynamic paths are found which all reach the same source locations for this error, the stack traces for these paths are displayed in this graphic with common elements of the duplicate stack traces merged into single nodes of the graphs.

In Fig. 2, we see the corrected version of the parallel application. In this version the data-race detected in Fig. 1 has been eliminated by the use of a "`critical`" pragma.

Running the example in Fig. 2 shows the observation that the execution time was much worse than the original sequential algorithm. Eliminating the fine grain usage of the "`critical`" pragma created the third version of the parallel application, as shown in Fig. 3. This example shows an interesting property of the Intel® Thread Checker. The analysis done for OpenMP applications is not to find bugs in the parallel algorithm, but instead to find differences between the parallel algorithm and the serial algorithm. The serial algorithm is defined as the parallel algorithm with the OpenMP directives suppressed. In this particular case Thread Checker is pointing out

that the intermediate values of the local variable "MY_D" have different values when using the parallel and serial algorithms. This is detected by noticing what would be a data-race on a private variable. This memory access instead of causing non-determinism to be introduced into the application instead causes the parallel algorithm to have a different memory access pattern than the serial algorithm.

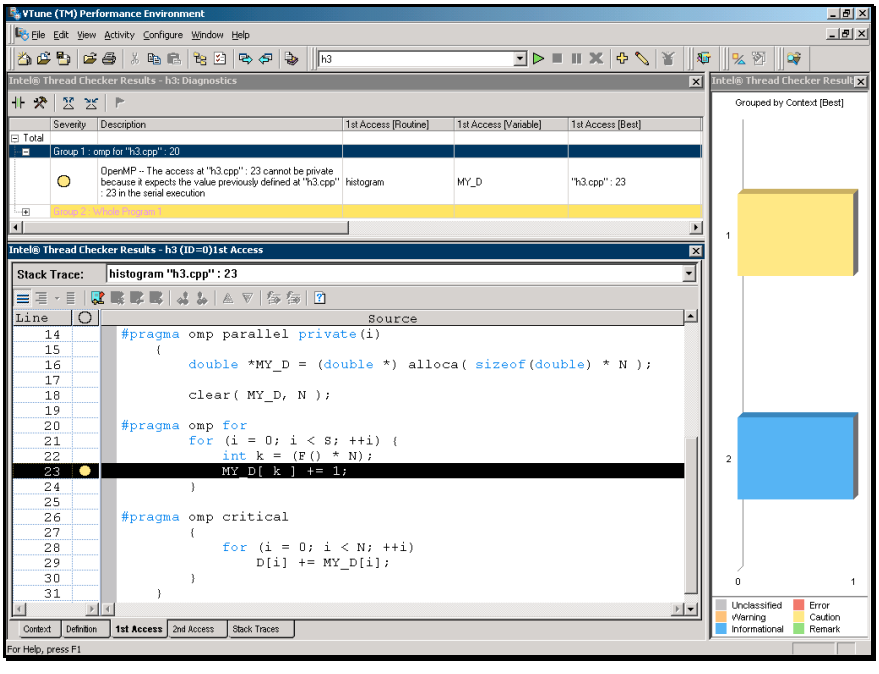


Fig. 3. A screen capture of the Intel® Thread Checker showing the analysis of the final algorithm. This algorithm uses a thread local array to accumulate partial results. The highlighted diagnostic indicates that the behavior the values in this temporary array will be different. Fortunately this difference is exactly what the application needs to be correct

7 Conclusion

The creation of parallel applications that are both correct and efficient is a challenging task. The use of OpenMP as the language to express parallelism allows for rapid specification of the parallel application. However, the programmer is faced with the same question as with any other parallel language: Is the parallel program correct? By restricting the domain of OpenMP applications to relaxed sequential programs, Intel® Thread Checker offers very powerful analysis and insight into potential failures for specific data sets, as has been demonstrated by it's use in several very large projects where the productivity and efficiency of the programmers and the time to completion of the project was positively impacted by the use of Intel® Thread Checker.

Acknowledgements

We would like to thank the contributions of the Threading Tools development team, the Compiler team and the VTune™ Performance Environment development team. This product has truly been a multidiscipline development effort. With out all of the groups hard work and dedication the Intel® Thread Checker product would not have been possible.

References

[1] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." Communications of the ACM (CACM), 21(7): 558-565, July 1978

[2] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. "Eraser: A dynamic data race detector for multithreaded programs", ACM Transactions on Computer Systems (TOCS), 15(4): 391-411, November 1997

[3] Michiel Ronsse, Koen De Bosschere. "RecPlay: A fully integrated practical record/replay system", ACM Transactions on Computer Systems (TOCS), 17(2): 133-152, May 1999

[4] KAI Software. "KAP/Pro™ Toolset Reference Manual Version 4.0", 2001 (see also http://developer.intel.com/software/products/kappro/).

[5] Intel Corporation. "Intel® Thread Checker and Thread Profiler", 2003 (see also http://www.intel.com/software/products/threading/)

[6] Intel Corporation. "Intel® C++ and Fortran Compilers", 2003 (see also http://www.intel.com/software/products/compilers/)

[7] Intel Corporation. "VTune™ Performance Environment", 2003 (see also http://www.intel.com/software/products/vtune/)

A C++ Infrastructure for Automatic Introduction and Translation of OpenMP Directives

Dan Quinlan, Markus Schordan, Qing Yi, and Bronis R. de Supinski

Lawrence Livermore National Laboratory**, USA
{dquinlan,schordan1,yi4,bronis}@llnl.gov

Abstract. In this paper we describe a C++ infrastructure for source-to-source translation. We demonstrate the translation of a serial program with high-level abstractions to a lower-level parallel program in two separate phases. In the first phase OpenMP directives are introduced, driven by the semantics of high-level abstractions. Then the OpenMP directives are translated to a C++ program that explicitly creates and manages parallelism according to the specified directives. Both phases are implemented using the same mechanisms in our infrastructure.

1 Introduction

The use of OpenMP within the OpenMP research community seems complicated by the lack of easy to use compiler infrastructure. Although much work is focused on OpenMP for FORTRAN 77 and FORTRAN 90, and there may be an abundance of C language compiler infrastructure; the unavailability of C++ compiler infrastructure has significantly limited the many research opportunities. In this paper, we present a useful infrastructure, ROSE [1], to assist the OpenMP research community generally, but particularly for OpenMP/C++ research.

Our infrastructure allows the automated introduction of OpenMP directives based on the semantics of user-defined abstractions. The introduction of pragmas, when adding OpenMP directives to a given code, is one of many possible applications. Another one is the translation of OpenMP directives; the recognition of specific pragma directives and the translation of associated code fragments to generate a program that explicitly creates and manages parallelism. We shall use a running example to illustrate both phases and how the ROSE infrastructure [1] can simplify these tasks. Through this example, we demonstrate the relatively simple specification of an OpenMP transformation to use the lower level Nanos Library for OpenMP [2]. We also discuss how to modify that transformation to implement the full OpenMP standard. Given the semantic similarity

** This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 13–25, 2003.

between most OpenMP runtime libraries, we expect that transformations for other OpenMP runtime libraries should be equally simple.

Since within ROSE we have the full type resolution within the AST, and not just syntax, the type information of specific user-defined types can be used as a basis for the optimization of applications that use them. And by including knowledge of the semantics of specific abstractions, fundamentally more information is available to the compiler and greater levels of optimization are often possible, depending upon the abstractions. We will show through the use of an array abstraction, that because the stronger array semantics is satisfied by the weaker OpenMP constraints we can automate the introduction of OpenMP directives into otherwise serial code. This approach permits fundamentally serial code to use the additional semantics of the array abstractions and be run as parallel code.

2 Infrastructure

The ROSE infrastructure offers several components to build a source-to-source translator. A complete C++ frontend is available that generates an object-oriented annotated abstract syntax tree (AST) as intermediate representation. Several different components can be used to build the midend of a translator: to operate on the AST, a predefined traversal mechanism, a restructuring mechanism, and an attribute evaluation mechanism can be used to implement a transformation. Other features are for example parsing of OpenMP directives and integrating these directives into the AST. A C++ backend can be used to unparse the AST and generate C++ code (see fig. 1).

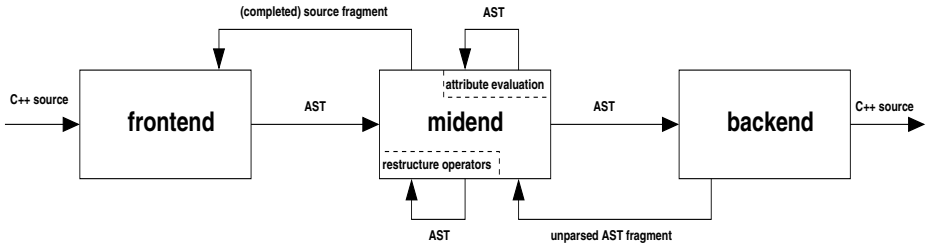


Fig. 1. ROSE Source-to-Source infrastructure with frontend/backend reinvocation

2.1 Frontend

We use the Edison Design Group C++ frontend (EDG) [3] to parse C++ programs. The EDG frontend generates an AST and performs a full type evaluation of the C++ program. The AST is represented as a C data structure. We translate this data structure into an object-oriented abstract syntax tree which is used by the midend as intermediate representation.

2.2 Midend

The midend supports restructuring of the AST. Code that is added to the AST can be specified as a source string, using C++ syntax, or by constructing subtrees node by node. An AST restructuring operation specifies a location in the AST where code should be inserted, deleted, or replaced. The code can be specified as C++ source string or an AST subtree. A program transformation consists of a series of AST restructuring operations.

The order of the restructuring operations is based on a pre-defined traversal. In a transformation the AST is traversed and different restructuring operations are invoked on the AST. The problem of restructuring the AST while traversing it, is addressed by making restructuring operations side-effect free functions that define a mapping from one subtree of the AST to another subtree. The new subtree is not inserted before the traversal of this subtree is finished. We provide interfaces for invoking restructuring operations that buffer these operations to ensure that no subtrees are replaced while they are traversed.

The attribute evaluation mechanism allows the computation of attribute values for AST nodes. Context information can be passed down the AST as inherited attributes and results of computations on a subtree can be computed as synthesized attributes (passing information upwards the tree). Examples for values of inherited and synthesized attributes are type information, size of arrays, the nesting level of loops, the scopes of associated pragma statements, etc. These values can be used to specify constraints on a transformation, i.e. to decide whether a restructuring operation should be applied.

Our infrastructure allows to use C++ source code strings to define code fragments. Any source string which represents a valid declaration, statement(list), or expression can specify a code pattern to be inserted into the AST. The translation of a source code string, s, into an AST fragment, is performed by reinvoking the frontend. The string, s, is extended by our system to form a complete program. This completed program is parsed into an AST by reinvoking the frontend. From this AST, we extract the AST fragment that corresponds to the source string s. This AST fragment is inserted into the AST.

2.3 Backend

The AST is unparsed and C++ source code is generated. It can be specified to unparse all included (header) files or the source file(s) specified on the command line only. This feature is important when transforming user-defined data types, for example when adding generated methods.

The backend can also be invoked during a transformation, to obtain the source code string that corresponds to a subtree of the AST. Such a string can be combined with new code (also represented as a source string) and inserted into the AST.

Both phases, the introduction of OpenMP directives and the translation of OpenMP directives, can be automated using the above mechanisms, as described in the following sections.

3 Semantics-Driven Introduction of OpenMP Directives

The use of high-level abstractions so greatly improves the productivity of developing scientific applications that we seek a way to address the numerous performance issues associated with it.

3.1 User-Defined Abstractions

User-defined abstractions permit a way to tailor the user-environment to be more domain specific than a general purpose language could allow. General purpose languages are expensive to develop and result from many years of work. The compilers that define the language are both expensive and difficult to develop. Such an investment is only possible for a sufficiently large user group.

Simplifying the development of many applications within a specific domain is commonly done through the development of domain-specific libraries. The libraries invariably define abstractions that hide numerous tedious details associated with the development of applications within a specific domain. The combination of a general purpose language and a domain specific library is not the same as a domain-specific language. The essential difference is that the complete semantics of a library's abstractions are unknown at compile time and, thus, some significant optimizations are impossible for the compiler to implement. The result is all too often that many essential abstractions are abandoned because they can't provide sufficiently high performance.

3.2 A++/P++ Serial and Parallel Array Class Library

We use a motivating example from the A++/P++ array class library [4] to show how the ROSE framework can be used by the library writer to develop a source-to-source translator that optimizes code based on high-level semantics. The example uses two classes which are implemented twice; once in the serial A++ library and again in the parallel P++ library. Within our motivating example we consider the following trivial five-point stencil array operation. In figure 2, `A` and `B` are multidimensional array objects of type `floatArray`. `I` and `J` are `Range` objects that together specify a two dimensional index space of the arrays `A` and `B`. The following sections demonstrate how ROSE supports the optimization of a scientific application code through our running example.

3.3 Automated Insertion of OpenMP Directives

Because of the parallel semantics of the A++ and P++ array objects, their use is interchangeable. This permits serial applications to be developed using A++ (serial arrays) and then recompiled to run in distributed memory mode using P++ (parallel arrays). Some simple constraints are that any use of non A++ array objects not constrain the data-parallel model that is hidden within the array semantics.

Since the parallel array semantics of A++ and P++ are consistent with those of OpenMP, OpenMP directives can safely introduce shared memory parallelism into all uses of A++ and P++ array objects. This is essential for the automated insertion of OpenMP directives without complex dependence analysis of the serial code.

3.4 Example C++ Code

The example codes in figure 2 and figure 3 demonstrate the transformation of high-level A++ code to highly efficient OpenMP code. The two codes are semantically equivalent, but the first code shows the use of high-level array abstractions. The semantics of the array abstractions are similar to those of array statements in FORTRAN 90, but the implementation is a (C++) class library instead of a (FORTRAN77) language extension. Clearly, the standard compilation process cannot take the semantics of the array class objects into account since those semantics are user defined. At this high level of abstraction, the C++ compiler is quite powerless to introduce any significant optimizations, precisely because the abstraction's semantics that are relevant to critical optimizations are user-defined and unknown.

```
int n;
Range I,J,K;
floatArray A(n,n,n);
floatArray rhs(n,n,n);
floatArray B(n,n,n);
...
A(I,J,K) = rhs(I,J,K) + ( B(I+1,J,K) + B(I-1,J,K) + B(I,J-1,K) +
           B(I,J+1,K) + B(I,J,K-1) + B(I,J,K+1) - 6.0 * B(I,J,K) );
```

Fig. 2. Example: Code fragment showing the use of A++/P++ array semantics

The high-level A++ code can be automatically transformed into the greatly expanded, but more efficient code shown in figure 3. The ROSE infrastructure allows the library implementer to leverage the semantics of the array class objects that are required to implement the transformation in a source-to-source translator that provides a library-specific compilation process. Specifically, the ROSE frontend creates an AST. The traversal mechanism allows the targeted array class statements to be located in the code. The restructuring mechanism is used to replace the high-level code with the corresponding, but more efficient code and the attribute mechanism supports important details of the transformation such as proper declaration of the loop control variables. A very small and almost trivial part of the transformation is the additional step to have the transformation also generate the OpenMP directive before the outermost loop.

3.5 Discussion

The ROSE mechanisms provide a general approach for the optimization of complex libraries that is not specific to the A++/P++ library. We use this example

```
#define SC(x1,x2,x3) /* case UniformSizeUnitStride */ (x1)+(x2)*_size1+(x3)*_size2
#pragma omp parallel for private (_3, _2, _1) \
       shared (AIJKpointer, rhsIJKpointer, BIJKpointer)
for (_3 = 0; _3 < _length3; _3++) {
     for (_2 = 0; _2 < _length2; _2++) {
        for (_1 = 0; _1 < _length1; _1++) {
           AIJKpointer[SC(_1,_2,_3)] =
               rhsIJKpointer[SC(_1,_2,_3)] +
               (BIJKpointer[SC((_1 + 1),_2,_3)] + BIJKpointer[SC((_1 - 1),_2,_3)] +
                BIJKpointer[SC(_1,(_2 - 1),_3)] + BIJKpointer[SC(_1,(_2 + 1),_3)] +
                BIJKpointer[SC(_1,_2,(_3 - 1))] + BIJKpointer[SC(_1,_2,(_3 + 1))] -
                6.0 * BIJKpointer[SC(_1,_2,_3)] );
        }
     }
   }
```

Fig. 3. Example: Transformed A++/P++ array class code fragment showing the insertion of an OpenMP directive (excluding preceding declarations)

because it is both a high-level abstraction specifically tailored to parallel scientific computing and because it is one with which we are familiar. Improving the performance of the A++/P++ library also has a direct impact on other applications and libraries using it (the Overture Framework [5] in particular).

4 Translation of OpenMP Directives

We use ROSE to build a specialized source-to-source translator that transforms OpenMP directives into lower-level code using an OpenMP runtime library. For our work, we have selected the Nanos OpenMP runtime library [2], but our intention is to demonstrate that any runtime library could be used. We believe our approach would be nearly the same for any OpenMP runtime library, given the seemingly strong semantic resemblance between the few that we have seen. An aspect of our effort is to show how easily other researchers within the OpenMP community could use the ROSE compiler infrastructure for OpenMP research. We hope that access to open compiler infrastructure for C, and particularly for C++, will be found useful.

4.1 Translation Specification

Before translating OpenMP directives into runtime library calls, we must first define a specification that maps the input and output of the translation. Figure 4 presents an example of such mapping, which translates the OpenMP `parallel-for` directive (with the `shared`, `private`, `default` and `schedule` clauses) into calls to the lower-level Nanos OpenMP runtime library [2]. We choose the `parallel-for` directive because it is suitable for illustrating our OpenMP source-to-source translator (shown in Figure 5) and because the ROSE infrastructure can automatically introduce it using the A++/P++ array semantics, as shown in Figure 3. After applying the mapping in Figure 4, our OpenMP source-to-source translator can further transform the OpenMP code in Figure 3 into the Nanos runtime library calls; the result is shown in Figure 6.

```
Input:
    #pragma omp parallel for schedule($scheduletype, $chunksize) default ($defaulttype) \
                                shared($shared_var_list) private($private_var_list)
    for ($i = $lb; $i <= $ub; $i + = $step) {
        $loop_body
    }

Output:
    void supportingOpenMPFunction$id( int* intone_me_01, int* intone_nprocs_01,
                                      int* intone_master01, $shared_var_decl_list )
    {
       $private_var_decl_list;
       int intone_start, intone_end, intone_last;

       intone_begin_for($lb, $ub, $step, $chunksize, $scheduletype);
       while (intone_next_iters( &intone_start, &intone_end, &intone_last)) {
              for ($i = intone_start; $i <= intone_end-1; $i + = $step) {
                     $loop_body
              }
       }
       intone_end_for(true)
    }

    int intone_nprocs_01 = intone_cpus_current();
    intone_spawnparallel( supportingOpenMPFunction$id, $numOfArgs, intone_nprocs_01,
                          $shared_var_list);
```

Fig. 4. Specification for translating the OpenMP parallel-for directive into Nanos runtime library calls (the bold text marks OpenMP keywords, and the $ sign denotes parameters of the input and output fragments.)

```
(1)Parse the C++/C input program and construct an Abstract Syntax Tree
   Parse the OpenMP directives in the constructed AST
(2)Traverse the Abstract Syntax Tree of the input program
   At each tree node astNode:
       if ( (pragma = PrevStatement(astNode)) is an OpenMP directive)
           string OpenMP_support_func = parameterized supporting-function string for pragma
           for (each parameter par in OpenMP_support_func)
               string par_val = Compute-Parameter-Value(par,astNode)
               String-Replace-Substring(OpenMP_support_func, par, par_val)
           Add OpenMP_support_func into global scope
           OpenMP_replace_pragma = parameterized intone_spawnparallel call for pragma
           Substitute parameters in OpenMP_replace_pragma with correct values
           replace pragma and astNode subtrees with OpenMP_replace_pragma
(3)Unparse the Abstract Syntax Tree
```

Fig. 5. Algorithm for translating OpenMP directives into runtime library within the ROSE infrastructure

In general, to provide translation support for the entire set of OpenMP directives, we need to specify a translation mapping, such as the one in Figure 4, for each OpenMP directive. These mappings should be easily constructed from the manual of an OpenMP runtime library. We then use these mappings to instantiate the general translation algorithm in Figure 5. Though currently we have implemented only the translation of the `parallel-for` directive within the ROSE infrastructure, other OpenMP directives can be translated in a similar fashion.

```
void supportingOpenMPFunction__0_0( int* intone_me_01, int* intone_nprocs_01,
   int* intone_master_01, float * AIJKpointer, float * rhsIJKpointer,
   float * BIJKpointer, int _length1, int _length2, int _size1, int _size2)
   {
     int _1, _2, _3;
     int intone_start, intone_end, intone_last;
     intone_begin_for(0,100,1,0,0);
     while(intone_next_iters(&intone_start,&intone_end,&intone_last)) {
          for (_3 = intone_start; _3 <= intone_end; _3++) {
               for (_2 = 0; _2 < _length2; _2++) {
                   for (_1 = 0; _1 < _length1; _1++) {
                        AIJKpointer[_1 + _2 * _size1 + _3 * _size2] =
                        rhsIJKpointer[_1 + _2 * _size1 + _3 * _size2] +
                         (BIJKpointer[(_1 + 1) + _2 * _size1 + _3 * _size2] +
                          BIJKpointer[(_1 - 1) + _2 * _size1 + _3 * _size2] +
                          BIJKpointer[_1 + (_2 - 1) * _size1 + _3 * _size2] +
                          BIJKpointer[_1 + (_2 + 1) * _size1 + _3 * _size2] +
                          BIJKpointer[_1 + _2 * _size1 + (_3 - 1) * _size2] +
                          BIJKpointer[_1 + _2 * _size1 + (_3 + 1) * _size2] -
                          6.0 * BIJKpointer[_1 + _2 * _size1 + _3 * _size2] );
                     }
                }
            }
       }
     intone_end_for(true);
   }
intone_nprocs_01 = intone_cpus_current();
intone_spawnparallel( supportingOpenMPFunction__0_0, 8, intone_nprocs_01, AIJKpointer,
                     rhsIJKpointer, BIJKpointer, _length1,_length2,_size1,_size2);
```

Fig. 6. Example: Transformed A++/P++ array class code fragment using the Nanos runtime library

4.2 Translation Algorithm

Figure 5 presents the structure of a ROSE source-to-source translator that transforms an arbitrary OpenMP directive into its corresponding runtime library calls. This source-to-source translator is separated into the following three phases.

The first phase uses the front end of ROSE to parse the input program into an AST, which provides support for most C++ high-level constructs and thus closely matches the structure of the original program. Within the same phase, the source-to-source translator then makes a second pass of the constructed AST to expand the OpenMP directives. Unlike the C++ front end, the OpenMP construct parser is not already implemented in ROSE and thus needs to be provided by the OpenMP source-to-source translator. It is our plan to provide a full implementation of this parser within our OpenMP source-to-source translator.

The OpenMP construct parser not only translates each string pragma into structured AST nodes, it also automatically collects all the implicit parallelization information pertinent to the OpenMP directive. For example, after this pass, even if the `parallel-for` directive in Figure 4 does not have a `shared` clause (assuming all variables are shared by default), the OpenMP parser will automatically collect the set of shared variables and then insert a `shared` clause into the parsed pragma. The exact behavior for variables in either $shared_var_list or $private_var_list is determined by the `default` clause (if present) and is im-

plemented entirely in the OpenMP parser. Thus, the subsequent phases of the translation algorithm can assume that all data storage attributes are explicit (this is equivalent to having a `default (none)` clause in the original work-sharing construct).

The second phase of the OpenMP source-to-source translator then traverses the AST and transforms the fully expanded OpenMP directives within the AST. At each node *astNode*, if the statement *pragma* immediately before *astNode* is an OpenMP directive, we translate this directive by first constructing a supporting function (*OpenMP_support_func*) for the original code (the subtree rooted at *astNode*). This supporting function is a parameterized string provided by the translation mapping specification (e.g., the section output in Figure 4). We then proceed to substitute all the parameters in the supporting-function string with their corresponding string values pertinent to the original code. Since the source-to-source translator has the pre-knowledge about all the parameters in the *OpenMP_support_func* string, it can compute the values for these parameters by invoking pre-defined AST analysis facilities within ROSE. We then insert the fully expanded *OpenMP_support_func* into the global scope and thus make it another function definition of the original C++ program. Next, we create a string, *OpenMP_replace_pragma*, that invokes the expanded supporting function using parallel threads (e.g., the *intone_spawnparallel* call in Figure 4). Finally, after substituting the parameters in *OpenMP_replace_pragma* with corresponding values, we use *OpenMP_replace_pragma* to replace both the original OpenMP directive (*pragma*) and the original code fragment (the subtree rooted at *astNode*).

Most steps described above can be realized in a straightforward fashion by simply invoking existing ROSE mechanisms. To illustrate the simplicity of this mapping, Figure 7 presents the ROSE C++ implementation for translating the `parallel-for` directive defined in Figure 4. Here we omit some parameter substitutions due to lack of space. Note that ROSE provides facilities to directly edit parameters in strings and to insert strings directly into the AST (they are parsed into *abstract syntax subtrees* before being inserted into the global AST).

As the final phase, after all the OpenMP directives have been translated, the source-to-source translator unparses the transformed AST to produce a C++ program that includes only calls to the OpenMP runtime library.

4.3 Discussion

Generalizing the source-to-source translator discussed in the preceding sections to provide support for the full OpenMP specification is the subject of on-going work. In this section, we discuss the modifications that our approach requires to provide that support. We consider all OpenMP directives, including any associated clauses.

The source-to-source translator presented thus far implements the OpenMP `parallel-for` construct, including the `private`, `shared`, `default` and `schedule` clauses. The source-to-source translator, as described, does not implement several possible clauses of the directive; extending it to support the remaining

```
OpenMPSynthesizedAttribute
OpenMPTraversal::evaluateRewriteSynthesizedAttribute (
                      SgNode* astNode, OpenMPInheritedAttribute inheritedAttribute,
                      SubTreeSynthesizedAttributes synthesizedAttributeList ) {
  OpenMPSynthesizedAttribute returnAttribute(astNode);
  if ( OmpUtility::isOmpParallelFor(astNode) ) {
    SgForStatement *forStatement = isSgForStatement(astNode);
    string supportFunction = " \n\
void supportingOpenMPFunction_$ID ( int* intone_me_01, int* intone_nprocs_01,
                                    int* intone_master01, $SHARED_VAR_DECL_LIST ) { \n\
    $PRIVATE_VAR_DECL_LIST; \n\
    int intone_start, intone_end, intone_last; \n\
    intone_begin_for($LB,$UB,$STEP,$CHUNKSIZE,$SCHEDULETYPE); \n\
    while ( intone_next_iters(&intone_start,&intone_end,&intone_last)) { \n\
      for ($LOOPINDEX = intone_start; $LOOPINDEX <= intone_end; $LOOPINDEX += $STEP) { \n\
        $LOOP_BODY; \n\
      } \n\
    } \n\
    intone_end_for(true); \n\
  } \n";
    string spawnParallel = " \
intone_nprocs_01 = intone_cpus_current(); \n\
intone_spawnparallel(supportingOpenMPFunction_$ID,$NUM_ARGS,intone_nprocs_01,\
$SHARED_VAR_LIST);\n";

    // Edit the function name and define a unique number as an identifier
    string uniqueID = buildUniqueFunctionID();
    supportFunction = StringUtility::copyEdit(supportFunction, "$ID",uniqueID);
    spawnParallel = StringUtility::copyEdit( spawnParallel, "$ID",uniqueID);

    // Edit the loop parameters into place
    string loopBody = forStatement->get_loop_body()->unparseToString();
    supportFunction = StringUtility::copyEdit(supportFunction, "$LOOP_BODY",loopBody);
    ... // similar copyEdits for $LOOPINDEX, $LB, $UB, $STEP

    // Edit the OpenMP parameters into place
    OmpUtility ompData (astNode);
    string privateVarDeclList = ompData.generatePrivateVariableDeclaration();
    string sharedVarList = ompData.generateSharedVariableFunctionParameters();
    string sharedVarDeclList = ompData.generateSharedVariableFunctionDeclarations();
    supportFunction = StringUtility::copyEdit(supportFunction,
                      "$SHARED_VAR_DECL_LIST",sharedVarDeclList);
    supportFunction = StringUtility::copyEdit(supportFunction, "$SHARED_VAR_LIST",
                      sharedVarList);
    spawnParallel = StringUtility::copyEdit(spawnParallel,
                      "$SHARED_VAR_LIST",sharedVarList);
    supportFunction = StringUtility::copyEdit(supportFunction,
                      "$PRIVATE_VAR_DECL_LIST",privateVarDeclList);
    ... //  similar copyEdits for $CHUNKSIZE,$SCHEDULETYPE, and $NUM_ARGS

    AST_Rewrite::addSourceCodeString(returnAttribute, "#include \"nanos.h\"",
                      inheritedAttribute, AST_Rewrite::GlobalScope,
                      AST_Rewrite::TopOfScope, AST_Rewrite::TransformationString, false);
    AST_Rewrite::addSourceCodeString( returnAttribute, supportFunction, inheritedAttribute,
                      AST_Rewrite::GlobalScope, AST_Rewrite::BeforeCurrentPosition,
                      AST_Rewrite::TransformationString, false);
    AST_Rewrite::addSourceCodeString( returnAttribute, transformationVariables,
                      inheritedAttribute, AST_Rewrite::LocalScope, AST_Rewrite::TopOfScope,
                      AST_Rewrite::TransformationString, false);
    AST_Rewrite::addSourceCodeString( returnAttribute, spawnParallel, inheritedAttribute,
                      AST_Rewrite::LocalScope, AST_Rewrite::ReplaceCurrentPosition,
                      AST_Rewrite::TransformationString, false);
  }
  return returnAttribute;
}
```

Fig. 7. Example: Code fragment showing translation of an OpenMP directive

clauses is straightforward. As discussed in section 4.2, parsing of the construct determines the lists of private and shared variables, including those for which the storage attribute is implicit. The construct parsing can easily be modified to build lists for the other data attribute clauses. As discussed in the Nanos documentation [6], variables with the `firstprivate` and `lastprivate` attributes become arguments to the call of the supporting function with corresponding internal variable names for the parameters. The only other change necessary to our source-to-source translator is to include the appropriate assignment between the internal variable name and the name used in the loop body in the supporting function string. The `reduction` clause requires similar changes, with the assignment guarded by a lock that is initialized prior to spawning the parallel region. The `if` clause requires that *OpenMP_replace_pragma* be extended to include the *intone_spawnparallel* call in an *if* statement with the original code cloned into the *else* clause, which is easily implemented with the ROSE restructuring mechanism.

Changes to the source-to-source translator that would support splitting the combined `parallel-for` directive are not difficult. In order to support the OpenMP `parallel` construct (i.e., without the *for* loop), the string used for the supporting function would only include the portions that establish the variable lists and the original code. We can support stand-alone OpenMP `for` constructs by replacing the pragma and original code with the body of the supporting function instead of the *intone_spawnparallel* call. In order to implement orphaned directives correctly with separate compilation, the runtime library must support this in-place replacement.

Straightforward modifications to the source-to-source translator will also extend it to implement the other work-sharing constructs and synchronization directives. The Nanos documentation discusses how to implement the `sections` construct and the `single` directive as variations of the `for` construct, while the replacement code for the synchronization constructs are even simpler. Although we could modify the replacement code to use other calls for runtime libraries that provide calls specific to the `sections` construct and the `single` directive, we plan to implement them as variants of the `for` construct initially.

We have not fully determined how to support `threadprivate` storage in our source-to-source translator. Our support for `threadprivate` storage is highly dependent on the support provided by the OpenMP run time library. The Nanos runtime library targets FORTRAN, and uses pseudo-dynamically allocated storage. More straightforward solutions are possible in C and C++ and one option is to provide an alternative mechanism. Whether or not we use the existing support of the runtime library, we expect that providing support for `threadprivate` storage will be fairly straightforward if it has static block-scope; while the support may be more complex for file-scope or name-space scope, particularly for initializating the storage.

The generality of the OpenMP translation in Figure 5 and the just discussed modifications depends on specific design properties of the OpenMP runtime library. In particular, given an OpenMP runtime library implementation, if a

translation interface similar to Figure 4 can be defined for each OpenMP directive, the source-to-source translator can easily be adapted to provide all the necessary translation support. Otherwise, if the translation of a particular OpenMP directive not only depends on itself and the source code that it applies to, but also depends on the subtle variations of its enclosing context, the algorithm in Figure 5 may not be directly applicable.

An example is the treatment of OpenMP `threadprivate` clauses. If the translation interface requires the OpenMP source-to-source translator to generate different output code patterns depending on whether or not `threadprivate` storage has been previously used, a straightforward adaptation of Figure 5 will not work. For such cases, more complicated global analysis and transformation techniques are required.

5 Related Work

Although a number of compilers were developed to support OpenMP applications, most OpenMP research projects [2, 7–9] only support applications written in C or FORTRAN. Because commercial C++ compilers, such as the SGI MIPSpro [10], the IBM XL [11], the Intel KAI Guide [12], and the Fujitsu for SPARC Solaris [13], target specific machine architectures and do not provide an open source-to-source transformation interface to the outside world, they cannot be used by the research community directly to plug in different OpenMP implementations. As the result, no OpenMP source-to-source translator was available for research into optimizing C++ applications. By providing a flexible source-to-source translator, we present an open research infrastructure for optimizing C++ constructs and OpenMP directives.

Previous research source-to-source translators provide various infrastructures for optimizing OpenMP directives. In particular, the OdinMP/CCp compiler [7] takes a C-program with OpenMP directives and produces a C-program for POSIX threads. In contrast, the Omni compiler [8] translates the OpenMP pragmas in C-programs into runtime library calls, which in turn then invoke either POSIX or Solaris threads. The NanosCompiler [2] and the Polaris compiler [9] translate Fortran programs with OpenMP directives in a similar fashion as the Omni compiler. In addition to OpenMP-directive translation, most of these infrastructures also investigate techniques to automatically generate OpenMP directives and to optimize the parallel execution of OpenMP applications. We complement the previous research by presenting an infrastructure for the C++ OpenMP pragma translation and for the automatic generation and optimization of C++ parallel applications.

6 Conclusions and Future Work

We have presented infrastructure for the transformation of C and C++ applications. We have used the semantics of high-level abstractions to demonstrate the automated introduction of OpenMP directives to parallelize serial codes. Finally

we demonstrated the translation of a representative OpenMP directive using the Nanos library.

In future work we will make available the OpenMP translation phase as a separate component. This will permit anyone defining transformations to specify them more simply via OpenMP directives and to then process the AST to generate the final code automatically using an OpenMP runtime library.

We are considering applying the ROSE infrastructure to the optimization of the use of OpenMP runtime libraries. This third aspect of ROSE-based OpenMP support would be similar to the A++/P++ source-to-source translator in that it would optimize library use, based domain-specific semantics. For example, we could specialize the use of the Nanos runtime library for special cases for which commercial compilers yield significant performance gains, such as when the number of threads is set to one.

References

1. Daniel Quinlan, Brian Miller, Bobby Philip, and Markus Schordan. Treating a user-defined parallel library as a domain-specific language. In *16th International Parallel and Distributed Processing Symposium (IPDPS, IPPS, SPDP)*, pages 105–114. IEEE, April 2002.
2. Eduard Ayguade, Marc Gonzalez, and Jesus Labarta. Nanoscompiler: A research platform for openMP extensions. In *European Workshop on OpenMP*, September 1999.
3. Edison Design Group. `http://www.edg.com`.
4. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.
5. Federico Bassetti, David Brown, Kei Davis, William Henshaw, and Dan Quinlan. OVERTURE: An object-oriented framework for high-performance scientific computing. In *Proceedings of Supercomputing'98 (CD-ROM)*, Orlando, FL, November 1998. ACM SIGARCH and IEEE. Los Alamos National Laboratory.
6. Centre Europeu de Parallelism de Barcelona, Spain. *Nanos Manual*. `http://nereida.deioc.ull.es/html/nanos.html`.
7. Christian Brunschen and Mats Brorsson. OdinMP/CCp - a portable implementation of openMP for c. In *European Workshop on OpenMP*, September 1999.
8. Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka. Design of openMP compiler for an SMP cluster. In *European Workshop on OpenMP*, September 1999.
9. Seung Jai Min, Seon Wook Kim, Michael Voss, Sang Ik Lee, and Rudolf Eighmann. Portable compilers for openMP. In *Workshop on OpenMP Applications and Tools*, July 2001.
10. Silican Graphics Inc. *Optimizing Compilers for High-Performance Computing*. `htpp://www.sgi.com/developers/devtools/languages/mipspro.html`.
11. IBM. *VisualAge C++ Professional for AIX V6.0*. `http://www-1.ibm.com/servers/eserver/ecatalog/us/software/6146.html`.
12. Xinmin Tian, Aart Bik, Milind Girkar, Paul Grey, Hideki Saito, and Ernesto Su. Intel openMP C++/Fortran compiler for hyper-threading technology: Implementation and performance. *Intel Technology Journal*, 6(1):36–46, 2002.
13. Fujitsu. *Fortran & C Packages for SPARC Solaris*. `http://www.fr.fse.fujitsu.com/devuk/solaris.shtml`.

Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization*

Zhenying Liu, Barbara Chapman, Yi Wen, Lei Huang,
Tien-Hsiung Weng, and Oscar Hernandez

Department of Computer Science, University of Houston
{zliu,chapman,yiwen,leihuang,thweng,oscar}@cs.uh.edu

Abstract. A so-called SPMD style OpenMP program can achieve scalability on ccNUMA systems by means of array privatization, and earlier research has shown good performance under this approach. Since it is hard to write SPMD OpenMP code, we showed a strategy for the automatic translation of many OpenMP constructs into SPMD style in our previous work. In this paper, we first explain how to interprocedurally detect whether the OpenMP program consistently schedules the parallel loops. If the parallel loops are consistently scheduled, we may carry out array privatization according to OpenMP semantics. We give two examples of code patterns that can be handled despite the fact that they are not consistent, and where the strategy used to translate them differs from the straightforward approach that can otherwise be applied.

1 Introduction

OpenMP has emerged as a popular parallel programming interface for medium-scale high performance applications on shared memory platforms. However, there are some problems associated with obtaining high performance under this model, and they are exacerbated on ccNUMA platforms. These include the latency of remote memory accesses, poor cache memory reuse, barriers and false sharing of data in cache.

SPMD-style programs access threads and assign computations to them using the thread ID, rather than via the straightforward loop level OpenMP directives. By means of the first-touch policy, data may be allocated to the local memory of a thread if they are first accessed by this thread. Therefore the affinity between the data and thread is constructed. The following loop iterations can be reorganized to reuse the data that are already in the local memory. For example, [18] showed how this method can be used to exploit memory affinity with the example of LU decomposition. The OpenMP LU code is rewritten so that each thread will reuse the data that are first accessed by the current thread. The performance of this kind of SPMD style code is better than the

* This work was partially supported by the DOE under contract DE-FC03-01ER25502 and by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 26–41, 2003.

original OpenMP code due to the data locality [18]. However, when the reused data are not stored consecutively in memory as can happen, for example, in an OpenMP Fortran program, if an array is accessed frequently by row dimension simultaneously by multiple threads, the array elements required by multiple threads may co-exist on the same page. So the performance of the program may suffer from false sharing problems at the page level, depending on the size of each array dimension and the size of a page. Even if we reuse data stored consecutively in memory, there may be some page allocation overheads which are not controlled by the compiler or the user.

More aggressive SPMD style coding privatizes arrays systematically by creating private instances of (sub-)arrays. These codes show good scalability for ccNUMA systems [5, 6, 25]; they are superior to an OpenMP program with straightforward parallelization of loops and an SPMD OpenMP program taking advantage of the first touch policy running on ccNUMA systems. The false-sharing problem is alleviated when accessing the privatized data, since the privatized arrays are allocated in the local stack of the current thread. However, a number of nontrivial program modifications are required to convert a program to the SPMD style. It is thus hard for a user to write SPMD style OpenMP code, especially for a large application. Ease of program development is a major motivation for adopting OpenMP and it is important to provide some help for users who wish to improve their program performance by array privatization. One method is to provide a tool that supports the generation of SPMD style OpenMP code from an OpenMP code with loop-level parallelism. We are building just such a tool based upon the open source Open64 compiler infrastructure.

We have developed a compiler strategy in order to translate loop-parallel OpenMP code into an SPMD-like form [17]. This requires automatic privatization of a large fraction of the data or arrays (that is, transformation of original *shared* arrays into *private* or *threadprivate* ones), the creation of shared buffers to store the resulting *non-local* array references (when one thread needs the private array elements from another thread), storage for the non-local data accessed (the halo area), and the generation of instructions to copy data to and from buffers with the required synchronization.

We show the components of the tool that translates OpenMP programs into equivalent SPMD ones in Fig. 1: compiler frontend, consistent loop scheduling, privatization analysis, overlap analysis, translation, optimization and compiler backend. Oval objects represent the *frontend* and *backend* of the existing OpenMP compiler; we are using the Open64 compiler [20] due to its powerful analyses and optimizations. The rectangular objects enclosed by dashed lines represent the main steps in our SPMD translation. An OpenMP program must be analyzed to

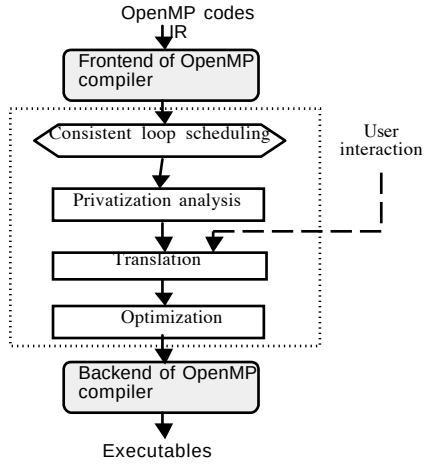


Fig. 1. The framework of SPMD translation

test for *consistent loop scheduling* before the *privatization analysis*, which determines the profitability of the privatization. Sometimes we may be able to enforce the consistency of loop schedules. However, our definition of consistency is relatively restrictive and there are a variety of additional situations in which we are able to perform our SPMD transformation despite the fact that this test fails. We give two examples of code patterns that can be handled despite the fact that they are not consistent, and where the strategy used to translate them differs from the straightforward approach that can otherwise be applied. In other cases, we need to interact with the user if we are to accomplish the task of array privatization. The basic *translation* and *optimization* have been thoroughly described in [17]. In this paper, we focus on issues related to consistency of loop scheduling and privatization analysis.

2 Consistent Loop Scheduling for Data Reuse

The goal of our SPMD transformation is to help the user achieve a scalable OpenMP code. To do so, we attempt to perform array privatization by following the semantics OpenMP loop scheduling. Therefore we need to ensure that the different parallel loops in the code lead to a consistent reuse of portions of individual arrays. If they do so, the area that they reference will be used to construct a private array. Without consistency in array accesses, array privatization will potentially lead to large private arrays and involve extensive sharing of data between threads, both of which are undesirable. We may exploit compiler technology to detect whether the loop scheduling is consistent, by using the semantics of loop scheduling directives in OpenMP to discover the loop partitioning and summarize the array sections associated with each thread within the loop using regular section analysis. In this section, we discuss the consistency test. Afterwards, we show how we deal with loops involving procedure calls, before considering how this information is exploited. Our ability to handle programs is not limited to those with consistency; this issue is explored later.

2.1 Consistency Test

In order to test the consistency of loop scheduling, we use regular section analysis to construct the array sections referenced by individual threads within parallel loops. Array section analysis is a technique to summarize rectilinear sub-regions of an array referenced in a statement, sequence of statements, loop or program unit. Here, our main focus is on forming a section to describe the references made by a thread within a parallel loop. An array section will suffice to describe the region of an array referenced by a thread as the result of a single occurrence of that array in a statement within the loop. We summarize the region of the array referenced by that thread by forming the union of the array sections that are derived from the individual references. Currently, we have decided to rely on a standard, efficient triplet notation based regular sections [10], rather than more precise methods such as simple sections [1], and region analysis [24]. However, when a union of two array sections cannot be precisely summarized, we do not summarize them immediately, but keep them in a linked list. A summary

of the linked list is forced if the length of the list reaches a certain threshold, and we mark that the access region of that parallel loop is approximate. We refer to this information as the *array section per thread* for the loop. We also compute the complete array section that is referenced in the parallel loop (by all threads). We check the consistency of the regions loop scheduling for each shared array. If the parallel loops are consistent for a shared array, we call this array *privatizable*.

Once we have created the array section per thread information for each parallel loop, we must check to determine whether or not threads consistently access the same portion of an array. Our *consistency test algorithm* works as follows. We first summarize the array section accesses for each parallel loop as indicated above. We then compare the array sections obtained for a given thread throughout the computation. We currently have strict requirements for consistency, as given in the three rules below. We hope to be able to relax them somewhat in future. Essentially, these rules describe a test between a pair of loop nests and cover a few cases. In one case, two loop nests are operating on entirely different regions of an array, so that the data referenced by a thread in one loop is distinct from the data referenced by any thread in the second loop. An example would be a pair of loops where one loop initializes the boundary of a mesh and the second loop initializes interior elements. Another case covers the situation when the array section per thread in one loop subsumes the array section per thread in the second loop. We give a simple example below. There are a number of ways in which the definition of consistency might be extended to cover cases where the array sections referenced in the loops are "roughly" the same. At present, we have chosen to rely upon the following definitions and to explore separately how to treat a variety of codes that only partially conform to it, or do not conform at all.

Let A_1 be the section of a shared array A that is accessed (by all threads) in a parallel loop L_1. Let A_1^t be the subsection of A_1 that is accessed by thread t. Similarly, let A_2 be the section of array A that is referenced in loop L_2 and A_2^t be the subsection of A_2 that is accessed by thread t in L_2. We consider the references to array A, and therefore the loop scheduling with respect to A, to be consistent between the parallel loops L_1 and L_2 if one of the following rules apply.

- **Rule 1**: In order to know if the loop scheduling is consistent, we first compute whether the intersection of A_1 $A_2 =$. If so, it is consistent; otherwise, we apply rule 2.
- **Rule 2**: We check whether the array section accessed by thread t in one of the loops contains the array section accessed by thread t in the other parallel loop by using the union operation. If A_1^t $A_2^t = A_1^t$ or A_1^t $A_2^t = A_2^t$ one array section contains the other. In this case, we consider the references in the pair of loops to be consistent. Otherwise, we apply rule 3 to further compute the array sections of C.
- **Rule 3**: We compute region $C = A_1^t$ A_2^t, where C is the intersection of array section A_1 and A_2, and it is the common subsection shared by A_1 and A_2. If it is 90% of the entire set of elements A accessed in the two loops by thread t, we consider the references in the pair of loops to be consistent.

The rules for the consistency test are conservative in that they handle only some of the cases in real world applications that can be handled. Further investigation is needed to extend this test. The test will be applied interprocedurally for each shared array.

2.2 Examples of Consistent and Inconsistent Schedules

```
!$omp parallel default(shared) private(i,j)
!$omp do
    do j = 2 , 999
       do i = 2 , 999
         A(i,j) = ( B(i-1,j)+B(i+1,j)
                + B(i,j-1)+B(i,j+1)) * c
       end do
     end do
!$omp do
      do i=1, 1000
        do j= 1, 1000
          B(i,j) = A(i,j)
        end do
       end do
!$omp end parallel
```

Fig. 2. A Jacobi code example with inconsistent loop scheduling

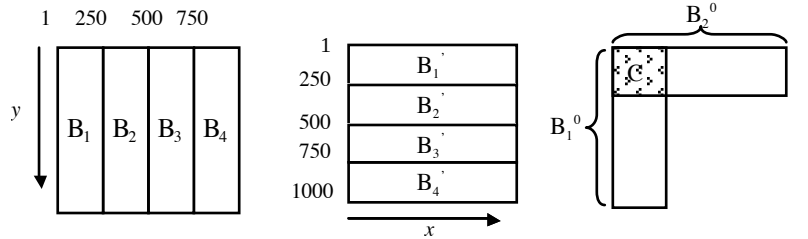
```
!$omp parallel default(shared) private(i,j)
!$omp do
    do j = 2 , 999
       do i = 2 , 999
         A(i,j) = ( B(i-1,j)+B(i+1,j)
                + B(i,j-1)+B(i,j+1)) * c
       end do
     end do
!$omp do
      do j=1, 1000
        do i= 1, 1000
          B(i,j) = A(i,j)
        end do
       end do
!$omp end parallel
```

Fig. 3. Another Jacobi code example with consistent loop scheduling

In OpenMP, a parallel do loop is partitioned into several sets of iterations according to the loop scheduling clauses (which may be the default clause). Threads will be assigned their own sets of iterations and will therefore require access to the subsections of shared arrays that occur in their iterations simultaneously. As mentioned, we use the term consistent loop scheduling to imply that threads access roughly the same regions of an array within multiple parallel loops in OpenMP.

We use the Jacobi program excerpt in Fig. 2 as an example to illustrate the concept of consistency and to show how to apply the consistency test. This code contains references to two arrays. We consider array B; similar reasoning applies to A. Our first rule for consistency checks whether the pair of loops access disjoint regions B_1 and B_2 of the array. However, the intersection of these regions is not empty and so we apply rule 2. Rule 2 is not satisfied either, because $B_1^t \quad B_2^t \neq B_1^t$ and $B_1^t \quad B_2^t \neq B_2^t$. In other words, B_1 and B_2 do not contain each other. Then we calculate the C as shown in Fig. 4(c), where thread 0 is selected to determine the consistency. Since C is far less than 90% of the entire set of elements A accessed in the two loops by thread t, the loop schedule is not consistent and B is not privatizable.

If we examine the loop nest informally, we will see that the strategy for paralleliz-ing the first loop implies that a given thread will access a block of rows of B, whereas n the second loop nest, the thread will access a block of columns of B. This lack of consistency in access is in some sense encouraged by OpenMP, as the user is advised that loop nests may be individually parallelized. Although better strategies exist, they may not have been chosen. The program in Fig. 3 shows a different parallelization strategy. In this case, threads will access blocks of rows of A and B in each loop nest. Furthermore, when we evaluate these regions, we determine that for array A the array sections per thread are identical. For array B, the region referenced in the first loop will subsume the region accessed in the second loop. Therefore the loop schedules are consistent with reference to both A and B, these arrays are privatizable, and we will be able to apply our privatization algorithm without further examination of the code.



(a) Array sections for 4 threads, first parallel loop

(b) Array sections for 4 threads, second parallel loop

(c) Array sections for thread 0, both parallel loop

Fig. 4. Jacobi program and consistency test of Fig. 2

3 Interprocedural Analysis

Our ability to determine consistency of access in a reasonable fashion relies on our ability to determine the array elements accessed by threads accurately. For this we may defer combining array sections in a loop nest. However, we also want to minimize the introduction of inaccuracies during interprocedural analysis. We consider how to do so next.

3.1 Call Graph

Parallel loops may contain calls to procedures. We must also evaluate access patterns throughout the entire program. In order to do so well, our interprocedural analysis is implemented on call graphs created by a precise call graph algorithm [26]. For any data flow problem including our privatization problem, one call site may have several different sequences of actual arguments as a consequence of several different call chains that will be followed at run time. The call graphs constructed by most call graph algorithms are not able to take the multiple sequences of actual arguments into consideration [3, 9, 22]. Hence they are sometimes imprecise. To overcome this, we have developed an algorithm that enables us to precisely determine the call chains that will occur. Our call graph is a multi-graph where each procedure in the program is represented by a single node, as usual. Directed edges between nodes indicate that the source procedure invokes the sink procedure. An edge records information on the predecessor edge in a call chain of the program. For each pair of nodes in the call graph, there may be zero, one or more edges connecting them. However, no two edges will be identical with respect to the associated information.

```
      Program Jacobi
   …
!$omp parallel default(shared)
!$omp&shared(A,B,size) private(k)
       do k=1,ITER
         call sub1(B,size)
         call sub2(A,B,size)
         call sub3(A,B,size)
       end do
!$omp end parallel
    …
```

```
subroutine sub1(A,size)
integer size,i,j
double precision A(0:size,0:size)
!$omp do
    do j=0,size
      do i=0, size
          A(i,j) = 1.0
      end do
    end do
!$omp end do
end
```

```
subroutine sub2(A,B,size)
integer i,j,size
double precision A(0:size,0:size)
double precision B(0:size,0:size)
!$omp do
    do i = 1, size-2
      do j = 1, size-2
          A(i,j) = (B(i,j-1) + B(i,j+1)
                  + B(i-1,j) + B(i+1,j))
      enddo
    enddo
!$omp end do
end
```

```
subroutine sub3(C,D,size)
integer m,n,size
double precision C(0:size,0:size)
double precision D(0:size,0:size)
!$omp do
     do m = 0, size-1
       do n = 0, size-1
         D(m,n) = C(m,n)
       enddo
     enddo
!$omp end do
end
```

Fig. 5. A Jacobi OpenMP program with several program units

We give a simple example to show the information that is appended to the edges in the graph. It considers a Jacobi program consisting of several program units in Fig. 5 and its call graph in Fig. 6. There are four nodes in the call graph to represent the procedures. The edges in the call graph represent the calling relations. For example, the edge e_1 depicts that the *main* calls *sub1*.

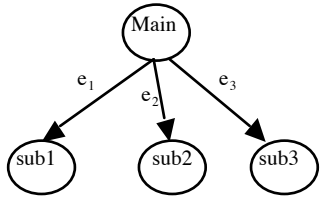


Fig. 6. The call graph for the Jacobi OpenMP program in Fig. 5

3.2 Interprocedural Algorithm

In order to handle parallel loops containing procedure calls, we must determine the array region accessed within the called procedure, using our call graph to help do so.

The algorithm in Fig. 7 operates on the call graphs we introduced in Section 3.1. It traverses the call graph in a depth-first manner. The algorithm is recursive. It follows a call chain from a program entry point to the end of this call chain. Upon returning, the algorithm gathers and binds information in bottom-up order which guarantees correct binding between formal parameters and actual arguments.

The initial invocation will be Inter_consistency_analysis(*Main*, *Null*) where *Main* is the program entry point and thus has no (*Null*) predecessor edges. In our call graph, each out edge of *Main* node has no predecessor. The *local consistency analysis* in line 3 involves the consistency test in Section 2.1, and records the corresponding array section information. When we execute line 7 to line 10, each out edge of *Main* node will be examined individually. The algorithm gathers and assembles information for determining consistency from their callees and callees' successors, then binds the information to these edges. The array section per thread is finally achieved by forming the union of the individual array sections corresponding to array references. Here, we also follow the strategy of deferring the merging of array sections unless a threshold is reached. We also use this call graph and the associated information to ensure that our subsequent handling of the program is accurate, as will be explained in the previous section.

```
1.  Procedure Inter_consistency_analysis(node v1, Edge e_prev)
2.    if v1.complete= False then
3.     v1.privatizaion =Local_consistency_analysis(v1)
4.     v1.complete=True
5.    end if
6.    Result = 1
7.    for each edge ej where ej.prev= e_prev in v1
8.     ej.privatizable=binding(Inter_consistency_analysis(ej,v2,ej),ej)
9.     Result = Result   ej.privatization
10.   end for
11.  Result = v1.privatization   Result
12.   return Result
13. End procedure
```

Fig. 7. The interprocedural consistency analysis algorithm

The algorithm needs to calculate local information for a procedure node only once, while the algorithm may visit a node more than once. Line 2 to line 5 of the algorithm in Fig. 7 guarantees this by examining the value of v_1.complete, which is true for completeness, and false for incompleteness. Lines 7 to line 11 gather and assemble array section information for every edge, for which e_{prev} is their predecessor edge in the call chain. The binding function binds the formal parameters to the actual ones, where e_j stands for one of the out edges of the current procedure node v_1, and v_2 for one of the callees of the procedure v_1.

4 Privatization Analysis

4.1 Privatization Algorithm

Our privatization analysis works as follows. We first test if the OpenMP program has consistent loop scheduling. If so, we will carry out the privatization according to OpenMP semantics. If it is largely consistent, then we may be able to modify the remaining "inconsistent" parallel loops to obtain consistency. This entails identifying which loop in the loop nest would result in the prevailing access pattern if parallelized; if dependence tests prove that this is legal, we then replace the originally parallelized loop with this one and an appropriate schedule. We may also be able to apply loop interchange to support this. Since only one level of parallelism is supported in OpenMP, we deal with one dimensional privatization at this stage. BLOCK partition is a default manner of privatization. Even if the loop scheduling is consistent, we still have to detect whether it is profitable to privatize arrays. For example, if we have large shadow areas (halos) for threads and share data extensively between threads, it is not likely to be profitable to privatize arrays. If the above methods do not enable us to privatize arrays, we may be able to detect and deal with a known special case. We illustrate this by discussing two important special cases, LU decomposition OpenMP program and ADI-like code, below. Although neither of these have consistent array usage, array privatization can be used in both cases to achieve good performance. In Fig. 8, we show the above process in the privatization analysis algorithm.

```
1.  Procedure Privatization_Analysis
2.     Consistency test
3.     if (inconsistent) then
4.        Data dependence test
5.          Applying loop transformation
6.     else
7.          Profitability test
8.          Translation OpenMP into SPMD style with array privation
9.     end if
10.          if ( known special cases ) then handling the special cases
11.          else   reporting to the users
12.          end if
13.       End procedure
```

Fig. 8. Privatization analysis algorithm

We use an LBE OpenMP example to illustrate the local privatization algorithm. In Fig. 9, part of the OpenMP LBE program is shown. LBE, a computational fluid dynamics code, solves the Lattice Boltzmann equation and is provided by Li-Shi Luo of ICASE, NASA Langley Research Center [11]. It uses a 9-point stencil, and the neighboring elements are updated at each iteration. The consistency test will have a positive result in this case: the loop schedule is consistent and array f and $fold$ are

privatizable, since both of the parallel loops distribute the iterations in the *j*-loop which sweeps the second dimension of array *f* and *fold*. Each thread will access a contiguous set of columns of the original array. When such results are obtained from the test, we can immediately determine the size and shape of the corresponding private arrays for individual threads: we simply use the union of the array sections reference in the different parallel loops. In this case, the transformation will be profitable because each thread only accesses the array elements inside an individual array section all the time. Once we privatize *f* and *fold*, only two columns of arrays are non-local and the data sharing between threads is trivial in contrast to the computation.

4.2 Special Case 1: LU

```
!$omp parallel
  do iter = 1, niters
!$omp do
  do j = 1, Ygrid
    do i = 1, Xgrid
            fold(i,0,j) = f(i,0,j)
            ……
            fold(i,8,j) = f(i,8,j)
   end do
  end do
!$omp end do
……
!$omp do
do j=1, Ygrid – 1
  do i=2, Xgrid -1
     f (i,0,j) = Fn(fold(i,0,j)
     f (i+1,1,j) = Fn(fold(i,1,j))
     f (i,2,j+1) = Fn(fold(i,2,j))
     ……
     f(i+1,8,j-1) = Fn(fold(i,8,j))
  end do
end do
!$omp end do
……
```

Fig. 9. OpenMP LBE code

```
!$omp parallel
  do k = 1, n-1
!$omp single
  lu(k+1:n,k) = lu(k+1:n,k)/lu(k,k)
!$omp end single
!$omp do
    do j = k+1,n
      lu(k+1:n, j) =
        lu(k+1:n,j) – lu(k,j)* lu(k+1:n,k)
   end do
!$omp end do
  end do
!$omp end parallel
```

Fig. 10. OpenMP LU decomposition

In some cases when the loop scheduling in OpenMP program is not consistent, we cannot arrive at a suitable array privatization by means of following its OpenMP semantics, for instance, the Jacobi program in Fig. 2 and the LU OpenMP program in Fi.g 10. If we apply loop interchange to the second loop nest, and parallelize the new outer loop to get another Jacobi program in Fig. 3, then threads will access roughly the same set of data in each loop and will moreover reference contiguous areas in memory.

In the case of LU decomposition, since its OpenMP program does not consistent schedule the loops, we may ask the user if we can privatize the arrays. In the LU program, the inner loop is a parallel one, while the loop bounds of the inner loop involve the upper level loop iteration variable. Therefore the inner parallel has the dynamic loop bounds and the associate array regions for each thread changes from

iteration to iteration and the size of array regions is shrinking. Although it is inconsistent loop scheduling, it is a good candidate to privatize arrays in the CYCLIC manner in the column dimension, and there is little data sharing between threads.

4.3 Special Case 2: ADI

ADI (Alternating Direction Implicit) application [12] is another special example in which each parallel loop sweeps a distinct dimension, and it spends almost the same execution time on each parallel loop. The data dependence in ADI prevents from loop interchanging. Therefore we are not able to privatize arrays by directly following OpenMP semantics.

```
!$omp parallel
!$omp do
        do j=1, N
          do i= 1, N
             A(i,j)=A(i,j)-B(i,j)*A(i-1,j)
          end do
        end do
!$omp end do
!$omp do
        do i= 1,N
          do j= 1, N
            A(i,j)=A(i,j)-B(i,j) * A(i,j-1)
          end do
        end do
!$omp end do
!$omp end parallel
```

Fig. 11. The ADI-like OpenMP code

```
Double precision A(0:N,N),B(0:N,N)
!$omp parallel
       chunk = N/omp_get_num_threads()
!$omp do
       do j=1, N
          do i= 1, N
             A(i,j) = A(i,j) - B(i,j) * A(i-1,j)
          end do
       end do
!$omp end do
     sync(1:N,omp_get_thread_num())=.F.
      do i= 1,N
!$omp flush ( A, sync )
       if(id.ne.0 .and..not. sync(i,id-1))then
         do while (.not. sync(i,id-1))
!$omp flush ( A, sync )
          end do
       end if
       do j=1+id*chunk,chunk + id*chunk
            A(i,j) = A(i,j)- B(i,j) * A(i,j-1)
       end do
       sync(i,id) = .T.
!$omp flush
     end do
!$omp end parallel
```

Fig. 12. ADI-like OpenMP code in SPMD style

```
!$omp threadprivate(Aloc, Bloc)
      sync = .F.
       chunk = N/omp_get_num_threads()
!$omp parallel shared(shadow, sync)
        do j=1, chunk
          do i= 1, N
             Aloc(i,j)=Aloc(i,j)  &
               Bloc(i,j)*Aloc(i-1,j)
          end do
        end do
      do i= 1,N
!$omp flush ( shadow, sync )
          if ( id .ne. 0 ) then
             do while (.not. sync(i,id-1))
!$omp flush ( shadow, sync )
             end do
             Aloc(i,0) = shadow(i,id-1)
          end if
          do j= 1, chunk
             Aloc(i,j)=Aloc(i,j)-   &
               Bloc(i,j)*Aloc(i,j-1)
          end do
          shadow(i,id) = Aloc(i,chunk)
          sync(i,id) = .T.
!$omp flush (shadow,sync)
       end do
!$omp end parallel
```

Fig. 13. ADI-like OpenMP code in SPMD style with array privatization

ADI OpenMP code can be rewritten into an SPMD style one containing consistent loop scheduling. Furthermore, we may privatize arrays according to the OpenMP semantics of this OpenMP SPMD code. In Fig. 14, we show results of executing three different versions of the ADI code. The experiments were conducted on Origin 2000 systems at the National Center for Supercomputing Applications (NCSA) in multi-user mode. MIPSpro 7.3.1.3 Fortran 90 compiler was used with the options: -mp. We set on the first touch policy and set off the page migration environmental variable. Altogether, the SPMD style code with array privatization outperforms its OpenMP and SPMD OpenMP code without array privatization.

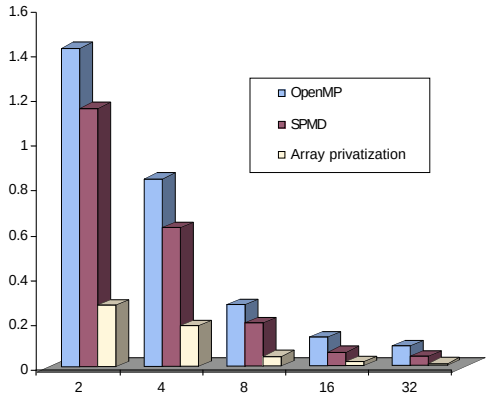


Fig. 14. The execution time of a pure OpenMP program, SPMD OpenMP programs without and with array privatization

5 Related Work

Scalability of OpenMP codes is naturally a concern on large-scale platforms. The data locality problem for OpenMP on ccNUMA architectures is well known and a variety of means have been provided by vendors to deal with it, including first touch allocation. Researchers have proposed strategies for page allocation [19] and migration; data distribution directives are implemented by SGI [23] and Compaq [2] to improve data locality, although their directives differ. Page granularity distributions are easier to handle, but may be imprecise. An element-wise (HPF-like) data distribution can distribute array elements to the desired processor or memory. For example, the DISTRIBUTE_RESHAPE directive provided by SGI uses the specification of the distribution to construct a "processor_array" which contains pointers to the array elements belonging to each processor, so as to guarantee the desired distribution. However, a consequence is that pointer arithmetic is introduced to realize accesses to the array elements, and performance of the resulting code is poor if options that optimize

pointers are not selected. Instead of this approach, our strategy is to base the translation on *threadprivate* arrays which are guaranteed to be local and do not involve the use of additional pointers.

A number of researchers investigated the problem of finding an efficient data layout automatically for codes that were being compiled for distributed memory systems in the context of HPF [8, 13]. The major steps that they identified in the search for a data distribution are as follows: alignment analysis [15, 14], decision on the manner of distribution (BLOCK or CYCLIC), adjustments to the block size of dimensions for a cyclic distribution, and determination of how many processors are to be used as the target for each distributed dimension.

But there are significant differences between this problem and ours. On distributed memory systems, each array must be distributed in order to enable parallelism: it is usually not feasible to replicate more than a few arrays, as memory costs will be prohibitive. This is not true for OpenMP programs. Even if some of the shared arrays in an OpenMP program are not privatized, we are still able to execute the code in parallel quite reasonably. Therefore, we have more options. In addition to that, automatic data distribution for distributed memory systems is based on sequential programs which have no analysis for potential parallelism. In contrast, OpenMP programs already include information that some loops are parallelizable. Moreover, if we follow the semantics of the user-specified OpenMP loops, we are provided with an assignment of work, and hence data references, to the individual threads. This information, in the form of array sections that are accessed by the threads, gives us initial information for determining an appropriate array privatization. Note, too, that the regions may overlap and that our analysis is greatly simplified by the fact that OpenMP specifies a simple and direct mapping of loop iterations to threads, rather than requiring us to consider each statement in a loop nest individually. While this may lead to a larger amount of data being referenced by multiple threads, it makes our problem much more tractable.

A large body of work exists that considers strategies for gathering and storing information on array sections accessed, and for summarizing these both within individual procedures and across a program. Among the best know strategies are regular sections [3], simple sections [1], atomic images and atoms [16]. We use bounded regular sections in this context. However, we have adopted a new strategy for obtaining precision, by merging regions where this does not lead to loss of precision and otherwise recording lists of references. We have also considered the problem of recording and using the correct call chains when performing this analysis.

In this paper, we have focused on the problem of array privatization for arrays that are accessed in regular (structured) patterns, which is quite widespread. In [21], the access pattern statistics are summarized for Perfect and SPEC benchmark suite. The authors note a very high percentage of simple affine subscripts (e.g., A(i)) in many applications. For instance, among the thirteen programs in its statistical table 1, four of them including *swim* and *tomcatv*, have at least 97% accesses that are simple affine subscripts. The *Ocean* program has only 32.3% affine subscripts; all the other 12 programs have at least more than 50% simple affine subscripts.

6 Conclusions and Future Work

OpenMP codes must contend with the data locality problems incurred by multiple caches and remote memory access latency. The SPMD style enhances the capability of OpenMP to achieve better data locality. This paper discusses the analyses and optimizations which will enable us to partially automate the translation of loop-parallel codes to SPMD style via a compiler. Thus the performance benefits are obtained without devolving this burden onto the users.

We have focused our attention here on applications with regular data access patterns. OpenMP programs provide an explicit mapping of computation to threads and hence specify which data is needed by an individual thread; we thus have a basis for implementing a privatization algorithm. However, these codes may lead to some difficult problems. In the ADI (Alternating Direction Implicit) kernel, the access pattern of shared arrays changes for different parallel loops makes the array privatization very hard to perform according to OpenMP semantics. We have determined that a transformation into a macro pipelined version (that can be described by using FLUSH directives) can help us obtain data locality and considerably lessen the synchronization cost for ADI-like applications. In the current state of our work, we must ask the user to help us decide how to deal with codes where the access patterns change.

More work is needed to improve our ability to perform this work automatically, to determine in some cases whether or not privatization is profitable, and to lessen the amount of global synchronization incurred in OpenMP codes. There are also many possible ways in which we could attempt to improve upon the code produced. We plan to perform experiments to learn more about these issues. We will also experiment to find out whether it is beneficial in practice to break one critical section into two or more smaller critical sections, and how beneficial it is to change global synchronization to point-to-point synchronization. We expect to replace the barriers in our generated SPMD style codes via FLUSH directives in order to improve our results.

Finally, we plan to test our SPMD style codes not only on ccNUMA systems, but also in a PC cluster running Omni/Scash, where our aggressive array privatization should have a strong impact on performance.

References

1. Balasundaram, V., and Kennedy, K.: A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations. Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, Oregon, June 21-23, (1989) 41-53
2. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C.A., and Offner, C.D.: Extending OpenMP for NUMA machines. Scientific programming. Vol. 8, No. 3, (2000)
3. Callahan, D. and Kennedy, K.: Analysis of Interprocedural Side Effects in a Parallel Programming Environment. Journal of Parallel and Distributed Computing. Vol. 5, (1988)
4. Chandra, R., Chen, D.-K., Cox, R., Maydan, D.E., Nedeljkovic, N., and Anderson,

J.M.: Data Distribution Support on Distributed Shared Memory Multiprocessors. Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation, Las Vegas, NV, June (1997)
5. Chapman, B., Bregier, F., Patil, A., and Prabhakar, A.: Achieving High Performance under OpenMP on ccNUMA and Software Distributed Share Memory Systems. Currency and Computation Practice and Experience. Vol. 14, (2002) 1-17
6. Chapman, B., Patil, A., and Prabhakar, A.: Performance Oriented Programming for NUMA Architectures. Workshop on OpenMP Applications and Tools (WOMPACT'01), Purdue University, West Lafayette, Indiana. July 30-31 (2001)
7. Gonzalez, M., Ayguade, E., Martorell, X., and Labarta, J.: Complex Pipelined Executions in OpenMP Parallel Appliations. International Conferences on Parallel Processing(ICPP 2001), September (2001)
8. Gupta M., and Banerjee, P.: PARADIGM: A Compiler for Automated Data Distribution on Multicomputers. Proceedings of the 7th ACM International Conference on Supercomputing, Tokyo, Japan, July 1993.
9. Hall, M.W., and Kennedy, K.: Efficient call graph analysis. ACM Letters on Programming Languanges and Systems, Vol. 1, No. 3, (1992) 227-242
10. Havlak, P., and Kennedy, K.: An Implementation of Interprocedural Bounded Regular Section Analysis. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 3, July (1991) 350-360
11. He, X., and Luo, L.-S.: Theory of the Lattice Boltzmann Method: From the Boltzmann Equation to the Lattice Boltzmann Equation. Phys. Rev. Lett. E, No. 56, Vol. 6, (1997) 6811
12. Jin, H., Frumkin, M., and Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. NAS Technical Report NAS-99-011, Oct. (1999)
13. Kennedy, K. and Kremer, U.: Automatic Data Layout for High Performance Fortran. Proceedings of the 1995 Conference on Supercomputing (CD-ROM), ACM Press, (1995)
14. Laure, E. and Chapman, B.: Interprocedural Array Alignment Analysis. Proceedings HPCN Europe 1998, Lecture Notes in Computer Science 1401. Springer, April (1998)
15. Li, J. and Chen, M.: Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Proc. Third Symp. on the Frontiers of Massively Parallel Computation, IEEE. October (1990): 424-433
16. Li, Z., and Yew, P.-C.: Program Parallelization with Interprocedural Analysis, The Journal Jin, H., Frumkin, M., and Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks of Supercomputing, Vol. 2, No. 2, October (1988) 225-244
17. Liu, Z., Chapman, B., Weng, T.-H., and Hernandez, O.: Improving the Performance of OpenMP by Array Privatization. In the Workshop on OpenMP Applications and Tools (WOMPAT 2002), Fairbanks, Alaska, August (2002)
18. Nikolopolous, D.S., Artiaga, E., Ayguadé, E., and Labarta, J.: Exploiting Memory Affinity in OpenMP through Schedule Reuse. Third European Workshop on OpenMP (EWOMP 2001), (2001)
19. Nikolopoulos, D.S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J., and Ayguadé, E.: Is Data Distribution Necessary in OpenMP? Proceedings of Supercomputing 2000, Dallas, Texas, November (2000)
20. The Open64 compiler. http://open64.sourceforge.net/
21. Paek, Y., Navarro, A., Zapata, E., Hoeflinger, J., and Padua, D.: An Advanced Compiler Framework for Non-Cache-Coherent Multiprocessors, IEEE Transactions on Parallel and Distributed Systems. Vol. 13, No. 3, March (2002) 241-259
22. Ryder, B.G.: Constructing the Call Graph of a Program. IEEE Transactions on Software

Engineering, Vol. 5, No. 3, (1979) 216-225
23. Silicon Graphics Inc. MIPSpro 7 FORTRAN 90 Commands and Directives Reference Manual, Chapter 5: Parallel Processing on Origin Series Systems. Documentation number 007-3696-003. http://techpubs.sgi.com/
24. Triolet, R., Irigoin, F., and Feautrier, P.: Direct Parallelization of CALL statements. Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction, July (1986) 176-185
25. Wallcraft, A.J.: SPMD OpenMP vs. MPI for Ocean Models. Proceedings of First European Workshops on OpenMP (EWOMP'99), Lund, Sweden, (1999)
26. Weng, T.-H., Chapman, B., and Wen, Y.: Practical Call Graph and Side Effect Analysis in One Pass. Technical Report, University of Houston, Submitted to ACM TOPLAS (2003)

A Runtime Optimization System for OpenMP*

Mihai Burcea and Michael J. Voss

Edwards S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto, Toronto, ON, Canada
{burceam,voss}@eecg.toronto.edu

Abstract. This paper introduces stOMP: a specializing thread-library for OpenMP. Using a combined compile-time and run-time system, stOMP specializes OpenMP parallel regions for frequently-seen values and the configuration of the runtime system. An overview of stOMP is presented as well as motivation for the runtime optimization of OpenMP applications. The overheads incurred by a prototype implementation of stOMP are evaluated on three Spec OpenMP Benchmarks and the EPCC scheduling microbenchmark. The results are encouraging and suggest that there is a large potential for improvement by the runtime optimization of OpenMP applications.

1 Introduction

In recent years, there has been an increased interest in the runtime optimization of applications. At compile-time, optimizers are severely restricted by incomplete knowledge of the program input, target architecture, and library modules. By performing optimization at runtime, more complete knowledge is available allowing programs to be highly tuned to their exact runtime environment and usage.

In this paper, we present stOMP: a specializing thread-library for OpenMP. stOMP is an implementation of the OpenMP API [1,2] based on the Omni compiler and runtime library [3]. In addition to the implementation of OpenMP provided by Omni, stOMP leverages properties of OpenMP applications and the Omni runtime library to provide an environment for the dynamic optimization and compilation of parallel regions. At runtime, stOMP is used to *specialize* parallel regions for runtime constant or well-behaved variables, and for the current configuration of the parallel environment.

In Section 2, we present related work in dynamic and adaptive program optimization. In Section 3, we discuss the unique properties of OpenMP applications that make them ideal for runtime optimization. In Section 4, we present an overview of the stOMP compilation and runtime subsystems, and details of the stOMP prototype that is currently under development. An initial evaluation of our prototype is provided in Section 5. In Section 6, we present our conclusions and plans for future work.

* This work is supported in part by the National Science and Engineering Research Council, Bell Canada University Labs, the Connaught Foundation and Dell Canada.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 42–53, 2003.

2 Related Work in Runtime Optimization

Runtime compilation and optimization has been most successfully applied in languages that have high abstraction penalties, such as Java, C# and Smalltalk [4,5,6,7]. Often in these languages, runtime compilation can be used to remove some of the high overheads incurred by advanced language features, such as virtual/polymorphic function invocation, dynamic class loading and array bounds checking. Just-in-time and adaptive compilation are now accepted practice for these languages.

There have been mixed results for dynamic optimization when applied to non-object-oriented imperative languages, such as C. The DyC, 'C, and Tempo projects have explored user-directed dynamic compilation of C programs [8,9,10]. These approaches require user-annotation of application programs to select code regions and variables for optimization. Unlike Java, C# and Smalltalk, C programs are less tolerant of the overheads incurred by runtime compilation, and therefore users must carefully select code regions where dynamic compilation will be profitable. In [11], a profile-directed tool, Calpa, is used to automate the selection of regions and specialization targets for DyC. However in [11], it is unclear if the approach scales to large programs.

In [12], ADAPT is proposed for the runtime optimization of loop-based Fortran applications. ADAPT is a generic system that allows compiler developers to easily add and experiment with adaptive optimizations. In [12], the dynamic serialization of parallel OpenMP regions is used as an example application of the system. Unlike stOMP, ADAPT cannot directly influence or leverage the OpenMP runtime library.

3 OpenMP as a Target for Runtime Optimization

The OpenMP API has bindings for both the Fortran [1] and C/C++ [2] languages. As discussed in Section 2, dynamic compilation systems that target these languages usually require users to hand-annotate regions for optimization [8,9,10]. However in the case of OpenMP applications, we believe that useful optimization directives can be inferred from the already existing OpenMP directives, and that OpenMP libraries provide needed features for easily implementing a runtime optimization system.

3.1 Runtime Characteristics of OpenMP Applications

To perform runtime optimization on an application, two basic choices must be made: (1) what code regions should be optimized and (2) what runtime values should be used to specialize these regions. A natural choice for the code regions to optimize in OpenMP applications are PARALLEL regions. These regions have been annotated by users because they are important to the application's performance and they have execution times that are large enough to tolerate parallelization overheads. These traits make them not only ideal for parallelization but for runtime optimization as well.

The second choice that needs to be made is to select the runtime values to use in optimization. Fig. 1 shows the behavior of shared variables in the C and Fortran77 SPEC OpenMP benchmarks [13]. In Fig. 1 (a), a histogram of the number of distinct values per shared-variable is shown by type. Fig. 1 (a) demonstrates that shared variables have only a few values over a program run.

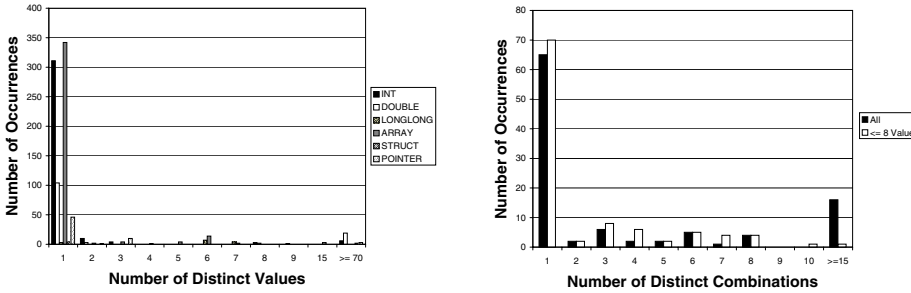


Fig. 1. Figure (a) is a histogram of the number of distinct values held by shared variables at entry to parallel regions in the C and Fortran77 SPEC OpenMP Benchmarks. For ARRAYs, STRUCTs and POINTERs, "Values" correspond to the addresses of the variables, not their content. Figure (b) is a histogram of the total number of combinations of variable values seen at entry to parallel regions. In (b), the combinations obtained when using all variables, as well as the well-behaving variables with less than 8 distinct values each are shown. In both (a) and (b), "Occurrences" is the static count of regions or variables that exhibit the behavior.

Fig. 1 (b) shows the number of distinct combinations of variable values. A combination of variable values is the set of values held by all variables of interest at entry to the region. For example, if a region has two shared variables, A and B, a combination might be $A = 1, B = 3$. In Fig. 1 (b), we present data for combinations derived from all shared variables, as well as combinations derived from the shared variables that have 8 or less distinct values (as shown in Fig. 1 (a)). Fig. 1 (b) suggests that if regions are optimized for frequently-seen combinations of values, only a small number of code versions need to be compiled and managed at runtime.

3.2 Features of OpenMP Implementations

Standard implementations of OpenMP provide the necessary hooks for easily implementing a runtime optimization system. Fig. 2 shows an example of an OpenMP parallel region as annotated by a user. In Fig. 3 (a) and 3 (b), the code generated by the Omni compiler to implement this region is shown.

The call to `_ompc_do_parallel` in Fig. 3 (a), causes the Omni runtime library to create a team of threads to execute the code found in the `__ompc_func_0` rou-

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
  iam = omp_get_thread_num();
  np = omp_get_num_threads();
  ipoints = npoints / np;
  subdomain(x, iam, ipoints);
}
```

Fig. 2. The source code for a simple parallel region

tine. In Section 4, we propose a system that uses the call to `_ompc_do_parallel` to select and create highly specialized and optimized versions of code for each parallel region at runtime.

3.3 Optimization Opportunities

If shared variables can be used to specialize OpenMP applications, there are a number of opportunities for enhancing the performance of these applications. An inspection of the SPEC OpenMP benchmarks shows that simple expressions of shared variables, the thread id and the number of threads, often determine loop bounds and branch conditions. Many static optimizations are severely handicapped when loop bounds and conditionals are unknown. By capturing these values at runtime, more accurate and aggressive loop transformations might be applied to these important parallel nests.

In addition, the transformations applied by OpenMP compilers often create complicated code that is less amenable to compiler optimization. For example, functions calls are added to determine loop schedules, and shared variables may be accessed through pointers, instead of directly, leading to added aliases. A runtime optimization system may be able to see through, or remove, these complications using runtime information, generating better code as a result.

4 The stOMP Runtime Optimization System

Fig. 4 shows an overview of our proposed system, stOMP. The system consists of three major components: a modified version of the Omni OpenMP compiler, a modified version of the Omni runtime library and a Dynamic Optimizer. We briefly describe these three components in the sections below.

4.1 The stOMP Compiler

For each parallel region in the OpenMP application, the stOMP compiler creates a new file that contains the source code for that region. For the code in Fig. 2, this new file would contain a copy of the code shown in Fig. 3 (b). Special attention is paid to ensure that global variables, static variables, and Fortran Common blocks are properly transformed to allow the code to be placed in a separate

```
(*(__ompc_argv)) =
  (((void *) (&x)));

(*((__ompc_argv) + (1))) =
  (((void *) (&npoints)));

_ompc_do_parallel (__ompc_func_0,
                   __ompc_argv);
```

(a)

```
static void
__ompc_func_0 (__ompc_args)
    void **__ompc_args;
{
  auto int _p_iam;
  auto int _p_np;
  auto int _p_ipoints;
  auto int **_pp_x = (((int **)
            (*__ompc_args)));
  auto int *_pp_npoints = (((int *)
    (*((__ompc_args) + (1)))));
  (_p_iam) = (omp_get_thread_num());
  (_p_np) = (omp_get_num_threads());
  (_p_ipoints) =
     ((*_pp_npoints) / (_p_np));
  subdomain (*_pp_x, _p_iam,
             _p_ipoints);
}
```

(b)

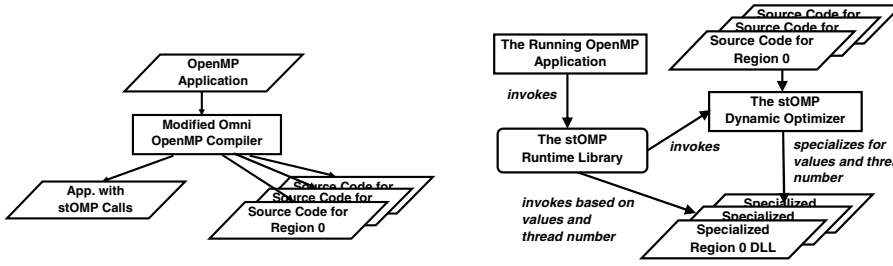
Fig. 3. The parallel region from Fig. 2 after being transformed by Omni: (a) the site of the parallel region and (b) the function created by Omni to encapsulate the body of the parallel region

file, while retaining the semantics of the original code. These files are stored for future processing by the stOMP Dynamic Optimizer. In addition to generating the parallel region files, the stOMP compiler also creates application-specific functions for runtime code management and compilation.

4.2 The stOMP Runtime Library

The runtime system, shown in Fig. 4 (b), tracks shared variables and invokes the stOMP Dynamic Optimizer. The runtime library maintains 8-element, per-region code caches that store dynamically generated code. At each call to the `_ompc_do_parallel` function, the code cache corresponding to the first argument is inspected for a version that matches the current values of the shared variables, number of threads and thread id (referred to as a *combination* of values in Section 3).

Fig. 5 shows an overview of the flow through the `_ompc_do_parallel` function. Before spawning worker threads, the master thread invokes the match function generated by the stOMP compiler for this parallel region. The match function returns a set of functions, one for each thread, that should be invoked to execute the parallel region. If this region has already been specialized for the current combination of runtime values, the set of specialized functions is returned.



(a) (b)

Fig. 4. An overview of stOMP: (a) the compiler and (b) the runtime system

If no matching functions are found, the number of times this combination of values has been seen for this region is compared against a user-set threshold. If the threshold has been exceeded, the stOMP_compile function for this region is invoked to create the corresponding set of functions. If the threshold has not yet been exceeded, the statically-compiled default set of functions is returned and a counter for this combination of values is incremented. If more than 8 combinations are seen for a region, the least-frequently-used cache block will be evicted.

In addition to the heuristic in Fig. 5, we also always specialize immediately for the combination seen during the first invocation of each parallel region. We immediately specialize for the first combination since Fig. 1 shows that most regions have only a single combination of values. This approach also allows regions that execute only once to benefit from runtime optimization.

4.3 The stOMP Dynamic Optimizer

The stOMP Dynamic Optimizer is a hand-coded preprocessor that calls gcc as a back-end to generate shared libraries. The `stOMP_compile_x` function, shown in Fig. 5, invokes the Dynamic Optimizer and dynamically loads the resulting libraries into the executable using standard `dlopen` and `dlsym` calls. When the stOMP_compile function is invoked by the runtime library, it is passed the current number of threads and the current values (or addresses) of the shared variables in use by the region.

A single call to the Dynamic Optimizer creates P specialized versions, one for each thread. The Dynamic Optimizer is coded in C and uses OpenMP pragmas to perform compilation of these P versions in parallel. However, the Dynamic Optimizer itself is currently not being optimized by stOMP. We have yet to fully explore mechanisms for decreasing the overhead of our runtime compilation system, but believe that significant reductions are possible.

Our current optimizer implements a very limited form of runtime constant propagation. The values or addresses of shared variables are explicitly assigned at the top of each parallel region function, providing more accurate values and

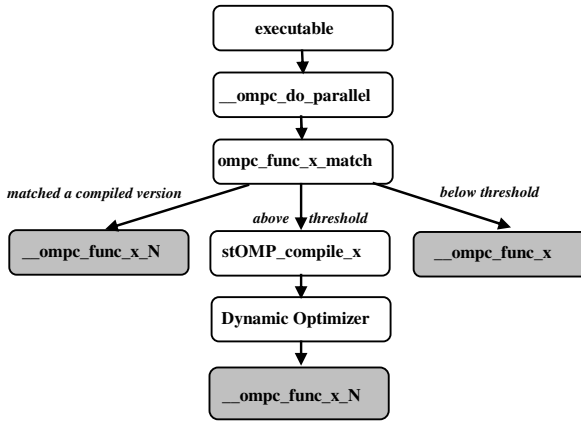


Fig. 5. The process used by the match function to select a copy of the parallel region to execute: The gray boxes represent the selected versions. The __ompc_func_x box is the statically-compiled default version of the code and the __ompc_func_x_N box is a region specialized for the current shared variables, number of threads and thread id

addresses to the back-end optimizer. Also, each call to `omp_get_thread_num` and `omp_get_num_threads` is replaced by its corresponding runtime value (as communicated by the runtime library). Fig. 6 shows an example of a snippet from a parallel region in Equake, a SPEC OpenMP benchmark, before and after optimization by our current Dynamic Optimizer.

The optimizations currently implemented in our Dynamic Optimizer, as Shown in Fig. 6, are unlikely to yield large performance improvements on most programs. However, our current prototype does allow us to gain invaluable insight into the lower-bound on the overheads we can expect from our full system. A study of these overheads is presented in the next section.

5 A Preliminary Evaluation

As described in Section 4, only limited optimizations have been implemented in the current stOMP prototype. In this Section, we therefore present an initial study of the overheads associated with our system. We first explore the overheads added to parallel regions by examining the EPCC Scheduling Microbenchmark (`schedbench`) [14]. Next, we show results from applying our system to three of the SPEC OpenMP Benchmarks: Apsi, Art, and Equake.

Our test system is a 4-processor Intel Xeon server with 1.6 GHz Hyper-threaded processors running Redhat Linux 7.3. We present results for 1 through 4 threads, restricting ourselves to the number of physical processors in the system. All code is compiled using gcc version 2.96 with the `-O2` optimization flag.

```
extern struct smallarray_s **w1;
extern int ARCHnodes;

void quake__ompc_func_31 (__ompc_args)
     void **__ompc_args;
{
  auto int *_pp_j;
  (_pp_j) = (((int *) (*__ompc_args)));
  {
    auto int _p_i;
    auto int _p_i_28;
    auto int _p_i_29;
    auto int _p_i_30;
    (_p_i_28) = (0);
    (_p_i_29) = (ARCHnodes);
    (_p_i_30) = (1);
    _ompc_default_sched (&_p_i_28,
                &_p_i_29, &_p_i_30);
```

```
extern struct smallarray_s **w1;
extern int ARCHnodes;

void
quake__ompc_func_31 (__ompc_args)
     void **__ompc_args;
{
  int _stomp_pp_j = (int) 0;
  auto int *_pp_j;
  _pp_j = &_stomp_pp_j;
  w1 = 0x8538d30;
  ARCHnodes = 30169;
  {
    auto int _p_i;
    auto int _p_i_28;
    auto int _p_i_29;
    auto int _p_i_30;
    (_p_i_28) = (0);
    (_p_i_29) = (ARCHnodes);
    (_p_i_30) = (1);
    _ompc_default_sched (&_p_i_28,
                &_p_i_29, &_p_i_30);
```

Fig. 6. An example of code from one of the parallel loop in Equake: (a) the code before being transformed by the runtime optimizer and (b) the code after runtime optimization

5.1 The Performance of the EPCC Scheduling Microbenchmark

Fig. 7 shows results from schedbench. This benchmark determines the overheads associated with starting and scheduling parallel regions. stOMP uses calls to `_ompc_do_parallel` to select and compile specialized code, and therefore its overheads are directly measured by this benchmark.

In Fig. 7, the improvement of `schedbench` when optimized by stOMP is presented. Fig. 7 (a) and (b) show the improvement for the static scheduling part of the microbenchmark. This benchmark executes a statically scheduled parallel region using varying chunk sizes (ranging from 1 to 128). The benchmark executes the region 50 times for each configuration, and calculates the average execution time of the region for each chunk size.

Fig. 7 (a) shows the improvement calculated from the average execution time measurements. These measurements include the overheads incurred by our Dynamic Optimizer. The Optimizer will be invoked once for each <processor number, chunk size> pair. Fig. 7 (b) shows the improvement calculated when the Dynamic Optimizer overhead is removed (this result includes 49 invocations of the parallel region).

Fig. 7 (b) suggests that the overhead incurred by the stOMP Dynamic Optimizer is the major reason for the poor performance shown in Fig. 7 (a). In fact, when the runtime compilation overhead is removed, stOMP outperforms the orginal Omni code in almost all cases, with an average improvement of 13.2% and gains as large as 33%.

The measurements shown in Fig. 7 (b) include all of the code cache lookup and management overheads incurred by the runtime library. These results show

that improvements are possible from the simple optimizations currently implemented in stOMP, but that the runtime of our Optimizer might need to be reduced.

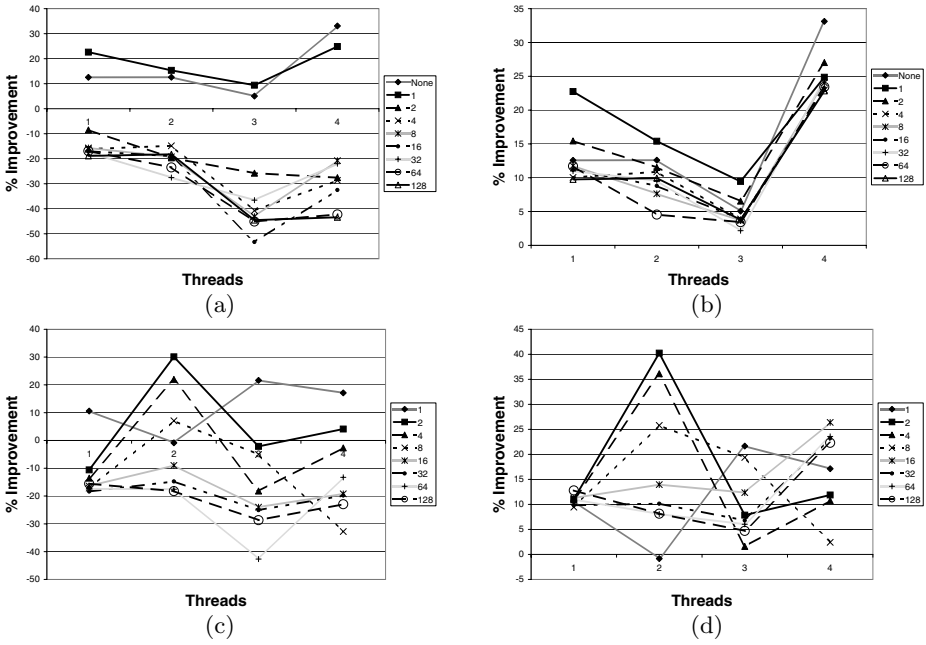


Fig. 7. Evaluation of the EPCC schedbench benchmark. The improvement of stOMP over Omni on the statically scheduled loop (a) including the time to spent in the Dynamic Optimizer and (b) excluding the time spent in the Dynamic Optimizer. The improvement of stOMP over Omni on the dynamically scheduled loop when (c) including the time spent in the Dynamic Optimizer and (d) excluding the time spent in the Dynamic Optimizer

The results for the dynamic scheduling part of `schedbench` are shown in Fig. 7 (c) and 7 (d). These results reinforce the conclusions drawn from Fig. 7 (a) and 7 (b). When the Dynamic Optimizer overhead is ignored, the average improvement on the dynamic loop is 14%, with improvements as large as 40%.

5.2 The Performance of Apsi, Art, and Equake

Fig. 8, 9 and 10 show the performance of three Spec OpenMP Benchmarks: Apsi, Art, and Equake. For both Apsi and Art we used the SPECOMP2001 train data set and for Equake we used the SPECOMP2001 reference data set. Due to the limited optimizations currently implemented in our system, both Apsi and Equake perform worse with stOMP than with the original Omni compiler and library.

Apsi: Apsi has the largest number of parallel regions (with 28) of the three benchmarks that we tested, and also has a large number of shared variables for each parallel region. All regions in Apsi have at least 10 shared variables, with some having as many as 18. We therefore expect to see a larger overhead with Apsi. Fig. 8 (a) shows that on 4 processors Apsi runs 10% slower than the original version. Fig. 8 (b) clearly indicates that this degradation is due to the time spent in the Dynamic Optimizer. Because of the large number of regions, the runtime optimizer is invoked frequently, generating 136 shared libraries during a single execution of the benchmark.

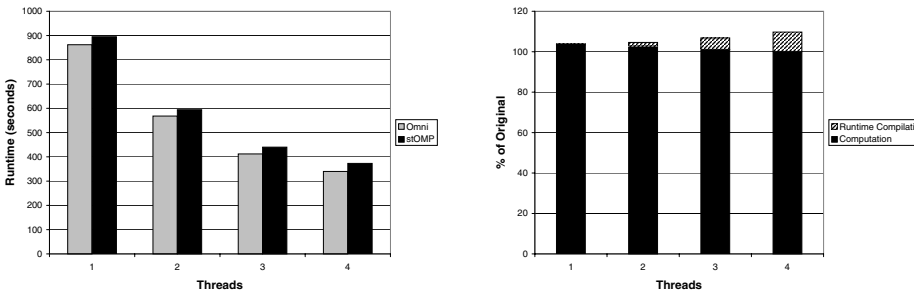
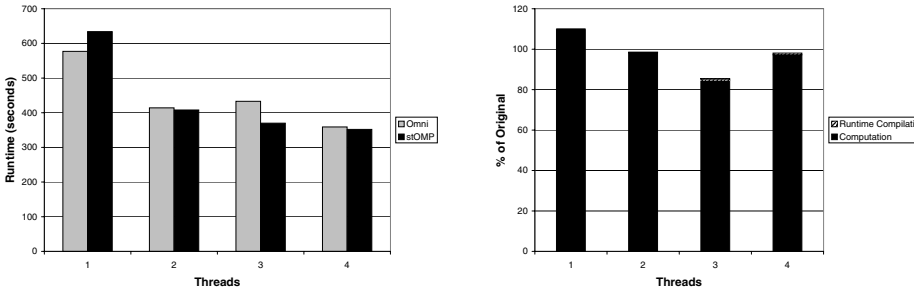


Fig. 8. The Spec OpenMP Benchmark Apsi: (a) the runtime of the original and stOMP versions and (b) the breakdown of the execution time of the stOMP version

Art: Art has 3 parallel regions, each of which is only invoked once. Therefore stOMP just-in-time compiles each region once and incurs no further overheads from compiling or matching. Consequently, the performance of Art in Fig. 9 (a) and (b) is the best of the three benchmarks. The stOMP version of Art shows improvements for all but the 1-processor run, with a gain of 14% on 3 processors. The runtime compilation overheads for Art are negligible.

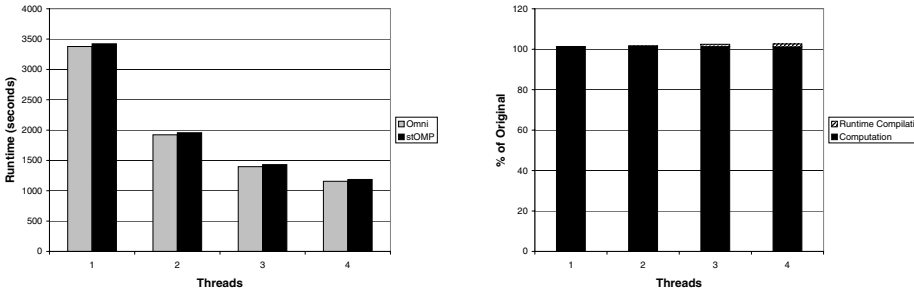
Eqake: Equake has 11 parallel regions, many of which are called multiple times. In Fig. 10 (a), the stOMP version of Equake is always within 3% of the original code. Fig. 10 (b) shows that while compilation overheads increase with the number of threads, the time spent in the Dynamic Optimizer is small.

The results from our initial evaluation of stOMP are encouraging. Both Art and Equake show small runtime compilation overheads, with Art already showing an improvement from the limited optimizations performed by our prototype. The overhead of stOMP when running Equake is always less than 3% and when running Apsi is always less than 10%. In the future, we plan to look at running the Dynamic Optimizer in the background, as is done in ADAPT [12], and thereby hide some of the latencies incurred in programs, such as Apsi, that have a large number of parallel regions.



(a) (b)

Fig. 9. The Spec OpenMP Benchmark Art: (a) the runtime of the original and stOMP versions and (b) the breakdown of the execution time of the stOMP version



(a) (b)

Fig. 10. The Spec OpenMP Benchmark Equake: (a) the runtime of the original and stOMP versions and (b) the breakdown of the execution time of the stOMP version

6 Conclusions

In this paper, we have introduced stOMP: a specializing thread library for OpenMP. The stOMP system is built on the Omni OpenMP compiler and library, and provides a system for the runtime optimization of parallel regions. In Section 3, we present motivation for the runtime optimization of OpenMP applications. In the SPEC OpenMP Benchmarks, it is shown that shared variables are in general runtime constant, or have only a few values during a program's execution.

In Section 4, we describe the architecture of stOMP. Using a combined compile-time and run-time system, stOMP specializes parallel regions for frequently-seen values and the configuration of the runtime system. In Section 5, a preliminary evaluation of stOMP is presented. Our results are encouraging and suggest that there is room for improvement using runtime optimization, but that our runtime compilation system needs to be refined to minimize overheads. In

future work, we will explore a range of runtime optimizations using the stOMP system.

References

1. OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface, V. 2.0*, 2002.
2. OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface, V. 2.0*, 2002.
3. The Omni OpenMP Compiler. `http://phase.etl.go.jp/Omni/`, 2003.
4. Sun Microsystems. The Java HotSpot Performance Engine Architecture. Technical White Paper, `http://java.sun.com/products/hotspot/whitepaper.html`, April 1999.
5. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno jvm. In *Proc. of the ACM SIGPLAN 2000 Conf. on Object-Oriented Programming Systems, Languages and Applications*, Minneapolis, MN, October 2000.
6. Standard ECMA-335: Common Language Infrastructure (CLI). `http://www.ecma.ch/ecma1/STAND/ecma-335.htm`, February 2002.
7. L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proc. of the Conf. on Principles of Programming Languages*, Salt Lake City, Utah, 1984.
8. Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. of the SIGPLAN'99 Conf. on Programming Language Design and Implementation*, pages 293–304, Atlanta, GA, May 1999.
9. Massimiliano Polettto, Wilson C Hsieh, Dawson R Engler, and M. Frans Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, March 1999.
10. Renaud Marlet, Charles Consel, and Philippe Boinot. Efficient incremental runtime specialization for free. In *Proc. of the SIGPLAN'99 Conf. on Programming Language Design and Implementation*, pages 281–292, Atlanta, GA, May 1999.
11. Markus Mock, Craig Chambers, and Susan Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *33rd Annual Symposium on Microarchitecture*, December 2000.
12. Michael Voss and Rudolf Eigenmann. High-Level Adaptive Program Optimization with ADAPT. In *Proc. of PPoPP'01: Principles and Practice of Parallel Programming*, Snow Bird, Utah, June 2001.
13. Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools (WOMPAT)*, pages 1–10, July 2001.
14. J. M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In *European Workshop on OpenMP (EWOMP)*, 1999.

A Practical OpenMP Compiler for System on Chips

Feng Liu[1] and Vipin Chaudhary[2]

[1] Department of Electrical & Computer Engineering, WSU, USA
fliu@ece.eng.wayne.edu
[2] Institute for Scientific Computing, WSU and Cradle Technologies, Inc.
vipin@wayne.edu

Abstract. With the advent of modern System-on-Chip (SOC) design, the integration of multiple-processors into one die has become the trend. By far there are no standard programming paradigms for SOCs or heterogeneous chip multiprocessors. Users are required to write complex assembly language and/or C programs for SOCs. Developing a standard programming model for this new parallel architecture is necessary. In this paper, we propose a practical OpenMP compiler for SOCs, especially targeting *3SoC*. We also present our solutions to extend OpenMP directives to incorporate advanced architectural features of SOCs. Preliminary performance evaluation shows scalable speedup using different types of processors and effectiveness of performance improvement through optimization.

1. Introduction

OpenMP is an industrial standard [1, 2] for shared memory parallel programming with growing popularity. The standard API consists of a set of compiler directives to express parallelism, work sharing, and synchronization. With the advent of modern System-on-Chip (SOC) design, chip multiprocessors (CMP) have become a new shared memory parallel architecture. Two major shared memory models exist today: Symmetric Multiprocessor machines (SMP) and distributed memory machines or clusters. Unlike these two models, SOCs incorporate multiple distinct processors into one chip. Accordingly, it has a lot of new features which normal OpenMP standard could not handle, or at least could not take advantage of.

By far there are no standard programming paradigms for SOCs or heterogeneous chip multiprocessors. Users are required to write complex assembly language and/or C programs for SOCs. It's beneficial to incorporate high-level standardization like OpenMP to improve program effectiveness, and reduce the burden for programmers as well. For parallel chips like *Software Scalable System on Chip (3SoC)* from Cradle, parallelism is achieved among different types of processors; each processor may have different instruction sets or programming methods. Thus, developing a standard parallel programming methodology is necessary and challenging for this new architecture.

Cradle's *3SoC* is a shared-address space multi-processor SOC with programmable I/O for interfacing to external devices. It consists of *multiple processors* hierarchically connected by two levels of buses. A cluster of processors called a Quad is connected

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 54–68, 2003.

by a local bus and shares local memory. Each Quad consists of four RISC-like processors called Processor Elements (PEs), eight DSP-like processors called Digital Signal Engines (DSEs), and one memory Transfer Engine (MTE) with four Memory Transfer Controllers (MTCs). The MTCs are essentially DMA engines for background data movement. A high-speed global bus connects several Quads. The most important feature for parallel programming is that *3SoC* provides 32 semaphore hardware registers to do synchronization between different processors (PEs or DSEs) within each Quad and additional 64 global semaphores [3].

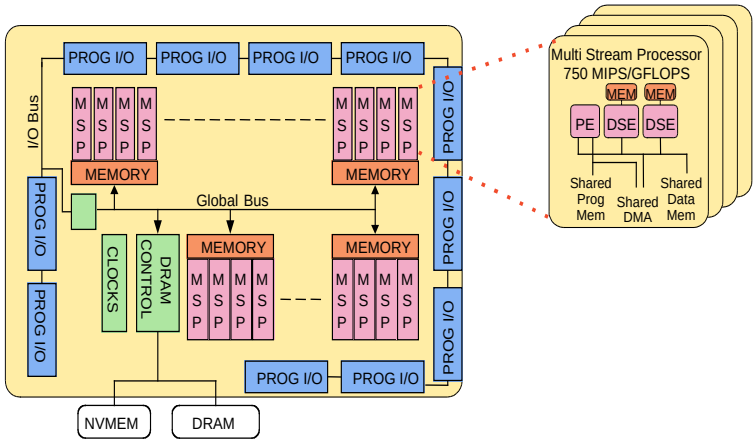


Fig. 1. *3SOC* Block Diagram

In this paper, we describe the design and implementation of our preliminary OpenMP compiler/translator for *3SoC*, with detailed focus on special extensions to OpenMP to take advantage of new features of this parallel chip architecture. These features are commonplace among SOCs; techniques used here may be targeted for other SOCs as well. Giants like Intel, Sun and IBM have addressed their own plans for SOCs and it's believed that CMP will dominate the market in the next few years. We give detailed explanation of each extension and how it should be designed in OpenMP compiler.

In the next section we briefly introduce the parallel programs on *3SoC*, targeting different processors: PEs or DSEs, respectively. In section 3 we present the design of our OpenMP compiler/translator with special focus on synchronization, scheduling, data attributes, and memory allocation. This is followed in section 4 with a discussion on our extensions to OpenMP. Section 5 outlines the implementation aspects of a practical OpenMP compiler for SOCs followed by a performance evaluation in section 6. Conclusion is given in section 7.

2. Parallel Programs on *3SoC*

In this section, we briefly describe the approach to program *3SoC*. Our OpenMP translator will attempt to create such parallel programs.

2.1 Programming Different Parallel Processors

A *3SOC* chip has one or more Quads, with each Quad consisting of different parallel processors: four PEs, eight DSEs, and one Memory Transfer Engine (MTE). In addition, PEs share 32KB of instruction cache and Quads share 64KB of data memory, 32KB of which can be optionally configured as cache. Thirty-two semaphore registers within each quad provide the synchronization mechanism between processors. Figure 2 shows a Quad block diagram. Note that the Media Stream Processor (MSP) is a logical unit consisting of one PE and two DSEs. We will now have a look at how to program parallel processors using PEs or DSEs.

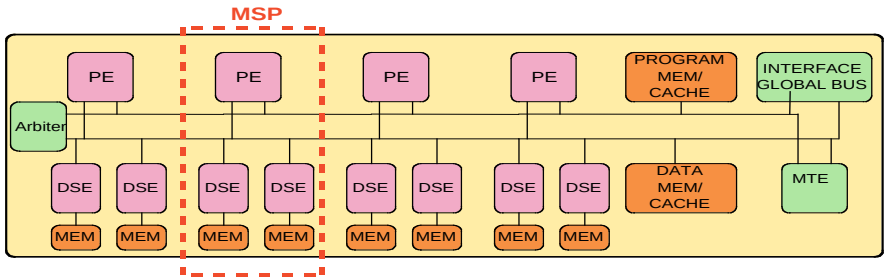


Fig. 2. Quad block diagram

2.2 Programming Using PEs

PE has a RISC-like instruction set consisting of both integer and IEEE floating point instructions. In *3SoC* architecture, there are a number of PEs which can be viewed as several "threads" compared with normal shared memory architecture. Each processor has its own private memory stack, similar to "thread" context. At the same time, each processor is accessing the same blocks of shared local memory inside Quad, or SDRAM outside Quad. These memories are all shared. In a typical *3SoC* program, PE0 is responsible for initializing other processors like PEs or DSEs, so that PE0 acts as the "master" thread while other processors act as "child" threads. Then PE0 will transfer parameters and allocate data movement among different processors. It will load MTE firmware and enable all MTCs. Through data allocation PE0 tells each processor to execute its own portion of tasks in parallel. PE0 will also execute the region itself as the master thread of the team. Synchronization and scheduling must be inserted properly. At the end of each parallel region, PE0 will wait for other processors to finish and collect required data from individual processor.

PEs are programmed using standard ANSI C. The *3SoC* chip is supplied with GNU-based optimizing C-compilers, assemblers, linkers, debuggers, and performance accurate simulators (refer to *3SoC* programmer's guide [4]). To incorporate several PEs to work in a parallel program, the approach is similar to conventional parallel programming [6].

We present the concept of translating an OpenMP program to a *3SoC* parallel program using several PEs. The *#pragma omp parallel* defines a parallel region whereas the directive *#pragma omp single* specifies that only one thread executes this scope. Correspondently, in the *3SoC* main program, each PE is associated with its ID, *my_peid*. The parallelism is started by PE0 *(my_peid==0)*, and only PE0 allocates and initializes all other processors. Once started, all PEs will execute *function()* in parallel. Within its own function context, each PE will execute its portion of tasks by associated processor ID. At the end of each parallel region, all PEs reach *an implicit barrier* where PE0 waits and collects data from other PEs. Finally, PE0 terminates other PEs and releases resources to the system.

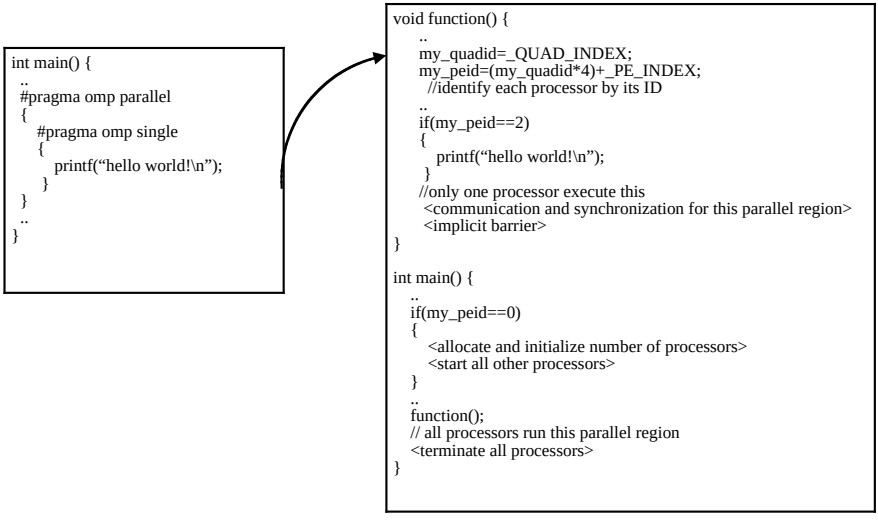


Fig. 3. Translation of a simple OpenMP program (left) to 3SoC parallel region (right)

2.3 Programming Using DSEs

DSE is a DSP-like processor which uses a different programming methodology. DSEs are programmed using C-like assembly language ("CLASM") combined with standard ANSI C. DSEs are the primary processing units within *3SoC* chip. Compilers, assemblers, and tools are supplied for DSE.

Writing OpenMP program for DSE requires a completely different approach. The controlling PE for a given DSE has to load the DSE code into the DSE instruction memory. Then PE initializes the DSE DPDMs (Dual Ports Data Memory) with desired variables and starts the DSE. The PE either waits for the DSE to finish by poll-

ing, or can continue its work and get interrupted when the DSE has finished its task. A number of DSE library calls are invoked. See Fig. 4.

First, the PE initializes the DSE library call via *dse_lib_init(&LocalState)*. Then the PE does some Quad I/O check and data allocation such as assigning initial values for the matrix multiplication. In the next for-loop, the PE allocates a number of DSEs and loads the DSE code into the DSE instruction memory by *dse_instruction_load()*. This is done by allocating within one Quad first, *dse_id[I]=dse_alloc(0),* if failed, it will load from other Quads. Afterwards, the PE loads the DPDM's onto the allocated DSEs, dse*_loadregisters(dse_id)*. Each DSE starts to execute the parallel region from the 0^{th} instruction, via *dse_start(dse_id[i],0)*. PE is responsible for waiting for DSEs to complete computation and terminating all resources.

```
void main() {
        ..
        int dse_id[NUM_DSE];
        dse_lib_init(&LocalState);
        pe_in_io_quad_check();
        ..
        <Data allocation>
        ..
        // load the MTE firmware and enable all MTCs
        _MTE_load_default_mte_code(0x3E);

        for(i = 0; i < NUM_DSE; i++) {
                // allocate a DSE in this Quad
                dse_id[i] = dse_alloc(0);

                if(dse_id[i] < 0) {
                        // allocate DSE from any Quad
                        dse_id[i] = dse_alloc_any_quad(0);
                        if(dse_id[i] < 0) {
                           <error condition>
                        }
                }
                // load the instructions on the allocated DSEs
                dse_instruction_load(dse_id[i], (char *)&dse_function, (char
*)&dse_function_complete, 0);
        }
        // Load the Dpdm's on the allocated DSEs
        DSE_loadregisters(dse_id);

        for(i = 0; i < NUM_DSE; i++) {
                // Start all DSEs from the 0th instruction
                dse_start(dse_id[i], 0);
        }

        for(i = 0; i < NUM_DSE; i++) {
                // Wait for the DSE's to complete and free DSEs
                dse_wait(dse_id[i]);
        }
        <other functions>
        ..
        dse_lib_terminate();
        ..
}
```

Fig. 4. Sample 3SoC parallel program using multiple DSEs

3. Design of OpenMP Compiler/Translator

To target the OpenMP compiler for *3SoC*, we have to cope with the conceptual differences from standard OpenMP programs. OpenMP treats the parallelism at the granularity of parallel regions, and each function may have one or more independent parallel regions; while *3SoC* treats the parallelism at the level of function, where each

private data structure is defined within one function. We also have to treat the data scope attributes between processors like *"Firstprivate", "Reduction"* along with appropriate synchronization and scheduling.

3.1 Synchronization

Hardware semaphores are the most important features that distinguish normal chip multiprocessors from "parallel" chips. Developers can use hardware semaphores to guard critical sections or to synchronize multiple processors. On *3SoC* platform, each Quad has 32 local and 64 global semaphore registers that are allocated either statically or dynamically. For parallel applications on *3SoC*, the semaphore library *(Semlib)* procedures are invoked for allocating global semaphore or for locking and unlocking.

Equipped with this feature, users can implement two major synchronization patterns.

1. By associating one hardware semaphore for locking and unlocking, user can define a *critical* construct which is mutually exclusive for all processors.
2. By combining one semaphore along with global shared variables, OpenMP *barrier* construct can be achieved across all processors. Sample *barrier* implementation is as follows:

```
semaphore_lock(Sem1.p);
done_pe++;        //shared variable
semaphore_unlock(Sem1.p);
while(done_pe<(NOS)); //total # of parallel processors
_pe_delay(1);
```

For OpenMP compiler, *barrier* implementation is important. There are a number of *implicit* barriers existing at the end of each OpenMP parallel region or work-sharing construct. Therefore, hardware semaphores allocation and de-allocation becomes a crucial factor. We take two steps to implement this. First step is to allocate all semaphores into local shared memory as "_SL" to improve data locality, i.e. *"static semaphore_info_t SemA _SL"*. (Type of variables like "_SL" is discussed in Section 3.3). Secondly we allocate semaphores dynamically at run-time. This provides more flexibility than static approach with respect to the limited number of semaphores in each chip. Each semaphore is initialized and allocated at the beginning of the parallel region; while after use, it's de-allocated and returned to the system for next available assignment. This is done by system calls at run-time [4].

3.2 Scheduling and Computation Division

In OpenMP programs, the computation needs to be divided among a team of threads. The most important OpenMP construct is the work-sharing construct, i.e., *for*-construct. The parallelism is started by the OpenMP directive *#pragma omp parallel*. Any statement defined within this *parallel* directive will be executed by each thread of the team. For the work sharing constructs, all statements defined within will be di-

vided among threads. Each thread will execute its own share, like *for*-construct, which is a small portion of loop iterations.

A common translation method for parallel regions employs a *micro-tasking* scheme, like OdinMP [5]. Execution of the program starts with a *master thread*, which during initialization creates a number of *spinner threads* that *sleep* until they are needed. The actual task is defined in *other threads* that are waiting to be called by the spinner. When a parallel construct is encountered, the master thread wakes up the spinner and informs it of the actual task thread to be executed. The spinner then calls the task thread to switch to specific code section and execute. We follow this scheme but take different approach. For CMP, each *"thread"* is a processor, which has its own processing power and doesn't wait for resources from other processors. The number of "threads" is the actual number of processors instead of a team of *virtual threads*. It's not practical to create two "threads", one for spinning, and another for actual execution. In our implementation, we assign each parallel region in the program with a unique identifying function, like *func_1()*. The code inside the parallel region is moved from its original place and replaced by a function statement, where its associated region calls this function and processors with correct IDs execute selected statements in parallel.

Scheduling is an important aspect of the compiler. Conceptually there are three approaches to distribute workload, known as: *cyclic, block,* and *master-slave* methods. Cyclic and block methods use static approach while master-slave method distributes dynamically. To implement common OpenMP work-sharing constructs like *for-loop*, we apply different methods.

First, we try to use the static approach. The OpenMP compiler will analyze and extract the for loop header to find the index, loop initial, loop boundary and the increment, then distributes the workload into small slices. Each processor will be assigned a small portion of loop iterations based on algorithms like cyclic or block allocation. It is the responsibility of the programmer to ensure that the loop iterations are independent. Secondly, we deploy dynamic scheduling. For Master-slave method, it needs one or two master threads which are responsible for distributing workload and several slave threads which fetch their own portions from the master thread dynamically. The master thread will be created at the beginning of parallel region and assign workload to slave threads in *request-grant* manner. In our implementation for CMP, we didn't create a master thread due to limited number of processors and potential waste of resources. Instead, we use a global shared variable which is accessible to all processors. The total amount of work is evenly divided into small slices; each time a processor obtains one slice by accessing and modifying the shared variable. With the growing value of this shared variable, workload is distributed dynamically at runtime. A semaphore, which guarantees that only one processor can modify a variable at any given time, protects this shared variable.

We implement both *dynamic* and *static* scheduling on *3SoC*. User can choose one of the two scheduling methods through OpenMP compiler parameter during compilation. This provides more flexibility to do performance evaluation for different applications. Among the above two approaches, static allocation shows better performance on average. The reason is that *3SOC* is a CMP environment where each "thread" is a processing unit. By contacting the master thread or accessing the shared variable a

dynamic approach involves more synchronizations that interrupt the processors more frequently. Sample OpenMP code using static scheduling is shown in Fig. 5.

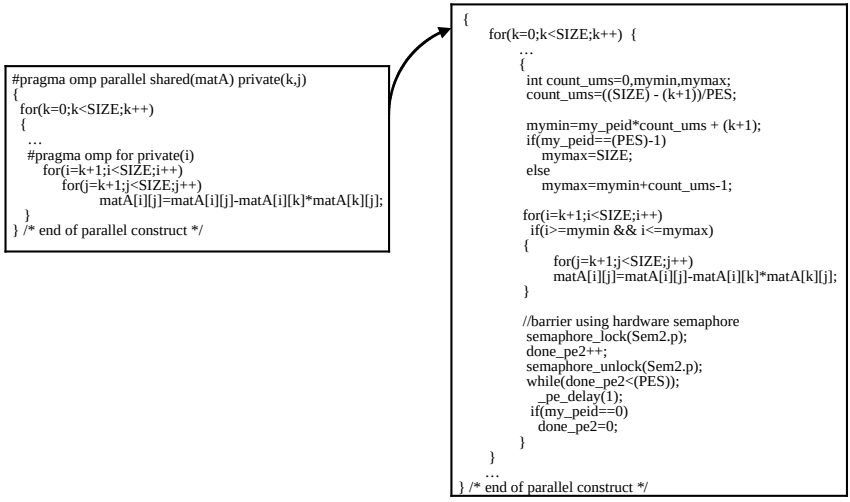


Fig. 5. Sample OpenMP code (LU decomposition) using static allocation

3.3 Data Attributes and Memory Allocation

In OpenMP, there are a number of clauses to define data attributes. Two major groups of variables exist: shared or private data. By default, all variables visible in a parallel region are shared among the threads. OpenMP provides several directives to change default behavior, such as variables defined in “FIRSTPRIVATE”, “PRIVATE”, and “REDUCTION”. For OpenMP compiler, some private variables need initialization and combination before or after parallel construct, like “FIRSTPRIVATE” and “REDUCTION”. Access to these data needs to be synchronized.

On *3SOC* platform, there are two levels of shared memory: local memory within Quad and shared DRAM outside Quad. The local data memory has a limited size of 64KB. Accordingly, there are four types of declarations for variables in *3SOC*, defined as:

- “_SD” - Shared DRAM, shared by all PEs.
- “_SL” - Shared Local memory of a Quad and shared by all PEs within Quad
- “_PD” - Private DRAM, allocated to one PE
- “_PL” - Private Local memory, allocated in local memory to one PE

By default, if none of these are declared, the variable is considered _PD in *3SOC*. The sample OpenMP code below illustrates data attributes:

```
1. int v1, s1[10];
2. void func() {
3.     int v2, v3, v4;
```

```
4.   #pragma omp parallel shared(v1,v2)
     firstprivate(v3) reduction(+: v4)
5.    {…}
6. }
```

If variables v1 and v2 are declared as shared, it should be defined in the global shared region. v1 is already in global shared memory, no special action needs to be taken, the only thing is to re-declare it as _SD(shared DRAM) or _SL(shared local). v2 is within the lexical context of *func()* which is a private stack area. This private stack is not accessible to other processors if it's not moved from *func()* to global shared memory during compilation. "FIRSTPRIVATE" is a special private clause that needs to be initialized before use. A global variable (v3_glb) needs to be defined and copied to the private variable (v3) before the execution of parallel construct. "REDUCTION" is another clause which needs global synchronization and initialization. A global variable is also defined in shared region and each local copy (_PL) is initialized according to reduction operation. At the end of parallel region, modification from each processor is reflected to the global variable.

For embedded systems, memory allocation is a crucial factor for performance because the local data memory is connected to a high-speed local bus. For standard OpenMP programs, data declaration in OpenMP should be treated differently for memory allocation in *3SoC*. This is a challenging task since the local memory has limited size for all CMP or equivalent DSP processors. It's not applicable to define all shared and private data structure into local memory first. We must do memory allocation dynamically in order to produce better performance.

We design to do memory allocation using "*request-and-grant*" model during compilation. At compilation, compiler will retrieve all memory allocation requests from each parallel region agreed on by some algorithm or policy-based methods. Then complier will assign different memory to different variable. For example, based on the size of data structure or frequency of referencing in the region, it will grant memory to different data structure. We also provide extensions to OpenMP to use the special hardware feature of CMP to allocate memory at runtime. (See 4.2.1 and 4.2.2)

4. Extensions to OpenMP

In a Chip Multiprocessor environment, there are several unique hardware features which are specially designed to streamline the data transfer, memory allocations, etc. Such features are important to improve the performance for parallel programming on CMP. In order to incorporate special hardware features for CMP, we extend OpenMP directives for this new parallel architecture.

4.1 OpenMP Extensions for DSE Processors

To deal with the heterogeneity of different processors within CMP, we try to incorporate DSEs into our OpenMP compiler. Unlike PE, DSE uses different programming

methodology which combines C and C-like Assembling Language (CLASM). We extend OpenMP with a set of DSE directives based on *3SOC* platform.

1. *#pragma omp parallel USING_DSE(parameters)*
   This is the main parallel region for DSEs.
2. *#pragma omp DSE_DATA_ALLOC*
   This is within DSE parallel region and used to define data allocation function.
3. *#pragma omp DSE_LOADCOMREG*
   Define data registers to be transferred to DSE
4. *#pragma omp DSE_LOADDIFFREG(i)*
   Define DSE data register with different value.
5. *#pragma omp DSE_OTHER_FUNC*
   Other user defined functions

The main parallel region is defined as *#pragma omp parallel USING_DSE (parameters)*. When the OpenMP compiler encounters this parallel region, it will switch to the corresponding DSE portion. The four parameters declared here are: *number of DSEs, number of Registers, starting DPDM number, and data register array,* such as *(8, 6, 0, dse_mem)*. "Number of DSEs" tells the compiler how many DSEs are required for parallel computation. Number of DSE data registers involved is defined in "number of Registers". There is an array where the PE stores the data that has to be transferred into the data registers of the DSE. The PE initializes the array and calls a MTE function to transfer the data into the DSE data registers. This array is defined in "data register array." In this example, it is *dse_mem*. The "starting DPDM number" is also required when loading data registers.

```
void main() {
          //other OpenMP parallel region
          #pragma omp parallel
          {...}
          …
          //OpenMP parallel region for multiple DSEs
          #pragma omp parallel USING_DSE(8,6,0,dse_mem)
           {
                #pragma omp DSE_DATA_ALLOC
                {
                    <initialization functions>
                }
                #pragma omp DSE_LOADCOMREG
                {
                    <define data registers to be transferred to DSE>
                }
                #pragma omp DSE_LOADDIFFREG(i)
                {
                    <define DSE data registers with different value>
                }
                #pragma omp DSE_OTHER_FUNC
                {
                     <other user defined functions>
                }
                //main program loaded and started by PE
                #pragma omp DSE_MAIN
                {
                     <order of executing main code>
                }
           }
}
```

Fig. 6. Sample OpenMP program using DSE extensions

Instead of writing*3SoC* parallel program using multiple DSEs in Fig. 4, user can write equivalent OpenMP program using DSE extensions shown in Fig. 6. For our OpenMP compiler, the code generation is guided by the parameters defined in *"parallel USING_DSE"* construct. The compiler will generate initialization and environment setup like *dse_lib_init(&LocalState),dse_allo(0),* DSE startup and wait call *dse_start(), dse_wait(),* and termination library call *dse_lib_terminate()*. So users will not do any explicit DSE controls, like startup DSE *dse_start()*. The DSE parallel region can also co-exist with standard OpenMP parallel regions that will be converted to parallel regions using multiple PEs.

The benefit of using OpenMP extensions is that it helps to do high-level abstraction of parallel programs, and allows the compiler to insert initialization code and data environment setup when necessary. Users are not required to focus on how to declare system data structures for DSE, and how PE actually controls multiple DSEs by complicated system calls. This will hide DSE implementation details from the programmer and greatly improve the code efficiency for parallel applications. Performance evaluation between different numbers of DSEs based on our OpenMP compiler will be given in section 6.

4.2 OpenMP Extensions for Optimization on SOCs

For SOCs or other embedded systems, memory allocation is critical to the overall performance of parallel applications. Due to the limited size of on-chip caches or memories, techniques have to be developed to improve the overall performance of SOCs. [7, 8].

4.2.1 Using MTE Transfer Engine.

Given the availability of local memory, programs will achieve better performance in local memory than in DRAM. But data locality is not guaranteed due to small on-chip caches or memories. One approach is to allocate data in DRAM first, then move data from DRAM to local memory at run-time. Thus, all the computation is done in local memory instead of slow DRAM. In *3SOC*, a user can invoke one PE to move data between the local memory and DRAM at run-time.

3SOC also provides a better solution for data transfer using MTE. MTE processor is a specially designed memory transfer engine that runs in parallel with all other processors. It transfers data between local data memory and DRAM in the background. We also incorporate MTE extensions to our OpenMP compiler.

1. *#pragma omp MTE_INIT(buffer size, data structure, data slice)*
   MTE_INIT initializes a local buffer for designated data structure.
2. *#pragma omp MTE_MOVE(count, direction)*
   MTE_MOVE will perform actual data movement by MTE engine.

MTE_INIT initializes a local buffer for *data structure* with specified *buffer size*. *MTE_MOVE* will perform actual data movement by MTE engine. Data size of equaling *count*slice* will be moved with respect to the *direction* (from local to DRAM or DRAM to local). Within a parallel region, a user can control data movement between

local memory and SDRAM before or after the computation. Accordingly, the MTE firmware needs to be loaded and initiated by PE0 at the beginning of the program. A number of MTE library calls will be generated and inserted by the OpenMP compiler automatically during compilation.

With the help of these extension, user can write OpenMP parallel program which controls actual data movement dynamically at run-time. The results show significant performance speedup using the MTE to do data transfers, especially when the size of target data structure is large. Performance evaluation of using the MTE versus using the PE to do memory transfer is given in section 6.

5. Implementation

In this section, we will discuss our implementation of the OpenMP compiler/translator for *3SoC*. However, this paper is not focused on implementation details. To implement an OpenMP compiler for *3SOC*, there are four major steps.

1. **Parallel regions**: Each parallel region in the OpenMP program will be assigned a unique identifying function number. The code inside the parallel region is moved from its original place into a new function context. The parallel construct code will be replaced by code of PE0's allocating multiple PEs or DSEs, setting up environment, starting all processors, assigning workload to each processor, and waiting for all other processors to finish.
2. **Data range**: Through analysis of all the data attributes in the OpenMP data environment clause, i.e., "SHARED", "PRIVATE", "FIRSTPRIVATE", "THREADPRIVATE", "REDUCTION", compiler determines the data range for separate functions and assign memory allocation like "_SL", or "_SD" in *3SOC*. Related global variable replication such as "REDUCTION" is also declared and implemented. Similar approach has been taken in Section 3.3.
3. **Work sharing constructs**: These are the most important constructs for OpenMP, referred as *for*-loop directive, *sections* directive and *single* directive. Based on the number of processors declared at the beginning of the *3SOC* program, each processor will be assigned its portion of work distinguished by processor ID. During runtime, each processor will execute its own slice of work within designated functions. For example, for the "sections" construct, each sub-section defined in *#pragma omp section* will be assigned a distinct processor ID, and run in parallel by different processors.
4. **Synchronization:** There are a number of explicit or implicit synchronization points for OpenMP constructs, i.e., *critical,* or *parallel* construct. Correspondingly, these constructs are treated by allocating a number of hardware semaphores in *3SOC*. Allocation is achieved statically or dynamically.

The first version of our compiler can take standard OpenMP programs. With our extension, user can also write OpenMP programs for advanced features of CMP, like using multiple DSEs.

6. Performance Evaluation

Our performance evaluation is based on *3SOC* architecture; the execution environment is the *3SOC* cycle accurate simulator, *Inspector* (version 3.2.042) and the *3SOC* processor. Although we have verified the programs on the real hardware, we present results on the simulator as it provides detailed profiling information. We present results of our preliminary evaluation.

To evaluate our OpenMP compiler for *3SOC*, we take parallel applications written in OpenMP and compare the performance on multiple processors under different optimization techniques: without optimization, using data locality (matrices in local memory), using the MTE for data transfer, and using the PE for data transfer. We also show the compiler overhead by comparing the result with hand-written code in *3SOC*.

Figure 7 shows the results of matrix multiplication using multiple PEs. The speedup is against sequential code running on single processor (one PE). Figure 8 is the result for LU decomposition using multiple PEs against one PE. By analysis of both charts, we conclude the following:

1. **Local memory vs DRAM:** As expected, memory access latencies have significant effect on performance. When the size of the data structure (matrix size) increases, speedup by allocation of data in local memory is obvious. For 64*64 matrix LU decomposition, the speedup is 4.12 in local memory vs 3.33 in DRAM.
2. **Using the MTE vs data in DRAM only:** As discussed in Section 5, we can deploy the MTE data transfer engine to move data from DRAM to local memory at runtime, or we can leave the data in DRAM only and never transferred to local during execution. For small size matrices below 32*32, the MTE transfer has no benefit; in fact, it downgrades the performance in both examples. The reason is that the MTE environment setup and library calls need extra cycles. For larger-size matrices, it shows speedup compared to data in DRAM only. For 64*64 matrix multiplication, the speedup is 4.7 vs 3.9. Actually 64*64 using MTE engine is only a 3.2% degrade compared to storing data entirely in the local memory. Therefore, moving data using MTE will greatly improve performance for large data structure.
3. **Using the MTE vs using the PE:** We observed scalable speedup by using the MTE over the PE to transfer memory. The extra cycles used in MTE movement do not grow much as the matrix size increases. For large data set movements, the MTE will achieve better performance than the PE.
4. **Using OpenMP compiler vs hand-written code:** The overhead of using the OpenMP compiler is addressed here. Since the compiler uses a fixed method to distribute computation, combined with extra code inserted to the program, it is not as good as manual parallel programming. In addition, some algorithms used in parallel programming cannot be represented in OpenMP. The overhead for OpenMP compiler is application dependent. Here we only compare the overhead of the same algorithm deployed by both the OpenMP compiler and hand-written code. It shows overhead is within 5% for both examples.

Figure 9 shows the result of matrix multiplication using multiple DSEs. The matrix size is 128*128.

1. **Scalable speedup using different number of DSEs:** 4 DSEs achieve 3.9 speedup over 1 DSE for the same program, and 32 DSEs obtain 24.5 speedup over 1 DSE. It shows that *3SOC* architecture is suitable for large intensive computation over multiple processors on one chip and performance is scalable.

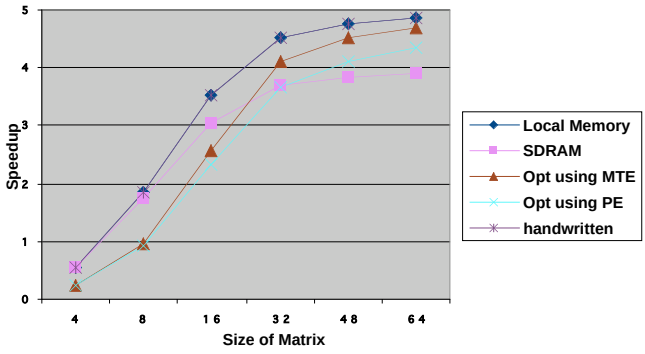


Fig. 7. Matrix Multiplication using four PEs

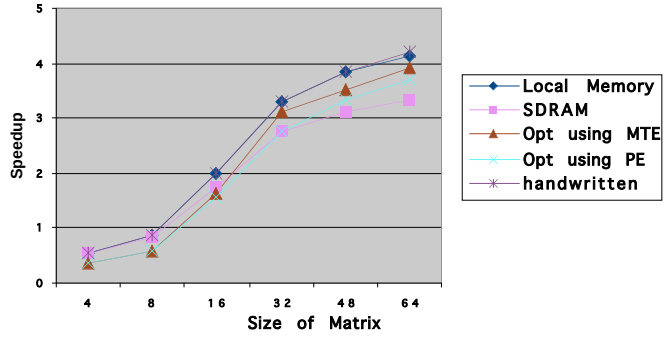


Fig. 8. LU decomposition using four PEs

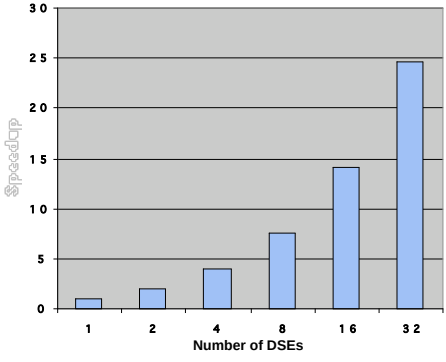


Fig.9. Matrix Multiplication using DSEs

7. Conclusion

In this paper, we present a practical OpenMP compiler for System on Chips, especially targeting *3SOC*. We also provide extensions to OpenMP to incorporate special architectural features. The OpenMP compiler hides the implementation details from the programmer, thus improving the overall efficiency of parallel programming for System on Chips architecture or embedded systems. Results show that these extensions are indispensable for performance improvement of OpenMP programs on *3SoC*, and such compilers would reduce the burden for programming SOCs. We plan to evaluate the translator/compiler on other large DSP applications and the new OpenMP benchmarks [9].

Acknowledgement

We thank the reviewers for their contributive comments on a draft of this paper. We also acknowledge the support of the Institute for Manufacturing Research at Wayne State University.

References

[1] OpenMP Architecture Review Board, OpenMP C and C++ Application Program Interface, Version 2.0, March, 2002. *http://www.openmp.org*
[2] OpenMP Architecture Review Board, OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering, Vol. 5, No. 1,* January/March 1998, http://www.openmp.org
[3] *3SOC* Documentation – *3SOC* 2003 Hardware Architecture, *Cradle Technologies, Inc.* March. 2002
[4] *3SOC* Programmer's Guide, *Cradle Technologies, Inc.,* Mar. 2002, http://www.cradle.com
[5] Christian Brunschen, Mats Brorsson, OdinMP/CCp – A portable implementation of OpenMP for C, *Lund Universtiy, Sweden, 1999*
[6] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001
[7] Chong-Liang Ooi, Seon Wook Kim, and Il Park. Multiplex: Unifying conventional and speculative thread-level parallelism on a chip multiprocessor, *Proceedings of the 15th international conference on Supercomputing* June 2001
[8] Non-Uniform Control Structures for C/C++ Explicit Parallelism, Joe Throop, Kuck & Associates, USA Poster Session at ISCOPE'98
[9] SPEC OMP Benchmark Suite, http://www.specbench.org/hpg/omp2001

Evaluation of OpenMP for the Cyclops Multithreaded Architecture

George Almasi[2], Eduard Ayguadé[1], Călin Caşcaval[2], José Castaños[2], Jesús Labarta[1], Francisco Martínez[1], Xavier Martorell[1], and José Moreira[2]

[1] CEPBA-IBM Research Institute, UPC - Barcelona, Spain
{eduard,jesus,fmartin,xavim}@ac.upc.es
[2] IBM Thomas J. Watson Research Center - Yorktown Heights, NY
{gheorghe,cascaval,castanos,moreira}@us.ibm.com

Abstract. Multithreaded architectures have the potential of tolerating large memory and functional unit latencies and increase resource utilization. The Blue Gene/Cyclops architecture, being developed at the IBM T. J. Watson Research Center, is one such systems that offers massive intra-chip parallelism. Although the BG/C architecture was initially designed to execute specific applications, we believe that it can be effectively used on a broad range of parallel numerical applications. Programming such applications for this unconventional design requires a significant porting effort when using the basic built-in mechanisms for thread management and synchronization. In this paper, we describe the implementation of an OpenMP environment for parallelizing applications, currently under development at the CEPBA-IBM Research Institute, targeting BG/C. The environment is evaluated with a set of simple numerical kernels and a subset of the NAS OpenMP benchmarks. We identify issues that were not initially considered in the design of the BG/C architecture to support a programming model such as OpenMP. We also evaluate features currently offered by the BG/C architecture that should be considered in the implementation of an efficient OpenMP layer for massive intra-chip parallel architectures.

1 Introduction and Motivation

Multithreaded architectures are a promising trend for the design of future high-performance microprocessor cores. Their ability to tolerate large memory and functional unit latencies and to increase resource utilization put them in the right position to achieve a high number of instructions per cycle (IPC). Tera MTA [28] and SMT [34] (an example of which is Intel Hyperthreading technology [6]) have followed this approach. Another trend is the integration of several microprocessor cores in the same chip, such as in the IBM Power4 [31]. Each processor has its own resources and shares the access to higher levels in the memory hierarchy such as off-chip main memory.

Multiprocessors systems-on-a-chip based on the replication of multithreaded cores offer a complexity-conscious alternative to future chip designs. The Blue

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 69–83, 2003.

Gene/Cyclops (BG/C) chip [9], which is the core of a new family of multithreaded architectures developed by IBM Research, consists of a large number of simple thread units simultaneously executing independent streams of instructions. Each thread behaves like a simple, single-issue, in order processor. Groups of threads share floating-point units and caches. All threads share a single address space implemented with an embedded DRAM memory in the same chip, resulting in a flat memory hierarchy with high bandwidth and low latency.

Making these new parallel architectures truly usable requires portable and easy-to-understand programming models that allow the exploitation of parallelism to applications written in standard high–level languages. Pthreads-like approaches are always possible but require a large programming effort. The user has to face the complexity of managing the parallelism at application level, manually handling thread creation, work distribution, allocation of variables and synchronization. The built-in parallel programming model provided by BG/C falls in this category. OpenMP [20] has emerged as the standard for shared-memory parallel programming. OpenMP applications are simple to program, portable across a range of shared-memory parallel platforms, and achieve near optimal parallel performance. The goal of this paper is to prove that OpenMP is a valid programming model for a machine that supports fine–grain multithreading, such as BG/C, and thus provide the user with a simple programming model for a complex machine.

We have ported the NthLib user-level threads library [19] to Cyclops in order to develop an experimental research platform, and used the Linux version of the NanosCompiler[10] to generate code. The current version does not consider some specific hardware features offered by the architecture. We will discuss about this limitation in Section 6, where we propose changes that will allow OpenMP to better exploit processor resources.

This paper is organized as follows: Section 2 describes the main characteristics of the BG/C architecture family and in particular, the configuration used in this paper. Section 3 describes the implementation of the OpenMP layer based on the NanosCompiler and NthLib. Section 4 describes the set of microbenchmarks and a subset of the NAS BT benchmarks used to obtain the experimental results presented in Section 5. The later section also shows the feasibility of programming OpenMP applications for BG/C. Section 6 discusses the explicit support for OpenMP that would be required in the architecture and the issues that should be considered to tune the implementation of the OpenMP layer. Finally, Section 7 outlines related work and Section 8 concludes the paper and outlines major directions in our future work.

2 The Blue Gene/Cyclops Architecture

The main characteristic of the BG/C design is the integration of embedded DRAM, processing logic and communications hardware on the same piece of silicon. The proximity of memory and processors results in a flat memory hierarchy which overcomes the von Neumann bottleneck (processor performance improves

faster than the capacity of memory to serve it) observed in conventional designs. Instead of hiding latencies through out-of-order or speculative execution, BG/C nodes tolerate latencies through massive parallelism. The solution adopted by BG/C is to use multiple threads in a single node so that, if a thread stalls for a memory reference, other threads can make progress. As a result each thread unit is simpler and expensive resources, such as FPUs and caches, are shared between different threads.

The organization of the BG/C chip is shown in Figure 1. At the base of the BG/C hierarchy are thread units. BG/C is a multithreaded design where thread units are simple computing processors that issue and execute instructions in program order. Each thread can issue an instruction at every cycle if resources are available and there are no dependences with previous instructions. Each thread unit consists of a register file (64 32-bit single precision registers, that can be paired for double precision values), a program counter, a fixed-point ALU, and an instruction sequencer.

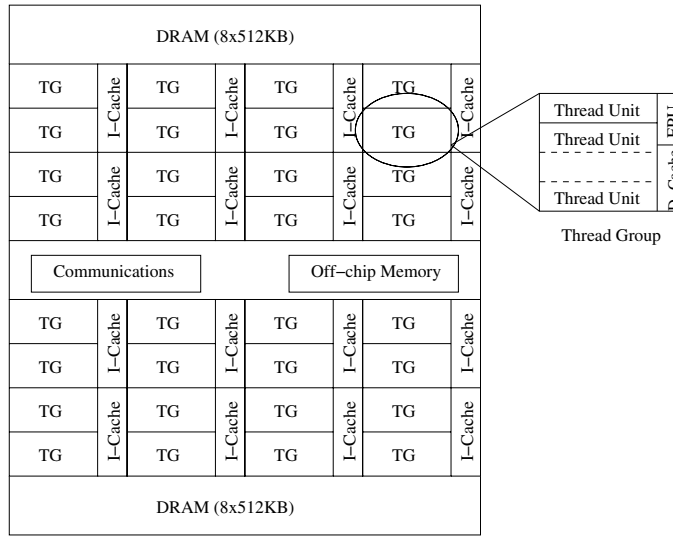


Fig. 1. Block diagram for a prototype of the BG/C architecture

Groups of threads units share an FPU and a data cache. Threads can dispatch a floating point addition and a floating point multiplication at every cycle. The base architecture in $0.18\mu m$ CMOS technology and 32 FPUs achieves a peak performance of 1 GFlops per FPU at a clock cycle of 500 MHz, for a total chip performance of 32 GFlops.

Each of the 32 16 KB data caches (one per thread group) has 64-byte lines and is 8-way set associative. By default, all data caches behave as a global, coherent cache. The data caches are shared among all threads in the chip. Thus,

a thread can access data in the cache of another thread group with lower latency than going to memory. Instruction caches are 32 KB, 8-way set-associative with 64-byte line size. In the base architecture, one instruction cache is shared by 2 thread groups. Unlike the data caches, the instruction caches are private to the threads in the thread groups. In addition, to improve instruction fetching, each thread unit contains a Prefetch Instruction Buffer (PIB) of 32 instructions.

The reference design considered in this paper has 16 banks of on-chip memory shared between thread units. Each bank is 512 KB for a total of 8 MB of embedded memory. The banks provide a contiguous address space to the threads. The latency to any bank is uniform. Addresses are interleaved to provide higher memory bandwidth. The unit of access is a 32-byte block, and threads accessing two consecutive blocks in the same bank will see a lower latency in burst transfer mode. The peak bandwidth of the embedded memory system is 40 GB/s (64 bytes every 12 cycles in each of the 16 banks).

In addition to the default all-shared cache behavior, the architecture supports an entire spectrum of access schemes through *interest groups* [5], from no sharing at all to caches shared at different levels. Any memory location can be placed in any cache under software control. The same physical address can be mapped to different caches depending on the logical address. An important use of this flexible cache organization is to exploit locality and shared read-only data. For example, data frequently accessed by a thread, such as stack data or constants, can be cached in the local cache by using the appropriate interest group. The hardware does not implement any coherence mechanism to deal with multiple copies of a memory line in different data caches.

Four global inter-thread hardware barriers are provided through a special purpose register (SPR). These barriers are implemented as a wired OR for all or a user defined subset of the threads on the chip.

The BG/C chip also provides six input and six output links. These links allow a chip to be directly connected in a three dimensional topology (mesh or torus). The links are 16-bit wide and operate at 500 MHz, giving a maximum I/O bandwidth of 12 GB/s. In addition, a seventh link can be used to connect to a host computer. These links can be used to build larger systems without additional hardware. Another port permits the access to external (off-chip) memory. However, these latter characteristics are not the focus of this paper.

BG/C executables (kernel, libraries, applications) are currently being generated with a cross-compiler based on the GNU toolkit, re-targeted for the BG/C instruction set architecture. This cross-compiler supports C, C++, and FORTRAN 77.

The performance results shown in Section 5 are generated by an architecturally accurate simulator which executes instructions from the BG/C instruction set, modeling resource contention between instructions, and thus estimating the number of cycles executed per instruction. The configuration parameters used for the simulations in this paper are listed in Table 1.

In addition, each chip runs a resident system kernel, which executes with supervisor privileges. The kernel supports single user, single program, multi-

threaded applications within each chip. The kernel exposes a single-address space shared by all threads. Due to the small address space and large number of hardware threads available, no resource virtualization is performed in software: virtual addresses map directly to physical addresses (no paging) and software threads map directly to hardware threads. The kernel does not support preemption (except in debugging mode), scheduling or thread priorities. Every software thread is preallocated with a fixed size stack per thread (selected at boot time), resulting in fast thread creation and reuse.

Table 1. Design parameters for the reference BG/C architecture. In (a) we show the number of cycles for execution and latency of the main instruction types. Execution is the number of cycles the functional unit is busy; latency is the additional delay until the results of the operation are available

(a) instructions

Instruction type	Execution	Latency
Branches	2	0
Integer multiplication	1	5
Integer divide	33	0
Floating point add, mult. and conv.	1	5
Floating point divide (double prec.)	30	0
Floating point square root (double prec.)	56	0
Floating point multiply-and-add	1	9
All other operations (except memory ops.)	1	0

(b) components

Component	# of units	Params/unit
Threads	1-256	single issue, in-order, 500 MHz
FPUs	32	1 add, 1 multiply
D-cache	32	16 KB, 8-way assoc., 64-byte lines
I-cache	16	32 KB, 8-way assoc., 64-byte lines
Memory	16	512 KB

3 Towards OpenMP for BG/C

OpenMP for BG/C is based on the NanosCompiler and the NthLib components. The OpenMP NanosCompiler is a source-to-source translator for Fortran77 based on Parafrase-2 [23]. NthLib is a runtime library designed to provide an efficient support to the OpenMP execution model on shared-memory multiprocessors. Fine grain parallel tasks are implemented as efficiently as other thread packages: the application creates work descriptors and supplies them to the participating threads [18]. A mechanism to spawn coarse grained parallel tasks, called *nanothreads* [19] is also available. This mechanism is more expensive but allows the exploitation of multiple levels of parallelism.

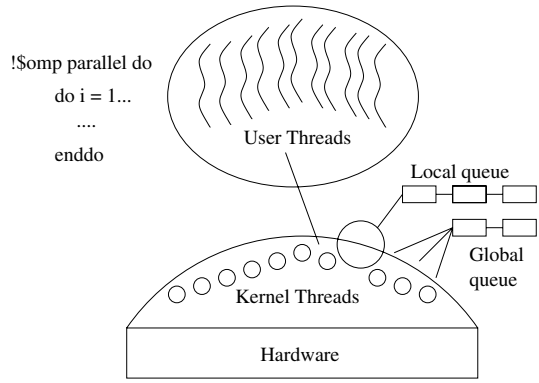


Fig. 2. Software architecture of OpenMP on BG/C

Figure 2 presents the software architecture used for supporting OpenMP. Kernel-level threads are the processor abstraction in our environment. User-level threads are supported on top of the kernel-level threads, and they represent the abstraction for work. Kernel-level threads are created when the application starts and are kept alive during the entire execution of the application. User-level threads are spawned as needed to create parallel regions – each thread executing a share of the whole work. Assigning the task to a thread is implemented by queuing the nanothread or work descriptor in one of the per-processor queues or in the global queue. The per-processor queues allows us to exploit locality; the global queue can be used for load balancing reasons, but it is not used for the experiments presented in this paper.

The implementation of NthLib for BG/C currently uses few specific system services and architectural features. Its only requirements are a processor allocation mechanism, control over stack placement and memory management, and a set of atomic memory operations. The thread creation mechanism provided by the BG/C system library is the *bg_svc_thread_create_specific* call. This primitive creates a kernel–level thread to run on a specific hardware thread. This way, the kernel–level threads may be mapped to specific processors. Other service calls similar to pthreads are used for thread management, like *bg_svc_thread_join*. Stack management was implemented giving fixed-size stacks to threads on creation, thanks to the ability of the thread creation service to use a designated stack as the thread stack. The implementation assumes the default all-shared cache organization inside the chip.

4 Benchmark Description

In order to evaluate the performance of our OpenMP implementation for BG/C, we have used a set of microbenchmarks and a subset of the NAS benchmarks, version 2.3. The purpose of the microbenchmarks is to compare the performance of

the OpenMP parallelization and the performance of the hand-optimized versions to execute on BG/C. The NAS benchmarks show that our OpenMP implementation scales to handle large, realistic applications.

4.1 Micro-Benchmarks

Throughout our evaluation we have used microbenchmarks to study specific properties of the architecture. In this paper, we present results for two scientific kernels: dense matrix multiplication and sparse matrix-vector product. Their simplicity allows the validation of results by direct study of the assembly listings, which increases our confidence in the correctness of the experiments.

Three versions for each kernel have been coded: Pthreads, OpenMP and Pthreads without optimizations (*wco*). In the Pthreads versions the program contains code to fork and join parallel threads. In addition, each thread executes code to determine the portion of work that has to execute and synchronizes with other threads by means of barriers. The OpenMP version simply contains the parallel and work-sharing directives necessary to express the same parallelization strategy (or the closest one, if not possible). The compiler takes care of generating the code for distributing the work and synchronizing across the threads.

MM. The matrix multiplication kernel, MM, computes $AB = C$ with $A_{m\times p}$, $B_{p\times n}$ and $C_{m\times n}$, where $m = 192$, $n = 192$ and $p = 100$ using the simple three-nested loops algorithm (high school matrix multiply). The data set results in a storage requirement of about 0.59 MB. That size essentially fits in the global D-cache.

The pthread-based implementation of MM distributes the matrix C evenly among $t = r \times s$ threads, resulting in each thread owning a rectangular section of C. Each thread computes only the portion of C it owns. The OpenMP implementation of MM distributes the work evenly among the columns of C: one-level STATIC block distribution of the iterations in the loop that traverse the columns of the result matrix. MM requires no synchronization between threads.

SPARSE, the multiplication of a sparse matrix by a vector, is the main kernel of many iterative linear solvers. Our implementation represents the sparse matrix S using *row-indexed sparse storage* [27,24]. This scheme stores the diagonal elements and the non-zero elements in a vector of values `val`. The columns of the non-zero elements are stored in an integer vector `idx`.

The inner loop of the sparse-matrix vector product $Sx = y$

```
for (k=idx[i]; d < idx[i+1]; k++)
   y[i] += val[k] * x[idx[k]]
```

requires three memory loads for every non-zero element k. The location of the dependent loads for the indirect access to x is particularly difficult to predict and the latency is difficult to hide. For that reason most sparse-matrix vector codes suffer from poor performance.

In both implementations (Pthreads and OpenMP), the rows of the matrix S and the solution vector y are partitioned between threads. This method does not require thread synchronization. A fill parameter f controls the sparsity of the matrix: one of every f elements in each row i is non zero, starting at column $i \bmod f$. Thus, the vector x is traversed in sequential order.

Threads multiply one or two rows of the matrix at the same time, and in the manual implementation the inner loop is unrolled 8 times. The test problem $Sx = y$ with matrix size 1024×1024 and fill factor $f = 4$ requires 3.03 MB of main memory.

4.2 NAS Benchmarks

We have also evaluated our BlueGene/Cyclops OpenMP implementation with a subset of the NAS benchmarks which are a representative set of computing intensive applications. We have used the OpenMP Fortran77 benchmarks in NAS PBN [12], version 2.3.

We have simulated fully both CLASS S and CLASS W benchmarks, although the sets are not the same because of memory capacity problems. The programs simulated from CLASS S (the smallest class) are MG, FT, SP, CG, and LU. Although they are small in data sizes and number of iterations, the complexity of the application is the same, enabling a complete evaluation of the OpenMP compilation environment in a reasonable amount of simulation time. The programs simulated in CLASS W are MG, BT, SP, CG, and LU. Their description can be found in [3][7].

5 Experimental Results

In this section, we show the results of our simulations. All the examples are naively implemented with no hand-coded manual optimizations for scalar performance (loop unrolling, blocking, ...). We briefly make some comments about how they impact performance and the reader is referred to [2]. That report evaluates specific features of the BG/C architecture using applications fine-tuned to execute at maximal performance.

For the micro-benchmarks, we plot performance results in MFLOPs when considering the parallelism fork and join overheads and when just considering the useful work executed in parallel (plot labeled *wco*). For the NAS subset we plot the speedup relative to the sequential version.

5.1 Micro-Benchmarks

Figures 3 and 4 show that the performance obtained by the OpenMP version of these benchmarks is similar to that of the hand-coded pthreads versions (in some cases, even better). They tend to scale at least as well as the hand-coded Pthreads versions.

There is an anomaly worth highlighting, though, in MM. The *OpenMP* plot shows that the performance improvement stalls at 96 threads. This is due to the different work distribution schemes used in the Pthreads and OpenMP versions. The Pthreads versions simultaneously distributes work from two of the loops in the nest where the matrix multiply is done. The OpenMP version just distributes work from one of the loops. Since the parallelized loop executes 192 iterations, using more that 96 threads results in a highly unbalanced assignment of iterations to threads (two iterations are assigned to some of them and one to the rest). Linearizing the loop or using two levels of parallelism would produce a balanced distribution of work, thus achieving the same performance.

By contrast, overheads are the problem for the degradation of performance in SPARSE – the code that distributes the work among threads is executed once in the Pthreads version, but multiple times in the OpenMP version.

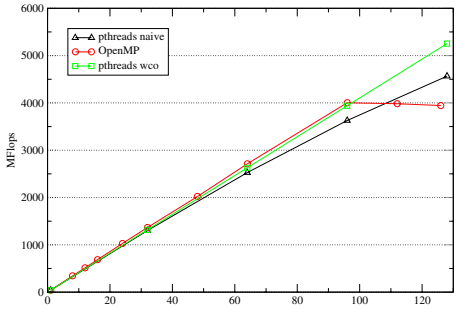


Fig. 3. MM performance

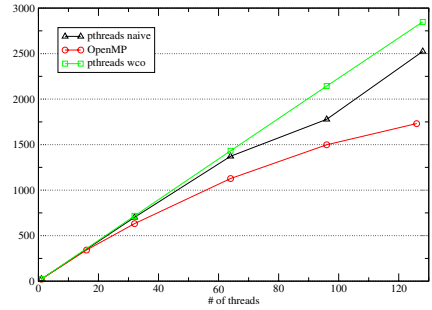


Fig. 4. SPARSE performance

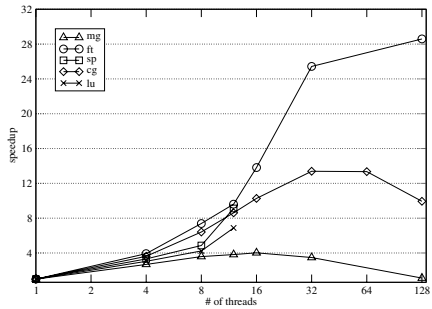


Fig. 5. NAS CLASS S scalability

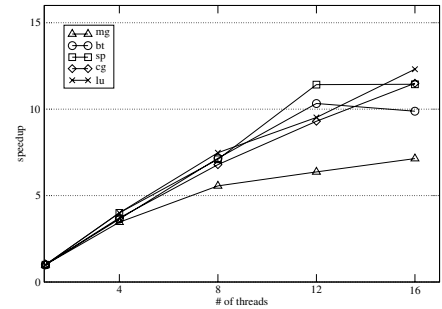


Fig. 6. NAS CLASS W scalability

5.2 NAS Benchmarks

Figure 5 shows the scalability of a subset of the CLASS S NAS benchmarks in BG/C. CLASS S has been selected to show how the architecture performs with small data sets. Figure 6 shows the scalability of some CLASS W benchmarks. The reason for missing some of the benchmarks is the incipient state of the OpenMP environment, the limited hardware memory size, and limitations in the simulator when running applications requiring a large memory size and a large number of threads.

The performance of most of the CLASS S benchmarks improves up to 16 processors. In this case, the scalability shown for CLASS S is better than the scalability obtained in other shared memory architectures, making us believe that further work in this direction will yield the expected results.

The first experiments done with CLASS W reveal that scalability improves as the data set grows. We believe that this is promising and shows that BG/C should be able to support large OpenMP applications.

6 Improving OpenMP Support for BG/C

As mentioned in Section 3, the NthLib library has been ported to BG/C without considering specific architectural features of the target machine. Some proposals to enhance the support to the OpenMP programming model in BG/C are described in this section.

- **Local versus global work descriptors**. In the current implementation of NthLib for BG/C, the master thread generates a work descriptor for each thread participating in the parallel execution, and supplies it to its per-thread local queue. This allows assignment of work in a very flexible way. However, in OpenMP this flexibility is not necessary, so that a single global work descriptor could be created and supplied to all per–thread local queues. This would reduce creation overheads, especially when the number of threads is large.
- **Take advantage of interest groups in data caches**. In the current implementation, all caches are shared. It would be possible to use other cache sharing possibilities in order to privatize variables. For example, when multiple levels of parallelism are exploited, groups of threads can be defined (OpenMP extensions supported by the NanosCompiler). In this case, threads in the same group share data that is privatized among groups.
- **Using hardware barriers**. BG/C offers an efficient implementation for barriers. However, this support is not sufficient in NthLib because a thread needs to look for work on the queues while waiting on a barrier to be open. We plan to solve this problem using split barriers (offering for example *barrier_enter* and *barrier_leave* primitives). In this way, a thread could execute useful work while waiting on a barrier. In addition, we can devise a tree-like scheme, in which different threads that map to the same physical thread use different hardware barriers from the 4 available. This approach trades the

number of global barriers available for a larger number of threads being able to synchronize.

- **Fine grain synchronization**. Locks could be implemented using hardware locks. These locks could stop the thread (or make it spin on an specific purpose register instead of in local cache) while waiting for the unlocking thread. In [35], it has been proposed to introduce two new instructions (*acquire* and *release*), as well as a locking hardware structure (*lock_box*). These locks could have different implementations for the threads inside the same Thread Group and for threads in different Thread Groups. This contrasts with the current BG/C implementation, that is a via the *test_and_set* atomic operation provided by the ISA. This could also help to reduce power consumption because resources used by sleeping threads could be turned off.
- **Eliminating the idle loop**. When a thread is waiting in the idle loop, it is wasting resources without a real need (for instance, cache bandwidth or energy). In a normal multiprocessor system, the idle loop spins in the local cache, thus wasting no resources at all; however, in BG/C the data cache is shared among threads so the spinning consumes a portion of its total bandwidth. A possibility would be to include hardware mechanisms to efficiently implement the idle loop similar to the one mentioned above to implement hardware locks. This would mean to stop the activity of a thread while waiting in the idle loop. The activity would be resumed as soon as work is queued in its per–thread local or global queues.
- **Eliminating user-level threads**. We can eliminate the user-level threads which causes many overheads, and implement thread creation directly via *bg_svc_thread_create_specific* calls. This would solve the barrier problem – we can use hardware barriers. This would also eliminate the idle loop. Although it has been proven that this is not viable in current multiprocessor systems, it should be studied for multithreaded systems as the BG/C with very low thread creation overhead. An hybrid implementation could allow choosing among the flexibility of user-level threads or the better performance of no scheduling levels. In this way, OpenMP applications that use a small number of threads (less than or equal to the number of hardware threads in a chip) could map the software threads to the physical threads, thus avoiding the overhead of context switches.

7 Related Work

Architectures that integrate processors and memories on the same chip are called Processor-In-Memory (PIM) or Intelligent Memory architectures. They have been spurred by technological advances that enable the integration of compute logic and memory on a single chip. These architectures deliver higher performance by reducing the latency and increasing the bandwidth of processor-memory communication. Examples of such architectures are EXECUBE [15], IRAM [22], Shamrock [14], Imagine [25], FlexRAM [13,32], DIVA [11], Active Pages [21], Gilgamesh [38], and MAJC [33]. The PIM chip is used either as a

coprocessor (Imagine, FlexRAM), or as the main engine in the machine (IRAM, MAJC, Shamrock), or as a *cell* in a larger system (MIT RAW [1,36], EXECUBE and BG/C). Another classification could be based on the number and type of the processors: FlexRAM and Imagine include many (more than 32) relatively simple processors, while EXECUBE, IRAM, MAJC, Piranha [4] and Shamrock include only a few (4-8). BG/C goes beyond what has been proposed, using hundreds of processors.

Simultaneous multithreading exploits both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. It was shown to be a more effective approach to improve resource utilization than superscalar execution. Results presented in [8,34] support our work by showing that there is not enough instruction-level parallelism in a single thread of execution, therefore it is more efficient to execute multiple threads concurrently.

The Tera MTA [28,29] is another example of a modern architecture that tolerates latencies through massive parallelism. In the case of Tera, 128 thread contexts share the execution hardware. This contrasts with BG/C, in which each thread has its own execution hardware. Both architectures can tolerate long latencies.

As far as we know, this is the first attempt to port an OpenMP runtime system to a massive parallel multithreaded system on–chip. The porting is based on the experience gained over the years on implementing such an environment on top of other execution environments, including small SMPs and large cc-NUMA. Vendors also provide fine-tuned implementations for their target machines, such as SGI IRIX MP[30] library or the IBM run-time library for AIX. For example, the SGI MP library provides a complete execution environment for each application, supporting thread creation, management, synchronization and NUMA features, such as memory placement. The library is aware of the machine load, trying to adjust the parallelism which is exploiting to the available resources. A number of projects also try to extend the use of OpenMP to clusters with DSM (Distributed Shared Memory) support. The long latencies experienced when accessing remote data and the memory granularity at the page level impose new constraints in these implementations [17,26].

The Nanos execution environment, which is the source for the two components used to implement OpenMP on top of BG/C, focus on adaptability at different levels, the effective exploitation of nested parallelism and the specification of precedence relations among computations that form pipelines. All these aspects form a set of extensions to OpenMP whose impact must be investigated in BG/C.

A number of studies have been recently published in which different compiler optimizations are evaluated for multithreaded architectures. For example, [16] relaxes and modifies some of the requirements on code scheduling and data access used by current compilers.

As stated before, the BG/C architecture is focused on the execution of a single multithreaded application within each chip. Other architecture proposals such as α–Coral [37] provides for mostly hardware managed simultaneous mul-

tiprogramming and multithreading environment. The Nanos environment also offers workload management at the software level with the CPUmanager component, specially designed for malleable OpenMP applications.

8 Conclusions

The Blue Gene/Cyclops architecture provides an excellent platform for studying programming environments for multithreaded architectures. Writing and porting applications for the BG/C architecture is not a simple process [2]. The very large number of threads (one or two orders of magnitude larger than similar architectures), the complexity of the cache organization, and the sharing of caches and floating point units are not yet easily modeled statically by compilers. The Pthreads execution model used until now for BG/C closely matches the hardware but adds another level of complexity to the process of writing software for BG/C.

In order to simplify this task, this paper introduces the implementation of an OpenMP environment for on–chip massive parallel architectures. This OpenMP environment together with the simulation environment for BG/C allows the exploration of a large number of SMT configurations with low programming effort. This permits us to better understand what are the trade-offs between multithreading characteristics and which properties are worth integrating in our implementation of the NthLib library.

This paper also shows that more tuning of our library is still required. With simple hand-optimized kernels, BG/C has demonstrated that its architecture is able to perform a very large percentage of the peak floating point performance [2] offered by the architecture. The results shown in previous sections showed that OpenMP applications can behave on par with Pthreads programs. Most of the optimizations used in [2], such as loop unrolling and register tiling, are orthogonal to the programming environment and will apply well to our OpenMP benchmarks. Nevertheless, there is also room for improvement in the case of large applications, such as the NAS benchmarks, as shown.

References

1. Anant Agarwal. Raw computation. *Scientific American*, August 1999.
2. George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Jr. Henry S. Warren. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. In *MEDEA Workshop on On-Chip Multiprocessor: Processor Architecture and Memory Hierarchy related Issues*, September 2002.
3. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and Maurice Yarrow. The NAS parallel benchmarks 2.0. Technical Report Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
4. L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

5. Călin Caşcaval, José Castaños, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, José E. Moreira, Karin Strauss, and Henry S. Warren, Jr. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the 8th International Symposium of High Performance Computer Architecture*, February 2002.
6. Intel Corporation. Intel hyperthreading technology. `http://www.intel.com/info/hyperthreading`. 2003.
7. D.Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, R. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report Technical Report RNR-94-007, NASA Ames Research Center, March 1994.
8. Susan Eggers, Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, pages 12–18, September/October 1997.
9. Frances Allen et al. Blue gene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2):310–328, 2001.
10. M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro, and J. Oliver. NanosCompiler: Supporting flexible multilevel parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(9), August 2000.
11. M. W. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCross, J. Brockman, W. Athas, A. Srivasava, V. Freech, J. Shin, and J. Park. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *Proceedings of SC99*, November 1999.
12. H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of the NAS parallel benchmarks and its performance. Technical Report Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
13. Yi Kang, Michael Huang, Seung-Moon Yoo, Zhenzho Ge, Diana Keen, Vinh Lam, Prattap Pattnaik, and Josep Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *International Conference on Computer Design (ICCD)*, October 1999.
14. P. Kogge, S. Bass, J. Brockman, D. Chen, and E. Sha. Pursuing a petaflop: Point designs for 100 TF computers using PIM technologies. In *Frontiers of Massively Parallel Computation Symposium*, 1996.
15. Peter M. Kogge. The EXECUBE approach to massively parallel processing. In *Intl. Conf. on Parallel Processing*, August 1994.
16. Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *International Symposium on Microarchitecture*, pages 114–124, 1997.
17. H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on network of workstations. In *Proc. of Supercomputing'98*, 1998.
18. X. Martorell, E. Ayguadé, J.I. Navarro, J. Corbalán, M. González, and J. Labarta. Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors. In *Proceedings of the 13th Int. Conference on Supercomputing ICS'99*, June 1999.
19. X. Martorell, J. Labarta, J.I. Navarro, and E. Ayguadé. A library implementation of the nano-threads programming model. In *Proceedings of Euro-Par'96*, August 1996.
20. OpenMP Organization. OpenMP Fortran application interface, v. 2.0. `http://www.openmp.org`, June 2000.

21. Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A computation model for intelligent memory. In *International Symposium on Computer Architecture*, pages 192–203, 1998.
22. David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM: IRAM. In *Proceedings of IEEE Micro*, April 1997.
23. Constantine D. Polychronopoulos, Milind B. Girkar, Mohammed Resa Haghighat, Chia Ling Lee, Bruce P. Leung, and Dale A. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing and scheduling programs on multiprocessors. In *1989 International Conference on Parallel Processing*, volume II, pages 39–48, St. Charles, Ill., 1989.
24. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes in C. In *Cambridge University Press*, 1992.
25. S. Rixner, W.J. Dally, U.J. Kapasi, B. Khailany, A. Lopez-Lagunas, P.R. Mattson, and J.D. Owens. A bandwidth-efficient architecture for media processing. In *31st International Symposium on Microarchitecture*, November 1998.
26. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for an smp cluster, 1999.
27. Scientific Computing Associates, Inc. *PCGPACK user's guide*.
28. A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchel, J. Feo, and B. Koblenz. Multiprocessor performance on the Tera MTA. In *Proceedings Supercomputing '98*, Orlando, Florida, Nov. 7-13 1998.
29. A. Snavely, G. Johnson, and J. Genetti. Data intensive volume visualization on the Tera MTA and Cray T3E. In *Proceedings of the High Performance Computing Symposium - HPC '99*, pages 59–64, 1999.
30. Silicon Graphics Computer Systems. Origin2000 and Onyx2 performance tuning and optimization guide. Technical Report Doc. num. 007-3430-002, 1998.
31. J. M. Tendler, J. S. Dodson, Jr. J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.
32. Josep Torrellas, Liuxi Yang, and Anthony-Trung Nguyen. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Sixth International Symposium on High-Performance Computer Architecture*, January 2000.
33. M. Tremblay. MAJC: Microprocessor architecture for Java computing. In *Hot Chips*, August 1999.
34. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.
35. Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *HPCA*, pages 54–58, 1999.
36. Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, pages 86–93, September 1997.
37. M. Yankelevsky and C. D. Polychronopoulos. α-Coral: A multigrain, multithreading processor architecture. In *Proceedings of International Conference on Supercomputing'01*, 2001.
38. H. P. Zima and T. Sterling. The Gilgamesh processor-in-memory architecture and its execution model. In *Workshop on Compilers for Parallel Computers*, June 2001.

Busy-Wait Barrier Synchronization Using Distributed Counters with Local Sensor

Guansong Zhang, Francisco Martínez*, Arie Tal, and Bob Blainey

IBM Toronto Lab,Toronto
ON, L6G 1C7, Canada

Abstract. Barrier synchronization is an important and performance critical primitive in many parallel programming models, including the popular OpenMP model. In this paper, we compare the performance of several software implementations of barrier synchronization and introduce a new implementation, *distributed counters with local sensor*, which considerably reduces overhead on POWER3 and POWER4 SMP systems. Through experiments with the EPCC OpenMP benchmark, we demonstrate a 79% reduction in overhead on a 32-way POWER4 system and an 87% reduction in overhead on a 16-way POWER3 system when comparing with a fetch-and-add implementation. Since these improvements are primarily attributed to reduced L2 and L3 cache misses, we expect the relative performance of our implementation to increase with the number of processors in an SMP and as memory latencies lengthen relative to cache latencies.

Keywords: Barrier, synchronization, multiprocessor, distributed counter.

1 Introduction

A barrier is a synchronization primitive used in parallel programming languages. The barrier point is a program position at which each thread of execution enters in parallel and waits to proceed until all threads have reached the same point. Figure 1 (a) illustrates a barrier synchronization of the execution of four threads. Barriers are necessary in many parallel programs to ensure data integrity between phases of a program which are to be executed using multiple threads.

Barriers are used in both shared and distributed memory programming models [1] [2] [3]. We restrict our attention in this paper to software barrier implementations for cache-coherent shared memory or symmetric multiprocessor (SMP) systems. Software barrier implementation for SMP systems can generally be done using hardware synchronization support such as fetch-and-add or test-and-set, by using busy-wait polling techniques or by using some combination of the two.

In the OpenMP shared memory parallel language specifications[4][5] , barrier synchronization plays a significant role. Parallel regions as well as all of

* Ph.D. candidate, research student visiting from the Technical University of Catalonia (UPC), Barcelona, Spain.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 84–98, 2003.

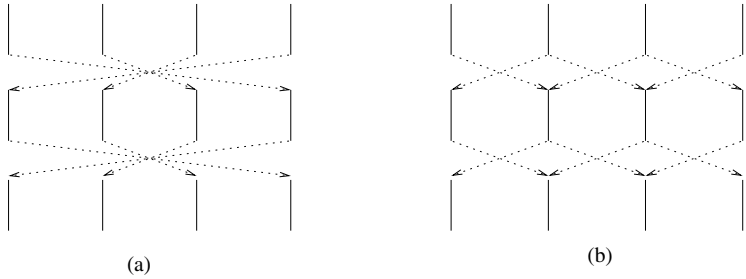


Fig. 1. Barriers

the defined work-sharing constructs are terminated by an implicit barrier synchronization (which may, in some cases, be avoided using a NOWAIT clause). OpenMP also defines an explicit barrier directive defined to be used whenever necessary.

In this paper, we will study different ways of implementing busy-wait barrier synchronization on a multiprocessor system with shared memory, typically for parallel programming with OpenMP. We use the EPCC micro-benchmarks[6] to measure performance overhead of the different barrier implementations discussed here. All of the test data is based on the use of an explicit barrier, though the results can be applied equally well to implied barriers.

In section 2, we describe the POWER4-based system on which we performed our experiments and the source of some of the costs implicit in barrier synchronization. In section 3, we describe several implementations of busy-wait synchronization and finally describe our new approach, *distributed counters with local sensor*. In section 4, we show the results of timing and hardware performance monitoring experiments using the various barrier implementations. Finally, in section 5, we summarize our findings and describe future work.

2 Overhead of a Barrier Synchronization

There are many ways of implementing a barrier synchronization. In this paper, we focus on the so-called *centralized barrier*, simply because in a globally addressed memory system with wide bandwidth among node processors, such as a POWER4, the centralized barrier will make the coding much easier, and less error prone[1].

Several ways to implement this kind of barrier were suggested in [7] and reexamined on a ccNUMA system in [8]. Basically, a shared variable is updated as each thread arrives at the barrier point. The thread will leave the position if testing finds out that all the threads have arrived. To understand the overhead of

[1] When there is a large number of processor nodes connected by a interconnection network with a specific topology, centralized barrier may not be preferable. While the barrier in Figure 1 (b) will be more interesting, where locality is emphasized.

the whole process, we first have a brief introduction on the hardware architecture used for testing, and our test environment.

2.1 POWER4 SMP Architecture and Software

The basic building block for a POWER4 SMP system can be found in [9]. It is a multi-chip module (MCM) with four POWER4 chips forming an 8-way SMP, as shown in Figure 2. Multiple MCMs can then be interconnected to form 16-, 24-, and 32-way SMPs.

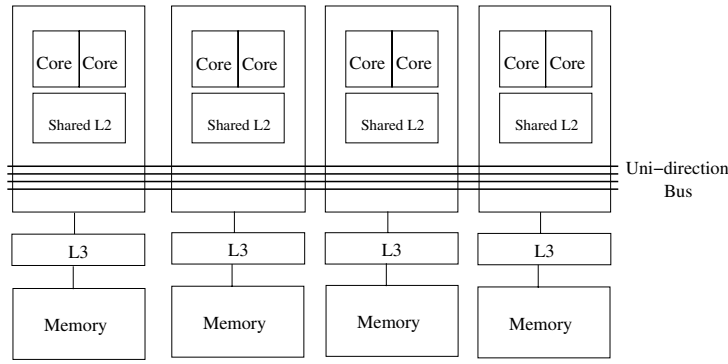


Fig. 2. POWER4 MCM structure

The logical interconnection of four POWER4 chips is point-to-point, with uni-directional buses connecting each pair of chips to form an 8-way SMP with an all-to-all interconnection topology. The fabric controller on each chip monitors (for example snoops) all buses and writes to its own bus, arbitrating between the L2 cache, I/O controller, and the L3 controller for the bus. Requests for data from an L3 cache are snooped by each fabric controller to determine if it has the data being requested in its L2 cache (in a suitable state), or in its L3 cache, or in the memory attached to its L3 cache. If any one of these is true, then that controller returns the requested data to the requesting chip on its bus. The fabric controller that generated the request then sees the response on that bus and accepts the data.

Up to four MCMs can be interconnected by extending each bus from each module to its neighboring module in one direction. Inter-module buses run at half the processor frequency and are 8-bytes wide. The inter-MCM topology is that of a ring in which requests and data move from one module to another module in one direction. As with the single MCM configuration, each chip always sends requests, commands and data on its own bus but snoops all buses for requests or commands from other MCMs.

The underlying software system is the SMP runtime library supporting the OpenMP standard. It is not a research system but production level software

available in the IBM® XL Fortran and VisualAge C++® products. The runtime system implements schedule-based barrier synchronization as well as a fallback for the busy-wait method.

2.2 Testing Benchmark

We use unmodified EPCC micro-benchmarks to test the performance of a barrier synchronization. As introduced in [10], the overhead here is considered as the difference between the parallel execution time and the ideal time, given by perfect scaling of the sequential program.

The parallel execution time is taken from the following code segment,

```
      dl = delaylength

      do k=0,outerreps
         start  = getclock()
!$OMP PARALLEL  PRIVATE(J)
         do j=1,innerreps
            call delay(dl)
!$OMP BARRIER
         end do
!$OMP END PARALLEL
         time(k) = (getclock() - start) *
     &             1.0e6 / dble (innerreps)
      end do
```

While the sequential reference time is measured through this code,

```
      dl = delaylength
      do k=0,outerreps
         start  = getclock()
         do j=1,innerreps
            call delay(dl)
         end do
         time(k) = (getclock() - start)*
     &             1.0e6 / dble (innerreps)
      end do
```

In the test program, the value of `outerreps` is set to 50, which is the default in the EPCC micro-benchmarks. The array variable `time` is used to compute the mean and standard deviation of the 50 measurements. Since we can exclusively access the machine, only the mean value is considered here.

2.3 Barrier Overhead on an SMP System

The two main reasons for the overhead of a barrier synchronization are memory contention and traffic. We define *contention* as the effect produced by many threads accessing simultaneously the same data, and *traffic* as the amount of data per time unit moving through the bus.

Nevertheless, contention and traffic are not independent as some accesses to memory increase both (so, contention in one memory position is likely to increase traffic). Studying memory behavior through hardware counters can show us an approximation to both the contention and traffic that a concrete barrier implementation puts on the system.

In order to reduce the complexity of the analysis, we separate the barriers into phases:

- Signaling: Time to enter the barrier and notifying that we are in
- Leaving: Time to realize that the synchronization is over and we can leave the barrier

This classification can be useful because some implementations focus on reducing the impact of the first while others focus on the second.

3 Design of Different Barriers

In this section we summarize different designs for barrier implementation and their characteristics. Based on the discussion presented in the previous section, we try to reduce the overhead of a barrier by decreasing the contention for a shared memory location and reducing the volume of data transmitted on the network. We start with a simple barrier implementation using *Fetch-and-Add*.

3.1 Barrier with Fetch-and-Add

In this case, a global counter is allocated in the shared memory of a parallel system. Before a barrier starts, the counter will be set to the number of threads participating in the parallel region. As the threads come to the barrier point, each one will decrease the counter with an atomic fetch-and-add operation and then spin on checking the counter value until the result is zero.

This implementation is quite similar to the barrier implementation introduced in [11]. The difference is that instead of doing scheduling, a busy-wait method is enforced by letting each thread constantly read the shared counter. The whole process is shown in Figure 3.

In the diagram, we use square blocks to represent our data structure, and arrowed lines with comments to represent actions that a thread may apply to the structure, in this case, the shared memory counter.

On a POWER4 system, the fetch-and-add operation is implemented using a `lwarx` (load and reserve) and `stwcx` (store conditional) instruction pair. As shown in the following assembly code, the two instructions work together to conditionally store a word to an indexed memory position.

```
static inline int fetch_and_add (volatile int *mem, int val)
{
  int tmp, result;
  asm volatile("\n\
0:      lwarx   %0,0,%2     \n\
```

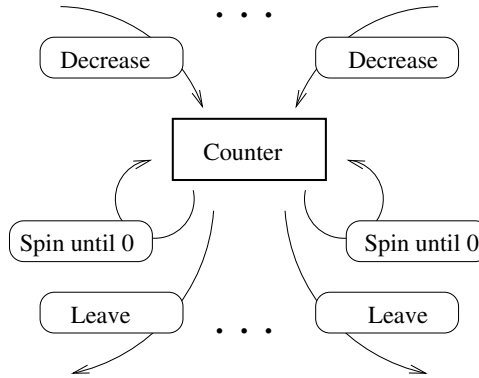


Fig. 3. Barrier implemented with fetch-and-add

```
        add%I3  %1,%0,%3    \n\
        stwcx.  %1,0,%2     \n\
        bne-    0b          \n\
" : "=&b"(result), "=&r"(tmp) : "r" (mem), "Ir\
"(val) : "cr0", "memory");
  return result;
}
```

The first instruction will create a reservation for use by the second instruction. The store can only be successful when the reserved bit is set. If not, the program will repeat the process. In this way, the procedure prevents two threads from updating the counter at the same time.

The advantage of this implementation is obvious. It has direct hardware support and is quite simple in terms of coding. It does not cost much memory either.

As we will find out, the performance of this design is acceptable only for a parallel system with a small number of processors, for example, less than eight nodes. We will see that as the number of processors increases, the memory contention increases sharply.

3.2 Distributed Counter

Since the fetch-and-add design has a high contention cost as the number of processors increases, we would like to coordinate threads using multiple memory locations to reduce contention.

As a result, we produced a new implementation we call a *distributed counter* shown in Figure 4. Instead of allocating one counter in the shared memory, we allocate multiple counters as a byte array. The size of the array is equal to the number of threads in the barrier operation and the value of each element in the counter is initialized to one.

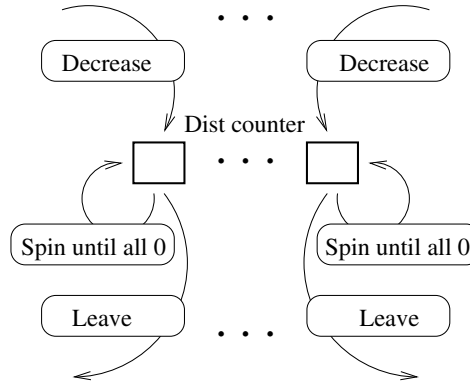


Fig. 4. Barrier implemented with distributed counter

Similarly to the fetch-and-add barrier design, each thread arriving at the barrier point will decrease the counter. However, unlike the fetch-and-add design, they will decrease only their own element of the counter. In this way, there is no need for the fetch-and-add function because simultaneous decrementing of the counter cannot happen.

In the spinning phase, each thread reads all the elements of the distributed counter array to make sure that all of the threads have decremented their own elements, thus arrived at the barrier point.

The idea of using different memory locations was also discussed in [12]. One difference here is that in their implementation the array element is increased by each thread continuously to mark the "milestones" of the synchronization.

From the test data shown later, we can see that this barrier design outperforms the fetch-and-add design and requires only a modest increase in memory (proportional to number of threads). With a 32-processor POWER4 system, we did not incur severe memory cost. As in the fetch-and-add design, we can assign multiple counters for multiple barriers, simplify the implementation.

3.3 Distributed Counter with Padding

One problem with the distributed counter design as described in the previous section is that false sharing can happen between elements of the counter array which reside on the same cache line. The result is that contention happens in the memory system even though counter storage is not strictly shared. False sharing can be eliminated by allocating the counter such that no two counter elements reside on the same cache line as shown in Figure 5.

For each thread, the operation sequence is exactly the same as the distributed counter design. The only thing different in the design is the distributed counter itself. The data structure is padded corresponding to the size of a cache line (128 bytes in a POWER4 system).

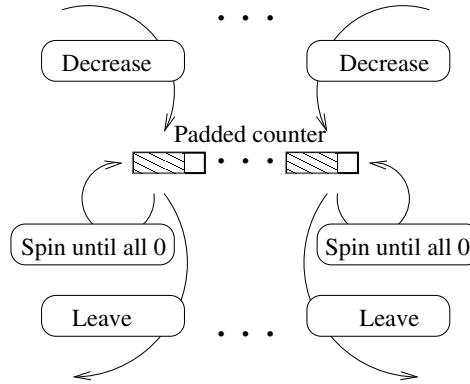


Fig. 5. Barrier implemented with padded distributed counter

In section 4, we will see that this further reduces the time needed by a barrier operation. The drawback of this scheme is that the memory impact will be amplified by the unused padding space. Among the 128 byte cache line, only one byte is used.

We solve this problem by setting up two counter arrays in each parallel region and allow all of the barriers in the same parallel region to share these two counter sets. This will reduce the memory consumption, while taking full advantage of a padded distributed counter.

To reuse the same data structure, we need to reinitialize the counter elements back to one after a synchronization. In case the program encounters multiple barriers in a small period of time, like the EPCC test. We need make sure that when we reinitialize the counter for the second barrier, we do not contaminate the previous one.

Suppose we have a very fast and a very slow thread, when the fast thread is free and encounters another barrier right away, it needs to reinitialize the counter before it can decrease the counter as designed. If this is the same counter as the one used in the previous barrier, the slow thread may still spin on checking whether all of the counter elements are zero. If the bit is reinitialized to one, the slow thread will not leave the first barrier nicely.

By having two counters, the threads can always initialize the alternative counter while leaving the current one, knowing that no threads are spinning on checking that one. Otherwise the current counter elements can not be all zero.

We can further reduce the memory cost by merging the two arrays into one position by using different bytes available in a cache line. If two barriers are not too close to each other, this won't affect the overall performance, but will halve the memory cost.

3.4 Combined with Local Sensor

Another interesting design was exploited in [7] and [8]. In comparison to the fetch-and-add barrier, the difference is that the authors did not let the threads spin on checking the hot accessed counter, but instead used a separate local sensor.

A bit is allocated in the shared memory local to each thread, and initialized to one before a barrier starts. Each thread, after decreasing the counter with a fetch-and-add operation, spins on checking the local bit. The last thread, the one that decreases the counter to zero, will set all the sensor bits to zero, signaling the spinning threads that they are free to leave. Thus, the counter is read many fewer times in the latter part of the barrier synchronization.

We implemented this barrier in our test environment and the performance data is available in the next section.

Finally, we can combine this idea with what we have in previous designs to create a new barrier scheme.

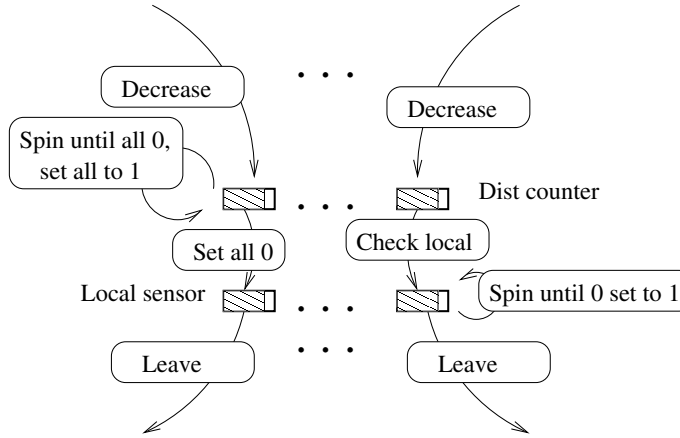


Fig. 6. Combined barrier with distributed counter and local sensor

In Figure 6, we have both a padded distributed counter and a local sensor. The local sensor is the same as the distributed counter, implemented as an array of cache lines.

Before a barrier starts, all the elements of the counter array will be set to one, as will the local sensor counter array. We let one thread in the group, for instance the master thread, act as if it is the last thread. It will decrease its own element of the distributed counter array and then spin to check whether all of the counter elements are zero. The rest of the threads will decrease their own counter elements and then spin on checking their own local sensors.

When the designated thread finds the counter elements are all zero, it will set all the counter elements back to one and then zero all of the elements in the

local sensor array. Finally, when all of the threads leave the barrier after their local sensor is zeroed, they reset their local sensor back to one.

Algorithm 1 describes this more formally[2].

Algorithm 1: Barrier with distributed counter and local sensor

```
Data  : Distributed counter with each element as one
Data  : Local sensor with each element as one
begin
    Decrease my own distributed counter element;
    if I am the master thread then
        repeat
            foreach element in distributed counter do check if it is zero
        until all distributed counter elements are zero;
        foreach element in distributed counter do
            set it back to one
        end
        foreach element in local sensor do
            set it to zero
        end
    else
        repeat
            check my local sensor element
        until it is zero;
    end
    Set my own local sensor element back to one;
end
```

We would like to avoid allocating two cache line arrays for every barrier. In our test code, we use the same idea to reduce memory cost as we did for padded distributed array by letting all the barriers in a parallel region share the same pair of counter and sensor.

Unlike the previous situation, we do not need two groups of counter here. The reason is that the last thread resets the counter values before any thread leaves the barrier, and each thread will reset its own sensor right after it is free.

In the combined barrier case, even if the fast thread is already spinning on checking its sensor for the second barrier, its counter element value will not affect the slow thread. This is so because, by the time the fast thread can decrease its counter element, the slowest thread must have passed the phase of resetting all the counter elements. The counter reset operation is done by the last thread (the slowest one) before it frees the remaining threads from the first barrier. In the worst case, the slowest thread will still be spinning on its sensor for the first barrier when the fast thread is spinning on its sensor for the second barrier.

[2] Note that we did not emphasize the data structure here, they can be either a padded one or not, or even share the same cache line as we discussed previously.

In order to save memory, we could also merge the counter and the sensor into the same cache line using a different byte position. However, this would increase the barrier overhead (largely nullifying the benefit of the local sensors) as the counter and the sensor will be accessed at the same time in the same synchronization.

4 Performance Data and Analysis

We measured the EPCC benchmark using various barrier designs to show the differences in performance overhead.

Figure 7 shows the barrier overhead for each kind of implementations on a 32-way POWER4 system with varying numbers of threads. We used different labels to tag different designs. *FetchAndAdd* is for the first and the simplest barrier design. *DistCounter* is used for the barrier with distributed counter. *DistCounterPad* is used for the barrier with padded distributed counter. *LocalSensor* is for the barrier using local sensor. And the last one, *Combined* represents our new barrier scheme, the barrier with distributed counter combined with local sensor.

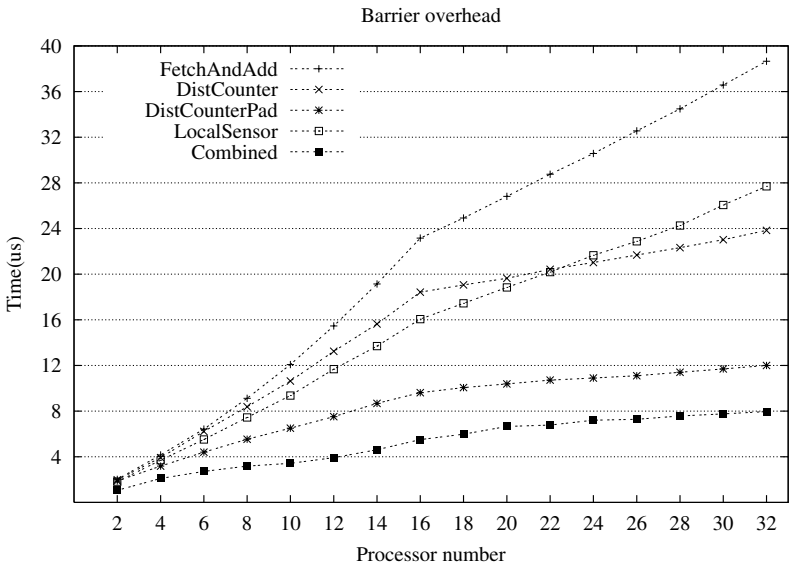


Fig. 7. POWER4 barrier overhead

To further understand the behavior of these barrier designs, we repeated the tests on a 16-way 375MHz POWER3 system. Figure 8 shows the overheads on POWER3.

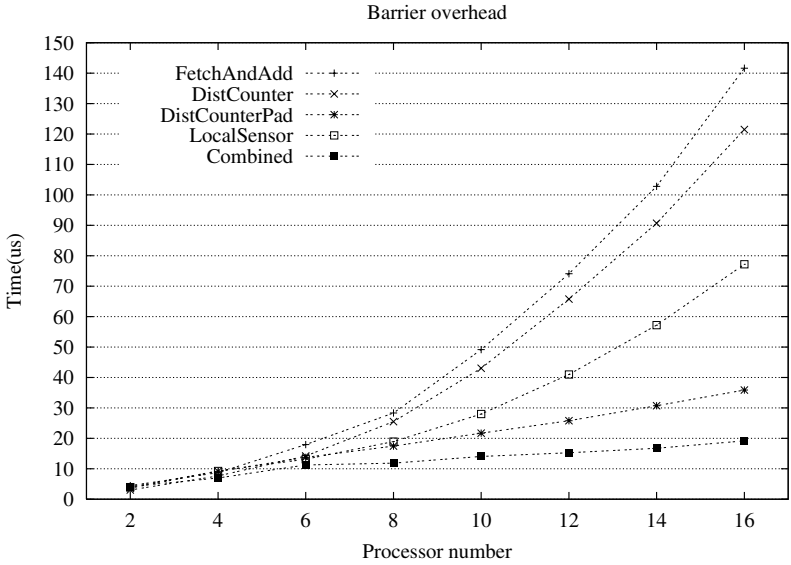


Fig. 8. POWER3 barrier overhead

Although POWER4 and POWER3 have different memory architectures[13], one can see that there is little difference in the relative overhead for different barrier designs. Of course, we could not compare the scalability of the designs beyond 16 threads on the POWER3 system so a completely equitable comparison was not possible.

From the figures we can see that the scalability of the overhead is not exactly the same, even considering the difference of the clock speed of the processors — recall that the POWER4 system we used is 1.1GHz and the POWER3 is 375MHz. The memory system in POWER4 system certainly gives it extra advantage.

To be able to compare the different implementations, we examine the performance counters on the more interesting POWER4 system, using `pmcount` program available on AIX 5.1. The program will print out the values of the different performance counters on each processor, when monitoring a given execution program.

First we do a simple measurement for the fetch-and-add barrier, using the exact same setting as we measure overhead previously. We check the performance counter for store conditional instruction to see the relationship between contention and the number of processors involved in a barrier operation. Figure 9 shows the average number of pass and fail for the store conditional instructions on each processor.

As one can see, while the number of pass remained almost the same(in theory, it should be identical since we have the same number of barrier for different

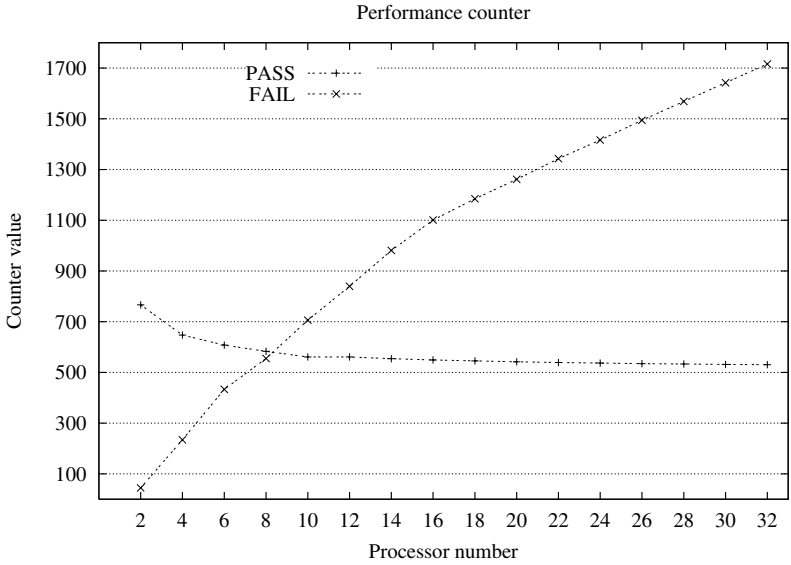


Fig. 9. Store conditional instruction pass and fail

number of processors, we consider it is measurement error here), the number of fail increased sharply as more processors are added. That explains why we can not have a scalable performance for fetch-and-add barrier.

Similarly, we can get the performance counter for cache misses, representing data traffic. An interesting case here is L2 miss for distributed counter.

For a POWER4 system, an L2 miss may be served by L2 from another L2 in the same MCM, L2 in a different MCM, L3 in the same MCM and L3 in a different MCM. We put them all together as L2 misses. Figure 10 pictures average L2 misses on different processors for barrier with distributed counter and padded distributed counter.

To our surprise, there are more cache misses in the barrier with padded counter, even though it cost less time. We are not sure what exactly caused this, but we suspect that the cache coherent algorithm used in hardware cause extra contention when different processors accessing the same cache line, which out weighs the traffic overhead caused by L2 misses.

5 Summary and Future Work

Barrier synchronization is a well-studied topic, and an important one in parallel programming. In this paper, we studied the performance characteristics of several alternative implementations of barrier synchronization on modern, complex shared memory multiprocessors.

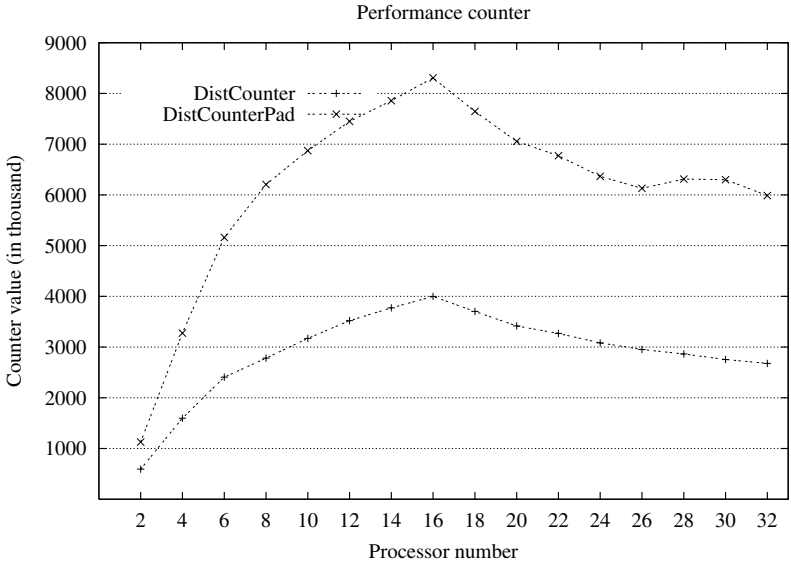


Fig. 10. L2 misses

We have implemented several different barrier schemes and measured their performance on the shared memory POWER3 system and the distributed shared memory POWER4 system. We analyzed barrier performance data through timing and hardware performance counters. In the future, we wish to study the impact of different barrier implementations on other shared memory platforms particularly looking at issues of non-uniform memory access and scalability.

Through the introduction of the *distributed counters with local sensor* implementation, we have demonstrated a dramatic 79% reduction in overhead on a 32-way POWER4 system when comparing to a fetch-and-add implementation and a 33% improvement on the same system over the next best technique, distributed counters with padding. While these experiments were done using the explicit barrier test in EPCC microbenchmarks, we get similar improvements in the performance of the implied barriers terminating work-sharing constructs and parallel regions. The availability of a low overhead barrier also provides more freedom to the compiler to automatically introduce parallelism in serial code.

Acknowledgments

Roch Archambault (from the IBM Toronto Lab) had useful discussions with us; Raul Silvera and Shimin Cui (from the IBM Toronto Lab) gave us valuable technical assistance during the integration of the new barrier scheme to the existing SMP runtime frame work.

References

1. Message Passing Interface Forum. MPI: A message-passing interface standard, 1994.
2. V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–339, December 1990.
3. Arvind Krishnamurthy and Katherine A. Yelick. Optimizing parallel programs with explicit synchronization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–204, 1995.
4. OpenMP Architecture Review Board. OpenMP specification FORTRAN version 2.0, 2000. `http://www.openmp.org`.
5. OpenMP Architecture Review Board. OpenMP specification C/C++ version 2.0, 2002. `http://www.openmp.org`.
6. Edinburgh Parallel Computing Center. OpenMP microbenchmarks, 1999. `http://www.epcc.ed.ac.uk/research/openmpbench`.
7. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, February 1991.
8. Dimitrios S. Nikolopoulos and Theodore S. Papatheodorou. A quantitative architectural evaluation of synchronization algorithms and disciplines on ccNUMA systems: The case of the SGI Origin2000. June 1999.
9. Steve Behling et al. The POWER4 processor introduction and tuning guide. Technical Report SG24-7041-00, International Technical Support Organization, November 2001. ISBN 0738423556.
10. J. M. Bull. Measuring synchronization and scheduling overheads in OpenMP. In *First European Workshop on OpenMP*, October 1999.
11. IBM Technical Disclosure Bulletin. Barrier Synchronization Using Fetch-and-Add and Broadcast. 34(8):33–34, 1992.
12. Rainer Kreuzburg. Method of synchronization, 2001. United States Patent, No. US 6,330,619.
13. Stefan Andersson et al. RS/6000 scientific and technical computing: POWER3 introduction and tuning guide. Technical Report SG24-5155-00, International Technical Support Organization, October 1998.

An OpenMP Implementation of Parallel FFT and Its Performance on IA-64 Processors

Daisuke Takahashi, Mitsuhisa Sato, and Taisuke Boku

Institute of Information Sciences and Electronics, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan
{daisuke,msato,taisuke}@is.tsukuba.ac.jp

Abstract. In this paper, we propose an OpenMP implementation of a recursive algorithm for parallel fast Fourier transform (FFT) on shared-memory parallel computers. A recursive three-step FFT algorithm improves performance by effectively utilizing the cache memory. Performance results of one-dimensional FFTs on the DELL PowerEdge 7150 and the hp workstation zx6000 are reported. We successfully achieved performance of about 757 MFLOPS on the DELL PowerEdge 7150 (Itanium 800 MHz, 4 CPUs) and about 871 MFLOPS on the hp workstation zx6000 (Itanium2 1 GHz, 2 CPUs) for 2^{24}-point FFT.

1 Introduction

OpenMP has emerged as the standard for shared-memory parallel programming. The OpenMP Application Program Interface (API) provides programmers with a simple way to develop parallel applications for shared-memory parallel computers.

The fast Fourier transform (FFT) [1] is one of the most widely used algorithms in science and engineering. Parallel FFT algorithms on shared-memory parallel computers have been well studied [2,3,4,5,6].

Many FFT algorithms work well when data sets fit into a cache. When a problem exceeds the cache size, however, the performance of these FFT algorithms decreases dramatically. One goal for large FFTs is to minimize the number of cache misses. A recursive algorithm is very good at improving the use of the cache. Thus, some previously proposed FFT algorithms [7,8] use a recursive approach.

In this paper, we propose an OpenMP implementation of a recursive algorithm for computing large one-dimensional FFTs on shared-memory parallel computers. We have implemented a recursive three-step FFT on IA-64 processors, the DELL PowerEdge 7150 (Itanium 800 MHz, 4 CPUs) and the hp workstation zx6000 (Itanium2 1 GHz, 2 CPUs), and in this paper we report the performance results.

Section 2 describes a recursive three-step FFT algorithm. Section 3 describes the in-cache FFT algorithm used for problems that fit into a data cache, and parallelization. Section 4 gives performance results. In section 5, we provide some concluding remarks.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 99–108, 2003.

```
      RECURSIVE SUBROUTINE RECURSIVE_FFT(A,W,U,N)
      DOUBLE COMPLEX A(*),W(*),U(*)
      IF (N .LE. CACHESIZE) THEN
        CALL IN_CACHE_FFT(A,W,U,N)
        RETURN
      END IF
! Step 1 (n1 simultaneous n2-point multirow FFTs with
!         twiddle factor multiplication)
      DO I=1,N/2
        W(I)=A(I)+A(I+N/2)
        W(I+N/2)=(A(I)-A(I+N/2))*U(I)
      END DO
! Step 2 (n2 individual n1-point multicolumn FFTs)
      DO J=1,2
       CALL RECURSIVE_FFT(W((J-1)*(N/2)+1),A,U(N/2+1),N/2)
      END DO
! Step 3 (Transpose)
      DO I=1,N/2
        A(2*I-1)=W(I)
        A(2*I)=W(I+N/2)
      END DO
      RETURN
      END
```

Fig. 1. A recursive three-Step FFT algorithm ($n_1 = n/2$ and $n_2 = 2$)

2 A Recursive Three-Step FFT Algorithm

The discrete Fourier transform (DFT) is given by

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, \quad 0 \le k \le n-1, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$.

If n has factors n_1 and n_2 ($n = n_1 \times n_2$), then the indices j and k can be expressed as:

$$j = j_1 + j_2 n_1, \quad k = k_2 + k_1 n_2. \tag{2}$$

We can define x and y as two-dimensional arrays (in Fortran notation):

$$x_j = x(j_1, j_2), \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1, \tag{3}$$

$$y_k = y(k_2, k_1), \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1. \tag{4}$$

Substituting the indices j and k in equation (1) with those in equation (2), and using the relation of $n = n_1 \times n_2$, we can derive the following equation:

$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \omega_{n_1}^{j_1 k_1}. \tag{5}$$

```
      SUBROUTINE PARALLEL_FFT(A,W,U,N)
      DOUBLE COMPLEX A(*),W(*),U(*)
      IF (N .LE. CACHESIZE) THEN
        CALL IN_CACHE_FFT(A,W,U,N)
        RETURN
      END IF
!$OMP PARALLEL
!$OMP DO
      DO I=1,N/2
        W(I)=A(I)+A(I+N/2)
        W(I+N/2)=(A(I)-A(I+N/2))*U(I)
      END DO
!$OMP DO
      DO J=1,2
      CALL RECURSIVE_FFT(W((J-1)*(N/2)+1),
     +   A((J-1)*(N/2)+1,U(N/2+1),N/2)
      END DO
!$OMP DO
      DO I=1,N/2
        A(2*I-1)=W(I)
        A(2*I)=W(I+N/2)
      END DO
!$OMP END PARALLEL
      RETURN
      END
```

Fig. 2. An OpenMP implementation of a recursive three-Step FFT algorithm

This derivation leads to a following three-step FFT algorithm:

Step 1: n_1 simultaneous n_2-point multirow FFTs with twiddle factor multiplication

$$x_1(j_1,\, k_2) = \sum_{j_2=0}^{n_2-1} x(j_1,\, j_2)\omega_{n_2}^{j_2 k_2}\omega_{n_1 n_2}^{j_1 k_2}.$$

Step 2: n_2 individual n_1-point multicolumn FFTs

$$x_2(k_1,\, k_2) = \sum_{j_1=0}^{n_1-1} x_1(j_1,\, k_2)\omega_{n_1}^{j_1 k_1}.$$

Step 3: Transpose

$$y(k_2,\, k_1) = x_2(k_1,\, k_2).$$

The distinctive features of the three-step FFT algorithm can be summarized as:

Table 1. Specification of machines

Platform	DELL PowerEdge 7150	hp workstation zx6000
Number of CPUs	4	2
CPU Type	Itanium 800 MHz	Itanium2 1 GHz
L1 Cache	I-Cache: 16 KB D-Cache*: 16 KB Associativity: 4-way	I-Cache: 16 KB D-Cache*: 16 KB Associativity: 4-way
L2 Cache	96 KB Unified Associativity: 6-way	256 KB Unified Associativity: 8-way
L3 Cache	4 MB Unified Associativity: 4-way	3 MB Unified Associativity: 12-way
Main Memory	32 GB	1 GB
OS	Linux 2.4.9-18smp	Linux 2.4.18-e.12smp

* The L1 D-Cache only caches data for the integer unit, not the floating-point unit.

- One multirow FFT and one multicolumn FFT are performed in steps 1 and 2, respectively.
- A matrix transposition takes place just once in step 3.

For each n_1-point multicolumn FFT in step 2 can be performed recursively. We will call it a recursive three-step FFT algorithm. Fig. 1 gives the pseudo-code for the recursive three-step FFT algorithm where $n_1 = n/2$ and $n_2 = 2$.

Here the arrays `A` and `W` are the input/output array and work array, respectively. The twiddle factors $\omega_{n_1 n_2}^{j_1 k_2}$ in step 1 are stored in array `U`. We can use the padding technique [9] for each recursive step.

3 In-Cache FFT Algorithm and Parallelization

The Stockham autosort algorithm [10] works well until the problem size exceeds the on-chip cache size. When the problem exceeds on-chip cache size, the recursive three-step FFT algorithm should be used.

We use the radix-2, 4 and 8 Stockham autosort algorithm for in-cache FFT. The higher radices are more efficient in terms of both memory and floating-point operations. A high ratio of floating-point instructions to memory operations is particularly important in a cache-based processor. In view of the high ratio of floating-point instructions to memory operations, the radix-8 FFT is more advantageous than the radix-4 FFT. A power-of-two point FFT (except for 2-point FFT) can be performed by a combination of radix-8 and radix-4 steps containing at most two radix-4 steps. That is, the power-of-two FFTs can be performed as a length $n = 2^p = 4^q 8^r$ ($p \geq 2,\ 0 \leq q \leq 2,\ r \geq 0$).

We parallelized the recursive three-step FFT by using OpenMP directives. Many existing OpenMP systems do not sufficiently implement nested parallelism [11]. Since the recursive three-step FFT has enough outermost parallelism, it is not necessary to use the nested parallelism. An OpenMP implementation of the recursive three-step FFT algorithm is shown in Fig. 2. The

Table 2. Performance of the recursive three-step FFT on the DELL PowerEdge 7150 (Itanium 800 MHz, 4 CPUs)

n	1 CPU		2 CPUs		4 CPUs	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00024	1039.63	0.00018	1350.97	0.00021	1157.54
2^{13}	0.00063	844.70	0.00044	1217.10	0.00042	1256.12
2^{14}	0.00131	875.25	0.00089	1288.95	0.00084	1372.04
2^{15}	0.00295	831.82	0.00190	1291.72	0.00172	1425.32
2^{16}	0.00810	647.08	0.00445	1178.72	0.00358	1462.89
2^{17}	0.02425	459.50	0.01235	901.97	0.00784	1421.90
2^{18}	0.05670	416.09	0.03348	704.73	0.02216	1064.87
2^{19}	0.12988	383.48	0.08044	619.15	0.06076	819.73
2^{20}	0.28271	370.90	0.17786	589.56	0.13086	801.30
2^{21}	0.62305	353.43	0.38794	567.62	0.27026	814.76
2^{22}	1.40723	327.86	0.88281	522.62	0.56689	813.86
2^{23}	3.08887	312.31	1.94727	495.41	1.17676	819.79
2^{24}	6.56055	306.87	4.17871	481.79	2.66016	756.82

parallelized subroutine `PARALLEL_FFT()` shown in Fig. 2 calls the recursive subroutine `RECURSIVE_FFT()` shown in Fig. 1.

Although the range of `DO J=1,2` loop shown in Fig. 2 is two, the loop length can be extended to $n_2 \geq 2$ shown in equation (5). Since we use the radix-8 FFT algorithm for the OpenMP implementation of the recursive three-step FFT, the range of `J` is 8.

Each directive of OpenMP may cause an overhead. In order to reduce fork/join overhead, three parallel regions can be fused shown in Fig. 2. Thus, the parallelized subroutine `PARALLEL_FFT()` has only one `PARALLEL` directive.

4 Performance Results

To evaluate the proposed recursive three-step FFT, we compared its performance against that of both the FFT library of FFTW (version 2.1.3) [8] and the ZFFT1D routine of Intel Math Kernel Library (MKL) on IA-64 processors. We averaged the elapsed times obtained from 10 executions of complex forward FFTs where input and output are in normal order. The FFTs were performed on double-precision complex data, and the table for twiddle factors was prepared in advance.

All routines were written in Fortran 90. The specifications for the two IA-64 platforms used are shown in Table 1.

4.1 Performance Results on the DELL PowerEdge 7150

The compiler used on the Intel Fortran Itanium Compiler (version 7.0) on the DELL PowerEdge 7150. For the recursive three-step FFT, the compiler options

Table 3. Performance of the FFTW on the DELL PowerEdge 7150 (Itanium 800 MHz, 4 CPUs)

n	1 CPU		2 CPUs		4 CPUs	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00024	1022.08	0.00045	548.07	0.00085	290.73
2^{13}	0.00069	767.53	0.00070	764.54	0.00093	574.75
2^{14}	0.00143	802.80	0.00126	911.47	0.00126	906.80
2^{15}	0.00348	706.94	0.00256	961.14	0.00205	1198.58
2^{16}	0.01152	455.07	0.00592	885.01	0.00422	1243.05
2^{17}	0.03438	324.05	0.01773	628.33	0.01082	1029.79
2^{18}	0.08081	291.95	0.04226	558.29	0.02482	950.51
2^{19}	0.16774	296.94	0.09047	550.57	0.05295	940.66
2^{20}	0.35264	297.35	0.19213	545.77	0.11206	935.69
2^{21}	0.80010	275.22	0.41954	524.87	0.24711	891.10
2^{22}	1.67996	274.63	0.95553	482.84	0.56100	822.41
2^{23}	3.95373	243.99	2.19172	440.15	1.43824	670.74
2^{24}	10.20131	197.35	4.21222	477.96	3.80907	528.55

Table 4. Performance of the Intel MKL on the DELL PowerEdge 7150 (Itanium 800 MHz, 4 CPUs)

n	1 CPU		2 CPUs		4 CPUs	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00022	1104.81	0.00024	1031.31	0.00028	869.13
2^{13}	0.00051	1048.54	0.00043	1233.23	0.00437	1217.10
2^{14}	0.00116	984.92	0.00842	1362.71	0.00768	1493.90
2^{15}	0.00237	1034.93	0.00206	1195.26	0.00159	1546.81
2^{16}	0.00529	990.19	0.00436	1201.39	0.00317	1655.89
2^{17}	0.01494	745.81	0.01059	1052.08	0.00688	1618.95
2^{18}	0.06174	382.15	0.02834	832.63	0.01877	1257.06
2^{19}	0.15503	321.28	0.08777	567.49	0.06027	826.37
2^{20}	0.33276	315.11	0.31836	329.37	0.27710	378.41
2^{21}	0.71387	308.46	0.78223	281.51	0.70068	314.27
2^{22}	1.48926	309.80	1.67383	275.64	1.48438	310.82
2^{23}	3.16602	304.70	3.51270	274.63	2.92090	330.27
2^{24}	6.61328	304.43	7.34668	274.04	7.02344	286.65

used were specified as, "`-O3 -tpp1 -openmp`". These options instruct the compiler to enable optimizations ("`-O2`") plus more aggressive optimizations, to optimize for Itanium processor ("`-tpp1`") and to enable the compiler to generate multi-threaded code based on the OpenMP directives ("`-openmp`"), respectively. For the FFTW and Intel MKL, the compiler options used were specified as, "`-O3 -tpp1`".

Tables 2, 3 and 4 compare the recursive three-step FFT, the FFTW and the ZFFT1D routine of Intel MKL in terms of their run times and MFLOPS.

Table 5. Performance of the recursive three-step FFT on the hp workstation zx6000 (Itanium2 1 GHz, 2 CPUs)

n	1 CPU		2 CPUs	
	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00012	2120.97	0.00012	2122.06
2^{13}	0.00027	1955.68	0.00027	1957.39
2^{14}	0.00062	1857.28	0.00062	1850.14
2^{15}	0.00129	1910.29	0.00128	1920.25
2^{16}	0.00303	1729.88	0.00302	1738.63
2^{17}	0.00951	1171.04	0.00491	2271.06
2^{18}	0.02370	995.61	0.01631	1446.39
2^{19}	0.05173	962.88	0.04034	1234.56
2^{20}	0.11658	899.47	0.08728	1201.39
2^{21}	0.24866	885.56	0.18909	1164.55
2^{22}	0.53857	856.66	0.42285	1091.10
2^{23}	1.28320	751.78	0.99658	968.00
2^{24}	2.87305	700.74	2.31055	871.34

The column headed by n shows the number of points of FFTs. The next six columns contain the average elapsed time in seconds and the average execution performance in MFLOPS. The MFLOPS values are each based on $5n \log_2 n$ for a transform of size $n = 2^m$.

The recursive three-step FFT is faster than the FFTW except for $2^{19} \leq n \leq 2^{22}$, 4 CPUs. On the other hand, for $2^{14} \leq n \leq 2^{19}$, 4 CPUs the recursive three-step FFT is slower than the MKL, whereas for $n \geq 2^{20}$, 4 CPUs the recursive three-step FFT is faster than the MKL. For each Itanium processor, power-of-two Stockham FFTs up to size 2^{11} points (Fortran complex*16) fit into the 96 KB L2 on-chip cache, and FFTs up to size 2^{16} points fit into the 4 MB L3 off-chip cache. The associativity of L3 cache of Itanium processor is 4-way, and the cache-miss ratio is high for a larger problem size on the radix-8 FFT algorithm of the recursive three-step FFT. This is the reason why for $2^{14} \leq n \leq 2^{19}$, 4 CPUs the recursive three-step FFT is slower than the MKL.

The speedup of the recursive three-step FFT is better than that of both the FFTW and MKL for a smaller problem size. These results clearly indicate that the OpenMP implementation of the recursive three-step FFT has low fork/join overhead.

4.2 Performance Results on the hp Workstation zx6000

The compiler used on the Intel Fortran Itanium Compiler (version 7.0) on the hp workstation zx6000. For the recursive three-step FFT, the compiler options used were specified as, "`-O3 -tpp2 -openmp`". These options instruct the compiler to enable optimizations ("`-O2`") plus more aggressive optimizations, to optimize for Itanium2 processor ("`-tpp2`") and to enable the compiler to generate multi-threaded code based on the OpenMP directives ("`-openmp`"), respectively. For

Table 6. Performance of the FFTW on the hp workstation zx6000 (Itanium2 1 GHz, 2 CPUs)

n	1 CPU		2 CPUs	
	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00011	2299.86	0.00028	882.93
2^{13}	0.00024	2207.81	0.00036	1484.61
2^{14}	0.00060	1906.05	0.00061	1874.89
2^{15}	0.00153	1610.49	0.00105	2336.47
2^{16}	0.00337	1555.03	0.00218	2409.17
2^{17}	0.01325	840.66	0.00815	1367.36
2^{18}	0.03711	635.71	0.02431	970.61
2^{19}	0.07607	654.72	0.04931	1010.09
2^{20}	0.16225	646.26	0.10083	1039.90
2^{21}	0.34453	639.14	0.22209	991.48
2^{22}	0.70362	655.71	0.45596	1011.87
2^{23}	1.58306	609.38	0.95822	1006.75
2^{24}	3.16589	635.92	1.90456	1057.08

Table 7. Performance of the Intel MKL on the hp workstation zx6000 (Itanium2 1 GHz, 2 CPUs)

n	1 CPU		2 CPUs	
	Time	MFLOPS	Time	MFLOPS
2^{12}	0.00016	1529.37	0.00013	1872.47
2^{13}	0.00033	1626.64	0.00024	2181.04
2^{14}	0.00073	1561.81	0.00051	2247.83
2^{15}	0.00171	1439.65	0.00102	2415.16
2^{16}	0.00384	1364.84	0.00227	2313.79
2^{17}	0.00790	1410.91	0.00493	2258.76
2^{18}	0.02856	825.96	0.01402	1683.38
2^{19}	0.08160	610.35	0.04141	1202.72
2^{20}	0.17932	584.75	0.20886	502.04
2^{21}	0.39111	563.01	0.50195	438.69
2^{22}	0.82227	561.10	1.02148	451.67
2^{23}	1.83887	524.61	2.38086	405.19
2^{24}	3.80566	529.02	5.11914	393.28

the FFTW and Intel MKL, the compiler options used were specified as, "`-O3 -tpp2`".

Tables 5, 6 and 7 compare the recursive three-step FFT, the FFTW and the ZFFT1D routine of Intel MKL in terms of their run times and MFLOPS. The column headed by n shows the number of points of FFTs. The next four columns contain the average elapsed time in seconds and the average execution performance in MFLOPS. The MFLOPS values are each based on $5n \log_2 n$ for a transform of size $n = 2^m$.

The recursive three-step FFT is faster than the FFTW except for $2^{14} \leq n \leq 2^{16}$ and $2^{23} \leq n \leq 2^{24}$, 2 CPUs. On the other hand, for $2^{13} \leq n \leq 2^{16}$ and $n = 2^{18}$, 2 CPUs the recursive three-step FFT is slower than the MKL, whereas for $n \geq 2^{19}$, 2 CPUs the recursive three-step FFT is faster than the MKL. For each Itanium2 processor, power-of-two Stockham FFTs up to size 2^{16} points which fit into the 3 MB L3 on-chip cache, are not parallelized. This is the main reason why for $2^{13} \leq n \leq 2^{16}$, 2 CPUs the recursive three-step FFT is slower than the MKL.

The performance of the recursive three-step FFT remains at a high level even for a larger problem size, owing to both recursive approach and padding. These results clearly indicate that for larger problem sizes the recursive three-step FFT is superior to the MKL.

We note that with the DELL PowerEdge 7150 as well as with the hp workstation zx6000, about 757 MFLOPS and about 871 MFLOPS, respectively, were realized with size $n = 2^{24}$ in the proposed recursive three-step FFT, as shown in Tables 2 and 5.

5 Conclusion

In this paper, we proposed the OpenMP implementation of a recursive algorithm for parallel FFT on shared-memory parallel computers. We show the parallelization of a recursive algorithm by using OpenMP directives. The performance of the recursive three-step FFT remains at a high level even for a larger problem size, owing to both recursive approach and padding. Furthermore, the speedup of the recursive three-step FFT is better than that of both the FFTW and MKL for a smaller problem size.

These results demonstrate that the proposed recursive three-step FFT utilizes cache memory effectively, and it has low fork/join overhead. We successfully achieved performance of about 757 MFLOPS on the DELL PowerEdge 7150 (Itanium 800 MHz, 4 CPUs) and about 871 MFLOPS on the hp workstation zx6000 (Itanium2 1 GHz, 2 CPUs).

Acknowledgments

We wish to thank Dr. Y. Ishikawa at the University of Tokyo for using of the hp workstation zx6000. This work was partially supported by the Grant-in-Aid for Scientific Research (A) (No. 14208026) of Japan Society for the Promotion of Science and the Grant-in-Aid for Young Scientists (B) (No. 14780185) of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

1. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex Fourier series. Math. Comput. **19** (1965) 297–301
2. Swarztrauber, P.N.: Multiprocessor FFTs. Parallel Computing **5** (1987) 197–210

3. Bailey, D.H.: FFTs in external or hierarchical memory. The Journal of Supercomputing **4** (1990) 23–35
4. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM Press, Philadelphia, PA (1992)
5. Wadleigh, K.R.: High performance FFT algorithms for cache-coherent multiprocessors. The International Journal of High Performance Computing Applications **13** (1999) 163–171
6. Takahashi, D.: A blocking algorithm for parallel 1-D FFT on shared-memory parallel computers. In: Proc. 6th International Conference on Applied Parallel Computing (PARA 2002). Volume 2367 of Lecture Notes in Computer Science., Springer-Verlag (2002) 380–389
7. Hegland, M.: A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. Numerische Mathematik **68** (1994) 507–547
8. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP98). (1998) 1381–1384
9. Panda, P.R., Nakamura, H., Dutt, N.D., Nicolau, A.: Augmenting loop tiling with data alignment for improved cache performance. IEEE Transactions on Computers **48** (1999) 142–149
10. Swarztrauber, P.N.: FFT algorithms for vector computers. Parallel Computing **1** (1984) 45–63
11. Tanaka, Y., Taura, K., Sato, M., Yonezawa, A.: Performance evaluation of OpenMP applications with nested parallelism. In: Proc. 5th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR 2000). Volume 1915 of Lecture Notes in Computer Science., Springer-Verlag (2000) 100–112

OpenMP and Compilation Issue in Embedded Applications*

Jaegeun Oh[1], Seon Wook Kim[1], and Chulwoo Kim[2]

[1] Advanced Computer Systems Laboratory
[2] Integrated System and Processor Laboratory
Department of Electronics and Computer Engineering
Korea University, Seoul, Korea
http://acsl.korea.ac.kr
{worms97,seon,ckim}@korea.ac.kr

Abstract. Currently embedded systems become more and more important and widely applied to everywhere around us, such as a mobile phone, a PDA, an HDTV, and so on. In this paper, we applied OpenMP to non-traditional benchmarks, i.e. embedded applications in order to examine the applicability of OpenMP in this area. We parallelized embedded benchmarks, called EEMBC, consisting of 5 categories and total 34 applications, and measure their performance in detail. From experiment, we could find 90 parallel sections in 17 applications, but we achieved speedup only in four applications. Since embedded applications consists of a chunk of small loops, we could not get speedup due to large parallelization overheads such as thread management and instruction overheads. Also we show that the OpenMP-inserted parallel code size is much larger than a serial version due to multithreaded libraries, which is critical to embedded systems because of their limited size of memory systems. We discuss an identified critical, but a trivial problem in the current OpenMP specification when we applied OpenMP to these applications.

1 Introduction

Over the past years, computer engineers and researchers have concentrated on accelerating scientific applications on high performance workstations and large-scale systems.

For this purpose, they have introduced various kinds of architectures [1,2], compilers [3,4] and parallel programming models and their APIs [5,6]. One of the efforts to provide easy programming to users on shared memory multiprocessors systems is OpenMP [5]. Currently OpenMP is a popularly used and standard API in many areas, but more in scientific applications.

The current focus in computing has shifted very fast from scientific engineering to multimedia, i.e. from high performance workstations to embedded systems, since multimedia applications become more and more important and

* This work is supported by a Korea University Grant.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 109–121, 2003.

broadly used than classical compute-bound scientific applications. The embedded applications are compute-bounded in general, but they are different from scientific applications. For example, the embedded applications consists of small workload in loops and functions, and their executable code size is very small due to a limited size of memory on embedded systems. Also it is well known that embedded applications intrinsically have data-level parallelism (DLP) instead of instruction-level parallelism (ILP) and/or thread-level parallelism (TLP).

In this study, we applied OpenMP API [5] to non-traditional application in this area, i.e. embedded multimedia benchmarks [7] for the following studies:

- Is OpenMP applicable to improve performance in embedded systems? We measured and analyzed in detail the performance of OpenMP-parallelized embedded benchmarks. Since embedded applications consist of a chunk of small loops, we could achieve speedup only in a few applications due to high parallelization overhead. More critically a compiled code size of an OpenMP-inserted parallel version is much higher than a serial, which may restrict to use OpenMP in embedded systems due to their limited size of memory systems.
- Is there any issue in OpenMP APIs when parallelizing embedded applications? We discuss a trivial, but a critical issue in OpenMP `for` directives. According to OpenMP specification, an induction variable in `for` statement should be declared as a signed integer. But, in most of our tested applications an induction variable is declared as an unsigned integer type. Even though a user's or a compiler's redeclaration of the variable is easy, we need to improve OpenMP specification for completeness.
- What are minimal functionalities of a parallelizing compiler to find parallel regions in these applications written in C? The embedded benchmarks are compute-intensive. But due to a simple pointer arithmetic, parallelizing compilers fail to parallelize them automatically.

This paper is organized as follows. In Section 2, we present an our tested multimedia benchmark, EEMBC. In Section 3 we show detailed performance in OpenMP parallelized codes. In Section 4 we discuss OpenMP specification issue which prevents a parallelizing compiler from automatically identifying parallel sections. Finally we make conclusion in Section 5.

2 EEMBC

For our study, we used a set of multimedia benchmarks, called EEMBC (EDN Embedded Microprocessor Benchmark Consortium) [7]. These codes have been developed to evaluate microprocessors, especially for embedded applications. Their codes were first released on April 2000, and now about 45 world leading semiconductors, intellectual property, and compiler companies are actively participating in this consortium. The benchmarks are categorized into five classes and their included algorithms as following:

- Consumer (digital cameras, set-top-boxes, and PDAs): RGB to CMYK/YIQ Conversion, high-pass gray-scale filter, JPEG (de)compression.

- Telecommunications (modem, ADSL, wireless): autocorrelation, convolution encoder, bit allocation, FFT, Viterbi decoder.
- Networking (network): Dijkstra's shortest-path, packet flow, and route lookup.
- Office automation (office machinery): Bezier curve calculation, dithering, image rotation.
- Automotive & Industrial (industrial controllers and auto applications): table lookup & interpolation, tooth to spark, angle to time conversion, pulse-width modulation, remote data request, road speed calculation, (in)finite impulse response filter, bit manipulation, basic arithmetic, pointer chasing, matrix arithmetic, cache buster, inverse DCT, FFT.

The main metric in this benchmark is execution throughput (iterations/second). Additionally, static code and data sizes are reported, since embedded programs typically allocate all the necessary buffer space in a static manner to avoid the overhead of dynamic memory management.

For performance measurement, there are two execution modes for measurement, an out-of-the-box mode and an optimized mode. The out-of-the-box mode does not allow any non-compiler's optimization. But in the optimized mode, any modification is allowed except for changing the underlying algorithm of the benchmark. For our study, we used the out-of-the-box mode. We applied only two simple parallelization techniques, and induction variable substitution [8] and a privatization [9] for code parallelization.

However, EEMBC benchmarks are not complete applications. We cannot evaluate system-level components and it does not include energy and power consumption. Also, the repeated large number of iterations may ignore the cost of accessing main memory.

3 Parallelization of EEMBC Benchmarks Using OpenMP

In this section, we discuss the detailed performance in all categories of EEMBC. Each application in EEMBC benchmarks consists of three parts: data preparation (data-in from input devices), an algorithm itself, and data post-processing (pass results onto output devices). We exclude pre and post data processing parts from time measurement, since they include only data accesses from/to external devices (we also report performance of an algorithm itself to EEMBC).

3.1 Experiment Methodology

For experiment, we used Intel Compiler 7.0 on an Intel/Xeon 2.0GHz two processor machine using Redhat 8.0 with `-O2 -openmp`. We measure the following performance metrics for our performance analysis:

- We measure a total and region-by-region execution time of serial codes, which do not include any OpenMP APIs. We derive the ratio of parallel regions to total execution and also ideal speedup by using Amdahl's Law.

- We measure a total and region-by-region execution time of the following two versions of parallel codes using 1 and 2 processors.
  - An unoptimized code: We insert OpenMP APIs in all identified parallel regions.
  - An optimized code: We used Intel OpenMP profiler (`-openmp_profile`) to collect OpenMP performance data such as parallel time, load imbalance, thread management overhead, and so on. We compare region-by-region performance from parallel code execution using 1 and 2 processors, and then we serialize an OpenMP-inserted region whose performance is degraded on 2 processors.
- We also measure sizes of `text`, `data`, and `bss` sections in compiled serial and OpenMP parallel codes.

3.2 Performance in OpenMP-Parallelized Codes

Overall. EEMBC benchmarks include 34 applications in 5 categories, as described in Section 2. We could find 90 parallel regions in 17 applications, but could achieve speedup only in 4 applications using 2 processors, `RGB high-pass grayscale filter` and `RGB to YIQ conversion` in `Consumer`, and `autocorrelation` in `Telecommunication`, and `Bezier cure interpolation` in `Office`. Since our studied applications consist of many small loops, unoptimized parallel versions incur high parallelization overhead such as load imbalance and instruction overhead.

Also, `text` size of the optimized OpenMP parallelized codes is twice larger than that of serial codes in most applications due to multithreaded libraries. It should be noted that an executable code size of an embedded application itself is very small. Therefore, when we parallelize codes and build executables with multithreaded libraries, the library code size becomes a major factor to determine a total code size. And `data` and `bss` sizes in parallel codes are ten times larger than those in serial ones in some applications. The executable code size increment is critical to embedded systems because they have only a few kilobytes of memory systems.

Consumer. Table 1 shows the basic property of five applications in `Consumer` benchmark. The execution time per iteration is less than one milli second in three applications. The executable code sizes of a parallel version is twice larger than a serial in `rgbcmy`, `rgbhpg`, and `rgbyiq`. But, in `cjpeg` and `djpeg` including several parallel regions in unoptimized codes, there is a little executable code size increment. It implies that a compiler's code conversion from an OpenMP API into a subroutine-based form does not increase executable code sizes. In other three applications, the executable code sizes are increased by twice even though they have a few parallel loops. We think that this is due to multithreaded libraries.

Table 2 shows execution time and speedup in three different schedulings, `static`, `guided`, and `dynamic`. There are high parallelism (an ideal speedup is close to 2) in `rgbcmy`, `rgbhpg`, and `rgbyiq`. Two applications, `rgbcmy` and `rgbhpg`

Table 1. Basic property of `Consumer` benchmark. The text, data and bss represent `text`, `data` and `bss` section sizes of an executable serial code. In a parallel column, text, data and bss imply the ratio of increment of an executable parallel code respect to a serial. `cjpeg` is `JPEG compression`, `djpeg` is `JPEG decompression`, `rgbcmy` is `RGB to YIQ conversion`, `rgbyiq` is `RGB to YIQ`, and `rgbhpg` is `RGB high-pass grayscale filter`

Application	Serial execution time (sec)	Number of iteration	Serial (Kb)			Parallel(%)					
						Unoptimized			Optimized		
			text	data	bss	text	data	bss	text	data	bss
`cjpeg`	8.53	1000	46	238	518	121↑	5↑	12↑	116↑	5↑	12↑
`djpeg`	6.13	1000	54	31	773	107↑	40↑	8↑	100↑	36↑	8↑
`rgbcmy`	0.9	1000	20	230	6	276↑	5↑	1095↑	276↑	5↑	1095↑
`rgbhpg`	0.12	100	20	80	6	276↑	14↑	1095↑	272↑	14↑	1095↑
`rgbyiq`	0.18	100	20	230	6	275↑	5↑	1095↑	275↑	5↑	1095↑

suffer from high instruction overhead[1]. When we used two processors in an unoptimized code, we can improve speedup only in two applications, `rgbhpg` and `rgbyiq`. For performance optimization, we removed OpenMP directives in parallel regions having poor performance. We could get speedup in two applications, `rgbhpg` and `rgbyiq`.

Table 2. Execution time and speedup of `Consumer` benchmark. The ratio of a parallel section implies the ratio of execution time in parallel regions to total execution. The ideal speedup is derived by Amdahl's Law. The `static`, `guided` and `dynamic` schedulings are used in an OpenMP `for` directive. # represents the number of OpenMP parallel regions

Application	Ratio of parallel section(%)	Ideal speedup (P=2)	Unoptimized speedup			Optimized speedup						
							`static`		`guided`		`dynamic`	
			#	1P	2P	#	1P	2P	1P	2P	1P	2P
`cjpeg`	48.42	1.32	7	0.87	0.29	0	1	1	1	1	1	1
`djpeg`	12.72	1.07	9	0.89	0.65	0	1	1	1	1	1	1
`rgbcmy`	92.22	1.85	1	0.49	0.86	1	0.49	0.872	0.50	0.93	0.5	0.07
`rgbhpg`	100	2	3	0.70	1.2	1	0.70	1.5	0.70	1.33	0.70	1.33
`rgbyiq`	88.89	1.8	1	0.95	1.63	1	0.94	1.63	0.95	1.64	0.95	0.15

There are two reasons in low speedup. One is an instruction overhead in parallel code, as we mentioned above. The other is a load imbalance. For example, an unoptimized parallel version of `cjpeg` incurs high load imbalance by 63% of total parallel time on thread 0 and 33% on thread 1. Similarly `djpeg` includes 59% on thread 0 and 20% load imbalance on thread 1. As shown in Table 2, we

[1] We assume that the thread management overhead in 1 processor run is negligible.

could not improve the performance even though we applied different scheduling techniques. The speedup is lowest in dynamic scheduling. We conclude that small parallel regions incur high load imbalance because a child thread finishes his task as soon as a master thread assigns a new task to his team. Also we think that the small speedup gap on different scheduling is a noise, since the execution time per iteration is so small.

Telecommunication. Tables 3 and 4 show the results from our experiment in the `Telecommunication` benchmark. `Telecommunication` benchmark consists of five applications that use different data set (It is distinguished by "_" in the table). The execution time per iteration is small in all applications. When we build an executable code with multithreaded libraries, the code size is increased by more than twice. A `bss` section size in a parallel code is blown up, which is about 11 times larger than a serial.

As shown in Table 4 we found a few parallel regions in `autcor`, `fft`, and `viterb`. But we could achieve a speedup only in `autocor`, which is a little far below an ideal speedup 2. The optimized parallel version of `autocor` incurs high load imbalance due to tiny parallel regions. The speedup does not change depending on schedulings. The unoptimized parallel version of `viterb` incurs

Table 3. Basic property of `Telecommunication` benchmark. The text, data and bss represent `text`, `data` and `bss` section sizes of an executable serial code. In a parallel column, text, data and bss imply the ratio of increment of an executable parallel code respect to a serial. `autcor00data` is `autocorrelation`, `conven00data` is `convolution encoder`, `fbital00data` is `bit allocation`, and `fft00data` is `FFT`

Application	Serial execution time (sec)	Number of iteration	Serial (Kb)			Parallel(%)					
						Unoptimized			Optimized		
			text	data	bss	text	data	bss	text	data	bss
autcor00data_1	0.01	5000	21	5	5	263↑	211↑	1120↑	259↑	207↑	1120↑
autcor00data_2	0.29	5000	21	7	5	263↑	153↑	1120↑	263↑	153↑	1120↑
autcor00data_3	0.27	5000	21	7	5	263↑	173↑	1120↑	263↑	173↑	1120↑
conven00data_1	0.17	5000	21	7	5	257↑	162↑	1120↑	257↑	162↑	1120↑
conven00data_2	0.13	5000	21	7	5	257↑	162↑	1120↑	257↑	162↑	1120↑
conven00data_3	0.11	5000	21	7	5	257↑	162↑	1120↑	257↑	162↑	1120↑
fbital00data_2	0.64	5000	21	7	5	258↑	150↑	1120↑	258↑	150↑	1120↑
fbital00data_3	0.05	5000	21	7	5	255↑	171↑	1120↑	255↑	171↑	1120↑
fbital00data_6	0.34	5000	21	7	5	258↑	162↑	1120↑	258↑	162↑	1120↑
fft00data_1	0.02	1000	21	12	5	252↑	98↑	1121↑	249↑	96↑	1120↑
fft00data_2	0.02	1000	21	12	5	252↑	98↑	1121↑	249↑	96↑	1120↑
fft00data_3	0.02	1000	21	9	5	252↑	121↑	1121↑	249↑	119↑	1120↑
viterb00data_1	0.34	3000	23	6	5	235↑	190↑	1177↑	230↑	185↑	1177↑
viterb00data_2	0.31	3000	23	6	8	235↑	190↑	715↑	230↑	185↑	715↑
viterb00data_3	0.3	3000	23	6	8	235↑	190↑	715↑	230↑	185↑	715↑
viterb00data_4	0.21	3000	23	6	9	235↑	190↑	713↑	230↑	185↑	713↑

Table 4. Execution time and speedup of `Telecommunication` benchmark. The ratio of a parallel section implies the ratio of execution time in parallel regions to total execution. The ideal speedup is derived by Amdahl's Law. The `static`, `guided` and `dynamic` schedulings are used in an OpenMP `for` directive. # represents the number of OpenMP parallel regions

Application	Ratio of parallel section(%)	Ideal speedup (P=2)	Unoptimized speedup			Optimized speedup						
							`static`		`guided`		`dynamic`	
			#	1P	2P	#	1P	2P	1P	2P	1P	2P
autcor00data_1	0	1	1	1	0.11	0	1	1	1	1	1	1
autcor00data_2	100	2	1	0.97	1.32	1	0.97	1.53	0.97	1.52	0.97	1.45
autcor00data_3	100	2	1	1	1.56	1	1	1.54	0.97	1.47	0.97	1.33
conven00data_1	0	1	0	1	1	0	1	1	1	1	1	1
conven00data_2	0	1	0	1	1	0	1	1	1	1	1	1
conven00data_3	0	1	0	1	1	0	1	1	1	1	1	1
fbital00data_2	0	1	0	1	1	0	1	1	1	1	1	1
fbital00data_3	0	1	0	1	1	0	1	1	1	1	1	1
fbital00data_6	0	1	0	1	1	0	1	1	1	1	1	1
fft00data_1	0	1	1	1	0.5	0	1	1	1	1	1	1
fft00data_2	50	1.33	1	1	0.67	0	1	1	1	1	1	1
fft00data_3	50	1.33	1	1	0.5	0	1	1	1	1	1	1
viterb00data_1	57.87	1.41	2	0.75	0.11	0	1	1	1	1	1	1
viterb00data_2	56.57	1.39	2	0.76	0.1	0	1	1	1	1	1	1
viterb00data_3	59.1	1.42	2	0.75	0.11	0	1	1	1	1	1	1
viterb00data_4	55.75	1.39	2	0.75	0.1	0	1	1	1	1	1	1

high instruction overhead, and its performance is severely degraded in parallel execution.

Networking. Table 5 shows the basic property of applications in this category. In `Networking` benchmark, we could not find any parallel region in all applications, as shown in Table 6. But when we build executable codes with multithreaded libraries, the code size increases by about twice.

Figure 1 shows a typical example of applications in this category. Most applications uses a linked list as a basic data structure, which is processed sequentially.

Office Automation. In `Office automation` category, as shown in Table 8, we could find parallel regions in four applications out of five, `bezier01fixed`, `bezier01float`, `rotate` and `text`. `bezier01fixed` and `bezier01float` are the same algorithm, and only difference is a processed data type. The application `bezier01float` incurs the largest instruction overhead among the applications in this category. We could achieve speedup only in `bezier01fixed`.

The compiler's code conversion from OpenMP API to a subroutine-based form does not contribute an increment of an executable code size, as shown in

Table 5. Basic property of `Networking` benchmark. The text, data and bss represent `text`, `data` and `bss` section sizes of an executable serial code. In a parallel column, text, data and bss imply the ratio of increment of an executable parallel code respect to a serial. `ospf` is `Dijkstra's shortest-path algorithm`, `pktflow` is `packet flow` and `routelookup` is `route Lookup`

Application	Serial execution time (sec)	Number of iteration	Serial (Kb)			Parallel(%)					
						Unoptimized			Optimized		
			text	data	bss	text	data	bss	text	data	bss
`ospf`	0.01	400	24	6	5	223↑	191↑	1114↑	223↑	191↑	1114↑
`pktflowb512k`	0.01	100	21	10	5	254↑	114↑	1107↑	254↑	114↑	1107↑
`pktflowb1m`	0.01	100	21	10	5	254↑	114↑	1107↑	254↑	114↑	1107↑
`pktflowb2m`	0.01	100	21	10	5	254↑	114↑	1107↑	254↑	114↑	1107↑
`pktflowb4m`	0.04	100	21	10	5	254↑	114↑	1107↑	254↑	114↑	1107↑
`routelookup`	0.02	100	22	14	6	243↑	81↑	1071↑	243↑	81↑	1071↑

Table 6. Execution time and speedup of `Network` benchmark. The ratio of a parallel section implies the ratio of execution time in parallel regions to total execution. The ideal speedup is derived by Amdahl's Law. The `static`, `guided` and `dynamic` schedulings are used in an OpenMP `for` directive. # represents the number of OpenMP parallel regions

Application	Ratio of parallel section(%)	Ideal speedup (P=2)	Unoptimized speedup			Optimized speedup						
							`static`		`guided`		`dynamic`	
			#	1P	2P	#	1P	2P	1P	2P	1P	2P
`ospf`	0	1	0	1	1	0	1	1	1	1	1	1
`pktflowb512k`	0	1	0	1	1	0	1	1	1	1	1	1
`pitiful1m`	0	1	0	1	1	0	1	1	1	1	1	1
`pktflowb2m`	0	1	0	1	1	0	1	1	1	1	1	1
`pktflowb4m`	0	1	0	1	1	0	1	1	1	1	1	1
`routelookup`	0	1	0	1	1	0	1	1	1	1	1	1

Table 7. The executable code size of parallel versions is increased because of multithreaded libraries.

Automotive and Industrial. In `Automotive & Industrial` benchmark, we could find parallel regions in five applications, `aifftr`, `aifirf`, `bitmnp`, `idctrn` and `matrix` out of 16 application, and it is shown in Table 10. The property of this category is that execution time is very short, as shown in Table 9. Also the applications include many tiny parallel regions. For example, we could find 28 parallel regions in `idctrn`, but we could not find effective parallel regions due to their short execution time. And all parallelized applications introduce high load imbalance. An executable code size of parallel versions is increased because of multithreaded libraries.

```
prev_node = NULL;
curr_node = (*front);
for ( ; ; ) {
   if ( tnode->dist > curr_node->dist) {
      if ( curr_node->next != NULL) {
         prev_node = curr_node; /* remember previous node */
         curr_node = curr_node->next; /* go to the next node */
      }
      else {
         ......
         curr_node->next = tnode;
         tnode->next = NULL;
         break;
      }
   }
   else {
      ......
      if(prev_node != NULL) {
         /* link into the middle, or end of the list */
         tnode->next = curr_node;
         prev_node->next = tnode;
         break;
      }
      ......
   }
}
```

Fig. 1. A typical code pattern in `Networking` benchmark. Data are sequentially processed, and we could not find any parallel region in this category

Table 7. Basic property of `Office automation` benchmark. The text, data and bss represent `text`, `data` and `bss` section sizes of an executable serial code. In a parallel column, text, data and bss imply the ratio of increment of an executable parallel code respect to a serial. `bezier01fixed` and `bezier01float` are `Bezier curve interpolation` in fixed point and float point. `dither` is `Floyd-Stein grayscale dithering` and `text` is `text parsing` and `rotate` is `bitmap rotation`

Application	Serial execution time (sec)	Number of iteration	Serial (Kb)			Parallel(%)					
						Unoptimized			Optimized		
			text	data	bss	text	data	bss	text	data	bss
`bezier01fixed`	0.66	1000	20	24	5	274↑	48↑	1114↑	274↑	48↑	1114↑
`bezier01float`	0.3	1000	20	43	5	271↑	27↑	1114↑	267↑	26↑	1114↑
`dither`	1.58	1000	20	69	5	271↑	16↑	1114↑	271↑	16↑	1114↑
`rotate`	0.33	1000	21	18	18	260↑	64↑	337↑	260↑	64↑	337↑
`text`	0.61	1000	21	23	45	256↑	50↑	134↑	252↑	49↑	134↑

4 Compilation Issue

In this section, we discuss restrictions in OpenMP APIs and present useful techniques found when manually parallelizing embedded applications. Figure 2 shows the typical code patterns in our tested applications. Inside `for` loop, an induction variable (`col`) is declared as an unsigned integer, and data are sequentially accessed by a post-increment pointer arithmetic.

Table 8. Execution time and speedup of `Office automation` benchmark. The ratio of a parallel section implies the ratio of execution time in parallel regions to total execution. The ideal speedup is derived by Amdahl's Law. The `static`, `guided` and `dynamic` schedulings are used in an OpenMP `for` directive. # represents the number of OpenMP parallel regions

Application	Ratio of parallel section(%)	Ideal speedup (P=2)	Unoptimized speedup			Optimized speedup						
							static		guided		dynamic	
			#	1P	2P	#	1P	2P	1P	2P	1P	2P
bezier01fixed	100	2	1	0.92	1.74	1	0.92	1.74	0.92	1.69	0.92	1.38
bezier01float	100	2	1	0.77	0.59	0	1	1	1	1	1	1
dither	0	1	0	1	1	0	1	1	1	1	1	1
rotate	100	2	1	0.83	0.93	1	0.84	0.93	0.84	0.72	0.84	0.49
text	0	1	1	1	1	0	1	1	1	1	1	1

Table 9. Basic property of `Automotive & Industrial` benchmark. The text, data and bss represent `text`, `data` and `bss` section sizes of an executable serial code. In a parallel column, text, data and bss imply the ratio of increment of an executable parallel code respect to a serial. `a2time` is `Angle-To-Time conversion`, `aifftr` is `Fast Fourier Transform`, `aifirf` is `Finite impulse response filter`, `aiifft` is `aiifft` is `Inverse Fast Fourier Transform`, `basefp` is `Basic floating-point`, `bitmnp` is `Bit manipulation`, `cacheb` is `Cache buster`, `canrdr` is `Response to remote request`, `idctrn` is `Inverse Discrete Cosine Transform`, `iirflt` is `Low-Pass Filter and DSP functions`, `matrix` is `matrix math`, `pntrch` is `pointer chasing`, `puwmod` is `pulse-width modulation`, `rspeed` is `road speed calculation`, `tblook` is `table lookup` and `ttsprk` is `Tooth-To-Spark`

Application	Serial execution time (sec)	Number of iteration	Serial (Kb)			Parallel(%)					
						Unoptimized			Optimized		
			text	data	bss	text	data	bss	text	data	bss
a2time	0.02	50000	22	7	6	242↑	158↑	1102 ↑	242↑	158↑	1102↑
aifftr	0.07	200	23	21	32	243↑	55↑	192↑	231↑	52↑	192↑
aifirf	0.01	10000	21	10	5	259↑	118↑	1121↑	251↑	115↑	1120↑
aiifft	0.08	200	23	21	32	236↑	52↑	192↑	236↑	52↑	192↑
basefp	0.01	10000	21	21	6	257↑	53↑	1060↑	257↑	53↑	1060↑
bitmnp	0.06	3000	23	6	7	234↑	178↑	873↑	228↑	173↑	873↑
cacheb	0.01	100000	22	6	6	248↑	189↑	1071↑	248↑	189↑	1071↑
canrdr	0.01	200000	22	11	5	242↑	102↑	1120↑	242↑	102↑	1120↑
idctrn	0.01	1500	25	13	7	244↑	92↑	851↑	218↑	83↑	850↑
iirflt	0.02	10000	24	10	5	223↑	117↑	1107↑	223↑	117↑	1107↑
matrix	0.01	110	26	32	6	223↑	39↑	1090↑	204↑	35↑	1089↑
pntrch	0.01	2500	21	10	5	255↑	107↑	1107↑	255↑	107↑	1107↑
puwmod	0.01	100000	22	15	6	246↑	77↑	1101↑	246↑	77↑	1101↑
rspeed	0.01	100000	20	7	5	264↑	158↑	1114↑	264↑	158↑	1114↑
tblook	0.01	100	21	17	5	252↑	65↑	1114↑	252↑	65↑	1114↑
ttsprk	0.02	10000	25	54	6	217↑	21↑	1048↑	217↑	21↑	1048↑

Table 10. Execution time and speedup of `Automotive & Industrial` benchmark. The ratio of a parallel section implies the ratio of execution time in parallel regions to total execution. The ideal speedup is derived by Amdahl's Law. The `static`, `guided` and `dynamic` schedulings are used in an OpenMP `for` directive. # represents the number of OpenMP parallel regions

Application	Ratio of parallel section(%)	Ideal speedup (P=2)	Unoptimized speedup			Optimized speedup						
							static		guided		dynamic	
			#	1P	2P	#	1P	2P	1P	2P	1P	2P
a2time	0	1	0	1	1	0	1	1	1	1	1	1
aifftr	14.28	1.07	10	0.875	0.64	0	1	1	1	1	1	1
aifirf	77.78	1.63	6	0.9	0.21	0	1	1	1	1	1	1
aiifft	0	1	0	1	1	0	1	1	1	1	1	1
basefp	0	1	0	1	1	0	1	1	1	1	1	1
bitmnp	100	2	3	0.86	0.3	0	1	1	1	1	1	1
cacheb	0	1	0	1	1	0	1	1	1	1	1	1
canrdr	0	1	0	1	1	0	1	1	1	1	1	1
idctrn	66.67	1.5	28	0.67	0.125	0	1	1	1	1	1	1
iirflt	0	1	0	1	1	0	1	1	1	1	1	1
matrix	100	2	15	0.5	0.08	0	1	1	1	1	1	1
pntrch	0	1	0	1	1	0	1	1	1	1	1	1
purmod	0	1	0	1	1	0	1	1	1	1	1	1
rspeed	0	1	0	1	1	0	1	1	1	1	1	1
tblook	0	1	0	1	1	0	1	1	1	1	1	1
ttsprk	0	1	0	1	1	0	1	1	1	1	1	1

```
  JDIMENSION col;
  /* copy these pointers into registers if possible */
  ......
  inptr0 = input_buf[0][in_row_group_ctr];
  ......
  outptr = output_buf[0];
  /* Loop for each par of output pixels */
#pragma omp parallel for private(toutptr,cb,cr,......)
  for (col = 0 ; col < cinfo->output_width >> 1 ; col++){
    /* Do the chroma part of the calculation */
    cb = GETJSAMPLE(*inptr1++);
    ......
    /* Fetch 2 Y values and emit 2 pixels */
    ......
    outptr[RGB_RED] = range_limit[y + card];
    ......
    outptr += RGB_PIXELSIZE;
    ......
  }
```

Fig. 2. Sample codes from EEMBC benchmarks. The code shows typical patterns in our studied embedded applications. An induction variable is declared as an unsigned integer, and data are sequentially accessed by a post-increment pointer arithmetic

```
  JDIMENSION col;
  /* copy these pointers into registers if possible */
  ......
  outptr = output_buf[0];
  /* Loop for each par of output pixels */
#pragma omp parallel for private(toutptr,cb,cr,......)
  for (col = 0 ; col < cinfo->output_width >> 1 ; col++){
    toutptr = outptr * RGB_PIXELSIZE * col;
    /* Do the chroma part of the calculation */
    cb = GETJSAMPLE(*(inptr1+col));
    ......
    /* Fetch 2 Y values and emit 2 pixels */
    ......
    toutptr[RGB_RED] = rangle_limit[y + card];
    ......
  }
```

Fig. 3. A parallel version of Figure 2 by an induction variable substitution and a privatization

OpenMP Specification. According to OpenMP C/C++ 2.0 specification [10], the `for` directive places restriction on the structure of the corresponding `for` loop that must have canonical shape. Also, inside `for` loop, an induction variable should be a signed integer.

We found from EEMBC benchmarks that most induction variables in `for` loops are declared as an unsigned integer type, and one of the examples is shown in Figure 2. It is popular that in embedded applications unsigned integers are used instead of signed because of data characteristics. When we tested with Intel Compiler 7.0, the compiler generates errors because of this OpenMP rule and fails to generate parallel codes from manually inserted OpenMP codes.

We could parallelize many loops when redeclaring an induction variable as a signed integer instead of an unsigned with execution validation. Even though a compiler's or a user's redeclaration is trivial for parallelization, we need to improve the OpenMP specification for completeness.

Parallelizing Compilers. The EEMBC benchmarks are compute-intensive codes, and includes many small parallel loops. When we applied a parallelizing compiler (Intel Compiler 7.0) to identify parallel sections, it failed to find them due to a simple pointer arithmetic, i.e, a post increment (`inptr1++` in Figure 2). It is well known that related dependences can be eliminated by an induction variable substitution [8]. Similarly, a dependence on a variable `outptr` can be broken by the same technique and a privatization [9]. The result code is shown in Figure 3.

There are still many existing technical challenges in compiler's automatic parallelization of C applications [3,4]. Since embedded applications do not include complex pointer manipulation, the applications can be parallelized using the currently available these minimal techniques by automatic parallelizing compilers.

5 Conclusion

In this paper, we applied OpenMP API to non-traditional benchmarks, i.e. embedded applications. From experiment, we discussed three important issues. First, we found that embedded applications have limited thread-level parallelism in execution due to tiny parallel regions, and we showed that the executable codes sizes of OpenMP-inserted parallel applications is much larger than a serial due to multithreaded libraries. The executable code size is critical to embedded systems because of their limited size of memory systems. Second, we discussed a simple, but a critical OpenMP API issue in `for` directive, which can be easily found in the embedded applications. We need to improve OpenMP specifications for completeness. Finally, we showed that an induction variable substitution and a privatization can be applied to a pointer arithmetic to remove dependences in many places of embedded codes.

References

1. Silicon Graphics Inc., `http://www.sgi.com/origin/2000/index.html`. *Origin 2000*.
2. Sun Microsystems Inc., Mountain View, CA, `http://www.sun.com/servers/enterprise/e4000/index.html`. *Sun Enterprise 4000*.
3. William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.
4. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, pages 84–89, December 1996.
5. OpenMP Forum, `http://www.openmp.org/`. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, October 1997.
6. MPI Documents. `http://www.mpi-forum.org/docs/docs.html`.
7. EEMBC (EDN Embedded Microprocessor Benchmark Consortium). `http://www.eembc.org/`.
8. William M. Pottenger. Induction variable substitution and reduction recognition in the polaris parallelizing compiler. Technical Report UIUCDCS-R-98-2072, 1998.
9. Peng Tu and David A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247–284, 2001.
10. OpenMP Architecture Board, `http://www.openmp.org/`. *OpenMP C and C++ Application Program Interface 2.0*, March 2002.

Parallelizing Parallel Rollout Algorithm for Solving Markov Decision Processes

Seon Wook Kim[1] and Hyeong Soo Chang[2]

[1] Advanced Computer Systems Laboratory
Department of Electronics and Computer Engineering
Korea University, Seoul, Korea
seon@korea.ac.kr
[2] Department of Computer Science and Engineering
Sogang University, Seoul, Korea
hschang@sogang.ac.kr

Abstract. *Parallel rollout* is a formal method of combining multiple heuristic policies available to a sequential decision maker in the framework of Markov Decision Processes (MDPs). The method improves the performances of all of the heuristic policies adapting to the different stochastic system trajectories. From an inherent multi-level parallelism in the method, in this paper we propose a parallelized version of parallel rollout algorithm, and evaluate its performance on a multi-class task scheduling problem by using OpenMP and MPI programming model. We analyze and compare the performance in two versions of parallelized codes, e.g., OpenMP and MPI on several execution environment. We show that the performance using OpenMP API is higher than MPI due to lower overhead in data synchronization across processors.

1 Introduction

The model of Markov Decision Processes (MDPs), a.k.a. stochastic dynamic programming, is widely used for solving sequential decision making problems that arise in various areas such as telecommunication networks, financial engineering, statistical physics, and etc. The decision maker perceives the world or the environment he belongs to at each decision time where the world changes probabilistically. Based on his description of the world or information of the perception or *state*, he makes a decision to react to the environment and obtains a certain reward/cost associated with his decision and the state of the world. How the world changes for the next decision of the decision maker depends on which decision has been made at which state by the decision maker now. The goal of the decision maker is simply to maximize/minimize the expected (discounted) rewards/costs sum over stochastic changes of the world over infinite horizon. Unfortunately, a stochastic control optimization problem formulated by an MDP experiences *the curse of dimensionality*, making it very impractical to apply the well-known exact algorithms, e.g., value iteration and policy iteration [1], to solve the MDP. There are numerous approximation-based schemes to get away

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 122–136, 2003.

with the dimensionality problem via various techniques, e.g., structural analysis, aggregation, sampling, feature extraction, learning, etc. (see, e.g., [2,1,3,4]).

The recently proposed algorithm called "parallel rollout" by the author et al. [5] breaks the curse of dimensionality in the other perspective. For this approach it is assumed that multiple heuristic policies are available to a decision maker such that for certain cases (stochastic changes of the world he perceives), a particular heuristic is "good" and for certain cases, another policy is good, and so forth. It is natural that the decision maker in this case wishes to combine these heuristics to make one policy out of those such that the policy adapts to the different cases automatically. Parallel rollout is a formal method of generating a policy given a set of heuristic policies and improves the performances of all of the heuristic policies. Furthermore, it can be implemented by Monte-Carlo simulation making the sampling complexity depend on the sampling horizon *linearly* rather than exponentially and *independent* of the state space size. Therefore, it is a good candidate method for solving MDPs in *on-line* sense in the context of "planning." As the name of parallel rollout signifies, parallel rollout contains an inherent parallelism that can be parallelized and implemented via multiple processors. Specifically, each action's utility measure is evaluated in parallel and for each action, the performance measure of each heuristic policy starting from the particular next (sampled) state from the current state is computed in parallel and this step is evaluated in parallel over a set of the (sampled) next states.

In this paper, first, we parallelize the (simulation-based) parallel rollout algorithm and evaluate its performance in terms of the speedup over the serial version of the parallel rollout algorithm on a multi-class task scheduling problem. Second, we use OpenMP and MPI parallel programming models to parallelize parallel rollout algorithm, and compare their performance. The performance in the OpenMP code is higher than the MPI code on the shared-memory system. We discuss the performance issues in these models with an example quoted from the proposed algorithm. Third, from the experiment, even if the parallel rollout algorithm has three levels of inherent parallelism and would get the ideal speedup in theory, we find that it incurs a high load imbalance and a synchronization overhead like in [4]. Also, there is no speedup on a network-based cluster due to synchronization overhead which dominates total execution time.

This paper is organized as follows. In Section 2, we formally describe what is an MDP for infinite horizon discounted cost criterion and we present parallel rollout. In Section 3, we study the parallelism that parallel rollout contains, and provide a pseudo-code for the parallelized version. We evaluate the serial and two versions of parallelized parallel rollout algorithms and show that the parallelized algorithm speeds up the computation time in Sections 4 and 5 on a multi-class task scheduling problem. We conclude this paper in Section 6.

2 Background

2.1 Markov Decision Processes

We briefly provide the formal model of Markov decision processes. For a substantial discussion, see [1]. An MDP M is 4-tuple (X, A, P, C), where X is a set of states in the system (the information that is available to a decision maker for the world he perceives) and $A(x)$ is the set of *admissible* actions at state $x \in X$. P is a state transition function that describes how the system evolves over time such that it is a mapping from $X \times A$ to a distribution over X. We denote a probability that a state $x \in X$ makes a transition to another state $y \in X$ by taking an action $a \in A$ at x as $P(y|x, a)$. C is a cost function that maps $X \times A$ to a non-negative real number. The state transition function P induces *the next state function* $f : X \times A \times [0, 1] \rightarrow X$ such that for a state x and an action $a \in A(x)$ taken at x and a random number generated from [0,1], the function f maps the particular next state with respect to x, a and the random number.

We define a policy or decision rule π be a mapping from X to A, and let Π be the set of all possible policies, and define *the value of following a policy* $\pi \in \Pi$ as $V^{\pi}(x) = E\left[\sum_{t=0}^{\infty} \gamma^t C(X_t, \pi(X_t))|X_0 = x\right], 0 < \gamma < 1, x \in X$, where the expectation is taken over all possible random system paths and which is a performance measure of the policy π, Define *the optimal value at* $x \in X$ as $V^*(x) = \min_{\pi \in \Pi} V^{\pi}(x), x \in X$. The goal is to find an optimal policy $\pi \in \Pi$ that achieves $V^*(x)$ for each $x \in X$. The discount factor is used to represent the degree of the emphasis on the immediate future performance. As the value of γ gets closer to 0, we emphasize the immediate future performance much more than the distant future performance and as the value of γ gets closer to 1, we emphasize the distance future performance equally to the immediate future. It is well known that $\forall\ x \in X$, $V^*(x) = \min_{a \in A(x)} \left\{C(x, a) + \gamma \sum_{y \in X} P(y|x, a)V^*(y)\right\}$ and $V^*(x), x \in X$ is unique and a policy π^* defined as $\pi^*(x) \in \arg\min_{a \in A(x)} \left\{C(x, a) + \gamma \sum_{y \in X} P(y|x, a)V^*(y)\right\}$, $, x \in X$ is an optimal policy [1].

As we mentioned before, there exists the two well-known algorithms to compute π^* or the optimal value function V^*, called value iteration and policy iteration [1]. However, applying the exact methods for solving MDPs is very difficult if the state and/or the action space are large because the time-complexities of the methods depend on the sizes of the state and the action spaces [6], which is true for many interesting problems, including our example multi-class task scheduling problem.

2.2 Parallel Rollout

Consider the case where a decision maker has a nonempty set $\Lambda \subseteq \Pi$ of available heuristic policies. It is naturally expected that he wishes to combine such heuristic policies dynamically generating a combined policy. In particular, if each policy is near-optimal for different system trajectories, the combined policy should adapt to the different system trajectories (stochastically), that is, it

should adaptively select one of the heuristic policies and improve the performances of all of the heuristic policies if each heuristic policy alone used by the decision maker.

Formally, the parallel rollout policy π^{pr} is defined such that for each $x \in X$,

$$\pi^{pr}(x) \in \arg\min_{a \in A(x)} \left\{ C(x,a) + \gamma \sum_{y \in X} P(y|x,a) \min_{\pi \in \Lambda} V^{\pi}(y) \right\} \quad (1)$$

It turns out that the policy π^{pr} improves all of the policies in Λ and adapts to the different system trajectories. See [5] for intuitive arguments and a formal proof. We refer the equation inside argmin as the *utility of taking an action* $a \in A(x)$ at x if it is evaluated with x and a.

When computing π^{pr} in practice, at each decision time at the current state, the decision maker uses an *on-line* simulation to predict the futures (via generating random numbers) and uses the information of the futures to evaluate each action's *utility* with respect to Equation (1). At each decision time, at the current state, the decision maker applies the simulation to obtain an action that achieves the minimum of Equation (1) and takes the action. In this manner, we apply the parallel rollout policy in the context of "planning," rather than invoking the parallel rollout policy computed in off-line manner. For the simulation-based implementation of parallel rollout defined as above, we use *the same random number sequence method*, which is well-known concept in the simulation literature [7]. Note that we only need to know the best action that achieves the minimum in Equation (1) but not the true utility values of each action $a \in A(x)$ for $x \in X$ to obtain the minimum, if our main interest is in obtaining a decision rule or a plan for the decision making process. Therefore, we use the same random number sequence method across the candidate actions to try to preserve only the relative ranking of the action utilities. It has been shown that this method also reduces the variances of the utility estimates obtained by the simulation across the candidate actions [8].

3 Parallelization of Parallel Rollout

Recall that V^{π}, $\pi \in \Pi$, is just the expected sum of costs obtained by the decision maker following the policy π. As we mentioned before, this can be estimated by a *sample mean* over a selected finite sampling-horizon by generating many random trajectories of the system or random number (in [0,1]) sequences of the length equal to the given sampling horizon, and then apply π to each generated trajectory, obtaining the cumulative (discounted) costs, and take an average of them. Because each policy in Λ can be evaluated *independently*, we can use the same set of random number sequences over these different policies in Λ, making a parallel implementation possible over the policies in Λ. Therefore, the term $\min_{\pi \in \Lambda} V^{\pi}(y)$ in the above equation can be evaluated in a coordinated manner once V^{π} is computed in parallel over the policies in Λ. We can easily see that if the number of processors is equal to $|\Lambda|$, the overall computation will be speeded

up by $|\Lambda|$. The minimization step requires a coordination over the computed value from each policy. After each processor finishes computing his part on the V^π-estimate, the coordinator needs to compute the minimal value among the computed V^π-estimates.

A possible next state from the current state x is determined stochastically with the state transition function P. We need to take an average of the values $\min_{\pi\in\Lambda} V^\pi(y)$ over all possible next states $y \in X$ with respect to the distribution given by P. We again estimate this with a sample mean by generating a set of random numbers in [0,1] and the set of randomly sampled next states is commonly used for each different action. In other words, for each sampled next state y, we need to obtain $\min_{\pi\in\Lambda} V^\pi(y)$ and the random trajectories we generated to estimate $\min_{\pi\in\Lambda} V^\pi(y)$ for a particular state y will be used again for another sampled next state y'. This provides another parallelism in parallel rollout. We can distribute the computation of the minimal value of the V^π estimates in Λ at each randomly sampled next state into multiple processors. If the number of processors is equal to the number of randomly sampled next states, the overall computation will be speeded up by the number of the sampled next states W_N. Finally, the random number sets generated to estimate a particular action's utility will be used for other actions. That is, each action will use the same random number sets. Therefore, we can distribute the tasks into multiple processors such that each processor computes an action's utility.

Overall, parallel rollout contains three-level parallelism. In theory, the computational complexity will be reduced by factor of $|\Lambda| \cdot W_N \cdot |A|$ by parallelizing parallel rollout. We present a pseudo-code for the parallelized parallel rollout policy with the same random number sequence method in Figure 1. When we parallelize the outermost parallel loop, e.g, over the action space, it should be noted that synchronization across actions (e.g. processors) is required at the end of the algorithm to identify an action with the highest utility. Also the best action needs to be broadcast to all processors. It incurs high synchronization overhead across MPI processes. This will be discussed in Section 5 in detail.

4 Evaluation on Multi-class Scheduling

To evaluate the parallelized parallel rollout policy, we consider the problem of scheduling randomly arriving tasks into a single server. We remark that the scheduling problem and the simulation set up for the problem is similar to the one in [5]. For the self-containedness, we describe the scheduling problem briefly.

Each task belongs to a predetermined finite set of classes $1, \cdots, m$ and each class is associated with a real number weight such that a cost of the class weight is incurred if the server does not serve a task in the class before its deadline. Every task takes one-unit time to serve. Tasks arrive into the queue within each time interval, and the server makes decisions at the boundary of each time step. Here we make further simplifying assumptions. We assume that at most one task can be generated per class per unit time and each class has the *same deadline* and there are no inter-dependencies between the tasks. Even under all

```
PARALLELIZED PARALLEL ROLLOUT:
  Inputs:
  H, W; /* sampling horizon and width for V^π-estimate */
  W_N; /* sampling width for the next states sampling */
  x; /* current state */
  C(), f(); /* cost and next state functions for the MDP */
  double r[W][H] /* sampled random number sequences, init each entry randomly in [0,1] */
  double Next[W_N] /* sampled random numbers for the next states, init each entry randomly in [0,1] */
  state x[H]; /* a vector of states for temporarily storing a state trajectory */
  double EstimateQ[]; /* cumulative estimate of the utility for action u, init to zero */
  double EstimateV[]; /* for each policy, the estimated value on current random trajectory */

  DO PARALLEL for each action u in A(x)
    DO PARALLEL for each I = 1 to W_N
      x[1] = f(x, u, Next[I]);
      for i = 1 to W do
        DO PARALLEL for each π in Λ
          EstimateV[π][I]=0;
          for t = 1 to H do
            u[t] = π(x[t]); EstimateV[π][I] += γ^t C(x[t], u[t]); x[t + 1] = f(x[t], u[t], r[i][t]);
          endfor
        ENDDO PARALLEL
      minV[I] = min_{π∈Λ} (EstimateV[π][I]/W); EstimateQ[u] += minV[I];
      endfor
    ENDDO PARALLEL
    EstimateQ[u] = C(x,u) + EstimateQ[u] / W_N;
  ENDDO PARALLEL
  take action arg min_{u∈A(x)} EstimateQ[u];
```

Fig. 1. Pseudo-code for parallelized parallel rollout using common random numbers. There are three-levels of parallelism in an action, a sampling width, and a policy, marked as `DO PARALLEL`

these simplifying assumptions, this on-line task scheduling has no known optimal policy for the weighted loss for a long finite time interval or infinite time interval.

When the server decides to process a particular task, there is always a bad future traffic that makes his current decision wrong. This is more particularly evident when the traffic is highly bursty. If we expect a heavy burst of important classes, the server better gives up serving relatively unimportant tasks even if the server can preserve the "throughput optimality" (maximizing the unweighted number of tasks served) to gain higher weighted throughput. That is, for this type of traffic pattern, serving simply the important class tasks as many as possible may well be a good policy. *Static Priority* (SP) is a policy where it serves the most important pending task, with breaking ties between the same class tasks by FIFO, making SP a near-optimal policy for those system paths. On the other hand, the server better serves unimportant tasks in an opposite situation. That is, if the average number of tasks generated over the unit time is small or the load of the queue is small (less than 1), serving the tasks to maximize just unweighted throughput is near-optimal. *Current Minloss* [9] (CM) is a policy whose performance is no worse than throughput optimal *Earliest Deadline First* (EDF) for any finite arrival sequence, where EDF serves the task to be expired soonest, with breaking a tie by serving the more important task. Thus, how well the decision has been made at each decision time depends on how likely such unfavorable/favorable traffic is to arrive at the queue in the future and a decision maker wishes to combine SP and CM into one scheduling policy such that the

policy selects between CM and SP automatically adapting to the stochastic future arrivals of the tasks. Note that the decision maker does not need to add EDF because CM is no worse than EDF.

4.1 MDP Formulation

In this subsection, we model the scheduling problem as an MDP. The traffic for each class i is modeled by Markov Modulated Bernoulli Process (MMBP) [10] where an MMBP is associated with a Markov chain where each state in a finite set S_i of the Markov chain is associated with a probability of generating a task at each time. Each class traffic generation starts at a particular state in S_i. After generating a task with a positive probability associated with the initial state, the state makes a transition to another state in the MMBP and generates a task with a probability associated with the new state and this process is repeated.

The state space X of this problem is $X = \Theta_1 \times \cdots \times \Theta_m \times \{0,1\}^{m \times d}$, where Θ_i is the probability distribution over the MMBP states in S_i for the class i (this distribution is updated from the task arrival information at each time by the Bayes update rule) and the last factor is a set of vectors where a vector represents current pending tasks in the buffer B_n at time n: if there exists a task T whose laxity (remaining time for the deadline to be expired) is $l_n(T)$, then the entry of the vector indexed with $K(T)$ and $l_n(T)$ is 1, where $l_n(T) \in \{0, ..., d\}$ and it signifies the remaining time until its deadline is expired and $K(T) \in \{1, \cdots, m\}$ is the class of the task T. The set of actions is $A = \{1, \cdots, m\}$, where action $a = i$ means that we serve the task with the smallest laxity in class i. We let the deadline of a particular task T be $d(T)$ and the task selected by a given scheduling policy π for a given buffer B as $\pi(B)$. The dynamics of the buffer over time is expressed as $B_{t+1} = B_t - \{T \in B_t : d(T) = t \text{ or } T = \pi_t(B_t)\} \cup \Omega_{t+1}$, where Ω_{t+1} is the set of tasks that have arrived to the server at time $t+1$.

The new buffer B_{t+1} at time $t+1$ is stochastically described by the random arrivals in Ω_{t+1} during the time interval $(t, t+1)$, where the random arrivals are generated by the given MMBPs. Therefore, the state transition function P is defined in the obvious manner representing underlying stochastic transitions in each of the class MMBPs, and the change in the buffer by adding new tasks in Ω_{t+1} generated by MMBPs as well as the expiry of unserved tasks and the removal of the task served by the action selected. The cost function C is defined such that $C(x, a)$ is the sum of the costs from lost tasks, i.e., the unserved tasks with the deadline of zero. For this MDP, we wish to find a scheduling policy that minimizes the expected weighted costs of unserved tasks over infinite horizon with a discount factor.

5 Simulation

5.1 Problem Setup

For the scheduling domain, it is unnatural to consider a discount factor much less than one for the performance measure of a scheduling policy because the performance in the distant future is equally important as in the immediate future. Therefore, we need to set the discount factor very close to one if we truly care about the technicalities. However, letting the discount factor one do not lose any practicality. For this reason, we used the discount factor of one through the simulation studies.

As with many problem domains (e.g., random propositional satisfiability problem), randomly selected problems from this scheduling domain are typically too easy. In scheduling, this problem manifests itself in the form of arrival patterns that are easily scheduled for virtually no loss (e.g., by EDF), and arrival patterns that are apparently impossible to schedule without heavy weighted loss (in both cases, it is typical that just blindly serving the highest class available in the buffer, e.g., by SP, performs as well as possible). Difficult scheduling problems are typified by arrival patterns that are close to being schedulable with no weighted loss, but that must experience some substantial weighted loss. We have conducted experiments by selecting MMBP models for the arrival distributions at random from a guided ad-hoc but reasonable single distribution over MMBPs.

All of the scheduling problems we consider involve several number of classes (8, 16, 32, 64, and 128) of tasks and we set the weights of the classes such that class i has a weight of w^{i-1}. By decreasing the parameter w in [0,1], we emphasize the disparity in importance between classes, making the scheduling problem more class-sensitive and by increasing the parameter w closer to 1, the relative importances between the classes get smaller. Note that for $w \approx 1$, CM must work near-optimal as CM maximizes the number of served tasks for any finite arrival sequences and for $w \approx 0$, SP is near-optimal as SP selects the tasks based on the priority only.

We select an MMBP model for each class, chosen from the same distribution. We selected the MMBP state space of size 3 arbitrarily and deliberately arrange the states in a directed cycle to ensure that there is interesting dynamic structure. Similarly, we select the self transition probability for each state uniformly in the interval [0.9, 1.0] and the arrival generation probability at each state is selected such that one state is "low traffic" (uniform in [0, 0.01]), one state is "medium traffic" (uniform in [0.2, 0.5]), and one state is "high traffic" (in [0.7, 1.0]). Finally, after randomly selecting the MMBPs for each of classes in a selected group from the distribution, we normalize the arrival generation probabilities for each class so that arrivals are (roughly) equally likely in high-cost, medium-cost, and low-cost and to make a randomly generated traffic from the MMBPs have overall arrivals at about one task per time unit to create a scheduling problem that is suitably saturated to be difficult. Even though we designed the scheduling problem domain in this way, note that a very broad range of arrival pattern MMBPs can be generated.

We selected two scheduling policies as the heuristic policies for the parallel rollout policy. As we discussed before, depending on the stochastic future traffic patterns, SP or CM is near-optimal. For this reason, we combine the two policies via parallel rollout. For the results that show the improvements of parallel rollout over SP, CM, and EDF, please refer [5]. But, due to irregular execution time of these two heuristic policies, the parallel rollout algorithm suffers from high load imbalance between processes/threads, and this issue will be discussed in the next subsection.

5.2 Measurement

For the simulation study of parallel rollout, we parallelized parallel rollout over only the action space by using either OpenMP or MPI, but not together in one simulation because of our limited experiment environment. That is, the computation of the utility measure of taking each action is distributed among the multiple processors. We believe that this setup is enough to convey the reader that parallelized parallel rollout improves the computational time over serial parallel rollout. We evaluated the serial parallel rollout policy with the parallelized parallel rollout policy with respect to five different traffics generated from five randomly selected traffic model parameter set.

For OpenMP experiment, we used two shared-memory machines: a SUN Enterprise 4000 having six 250MHz UltraSPARC processor and an Intel Xeon dual processor systems. On the SUN machine, we used SUN Forte 6.2 OpenMP compiler on Solaris 5.8, and `-xO3 -xtarget=ultra2 -xcache=16/32/1:1024/64/1` for code compilation. On the Intel machine, we used Intel C/C++ 7.0 compiler with `-O3` optimization on Redhat 7.1 Linux. For MPI experiment, we used the same SUN Enterprise as OpenMP experiment with shared-memory version of MPICH 1.2.4. Also, we evaluated our code on a 4 node cluster connected with Ethernet, and each has AMD 1GHz Athlon, 256MB main memory, Redhat Linux 7.1, and MPICH 1.2.3 libraries.

Table 1 shows ideal speedup of our experimental problem on our two experimental machines. The ideal speedup is driven by Amdahl's Law to use serial and parallel sections' execution time on a single processor system. The table shows that the parallelized algorithm has high-level parallelism and the ideal speedup on SUN Enterprise is a little higher than an Intel Xeon system. The serial sections of the code include many file read operations. We think that Solaris system has more efficient and fast methods to deal with these operations than Linux.

Parallelization Using OpenMP. In order to exploit parallelism over the action space in OpenMP, we used variable renaming and array expansion techniques from the serial version of the code.

Table 2 shows the speedup of parallelized parallel rollout algorithm using OpenMP on SUN Enterprise and Intel Xeon systems in five different classes. The speedup is defined as the ratio of execution time in a serial parallel rollout to that in its parallel version. The achieved maximum speedup on the SUN

Table 1. Ideal speedup of our problem on different classes and processors. The ideal speedup is driven by Amdahl's Law to use execution time of serial and parallel regions on a single processor system. The proposed algorithm has high-level parallelism

Number of classes	Sun Enterprise						Intel Xeon	
	1 Proc	2 Procs	3 Procs	4 Procs	5 Procs	6 Procs	1 Proc	2 Procs
8	1.00	1.80	2.45	3.00	3.46	3.85	1.00	1.51
16	1.00	1.88	2.66	3.35	3.97	4.53	1.00	1.65
32	1.00	1.79	2.44	2.97	3.43	3.81	1.00	1.42
64	1.00	1.92	2.76	3.54	4.26	4.94	1.00	1.73
128	1.00	1.92	2.78	3.57	4.31	5.00	1.00	1.79

system is 3.47 using six processors, which is far below an ideal speedup. On the Intel system, we achieved a maximum speedup 2.18 using two processors which is higher than an ideal speedup. Overall, the speedup is linearly proportional to the number of processors except for 8 and 32 classes.

Table 2. Speedup of parallelized parallel rollout using OpenMP parallel programming on SUN Enterprise and Intel Xeon systems. The speedup is linearly proportional to the number of processors. We achieved ideal speedup in some cases

Number of classes	Sun Enterprise						Intel Xeon	
	1 Proc	2 Procs	3 Procs	4 Procs	5 Procs	6 Procs	1 Proc	2 Procs
8	1.33	2.00	2.00	2.40	2.40	2.40	1.01	1.23
16	0.94	1.57	1.94	2.36	2.54	2.75	1.13	1.65
32	1.08	1.30	1.53	1.53	1.63	1.63	0.97	1.15
64	1.20	1.88	2.42	3.00	3.22	3.47	1.18	1.84
128	0.91	1.38	1.89	2.30	2.61	2.98	1.29	2.18

There are two findings to explain the performance gap between ideal and achieved speedup, especially in 8 and 32 classes. One of the findings is a load imbalance in our algorithm. In order to get data in Table 2, we used dynamic scheduling in an OpenMP parallelized loop. Table 3 shows the speedup when we used static scheduling. We see the performance difference between two schedulings, and the performance of static scheduling is worse than dynamic. Also, Table 4 shows the load imbalance between two threads on Intel Xeon processors, which is the ratio of load imbalance to the total execution in parallel regions. We used Intel compiler's OpenMP performance profiler (`-openmp_profile`) to collect the data. On average the load imbalance is about 20% of the parallel execution time, and it becomes worse at 8 and 32 classes. As we mentioned earlier, we selected two scheduling policies, SP and CM for the parallel rollout policy.

These two heuristics' execution times are irregular depending on the number of classes, and which incurs large load imbalance. We predict that this problem becomes worse on the SUN system due to the overhead to support large number of threads. As a result, the achieved speedup on the SUN system is far below the ideal speedup.

Table 3. Speedup of statically scheduled parallelized parallel rollout using OpenMP parallel programming on SUN Enterprise and Intel Xeon systems. The speedup is lower than in Table 2

Number of classes	Sun Enterprise						Intel Xeon	
	1 Proc	2 Procs	3 Procs	4 Procs	5 Procs	6 Procs	1 Proc	2 Procs
8	1.33	1.50	1.71	2.00	2.00	2.00	1.07	1.14
16	0.94	1.38	1.50	1.83	1.74	1.94	1.17	1.49
32	1.04	1.25	1.39	1.39	1.47	1.47	0.93	0.96
64	1.20	1.64	2.03	2.13	2.36	2.61	1.20	1.60
128	0.91	1.34	1.65	1.93	1.96	2.06	1.37	1.91

Table 4. Load imbalance (%) between threads on Intel Xeon dual processors using OpenMP. The load imbalance is defined as a ratio of load imbalance to the total execution in parallel regions

Number of classes	Intel Xeon	
	Thread 0	Thread 1
8	25.70	35.91
16	19.57	18.92
32	49.60	34.57
64	10.39	25.71
128	10.15	17.34

Another reason for the performance gap is explained by data synchronization between serial and parallel regions, which introduces higher cache coherence overhead on a large shared-memory multiprocessor system, e.g., the SUN system. For example, in Figure 2, `Vminavg` is updated on child threads (in a parallel region), and it is read by a master thread in a serial region. Similarly, `Qstate` is updated on a master thread, and it is read by child threads. On the Intel Xeon system, this overhead seems to be negligible because the system has only 2 processors.

Parallelization Using MPI. Table 5 shows the speedup of the parallelized code using MPI APIs on the SUN Enterprise and the cluster systems. The

```
   // parallel rollout
#pragma omp parallel for private(pinum,policyeval,prio,slot,......) schedule(dynamic,1)
   for(class = 1; class <= MAX_CLASS; class++) {
      ......
      for(i = 1; i <= MAX_CLASS; i++)
         ...... = Qstate[i];
      ......
      Vminavg[class] += .....
   }

   // find the best action
   maxQ = -99;
   for(class = 1; class <= MAX_CLASS; class++) {
       if(feasible[class] != -1) {
             Qvalue[class] = ...... + Vminavg[class];
             if(Qvalue[class] >= maxQ) {
                   whichclass = class; maxQ = Qvalue[class];
             }
       }
   }
   Qstate[whichclass] = ......
```

Fig. 2. Serial and OpenMP codes to identify the best action. The hardware provides fast data synchronization of variable `Vminavg` between processors across the first and the second loops, and `Qstate` between a serial and a parallel regions. The second loop is not parallelized due to its small execution time

achieved speedup in MPI is less than that in OpenMP on the SUN Enterprise system, and there is no speedup on the cluster. The performance gap between in OpenMP and MPI codes is discussed next.

Table 5. Speedup of parallelized parallel rollout using MPI parallel programming on SUN Enterprise and 4-node AMD cluster systems. The speedup on SUN is proportionally linear to the number of processors. But we failed to get speedup on the cluster system

Number of classes	Sun Enterprise						AMD cluster		
	1 Proc	2 Procs	3 Procs	4 Procs	5 Procs	6 Procs	1 Proc	2 Procs	4 Procs
8	1.29	1.68	1.85	1.95	1.88	2.20	0.95	0.73	0.05
16	1.00	1.52	1.92	2.01	2.19	2.53	1.02	0.76	0.07
32	1.28	1.46	1.69	1.69	1.78	1.77	1.08	0.77	0.07
64	1.28	1.90	2.20	2.50	2.54	2.75	0.97	0.81	0.24
128	1.01	1.65	2.07	1.91	2.59	2.68	0.98	0.84	0.44

Comparison between OpenMP and MPI. As we mentioned earlier, the parallel algorithm needs synchronization to find the best action. The OpenMP codes are constructed as shown in Figure 2.

On the shared-memory multiprocessor system, OpenMP does not need explicit data synchronization across processors. The hardware cache-coherence pro-

tocol takes care of dependences between processors, i.e. between a serial and a parallel regions. The array elements `Vminavg` are gathered by a master thread and the master thread finds the best action. Similarly dependence on `Qstate` is resolved by the protocol at high speed. Since the data synchronization is done by hardware, its overhead is small compared with MPI methods.

In MPI, process-local best action is calculated per process. Then all local best actions are reduced by all processes to find a global best action. All processes need the best action to update variable `Qstate`, which is read inside the parallel loop in the parallel rollout, as shown in Figure 3. This `MPI_Allreduce` collective operation is spread over all execution, and incurs high overhead due to slow synchronization. This is shown in Figure 4 using a VAMPIR [11] MPI performance analysis tool. In the figure, the lines between processes show the collective communication between processes (all of them are `MPI_Allreduce` operations), and dark gray in process lines represents MPI execution overhead.

```
// parallel rollout
for(class = mypid; class <= MAX_CLASS; class+=rank) {
   ......
   for(i = 1; i <= MAX_CLASS; i++)
      ...... = Qstate[i];
   ......
   Vminavg[class] += .....
}

// find the best action
maxQ = -99;
for(class = mypid; class <= MAX_CLASS; class+=rank) {
    if(feasible[class] != -1) {
          Qvalue[class] = ...... + Vminavg[class];
          if(Qvalue[class] >= maxQ) {
                whichclass = class; maxQ = Qvalue[class];
          }
    }
}
in.value = maxQ; in.class = whichclass;
MPI_Allreduce(&in, &out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, MPI_COMM_WORLD);
whichclass = out.class;
Qstate[whichclass] = ......
```

Fig. 3. MPI version to identify the best action. MPI needs explicit collective operation to synchronize local best action

The sent and received total bytes in the collected operation per each simulation between processes is only 1K. But MPI execution, `MPI_Allreduce`, dominates overall execution, as shown in Table 6. Larger number of participant nodes incurs higher overhead, and finally dominates the total execution time. The synchronization overhead can be reduced on the SUN Enterprise experiment through the shared-memory execution model. Also the load imbalance between processes due to the used heuristic policies affects MPI operations. In 32 classes of Table 6, the MPI overhead in 2 processor run is higher than 4 because of uneven workload distribution.

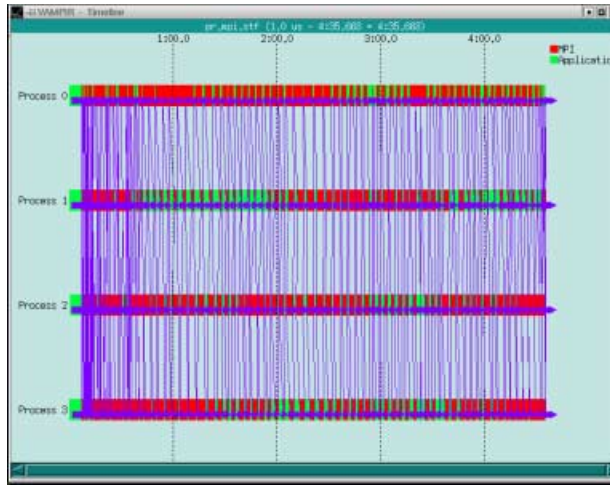


Fig. 4. Timeline display in 128 classes using 4 processes on the Ethernet connected cluster. In a legend, `MPI` (dark gray) means spent-time in MPI execution, and `Application` means execution excluding MPI operations. The lines between processes represent MPI communication between processes, which is `MPI_Allreduce` collective operation

Table 6. MPI overhead (%) on the cluster, which is the ratio (%) of MPI execution time to the total execution time. MPI operation dominates overall execution

Number of classes	AMD cluster		
	1 Proc	2 Procs	4 Procs
8	0.04	52.84	88.20
16	0.02	41.68	93.26
32	0.00	66.40	63.20
64	0.00	29.43	79.34
128	0.00	28.31	62.49

6 Conclusion

In this paper, we proposed a parallel algorithm of parallel rollout, which is a formal method of combining multiple heuristic policies available to a sequential decision maker for solving problems formulated in the framework of Markov Decision Processes (MDPs). We used OpenMP and MPI parallel programming to parallelize the algorithm, and evaluated their performances on several machine configurations for a multiclass scheduling problem. Overall, the speedup is proportionally linear to the number of processors on the shared-memory multi-

processor systems both in OpenMP and in MPI codes. The performance using OpenMP is higher than MPI due to lower data synchronization overhead across processors. We failed to get speedup on the network-based cluster system.

The parallel rollout algorithm has the three levels of inherent parallelism. But it needs a synchronization at the final phase to identify the best action or the action with the highest utility. In MPI the synchronization needs a MPI collective operation across all processes, and we found that it incurs a high overhead of the synchronization and the selected heuristic policies introduce a load imbalance over the action space. Note that even though the synchronization overhead is unavoidable, the load imbalance problem is problem specific. This high load imbalance results in high synchronization MPI operations.

To the best of our knowledge, the present work is the first parallelization and evaluation of on-line simulation-based solution scheme for solving MDPs, rather than off-line computation. Even though the parallelized parallel rollout cannot avoid a certain synchronization overhead, the overhead is independent of structural properties in problems themselves formulated by MDPs. Because of the inherent parallelism of parallel rollout, the parallel rollout does not cause complicated parallel implementation issues that arise in the previous works mentioned in the introduction of the present paper, especially in the perspective of the speed up of the parallel implementation.

References

1. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York, 1994.
2. D. P. Bertsekas and J. N. Tsitsiklis. Neuro dynamic programming. *Athena Scientific*, 1996.
3. R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, 2000.
4. A. Printista, M. Errecalde, and C. Montoya. A parallel implementation of Q-learning based on communication with cache. *Journal of Computer Science and Technology*, 1(6), 2002.
5. H. S. Chang, R. Givan, and E. K. P. Chong. Parallel rollout for on-line solution of partially observable markov decision processes. *Discrete Event Dynamic Systems (Revised)*, 2002.
6. M. Littman, T. Dean, and L. Kaelbling. On the complexity of solving markov decision problems. In *Proc. 11th Annual Conf. on Uncertainty in Artificial Intelligence*, pages 394–402, 1995.
7. A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis, 3rd Ed.* McGraw-Hill, New York, 2000.
8. D. P. Bertsekas. Differential training of rollout policies. In *Proc. 35th Allerton Conf. on Comm., Control, and Computing*, 1997.
9. R. Givan, E. K. P. Chong, and H. S. Chang. Scheduling multiclass packet streams to minimize weighted loss. *Queueing Systems*, 41:241–270, 2002.
10. W. Fischer and K. Meier-Hellstern. The markov-modulated poisson process (mmpp) cookbook. *Performance Evaluation*, 18:149–171, 1992.
11. Wolfgang E. Nagel, Alfred Arnold, Michael Weber, and Hans-Christian Hoppe. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, (1):69–80, January 1996.

DMPL: An OpenMP DLL Debugging Interface

James Cownie[1], John DelSignore, Jr.[1], Bronis R. de Supinski[2], and Karen Warren[2]

[1] Etnus, LLC, 24 Prime Parkway, Natick, Massachusetts 01760
{jcownie,jdelsign}@etnus.com
http://www.etnus.com/Products/TotalView/index.html
[2] Lawrence Livermore National Laboratory, P.O. Box 808, L-560
Livermore, California 94551-0808*
{bronis,kwarren}@llnl.gov
http://www.llnl.gov/icc/lc/DEG/TV.html

Abstract. OpenMP is a widely adopted standard for threading directives across compiler implementations. The standard is very successful since it provides application writers with a simple, portable programming model for introducing shared memory parallelism into their codes. However, the standards do not address key issues for supporting that programming model in development tools such as debuggers. In this paper, we present DMPL, an OpenMP debugger interface that can be implemented as a dynamically loaded library. DMPL is currently being considered by the OpenMP Tools Committee as a mechanism to bridge the development tool gap in the OpenMP standard.

1 Introduction

OpenMP is a widely used parallel programming model for shared-memory multiprocessor (SMP) architectures [1], [2]. The OpenMP organization initially focused on language design and run-time support. This focus has been successful - OpenMP now provides relative ease in writing codes that efficiently utilize SMP architectures.

While the initial focus has been successful, key areas for usability still need to be addressed. In particular, accurate information needed for debugging and analyzing performance of OpenMP applications can be difficult to obtain. Even when the information is provided, it is very difficult to present the information consistently with the semantics of the OpenMP programming model.

Recently, the OpenMP organization created a Tools Committee to address some of these difficulties. That committee is considering the POMP interface [3] as a standard method to instrument OpenMP applications and to gather performance data. However,

* This work was partially performed under the auspices of the U.S. Department of Energy by University of California LLN Laboratory under contract W-7405-Eng-48. UCRL-JC-151670.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 137–146, 2003.

the POMP interface does not address problems that arise when one attempts to debug an OpenMP code. In this document we propose DMPL (pronounced “dimple”), an interface for OpenMP debugger support that can be implemented as a dynamically loaded library (DLL). We propose that each compiler vendor implement this library and provide it along with the OpenMP run-time library. Debuggers would then dynamically load this library to obtain the information needed for users to debug OpenMP codes.

The rest of this paper first explores key background issues for debugging OpenMP application codes. We then discuss why a DLL-based OpenMP debugger interface is the right solution. Next, we present an initial proposal for the definition of DMPL. We conclude with a discussion of why it is inappropriate to add the required debugger support to a performance instrumentation API and remaining open issues for OpenMP debugger support.

2 Background

2.1 Multiple Compilers, User Levels

Debugging features required for OpenMP codes are similar to those required for sequential code. Users want to work with the "source-level" code and to plant breakpoints within OpenMP regions. They want to step and otherwise control execution in those regions and to examine variables within them. However, the underlying threading provided by the OpenMP compiler and run-time complicates these goals. Should the debugger present outlined routines that the compiler creates so that OpenMP regions can be executed by multiple threads? Does the user expect the same view for shared, private, and thread private variables?

Some users understand how the OpenMP directives accomplish the desired parallelism; other users don’t care about such details. Ideally, the debugger presents a serial code to the latter users while supporting a more detailed view that sophisticated users may desire. More realistically and practically, the debugger allows the user to see thread private variables and manipulate the threads individually. Regardless, a portable debugger must understand the output from various compilers (e.g., HP/Compaq, IBM, Intel-Guide, SGI and Sun) and present it to the user in a way that avoids involving the user in the underlying transformations.

2.2 Compiler Transformations

The OpenMP language specifications allow a variety of implementations. Portable debuggers must directly solve some possible implementation differences. However, a standard interface can assist the debugger in hiding many implementation details.

Depending on the compiler, the user's code may first be preprocessed and the resulting code actually compiled. The debugger deals with the preprocessed code. Since the user wants to debug the original code, the preprocessor must insert line number direc-

tives. The debugger must interpret the directives. Although line number directives are already standardized, some OpenMP implementations that use preprocessing have not always included them.

The compiler uses threads to achieve the necessary parallelism. Various threading packages can be used: pthreads, sproc, and other proprietary threading packages. The debugger user must be able to asynchronously control these threads. This goal means that the underlying thread package must include support to synchronize thread execution, to single step threads in lockstep and for thread-specific breakpoints. Although the debugger must handle the underlying thread package, OpenMP implementers can help ensure that the package includes needed support for asynchronous thread control.

To achieve the desired parallelism, the compiler constructs outlined routines that the master thread calls and the worker threads execute. There may be multiple outlined routines for a single worksharing construct. Calls to the outlined routines become part of the stack traces. The stack frames from the routines become part of the calling stack frame. Ultimately, what looks like straight-line code to the user isn't. The debugger needs a standard method to recognize outlined routines and to associate them with worksharing constructs. Similarly, the debugger needs a standard mechanism to identify run-time library routines, for which the names vary for every compiler. Given these mechanisms, the debugger can then eliminate the calls from stack frames and support the appearance of straight-line code, if the user desires it.

Each compiler mangles the source code names differently when it makes up names for the outlined routines and their variables. The debugger must demangle a mangled name in order to present the user with a recognizable name. Further, the debugger may need to determine language meaning for the mangled name, such as whether a variable is shared or private. The debugger also must use the proper addressing mode for the variable, which can vary substantially between compilers, particularly for thread private data. The OpenMP implementation must provide the debugger with a standard mechanism for name demangling and this related information.

3 DMPL Objective

We propose DMPL, a dynamically loaded library interface that provides the debugger with information needed to support OpenMP applications. The TotalView parallel debugger [4] already uses this paradigm for debugging pthreads, MPI message queues [5], and UPC [6]. Similarly to run-time libraries, OpenMP compiler vendors would provide a DMPL library. The debugger, which already contains information about the run-time parallel environment, will load the DLL.

This paper defines the DMPL interface, which separates the OpenMP implementation from the debugger. This interface must provide a debugger with all the information that it needs to present the user with a view of his code both in terms of the original source and with additional thread details. The routines supplied by the vendor in the DLL can be linked dynamically in the debugger allowing callbacks to the debugger. The DLL is an OpenMP implementation-specific product and its implementation details are left to the OpenMP implementer.

4 DMPL Interface

The proposed DLL is a two-way interface between the debugger and the DLL itself. When the debugger needs to display the value or address of a thread private object, it will make a call to a DLL function to extract the absolute address of the object. The DLL itself will make calls to the debugger to access information about the target process or thread. The DLL must not use global data internally because the debugger may be debugging independent processes simultaneously. Instead, it must associate data between calls with the specific object.

4.1 DMPL Types

DMPL includes several defined types in order to pass target architectural information to the debugger and to simplify interface function definitions. Other types provide function return codes, language values and codes for demangling information.

```
typedef struct {
    int short_size;
    int int_size;
    int long_size;
    int long_long_size;
    int pointer_size;                    /* sizeof (void *) */
} DMPL_target_type_sizes_t;

typedef unsigned long long               DMPL_taddr_t;
typedef long long                        DMPL_tword_t;

typedef struct _DMPL_process_t           DMPL_process_t;
typedef struct _DMPL_thread_t            DMPL_thread_t;
typedef struct _DMPL_type_t              DMPL_type_t;
```

Fig. 1. Types for DMPL target architectural information

Figure 1 shows that target architectural information includes the sizes of pointers and integer types and address values. Since `DMPL_taddr_t` is an unsigned long long, it allows for addresses on any architecture. Specifically, a 32-bit debugger should be able to debug a 64-bit target process.

Figure 1 also includes opaque types for process, thread and type information. The debugger and the DLL will each determine what process/thread information it needs. The opaque types, which will be cast to concrete types in the debugger, preserve types across the DMPL interface. We use these undefined structures instead of void pointers in order to provide more compile-time checking at the cost of explicit casts in the library and support code.

```
Enum {                                  Explanation
     DMPL_ok=0,                         Success
     DMPL_tls_unallocated               No space allocated
     DMPL_name_too_long,                Buffer too small
     DMPL_name_changed,                 Name wasn't demangled
     DMPL_first_user_code = 100         Allow more pre-defines
};
```

Fig. 2. DMPL result codes

```
enum {
     DMPL_lang_unknown        =      0,
     DMPL_lang_c              =      'c',
     DMPL_lang_cplus          =      'C',
     DMPL_lang_f77            =      'f',
     DMPL_lang_f9x            =      'F'
};
```

Fig. 3. DMPL language codes

We have included three enumerated types in DMPL. In order to avoid potential issues with different compilers implementing enumerated types as different sized objects, we actually use `int` when they are used as a result or parameter type. Most DMPL functions return one of the result codes listed in Fig. 2 Although both the DLL and the debugger will use values starting with `DMPL_first_user_code`, calling context will eliminate any confusion.

Figure 3 shows the DMPL language values, which support providing the debugger with language-specific information from the OpenMP run-time library. The DLL uses DMPL demangling codes that are listed in Fig. 4 to communicate scoping information to the debugger. `DMPL_varinfo_needs_dereference` should be 'or-ed' into the appropriate scoping code in order to indicate that dereferencing is necessary to access the variable.

4.2 DMPL Functions

In order to access the information needed for OpenMP codes, the debugger loads the DMPL DLL. The debugger then calls the DMPL function `DMPL_initialize`, which performs steps necessary to initialize the DLL, including instantiation of a DMPL callback table that supports communication between the DLL and the debugger. The rest of this section describes DMPL functions and function types, beginning with the DMPL callback table.

```
enum {
     DMPL_varinfo_needs_dereference                    = 0x8000,
     DMPL_varinfo_none                                 = 0,
     DMPL_varinfo_private,
     DMPL_varinfo_shared,
     DMPL_varinfo_firstprivate,
     DMPL_varinfo_lastprivate,
     DMPL_varinfo_firstlastprivate,
     DMPL_varinfo_reduction,
     DMPL_varinfo_threadprivate,
     DMPL_varinfo_threadshared,
     DMPL_varinfo_copyin
};
```

Fig. 4. DMPL demangling codes

4.2.1 DMPL Callback Table

In order to provide the information needed by the debugger, the DLL must make calls to the debugger. This architecture supports a clean interface between the DLL and the debugger, avoiding the use of global data. The DLL uses functions contained in the DMPL callback table, as defined in Fig. 5. The function types used to define the callback table are shown in Fig. 6. The primary callback functions communicate process, thread and type information between the DLL and the debugger.

```
typedef struct {
     DMPL_put_process_info_ft       DMPL_put_process_info_fp;
     DMPL_get_process_info_ft       DMPL_get_process_info_fp;
     DMPL_put_thread_info_ft        DMPL_put_thread_info_fp;
     DMPL_get_thread_info_ft        DMPL_get_thread_info_fp;
     DMPL_get_process_ft            DMPL_get_process_fp;
     DMPL_get_type_sizes_ft         DMPL_get_type_sizes_fp;
     DMPL_find_symbol_ft            DMPL_find_symbol_fp;
     DMPL_find_type_ft              DMPL_find_type_fp;
     DMPL_get_data_ft               DMPL_get_data_fp;
     DMPL_get_pthread_key_ft        DMPL_get_pthread_key_fp;
     DMPL_target_to_host_ft         DMPL_target_to_host_fp;
     DMPL_malloc_ft                 DMPL_malloc_fp;
     DMPL_free_ft                   DMPL_free_fp;
     DMPL_error_string_ft           DMPL_error_string_fp;
     DMPL_prints_ft                 DMPL_prints_fp;
} DMPL_callbacks_t;
```

Fig. 5. DMPL callback table definition

The DLL must be able to identify the thread or process associated with debugger calls. A call to `DMPL_put_process_info_fp` stores process information in the debugger while `DMPL_get_process_info_fp` retrieves the previously stored process information. The functions `DMPL_put_thread_info_fp` and `DMPL_get_thread_info_fp` provide similar functionality for threads. To pro-

vide access to process-wide information, the function DMPL_get_process_fp returns the process within which a thread resides.

Several callback functions combine to provide the needed mechanisms to communicate type and data information between the DLL and the debugger. The DLL uses DMPL_get_type_sizes_fp to get fundamental type sizes for a specific process. DMPL_find_symbol_fp provides a mechanism to look up a symbol in a process. Given a type name, the DLL can retrieve the associated information in the process with DMPL_find_type_fp. Data can be read from an address within a specific thread by using DMPL_get_data_fp while the value of a pthread key can be read by using the function DMPL_get_pthread_key_fp. Data can be converted into the host format with DMPL_target_to_host_fp.

```
typedef void (*DMPL_put_process_info_ft)
             (DMPL_process_t *, DMPL_process_info_t *);
typedef DMPL_process_info_t * (*DMPL_get_process_info_ft);
typedef void (*DMPL_put_thread_info_ft)
             (DMPL_thread_t *, DMPL_thread_info_t *);
typedef DMPL_thread_info_t * (*DMPL_get_thread_info_ft)
             (DMPL_thread_t *);
typedef DMPL_process_t * (*DMPL_get_process_ft)
             (DMPL_thread_t *);
typedef int (*DMPL_get_type_sizes_ft)
             (DMPL_process_t *, DMPL_target_type_sizes_t *);
typedef int (*DMPL_find_symbol_ft)
             (DMPL_process_t *, const char *, DMPL_taddr_t *);
typedef DMPL_type_t * (*DMPL_find_type_ft)
             (DMPL_process_t *, const char *, int);
typedef int (*DMPL_get_data_ft)
             (DMPL_thread_t *, DMPL_taddr_t, void *, int);
typedef int (*DMPL_get_pthread_key_ft)
             (DMPL_thread_t *, DMPL_tword_t, DMPL_taddr_t *);
typedef void (*DMPL_target_to_host_ft)
             (DMPL_thread_t *, const void *, void *, int);
typedef int (*DMPL_sizeof_ft) (DMPL_type_t *);
typedef int (*DMPL_field_offset_ft)
             (DMPL_type_t *, const char *);
typedef void * (*DMPL_malloc_ft) (size_t);
typedef void (*DMPL_free_ft) (void *);
typedef const char * (*DMPL_error_string_ft) (int);
typedef void (*DMPL_prints_ft) (const char *);
```

Fig. 6. DMPL callback function type definitions

The debugger uses two function types extensively to implement the type and data callback functions. A function of type DMPL_sizeof_ft determines the size of a specific type. For structs and similar types, the DLL can find the field offset pointer of a member name of the type with a DMPL_get_field_offset_ft function.

The remaining callback functions provide important utility operations to the DLL. The DLL can allocate and free debugger memory with `DMPL_malloc_fp` and `DMPL_free_fp`. The DLL should not call malloc and free directly. The DLL can determine the error string associated with an error code by calling `DMPL_error_string_fp`. `DMPL_prints_fp` allows the DLL to print any messages. The DLL should not print messages directly.

4.2.2 DMPL Debugger Functions

The debugger accesses several functions in the DMPL interface. To display the value or address of the thread variable, the debugger will make calls to the DLL to get the absolute address of the object. The DLL also provides initialization, version information and the abilities to release process and thread information and to convert an error code to a string. The rest of this section describes the functions for debugger use, which are defined in Fig. 7.

```
extern int DMPL_initialize (const DMPL_callbacks_t *);
extern const DMPL_rtl_names *DMPL_get_rtl_names (int);
extern int DMPL_demangle_name (const char * mangled_name,
             int * res_len, char * demangled_name,
             int buf_len, int * flags, int * decl_line);
extern int DMPL_get_tls_address
             (DMPL_thread_t *, DMPL_taddr_t, DMPL_taddr_t *);
extern int DMPL_destroy_thread_info (DMPL_thread_info_t *);
extern int DMPL_destroy_process_info (DMPL_process_info_t *);
extern const char *DMPL_version_string (void);
extern int DMPL_version_compatibility (void);
extern const char *DMPL_error_string (int);
```

Fig. 7. DMPL functions accessed by the debugger

The debugger calls `DMPL_get_rtl_names` in order to obtain the names of the run time library names invoked by a given OpenMP construct. The parameter of this function is a DMPL language code; as described previously, the parameter is passed as an `int` to avoid size conflicts. The return value is a pointer to a struct of two functions:

```
typedef struct {
  const char *main_task_dispatcher;
  const char *microtask_invoker;
} DMPL_rtl_names;
```

The main_task_dispatcher function is the outlined routine called by the main task to invoke the OpenMP parallel region or worksharing construct, which we refer to as a microtask. The second function invokes the actual microtask.

The DMPL interface provides the debugger with scoping functions to demangle mangled names and to locate the actual storage used for thread private variables. The `res_len` parameter of the `DMPL_demangle_name` function allows the debugger to allocate more space for the result if it is too long for the initial `deman-`

`gled_name` buffer. Thread private data can be implemented in various ways: thread local storage system facilities, virtual-address-map page aliasing, and pthread key specific data. Thus, we provide `DMPL_get_tls_address` to obtain the actual address of a thread private object in this thread, given its apparent address in the process or thread.

The remaining functions in the DMPL interface that are used by the debugger provide important utility functions. The debugger releases any process or thread information associated by the DLL with a process (or thread) by calling `DMPL_destroy_process_info` (or `DMPL_destroy_thread_info`). The debugger ensures that the DLL provides the expected interface through the two versioning functions `DMPL_version_string` and `DMPL_version_compatibility`. Finally, `DMPL_error_string` converts error codes to strings for the debugger.

5 Discussion

Although standards exist for OpenMP directives, there is currently no standard for the information that compilers or other tools require to present information consistently with the OpenMP programming model. Since compilers implement OpenMP directives differently, we propose that DMPL be adopted as a standard interface for providing debuggers and similar tools that information.

The POMP interface has already been proposed for providing information to performance tools. A legitimate question is whether performance tools and debuggers could be properly served by a unified information interface. However, the interfaces work in fundamentally different ways - the performance interface works within the application while the debugger interface is external. Compiler, run-time library and tool implementers have agreed during meetings of the OpenMP Tools Committee that they prefer two interfaces. The compiler writers and run-time system implementers have committed to providing two interfaces if the committee adopts them.

Even though the access mechanisms are sufficiently different to justify two standard information interfaces, there is significant intersection of the information needed for them. For example, DMPL includes functions to demangle names. Name demangling is also useful to performance tools. For this reason, we anticipate a set of low-level information standards or vendor supplied tools, such as a name demangler.

DMPL is a basic interface for the OpenMP debugging DLL. Implementations of this library with slightly different naming conventions are already available from IBM and Intel for its Guide compilers. OpenMP debugging on those platforms with current versions of TotalView demonstrates that DMPL provides significant support for the OpenMP programming model [4]. However, DMPL is an evolving interface and we recognize that additional functions and types may be needed. The Tools Committee will iterate on this specification and will eventually adopt a DLL-based debugger interface standard.

References

1. OpenMP Architecture Review Board: OpenMP Fortran Application Program Interface, Version 2.0. OpenMP Architecture Review Board (2000)
2. OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface, Version 2.0. OpenMP Architecture Review Board (2002)
3. Mohr, B., Malony, A.D., Hoppe, H.C., Schlimbach, F., Haab, G., Hoefflinger, J., Shah, S.: A Performance Monitoring Interface for OpenMP. In Proceedings of the Fourth European Workshop on OpenMP (EWOMP 2002). Rome (2002)
4. Etnus LLC: TotalView Reference Guide, Version 6.0. Etnus LLC (2002)
5. Cownie, J., Gropp, W.: A Standard Interface for Debugger Access to Message Queue Information in MPI. Sixth European PVM/MPI Users' Group Meeting. (1999)
6. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, Institute for Defense Analysis, Center for Computer Sciences, Bowie, Maryland (1999)

Is the *Schedule* Clause Really Necessary in OpenMP?

Eduard Ayguadé[1], Bob Blainey[2], Alejandro Duran[1], Jesús Labarta[1], Francisco Martínez[1], Xavier Martorell[1], and Raúl Silvera[2]

[1] CEPBA-IBM Research Institute
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3, Barcelona, Spain
{eduard,aduran,jesus,fmartin,xavim}@ac.upc.es
[2] IBM Toronto Lab, 8200 Warden Ave
Markham, ON, L6G 1C7, Canada
{blainey,rauls}@ca.ibm.com

Abstract. Choosing the appropriate assignment of loop iterations to threads is one of the most important decisions that need to be taken when parallelizing Loops, the main source of parallelism in numerical applications. This is not an easy task, even for expert programmers, and it can potentially take a large amount of time. OpenMP offers the *schedule* clause, with a set of predefined iteration scheduling strategies, to specify how (and when) this assignment of iterations to threads is done. In some cases, the best schedule depends on architectural characteristics of the target architecture, data input, ... making the code less portable. Even worse, the best schedule can change along execution time depending on dynamic changes in the behavior of the loop or changes in the resources available in the system. Also, for certain types of imbalanced loops, the schedulers already proposed in the literature are not able to extract the maximum parallelism because they do not appropriately trade–off load balancing and data locality. This paper proposes a new scheduling strategy, that derives at run time the best scheduling policy for each parallel loop in the program, based on information gathered at runtime by the library itself.

1 Introduction

Parallel loops are the most important source of parallelism in numerical applications. OpenMP, the standard shared–memory programming model, allows the exploitation of loop–level parallelism thorough the `DO` work–sharing and `PARALLEL DO` constructs. Iterations are the work units that are distributed among threads as indicated in the `SCHEDULE` clause: `STATIC`, `DYNAMIC` and `GUIDED` (all of them with or without the specification of a `chunk` size). While in a `STATIC` schedule the assignment of iterations to threads is defined before the computation in the loop starts, both `DYNAMIC` and `GUIDED` do the assignment dynamically as the work is being executed. In `DYNAMIC` threads get uniform chunks while in

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 147–159, 2003.

`GUIDED` chunks are progressively reduced in size in order to reduce scheduling overheads at the beginning of the loop and Cavour load balancing at the end.

Deciding the appropriate scheduling of iterations to threads may not be an easy task for the programmer, specially when it depends on dynamic issues, such as input data, or when memory behavior is highly dependent on the schedule applied. Load unbalancing or high cache miss ratios, respectively, are usually symptoms of inappropriate iteration assignments. In OpenMP, the programmer can play with the predefined schedules mentioned above or embed its own scheduling strategy in the application code if none of them is appropriate. The `chunk` size (or number of contiguous iterations assigned to a thread) is a parameter that needs to be appropriately set in order to avoid non–friendly memory assignments of iterations and/or excessive run-time overheads in the process of getting work. Even worse, the decisions may depend on parameters of the target architecture (going against performance portability, one of the key issues in OpenMP).

The standard offers the possibility of specifying that the loop needs to be serialized if a certain condition is met (`IF` clause in OpenMP). Some OpenMP runtime systems can also decide to serialize the execution if certain conditions (e.g. loop bounds, number of threads, ...) are met.

In order to decide a schedule strategy, some simple rules of thumb are usually applied: `STATIC` for those loops with good balance among iterations; unbalanced loops should use an interleaved schedule (`STATIC` with `chunk`) or some sort of dynamic schedule (`DYNAMIC` or `GUIDED`). However, the use of dynamic schedules usually incurs high scheduling overheads and its non-predictive behavior tends to degrade data locality (non–reuse of data across loops or multiples instances of the same loop). Although these rules work for a large number of simple cases, they are far from complete and can lead to poor decisions. Other schedules need to be built by the user, embedding code and data structures to implement them.

In this paper we will present a proposal to remove such burden from the programmer by letting the runtime decide which is the most appropriate schedule for a given loop. In the next section we motivate the work by using a simple unbalanced and applying different schedules. In section 3 we present the generic framework of our work. In section 4 we describe our current prototype implementation. In section 5 we show the results obtained with some benchmarks. Finally in section **??** concludes the paper and shows future directions of research.

2 Motivation and Related Work

In order to motivate the proposal presented in this paper, we will consider a synthetic loop in which the cost of each iteration is $cost(i) = k/i$, k being a parameter that depends on the number of iterations of the loop. This distributes almost all the weight of the loop to the first iterations (the first 1% of the iteration space accounts for 50% of the cost of the loop). During the execution, each thread accesses a matrix indexed with its thread identifier. Therefore, the loop only has

temporal locality. The loop has been executed 500 times on a 4-way IBM Power4 system. Figure 1 shows the results obtained for different schedules.

In this loop, using a `STATIC` schedule leads to a highly unbalanced execution, with a speedup of 1.64 with respect to the execution with one thread. Although the use of a `STATIC` schedule with a chunk size of one increases the speedup to 1.97, it still does not achieve good balance. For example, if $k = 10000$, the work of the first thread is 1.6 times the work of the fourth. Therefore, this is an example in which it seems appropriate to use either a `DYNAMIC` or `GUIDED` schedule. Using a `DYNAMIC` schedule we get a speedup of 1.82 due to the high scheduling overheads and degradation of temporal locality. A `GUIDED` schedule, which usually tends to reduce these scheduling overheads, is decreasing the speedup to 1.63; this is due to the fact that some threads get excessively large chunks at the beginning that are not well balanced with the remaining ones.

Probably, the best schedule would be ad–hoc trying to reduce scheduling overheads, optimize load balancing, avoid false sharing or a combination of these issues that compensate them. However, in other cases it may be even impossible for the the programmer to calculate this "ideal schedule" because it depends on variables only available at runtime (architecture, input data, interaction between loops or processes, ...).

Other schedules, similar in nature to `GUIDED`, have been proposed in the literature, such as trapezoid scheduling [1], factoring [2], and tapering [3]. These schedules are variations of the previous ones and are tailored for certain load unbalance patterns. Some other schedules try to take in account the geometric form of the iteration space. For example folding is a variation of `STATIC` in which iterations i and $N - i$ are assigned to the same thread, N being the total number of iterations. For the previous synthetic example, this schedule is unable to improve the behavior achieving a speedup of 1.77. A study of the most suited schedule for different loops, grouped by their iteration execution time variance is presented in [4].

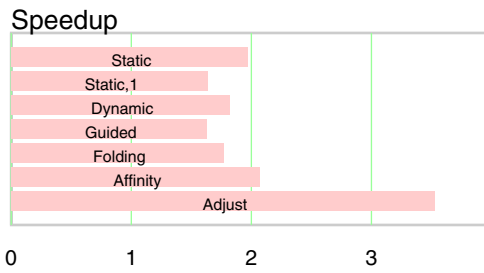


Fig. 1. Speedup for different schedules on a 4-way IBM Power4

Other proposals try to achieve load balancing by applying work stealing. They usually assign iterations to threads in a `STATIC`–like manner; in Affinity Scheduling (AS) [5] threads steal chunks of work from other processors as soon as they finish with the initially assigned work. This work stealing adds a dynamic part to the work assignment that does not Cavour memory behavior. Affinity Scheduling is available in the IBM OpenMP runtime system as a non–standard feature that can be specified in the `OMP_RUNTIME` environment variable. In our synthetic example, affinity scheduling achieves the highest speedup of all the available schedules (2.07). Dynamically partitioned affinity scheduling (DPAS) [6] learns from work stealing in order to derive a new `STATIC`-like schedule, to be used in subsequent instances of the loop, in which each thread has a different chunk size. Other proposals such as Feedback Guided Dynamic Schedule (FGDS) [7] and Feedback Guided Load Balancing [8] avoid the dynamic part by simply measuring the amount of unbalance (without applying work stealing) and derives a similar `STATIC` schedule. In [9] a different approach is used where a processor is reserved to compute partial schedules based on the load of each processor that are placed on the processors work queues.

The main objective of this paper is to advance one step further in the use of dynamically derived schedules and show how they can optimize the behavior of real applications. We propose a general framework oriented towards having a self–tuned OpenMP runtime system and show an implementation on a real commercial system. The runtime is able to characterize the execution of a loop and learn from past executions in the same run in order to gradually enhance the assignment of iterations to threads. The objective is to relieve the user from the task of deciding the best schedule for each loop and ideally lead to better performance. For instance, in the same synthetic example described above, our proposed framework achieves an speedup of 3.53. The scheduling is achieved in a completely transparent way with no additional specification from the programmer in the source code.

3 Dynamic Derivation of Loop Schedules

In order to decide the most suited schedule, the runtime needs to collect information that characterizes the behavior of the loop. Although the compiler could provide static information derived from the analysis of the source code (or even could be provided by the user as hints), such as the initial schedule for the loop or the identifiers of other loops with similar memory access and/or workload patterns, most of the information can only be gathered at runtime. Our main goal is to show that this information gathering, loop characterization and optimization can be done at runtime with minimal (or no) information from the user and/or compiler and with reasonable (or even negligible) overhead.

At runtime, the information that can be dynamically observed and collected includes: size and bounds of the iteration space, variation in the cost of the iterations, memory access patterns and conflicts in the access to memory containers (cache lines or pages), etc. In the process of observing these metrics, granularity

is an important issue to consider: Overall per–thread execution time versus execution time for iteration (or groups of iterations), overall per–thread cache miss ratio (or page fault ratio) versus detailed correspondence between cache misses (page faults) and loop iterations, ... The accuracy level (granularity) may not be constant during the execution of the program and vary according to the characterization process itself. The first time a loop is executed, or after detecting a high perturbation in its current characterization, the runtime could switch to fine–grain measurement status. Once the runtime detects a stable characterization, it could switch to a coarser–grain mode of operation, in order to minimize unnecessary overheads.

When no information is available for a loop (e.g it is the first time the loop is executed), the runtime could start with a predefined schedule (or even the one suggested by the programmer) and try to characterize the loop doing fine-grain measurements. Another possibility could be to adopt the characterization for a another loop (for which a characterization has been done) and make fine-grain measurements. This characterization reuse may be important in order to reduce the time required to reach a stable characterization state. Reuse hints could also be provided by the programmer or the compiler (e.g. providing information about affine loops). It could even be possible that the runtime discovers affinity relationships between loops (i.e. loops whose characterization is the same or changes in the same way) that are executed inside an iterative sequential loop.

Our belief is that the use of work-stealing strategies during the characterization process should be avoided in order to prevent perturbations with the characterization process itself. For example, work stealing adds a dynamic part to the assignment of iterations that may worsen memory locality and increase memory latencies both for the stealing and stolen threads. However, in some cases this extra overhead may be compensated with load balancing, thus reducing the overall time to reach a (new) efficient schedule for the loop. Loops that are executed only once could also have a better behavior if work stealing is applied.

Based on the available characterization, and in order to decide the best suited assignment of iterations to threads, the runtime should try to:

- Preserve spatial locality, by assigning contiguous chunks of iterations to the same thread whenever possible. This will optimize the access to memory containers (cache lines or pages) and reduce the likelihood of false sharing.
- Preserve temporal locality, by reusing the same schedule in subsequent execution instances of the same loop (or an affine one). This will Cavour data reuse.
- Balance loops, so that all threads get the same amount of work; this does not imply the same amount of iterations.

Once the scheduling is decided, the characterization process continues in order to detect further opportunities for refinement. As mentioned before, and since this information gathering could have a significant impact on performance, the runtime should be able to switch to the most appropriate granularity level, depending on the status of the characterization itself.

Up to this point, we have not addressed possible interferences between the schedules applied to different loops. The use of different iteration assignments in different loops may degrade memory locality and be counter–productive. To this end, the characterization process could consider sequences of loops and derive decisions that optimize the behavior of the sequences and not the individual loops.

4 Current Implementation

In this section, we describe the current prototype for the self–tuning OpenMP runtime system that has been implemented in the XL IBM Runtime. In the description we consider both the characterization and the decision processes. In this prototype implementation, mainly load balancing issues are addressed.

The runtime identifies each parallel loop instance with a tuple $\{L,\ IS\}$. The first component (L) of this tuple identifies the loop in the program (using the pointer to the routine generated by the compiler that encapsulates the loop). The second component (IS) identifies different instances of the same loop (using the iteration space of the loop: iteration limits). Whenever possible, the information derived by the runtime for a tuple $\{L_i,\ IS_j\}$ will be re-used to initially characterize other tuples $\{L_i,\ IS_k\}$ that correspond to the same loop. All tuples that correspond to the same loop L_i summarize the past behavior for that loop.

For each tuple $\{L,\ IS\}$ the following information is recorded:

- The pointer to the routine that identifies the loop.
- The iteration space description.
- The $\{L,IS\}$ balancing information.
- The last subchunk information gathered by the runtime for the tuple (See below).
- The relation of weights between iterations. In the current implementation only two patterns are handled: constant weight, when all iteration have the same weight, and unknown. Others patterns could be recognized if their properties are useful for scheduling.
- The last schedule applied to the tuple.

The balance information is composed of:

- A state that indicates the actual knowledge of the balance of the {loop/iteration space}.The possible states and their meaning are summarized in table 1.
- The number of consecutive executions that this balance state has been maintained.
- The actual definition of balanceness for the tuple, i.e the percent of unbalance allowed. This definition varies among time. When there is no knowledge about balance the limit is 10% of imbalance. Later on, as there is more confidence the limit is increased first to 20% and later to 25%. The increment of our definition of balance enables to elude minimal perturbations of the system.

Table 1. Possible balance states

State	Meaning
Unknown	No balance information is known yet.
Unbalanced	Runtime found that it is unable to balance the tuple.
Balanced	Runtime found a schedule that balanced the tuple.
Highly balanced	Runtime feels really confident that the schedule applied
	will be balanced.

The transitions of the balance state are shown in figure 2. If no information is inherited from other states, the first balance information is the *Unknown* state. When a balanced execution is done a transition to the *Balanced* state is done. After N unsuccessful executions the *Unbalanced* state is achieved. While in this state a balanced execution, normally due to a change in behavior, takes a transition to the *Balanced* state. While in the *Balanced* state any imbalanced execution reverts to the *Unknown* state. N balanced executions in the *Balance* state increases the confidence on the decision and goes to the *Highly Balanced* state. An unbalanced execution in this last state reverts to the *Balanced* state. Note that when there is confidence in the balance of the loop there have to be two consecutive unbalanced executions in the last N to revert from *Highly Balanced* to *Unknown*. This gives some tolerance to perturbations while being able to adapt to changes in the behavior. The actual value for N is 10 times.

When a {loop/iteration space} is executed for the first time a new state is allocated for it. If it is also the first execution of the loop the state is initialized to an *Unknown* balance state and a hypothesis that the iterations weights are constant is tried. If other iteration spaces were executed for the same loop before, the initialization is inherited from the most similar iteration space. In other words, the balance information, the iteration information and a modified version of the schedule applied to the other iteration space (adding or subtracting iterations) are copied.

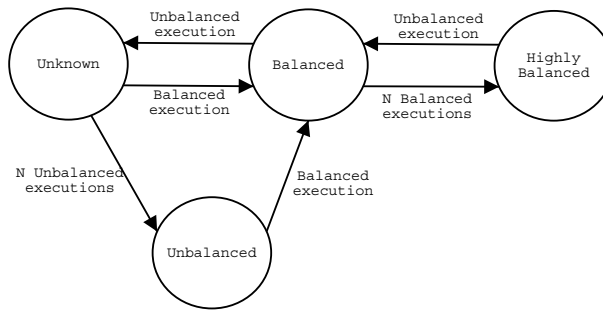


Fig. 2. Balance information transitions

Every time the loop starts the execution in a given iteration space, the schedule to be used (and its parameters) are decided. This decision is based on the current state of the tuple. Currently, one of two the following schedules can be chosen:

- *OpenMP* `STATIC`.
- *Non–uniform* `STATIC`. This schedule is very similar to the previous one. Each thread is also assigned a chunk of contiguous iterations that are determined prior to the loop execution. However, chunks assigned to threads may be of different size. When the size of each chunk is properly chosen a very good load balance is achieved. Temporal and spatial localities are achieved as in the `STATIC` case because of the assignment of contiguous iterations and schedule reuse.

The scheduling decision is summarized in table 2. When something is known about the balance of the loop, either that loop is balanced or that it is unbalanceable, the last schedule applied is reused. In case the loop is considered balanced this schedule will be the the one that achieved the balance. In case the loop is considered unbalanceable the schedule will be the best schedule found which will be used there on. When nothing is known about the balance of the loop, either because a proper schedule that balances it has not been found yet or because it hasn't reached the threshold to give up, the schedule used is static if the iterations were found to be constant, otherwise an non-uniform static schedule is used with the assignment of iterations for each thread calculated based on previous gathered measurements.

Table 2. Schedule decision function

Balance state	Iteration cost	Schedule
Unknown	Constant	Static
Unknown	Non-constant	Non-uniform Static
Other	*	Reuse previous

The assignment of iterations for each thread when the non-uniform static schedule is used works as follows. First, the weight each thread should have is calculated dividing the total time by the number of threads. Afterwards, subchunks are assigned sequentially to the first thread. When the sum of the subchunks is greater that the estimated weight per thread, the last subchunk is broken assuming all iterations in the subchunk have the same weight, and the number iterations of the first thread is adjusted in consequence. The remaining iterations of the last subchunk are left to be assigned to other threads. Next, we start assigning iterations to the second thread, and so on. If we arrive to the last thread there are still some iterations left are assigned to the last thread.

Also when going to execute the loop, the granularity of measuring has to be decided. Two granularities are supported: subchunk (fine granularity) and

thread (coarse granularity). When subchunk granularity is used, to avoid excessive overhead of measuring every single iteration, groups of iterations called subchunks are measured. The number of these subchunks is variable for each thread and depends of the number of iterations that have been assigned. When thread granularity is chosen, the measures are done for the overall iterations of each thread (so there is only one subchunk per thread). The measures right now include only execution times. The decision of choosing between the two granularities is taken based on the balance state of the loop as shown in table 3.

Table 3. Time measures granularity decision function

Balance state	Granularity used
Unknown	Fine measuring
Other	Coarse measuring

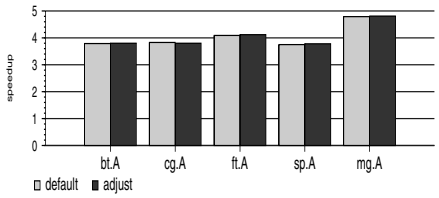
After the execution of the loop a new state has to be generated. It is calculated using the actual state and the measures taken from the execution. If the execution time of each thread does not deviate from the average more than the actual definition of balance the execution is considered balanced, otherwise the execution was unbalanced. With this information a transition in the balance state automaton is done. Also, based on the taken measures, if the mean iteration weight per thread does not deviate from a certain threshold the iterations are considered to be constant, otherwise iteration weights are calculated from subchunk information. Finally, current schedule decision are saved as the last schedule applied. If this schedule also resulted in the best schedule applied so far it is saved as the best schedule used.

With this runtime environment, loops that would require the use of `STATIC`, `DYNAMIC`, `GUIDED` or even other schedules not available in OpenMP (such as folding) can be efficiently executed, as shown in the evaluation section.

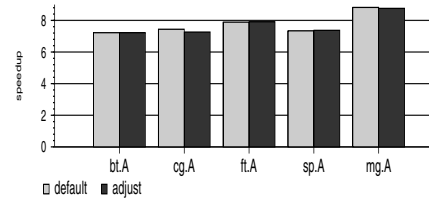
5 Evaluation

In order to evaluate the proposed schedule, we have used some programs from the SPEComp suite [10] (swim, ammp, gafort, apsi, wupwise, and art), class A NAS OpenMP benchmarks [11] (bt, ft, cg, sp, and mg), and a computational kernel that calculates the Legendre polynomial. They are OpenMP versions that make use of the `SCHEDULE` clause.

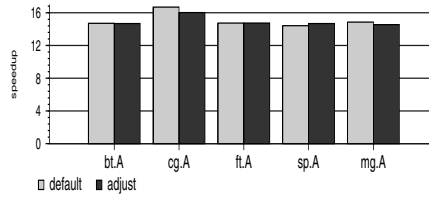
When a *parallel do* loop does not specify an schedule (using the `SCHEDULE` clause) it defaults to a special value: `RUNTIME`. This value means that the actual schedule to be used can be specified in the environment variable In order to specify a schedule not specified in the standard, we use `OMP_RUNTIME`. If this variable is not specified the schedule used is implementation dependent, thought typically is `STATIC`. Two different kind of tests have been run: the first run



(a) 4 processors



(b) 8 processors



(c) 16 processors

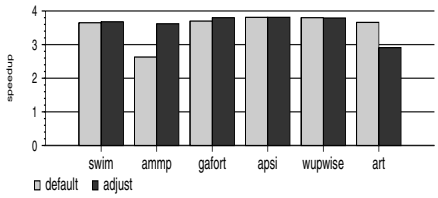
Fig. 3. Speedups for the NAS benchmarks

with the schedules originally set in the benchmark, and `OMP_RUNTIME` defined as `STATIC`. In the second test, we eliminated the `SCHEDULE` clauses of all parallel loops and set the `OMP_RUNTIME` environment variable to `ADJUST`, specifying that the schedule described in Section 4 should be applied.

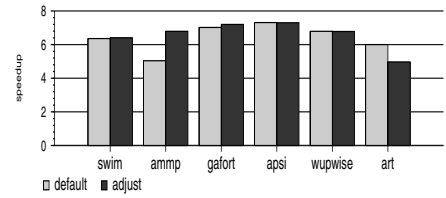
The benchmarks were run in a p690 32-way Power4 [12] machine at 1.1 Ghz with 128 Gb of RAM. We used the IBM's XLF compiler with the `-O3 -qipa=noobject -qsmp=omp` flags, and the operating system was AIX 5.1 .

Programs *bt*, *ft*, *cg*, *sp*, *mg*, *swim*, *apsi*, and *wupwise* use `STATIC` schedules. The programs *ammp*, *gafort* and *art* use `GUIDED` schedules. The *Legendre* kernel has two triangular loops that are programmed with a "folding" schedule embedded in the application code. In *mg*, is interesting to note that the iteration space changes from one execution to another.

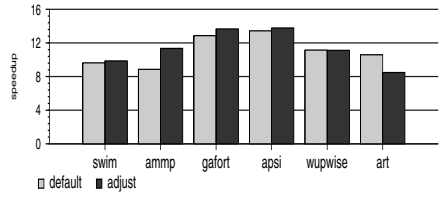
Figure 3 shows the results for the NAS benchmarks, on the x-axis are the benchmarks and on the y-axis is the speedup achieved for the default schedule and for the adjust runtime. It can be seen that the results obtained from both methods are almost equivalent (the difference is always below 5%). This is due the NAS benchmarks have loops very balanced with good locality by default and there is little room for improvement here. The important thing is that our mechanism is able to decide also a `STATIC` schedule for this type of loops, that are quite common, with a negligible overhead in reaching that decision.



(a) 4 processors



(b) 8 processors

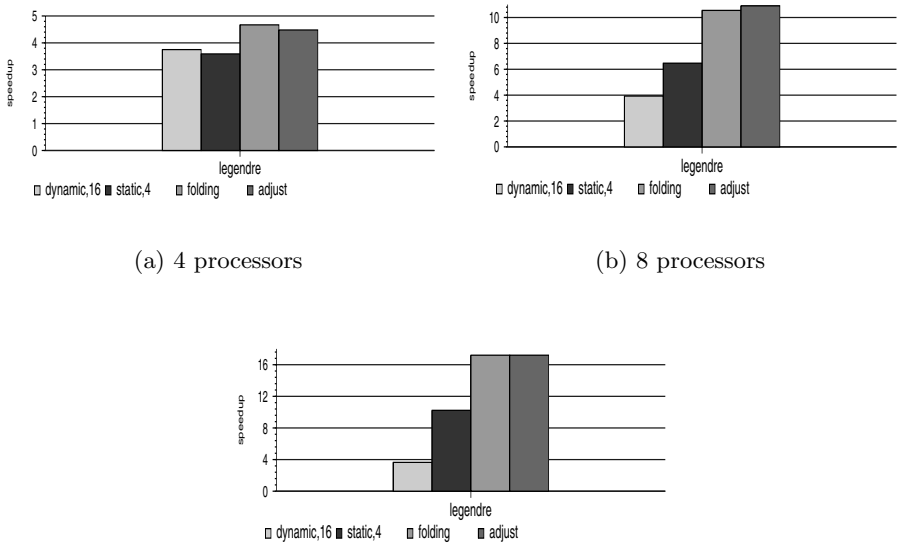


(c) 16 processors

Fig. 4. Speedups for the SPEComp benchmarks

Figure 4 shows the results for the SPEComp benchmarks. Notice that the ones that use `STATIC` schedules achieve the same performance with differences below 3%. Of the programs that use `GUIDED` schedules, *ammp* is really improved by our method (27% with 4 processors, 22% with 16 processors). The reason for this is that the *non–uniform static* schedule derived by the runtime has much better locality, because iterations are executed contiguously and the schedule is reused, than the original schedule. The improvement of *gafort* is negligible (below 4%). As *gafort* makes random vector accesses thus we do not obtain the locality gains seen in *ammp*. The *art* benchmark is the worse case our method can find, as the main code is a loop executed just one time, and we cannot make use of the knowledge obtained in that first execution. As we see, our method needs loops that are executed iteratively. To solve this cases a schedule that is more flexible to imbalanced codes than `STATIC` should be used the first time (such as the Affinity schedule), but right now the original code performs a 20% better that our proposal.

In figure 5 are the results for the *Legendre* kernel results. In addition to the default folding schedule and adjust, the best OpenMP schedules found (dynamic, 16 for 4 processors and static, 4 for 8 and 16 processors) are also shown. Note that one user that decided to use as schedule *dynamic,16* after some basic analysis in a minor configuration would be fooled if later he would run it in a high end



(a) 4 processors

(b) 8 processors

(c) 16 processors

Fig. 5. Speedups for the Legendre kernel

production system. In any case, our proposal is able to find a *non–uniform static* schedule that performs as well as the "hardwired" folding schedule (differences are below 5%) and much better than any OpenMP schedule (from 16% to 41% better).

Acknowledgments

Authors want to thank Julita Corbalan for her insightful comments. This work has been supported by the IBM CAS program, the POP European Future Emerging Technologies project under contract IST-2001-33071 and by the Spanish Ministry of Science and Education under contract TIC2001-0995-C02-01.

References

1. T.H. Tzen and L.M. Ni. Trapezoid self-scheduling scheme for parallel computers. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):87–98, 1993.
2. E. Schonberg, S.F. Hummel, and L.E. Flynn. Factoring: A practical and robust method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

3. S. Lucco. A dynamic scheduling method for irregular parallel programs. In *Proceedings of ACM SIGPLAN'92 Conference on Programming Language Desing and Implementation*, pages 220–211, 1992.
4. Kelvin K. Yule and David J. Lilja. Categorizing parallel loops based on iteration execution time variances. Technical Report HPPC-94-13, University of Minnesota, 1994.
5. E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. Technical Report TR410, 1992.
6. S. Subramaniam and D.L. Eager. Affinity scheduling of unbalanced workloads. In *SuperComputer'94 Conference Proceedings*, 1994.
7. J. Mark Bull. Feedback guided dynamic loop scheduling: Algorithms and experiments. In *European Conference on Parallel Processing*, pages 377–382, 1998.
8. Francis H. Dang and Lawrence Rauchwerger. Speculative parallelization of partially parallel loops. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 285–299, 2000.
9. Babak Hamidzadeh and David J. Lilja. Self-adjusting scheduling: An on-line optimization technique for locality management and load balancing. In *International Conference on Parallel Processing*, pages 39–46, 1994.
10. Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. *Lecture Notes in Computer Science*, 2104, 2001.
11. D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
12. Steve Behling et al. The POWER4 processor introduction and tuning guide. Technical Report SG24-7041-00, International Technical Support Organization, November 2001. ISBN 0738423556.

Extended Overhead Analysis for OpenMP Performance Tuning

Chen Yongjian, Wang Dingxing, and Zheng Weimin

Institute of HPC, Tsinghua University, China
chenyj99@mails.tsinghua.edu.cn
{dxwang,zwm-dcs}@tsinghua.edu.cn

Abstract. Overhead analysis was developed as a performance tuning approach for parallel programming and were adopted by several performance analysis systems for OpenMP programs. In this paper, an extended overhead analysis scheme based on layered model is proposed for OpenMP programming, to further enhance the capability of overhead analysis and thus make the OpenMP performance tuning easier. An example case called ILP/TLP overlap is studied in detail to show the idea of layered overhead model, and a new way to organize the overhead hierarchically is also presented based on the layered overhead model.

1 Introduction

OpenMP was designed to be easy to use for parallelization. However, to obtain high performance, the programmer still needs knowledge about both the application and the underlying system, which imposes too many obstacles in the way to effective parallel programming using OpenMP. In order to address such problem, the performance tuning assistant tools special for OpenMP, should be more powerful to make OpenMP programming even easier.

Overhead analysis is just another approach for performance tuning. Generally, overhead is defined and broken down in the following way:

$$T_p = \frac{T_s}{p} + \sum_i O_i \Rightarrow O_i = T_p^i - \frac{T_s^i}{p} \tag{1}$$

or in a more familiar Amdahl's law view:

$$S = \frac{T_s}{T_p} = \frac{p}{1 + \frac{\sum_i O_i}{T_s}} = \frac{p}{1 + \sum_i O_i^r} \tag{2}$$

T_p and T_s are parallel time and the best sequential time of the program respectively, and p is the processor number. O_i is the overhead of category "i". In (2), O_i^r is called relative overhead of category "i".

To approach the ideal speedup, overhead analysis assists the programmer to eliminates these overheads according to the priority of the relative overheads. The rationale behind is that, if we can break down the overhead into detailed

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 160–169, 2003.

enough categories, so that each overhead class is just corresponding to one identified cause, then by measuring the overhead, we can directly trace back to their causes and thus reduce or even eliminate performance overhead in a recipe way. Since OpenMP is rather easy, it is possible to enumerate most of the overheads in OpenMP programs for typical OpenMP constructs. In fact, the OpenMP constructs used in practical programs fall into several basic sets, and to these typical constructs, not too much parameters can be adjusted for performance reason.

For the usage of overhead analysis, we take the philosophy that performance tuning system should provide tools to help programmers to understand the performance and answer performance questions, and the programmers decide the others([11]). As stated in [4], the performance tuning process is essentially an optimization problem, so the most hopeful benefits of overhead analysis is to reduce the search effort to find the optimal implementation. A possible use case of overhead analysis is that it can identify performance anomalies, and relate these anomalies back to predefined causes and improvement suggestions. Here we use the phrase "performance anomaly" to represent those "unexpected performance".

In this paper, we use a layered model for overhead analysis, and an example case called TLP (Thread Level Parallelism) and ILP (Instruction Level Parallelism) overlap in OpenMP programs is discussed to demonstrate the idea, and show that how this layered model can help programmers to locate and understand the performance at OpenMP language level. And then we extend current overhead analysis for OpenMP programs by introducing a new category of overhead and reorganizing the overhead classifications, to apply the layered model and to allow more direct tuning.

The rest of the paper is organized in the following way: related work on overhead analysis is presented in section 2. A layered model for overhead analysis is discussed in section 3, and to illustrate the idea, a case study using an example called ILP/TLP overlap is presented in section 4. In section 5, an extended overhead analysis based on the layered model is given out. Conclusions and future works come as section 6, while acknowledgment and bibliography come as the last parts.

2 Related Work on Overhead Analysis

Use of overhead as performance metrics is not new([13]). The definition of overhead in this paper is raised in [6], and it's a natural extension to the conventional cost metrics by making distinction between raw performance and non-achieved performance.

[6] further suggested using overhead classification and measurement as a framework to describe the behavior of parallel programs. A hierarchical classification of overhead is developed because such organization are flexible to provide overhead information in different detail level and more suitable for refinement style performance tuning. Other classification schemes are proposed in [9] and [8], with only slight differences.

Tools based on overhead analysis have been developed in past five years, include Ovaltine([9]) and SCALEA([8]). In the latter, the measurement and analysis is on user defined regions rather than functions or routines, which helps to isolate the performance anomalies and is natural to OpenMP programs since most OpenMP constructs have region scopes. Since OpenMP is intended to be used in an incremental way, which means that, the parallelization often involves only local changes, and in this way, constructs in OpenMP programs can be mapped to corresponding constructs in the sequential version. This forms the base of region based overhead analysis.

As an early effort towards overhead analysis, [10] used lost cycle analysis to build up analytical performance models based on measured overhead, and intended to use it to predict performance. It's possible to model the impact of some OpenMP construct parameters on the performance, e.g., the schedule type and chunksize of schedule, but in general, such a model may be too complex to be built up.

Overhead Analysis is powerful in that it not only provides the programmer a way to get insight about the performance characteristics, but also provides a clearer clue about why the programs behave badly by identifying overhead components. However, there are still many difficulties for it.

Overhead analysis tries to quantify all the non-achieved performance for parallel programs. Often, performance lost may be caused by low level issues rather than OpenMP level, and moreover, Overhead Analysis still doesn't imply a direct way to further tune the programs in some cases, since the overheads of some high level structures in the OpenMP programs may span across multiples categories. The problem is that it doesn't directly point back to the original reason.

3 Overhead Analysis: A Layered Model

OpenMP is a high level language, and generally, OpenMP applications are implemented by several abstract layers, as depicted in Figure 1.

In such a model, each low abstract layer implements a higher abstract layer and the implementation details of lower level abstract layers are hidden by the higher abstract layer. Overheads can be viewed as inefficiencies in the implementation, and these inefficiencies can be injected into the code at any of these abstract layers, and the performance of programs written in high level is determined by the implementation efficiency of each lower level abstract layer. Several works have been done to evaluate the efficiencies of special abstract layers. For example, special benchmarks ([5] and [14]) were developed to measure the efficiency of OpenMP implementation abstract layer.

In this layered model, overhead at a specific abstract layer can be inherent inefficiency, such as those caused by non-optimal implementations, or inefficiencies induced by high level reasons. We call these high level reasons the original overhead sources, and reasons at low level are called direct overhead sources(or derived overhead sources). For inherent inefficiency, there are not many ways to

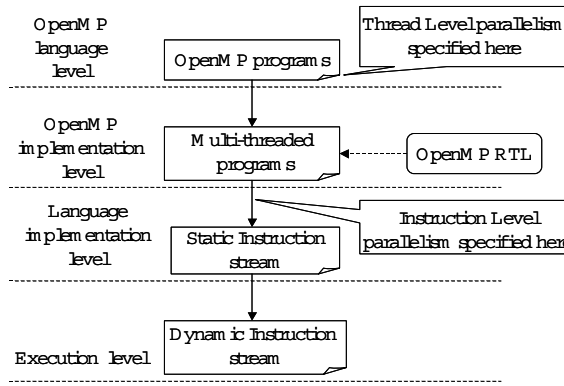


Fig. 1. The abstract layers to implement OpenMP API

attacked them at high levels. Reducing or eliminating the induced inefficiency by removing the overhead source is the target of overhead analysis.

In the performance tuning process, the system tends to collect performance data at lower levels and the programmer are supposed to modify the programs at higher levels. And to explain the overhead at some specific abstract layers to the programmer, knowledge about these layers is also required for the programmer, and again to fulfill this requirement is too much for average programmers. What needed is to find a way to interpret low level overheads, and associate the direct overhead sources with their original overhead sources(This is called mapping of low-level cost and high-level structure in [12]). The programmer needs to view the performance problems at the programming level, other than something that he can not control.

Most of current performance tuning approaches don't applies to this multi-layer model, and performance metrics from different abstract levels are present all at once to programmers for their choice. To simplify the tuning process, the details about lower levels should be hidden away from the programmer as much as possible while only a high level summary should be presented.

In the next section, we use an example, called TLP and ILP overlap to illustrate this idea.

4 Case Study: TLP and ILP Overlap

The OpenMP program presented in Code 1 causes big performance trouble when executing on a 4 Itanium2 CPU SMP Linux box comparing with its sequential execution.

Code 1: pi.f to calculate π

```
      program compute_pi
      integer n, i
      double precision w, x, sum, pi, f, a
      f(a) = 4.d0 / (1.d0 + a * a)
      n = 100000000
      w = 1.0d0/n
      sum = 0.0d0
!$OMP PARALLEL DO private(x) schedule(STATIC,1) reduction(+: sum)
      do i = 1, n
           x = w * (i - 0.5d0)
           sum = sum + f(x)
      enddo
!$OMP END PARALLEL DO
      pi = w * sum
print *, 'computed pi = ', pi
```

Refinement methods can be used to isolate the performance bottleneck, and locate it to the parallel-do loop construct in above code. Table 1 presents execution level performance metrics such as the execution cycle count of the parallel-do loop, and overheads are derived from these metrics.

Table 1. Execution cycle count of compute_pi. The data were collected using pfmon2.0 ([1]). And for comparison, different compilation options are used to produce a sequential version, an OpenMP version with SWP enabled and an OpenMP version with SWP disabled. Data are presented with a postfix: "b" means "billion" and "m" means "million". CPU_CYCLES, INST_DISPERSED, IA64_INST_RETIRED, NOPS_RETIRED and DISP_STALLED are PMU event code name ([2]), stand for total CPU cycles, dispersed instruction number, retired instruction number, retired NOP instruction number and stalled CPU cycles respectively. DISP_STALLED count is presented to show that the load is even distributed and synchronization overhead is small. The machine used to produce the result is a 4–Itanium2 SMP box, running Linux with kernel 2.4.19smp, and the code is compiled using ORC1.1 (Open Research Compiler, [3]) with OpenMP enabled, running above Intel's GUIDE OpenMP runtime library

	sequential	w/SWP enabled				w/SWP disabled			
		CPU0	CPU1	CPU2	CPU3	CPU0	CPU1	CPU2	CPU3
CPU_CYCLE	0.78b	2.85b	2.85b	2.85b	2.85b	1.85b	1.85b	1.85b	1.85b
INST_DISPERSED	4.77b	15.5b	15.5b	15.5b	15.5b	2.25b	2.25b	2.25b	2.25b
IA64_INST_RETIRED	4.68b	10.2b	10.2b	10.2b	10.2b	2.23b	2.23b	2.23b	2.23b
false predicated inst.	0.09b	5.3b	5.3b	5.3b	5.3b	0.02b	0.02b	0.02b	0.02b
NOPS_RETIRED	2.73b	8.8b	8.8b	8.8b	8.8b	1.18b	1.18b	1.18b	1.18b
DISP_STALLED	0.06m	75.2m	75.6m	75.5m	75.6m	1.09b	1.09b	1.09b	1.09b

The relative overhead is 3.4, rather bigger than what is expected. Current overhead analysis schemes have problem to explain this overhead. Because the retired instruction number increased by a factor of over 8 (mostly NOP operations in this example), in Bull's ([6]) classification, this overhead should be conceptually identified as "additional computation", and other overheads are to be identified as "unidentified overhead". Using the method described in ([7]) to quantify "additional computation", by comparing assembly codes, can't answering why the retired instruction numbers differ so much. Even when this overhead can be obtained by using region profiling and excluding the runtime library cost, the programmer still have no idea about what causes this overhead. This case well exhibits the difficulties current overhead analysis faced.

A top-down query process based on the layered model can help to understand what happens. Since at the OpenMP language abstract layer, nothing can be found out to explain the overhead directly, we must look down to lower abstract layers. At the OpenMP implementation abstract layer, the OpenMP compiler will process the parallel do loop and produce code like the following(C style).

Code 2. Possible OpenMP compiler generated code for the loop in Example 1.

```
_rtl_static_init(1,100000000,1,1, /*lower,upper,stride,chunksize*/
                 &current_lower, &current_upper, &current_stride);

while (current_lower<=100000000)
{
    if (current_upper>100000000) current_upper = 100000000;
    for (local_i=current_lower; local_i<=current_upper; local_i++)
    {
        /* do loop content, will be pipelined by SWP */
    }
    current_lower += current_stride;
    current_upper += current_stride;
}
```

But knowledge at this level still can not explain the overhead measured at execution level. While OpenMP implementation level profiling can provide information about the schedule overhead, it can only tell that the main overhead lies in the while loop.

The direct overhead source lies at the language implementation level. At the language implementation layer, the inner most loop in Code 2 will be Software-pipelined by the compiler to pursuit ILP, and at the execution layer, when chunk size set to 1, the software-pipelined loop doesn't bring expected performance since the loop only runs for one iteration (tripcount=1). The SWP needs more iterations to get full profit.

This suggests that back to the OpenMP language level, the schedule chunksize is the original overhead source. Figure 2 depicts the cycle counts of the above program with the chunk size varying from 1 to 256 by power 2. The result shows,

with chunk size set to 128, the performance lost against best case (using static even schedule) is about 10% (0.22b vs. 0.2b).

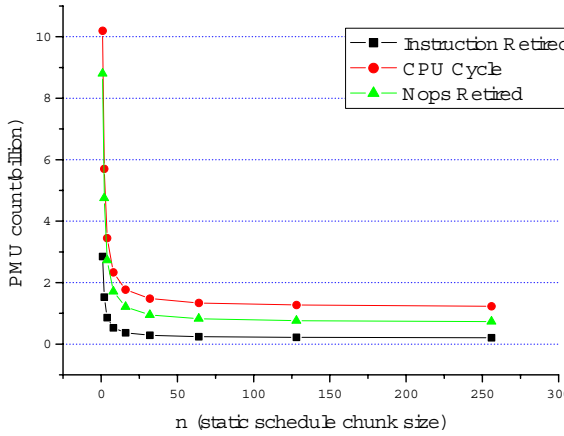


Fig. 2. Execution cycle count decrease when using larger chunk size for schedule

To associate the direct overhead source at language implementation level with the original overhead source at OpenMP language level, we can answer the performance questions at OpenMP language level instead of presenting the direct overhead reason at low level. This example also shows that when the thread task granularity is too small, ILP profits will be eliminated, and in the case of OpenMP, we call this overlap between ILP and TLP since OpenMP is generally used to explore parallelism at loop level.

5 A New Overhead Classification Scheme

For OpenMP programmer, three basic issues in OpenMP programming are parallelism identification, parallel task distribution/scheduling and synchronization. Thus OpenMP directives can be naturally grouped, and the classification of overheads in previous work can be reorganized to associate with them. The idea can be illustrated in Figure 3.

The overhead are broken down according to OpenMP construct elements in a non-intersectant way, and the mapping between OpenMP construct elements and low level overhead is one-to-many. The overhead measurement process is directed by high level instructions and performance explanations are summaries from lower abstract layers up to OpenMP language layer.

The classification is still in a hierarchical way (also note that the following classification is only to demonstrate the idea and many sub-categories are not

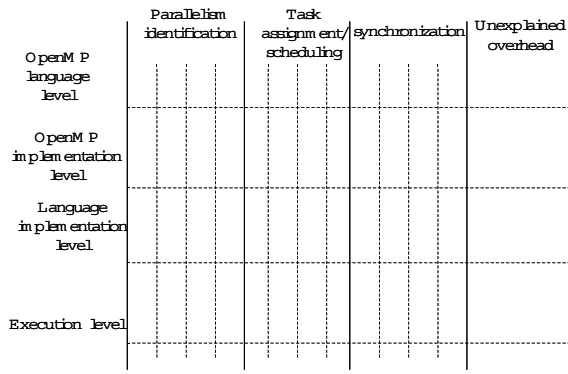


Fig. 3. Extended Overhead classification based on layered model

listed). All the overheads are defined for regions in the programs. Note that we distinguish static regions in source code with dynamic regions in the execution.

5.1 Parallelism Identification

Parallel Region construct and Work-Sharing constructs (and Combined Parallel Work–Sharing constructs, and associated data environment constructs) fall into this group, but not include the SCHEDULE subclause. They identify and enable the parallelism of code regions, but don't specify exact tasks executed by threads. Two categories of Overhead are in this group.

- Fork–join threads, initialization and combining cost of Reduction, firstprivate, lastprivate, and threadprivate associated overhead
- Implicit synchronization associated with these constructs
- Some sub-categories of data movement caused by program transformation
- Loss of Parallelism
  - Unparallelized code region
  - Replicated code region
  - Partial parallelized code region, and ordered clause

We take the concept of [9] that the parallelism should be checked in Runtime regions rather than in static code region because multiple static code regions may run in parallel (e.g., the effect of NOWAIT subclause). Runtime regions always consist of one or more static code regions.

Overheads in this group suggest that more parallelism is needed, either by specifying NOWAIT or identifying new Work-Sharing constructs.

5.2 Parallel Task Assignment/Scheduling

Any task assignment and schedule directives/subclauses falls into this group. For OpenMP, it mainly includes SCHEDULE clause. This kind of constructs specify the tasks executed by each thread, and it brings many effects on the execution behavior.

Overheads produced by this group include:

- Scheduling
- Some sub-categories of data movement, caused by schedule
- Load imbalance caused by schedule
- Impact on ILP

Note that data-movement overhead is also presented in this group.
Overheads in this group suggest that an better schedule scheme may exist.

5.3 Synchronization Clause

This includes all the synchronization clauses.

It contains the synchronization overhead.

- Synchronization overhead
  - locks
  - barriers
  - atomic operations

This classification is mainly a reorganization of previous identified overheads, and a few more categories are added. But the benefits is obvious in that it provides the programmer a way to tuning their OpenMP programs in a "recipe" style. For the example discussed above, the overhead produced is categorized into scheduling class and thus related with the schedule scheme. In this way, the overhead analysis can remind the programmer that there are something wrong with the schedule parameters, and that's just the original overhead source.

6 Conclusion and Future Work

A layered model can help us understand the problem of overhead analysis clearer and organize the overhead classification in a better way. And the example, the TLP and ILP overlap in OpenMP programs, not only raises a new category of overhead, but also provides us a case to study how to explain performance using the layered model. The layered model and the case study shows that an extension to current overhead classification is needed to help the OpenMP programmer to do performance tuning in a more direct way. This may further simplifies the work for OpenMP performance tuning and we also hope that the work can be used by parallel compilers to generate more efficient OpenMP code.

The work is based on our previous work of enabling the OpenMP processing ability for ORC. A new project of adding OpenMP profiling support to ORC using perflib is ongoing, as a base to support the Overhead Analysis described above.

Acknowledgment

The work is supported by Intel's research funding, and partly supported by HP's gelato project. And at here, the advices given by reviewers should be sincerely acknowledged.

References

1. Performance monitor for IA64. `http://www.hpl.hp.com/research/linux/perfmon/`
2. Intel Inc.: Intel Itanium 2 Processor Reference Manual for Software Development and Optimization (June 2002)
3. Open Research Compiler. `http://ipf-orc.sourceforge.net`.
4. G. D. Riley, J. M. Bull, J. R. Gurd: Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics. In Proc. of 11^th Supercomputing (July 1997) 36–43
5. J. M. Bull: Measuring Synchronisation and Scheduling Overheads in OpenMP. In Proc. of First European Workshop on OpenMP(EWOMP1999) (September 1999) 99–105.
6. J. M. Bull: A hierarchical classification of overheads in parallel programs. In Proc. of First IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems (March 1996) 208–219
7. M. K. Bane, G. D. Riley: Automatic Overheads Profiler for OpenMP Codes. In Proc. of the Second European Workshop on OpenMP (EWOMP2000)(September 2000)
8. Hong-Linh Truong, Thomas Fahringer: SCALEA: A Performance Analysis Tool for Distributed and Parallel Programs. In Proc. of the 8^th International EuroPar Conf. (August 2002)
9. M. K. Bane: Extended Overhead Analysis for OpenMP. In Proc. of the 8^th International EuroPar Conf. (August 2002)
10. M. E. Crovella, T. J. LeBlanc: Parallel Performance Prediction Using Lost Cycles Analysis. In Proc. of 8^th Supercomputing (1994) 600–610
11. J. K. Hollingsworth: Finding Bottlenecks in Large Scale Parallel Programs. Doctor Thesis, Department of Computer Science, University of Wisconsin-Madison. (1994)
12. R. B. Irvin: Mechanisms for Mapping High-Level Parallel Performance Data. In Proc. of the ICPP Workshop on Challenges for Parallel Processing. (August 1996)
13. J. Kohn, W. Williams: ATExpert. Journal of Parallel and Distributed Computing, Vol. 18 (1993) 205–222.
14. Matthias Müller: Some simple OpenMP optimization techniques. In the Workshop on OpenMP Applications and Tools (WOMPAT 2001) (July 2001)

Supporting Realistic OpenMP Applications on a Commodity Cluster of Workstations

Seung Jai Min, Ayon Basumallik, and Rudolf Eigenmann*

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
http://www.ece.purdue.edu/ParaMount
{smin,basumall,eigenman}@ecn.purdue.edu

Abstract. In this paper, we present techniques for translating and optimizing realistic OpenMP applications on distributed systems. The goal of our project is to quantify the degree to which OpenMP can be extended to distributed systems and to develop supporting compiler techniques. Our present compiler techniques translate OpenMP programs into a form suitable for execution on a Software DSM system. We have implemented a compiler that performs this basic translation, and we have proposed optimization techniques that improve the baseline performance of OpenMP applications on distributed computer systems. Our results show that, while kernel benchmarks can show high efficiency for OpenMP programs on distributed systems, full applications need careful consideration of shared data access patterns. A naive translation (similar to the basic translation done by OpenMP compilers for SMPs) leads to acceptable performance in very few applications. We propose optimizations such as *computation repartitioning*, *page-aware optimizations*, and *access privatization* that result in average 70% performance improvement on the SPEC OMPM2001 benchmark applications.

Keywords: OpenMP Applications, Software Distributed Shared Memory, benchmarks, performance characteristics, optimizations.

1 Introduction

OpenMP [1] has established itself as an important method and language extension for programming shared-memory parallel computers. While OpenMP has clear advantages on shared-memory platforms, message passing is today still the most widely-used programming paradigm for distributed-memory computers. In this paper, we explore the suitability of OpenMP for distributed systems as well.

* This material is based upon work supported in part by the National Science Foundation under Grant No. 9703180, 9975275, 9986020, and 9974976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 170–179, 2003.

Our basic approach is to use a Software DSM (Distributed Shared Memory) system, which provides the view of a shared address space on top of a distributed-memory architecture. To this end, we have implemented a compiler that transforms OpenMP programs into Treadmarks Software DSM programs [2]. This paper makes the following specific contributions.

- We describe our compiler infrastructure that can translate OpenMP applications into Software DSM programs.
- We measure the baseline performance of the application-level SPEC OMPM2001 benchmarks on a distributed memory system, and we analyze the performance behavior.
- We present optimization techniques that enhance the baseline performance of real OpenMP applications on distributed memory system.

Our work is closely related to the following projects. In [3], the authors implemented OpenMP on a network of shared memory multiprocessors and showed their performance using a subset of the SPLASH-2 and NAS benchmark suites, without the help of compiler optimization. In [4], compiler optimizations have been introduced for OpenMP programs on Software DSM. Several recent papers have proposed language extensions. For example, in [5–7], the authors describe data distribution directives similar to the ones designed for High-Performance Fortran (HPF) [8].

From these related efforts, we found that, while results from small kernel programs have shown promising performance, little information on the behavior of realistic OpenMP applications on Software DSM systems is available. In this paper, we show how application-level benchmarks performs on Software DSM and propose optimization techniques to improve the speedups. Also, the optimization techniques that we are presenting in this paper use standard OpenMP as input and do not rely on the user's data distribution input.

The paper is organized as follows. Section 2 will present basic compiler techniques for translating OpenMP into Software DSM programs. Section 3 will discuss the performance behavior of such programs. Section 4 will present advanced optimizations. In Section 5, we will quantitatively evaluate our proposed techniques, followed by conclusions in Section 6.

2 Translating OpenMP Applications into Software DSM Programs

In the transition from shared-memory to distributed systems, the major challenge is that each participating node now has its own private address space, which is not visible to other nodes. This difference in address spaces affects the OpenMP translation. In case of SMPs, implementing OpenMP shared variables is straightforward, because all variables are accessible by all threads. By contrast, in Software DSM systems, all variables are private by default, and shared data has to be explicitly allocated as such. This creates a challenge for the translation of most OpenMP programs, where variables are shared by default. To address

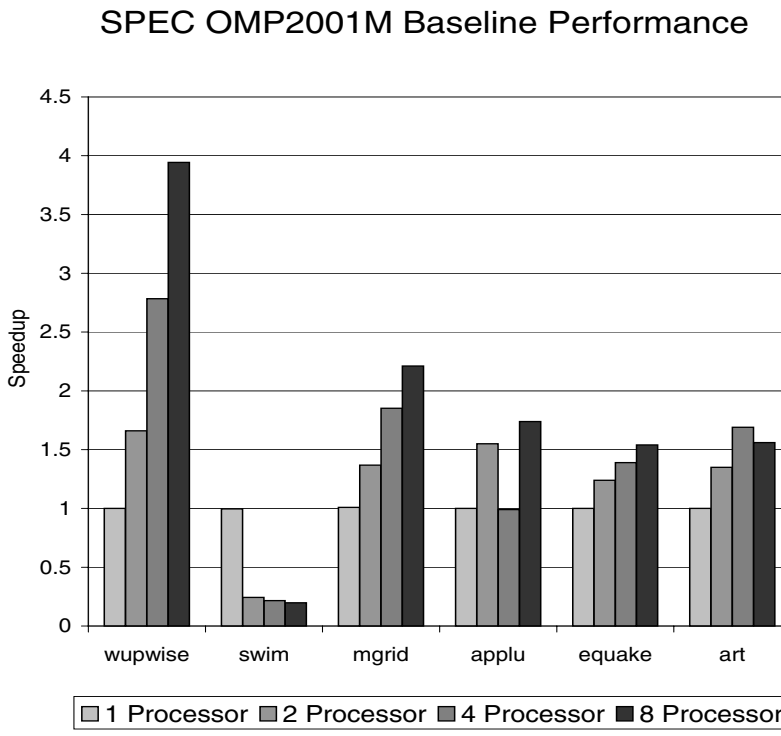


Fig. 1. Baseline performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines

this challenge, we have developed a compiler infrastructure that performs the following translations. First, our compiler converts the OpenMP application into a micro-tasking form [9]. Second, it converts OpenMP shared data into the form necessary for Software DSM systems. Also, OpenMP shared data may include subroutine-local variables, which reside on the process stack. Stacks on one node are not visible to other Software DSM nodes. We have developed a compiler algorithm that identifies shared variables, using inter-procedural analysis, and changes their declaration to an explicit allocation in shared space.

3 Performance Evaluation of Real Application Benchmarks

In previous work [10], we measured the performance of kernel programs to estimate an upper bound for the performance of OpenMP applications on our system. We also used micro-benchmarks to quantify the performance of specific OpenMP constructs. In this section, we describe and discuss the measurements carried out on the baseline performance of real-application benchmarks on our

cluster. The SPEC OMPM2001 suite of benchmarks [11] consists of realistic OpenMP C and Fortran applications.

To summarize our measurements, we note that a naive transformation of realistic OpenMP applications for Software DSM execution does not give us the desired performance. The baseline performance is illustrated in Figure 1. We translated four SPEC OMPM2001 Fortran programs (WUPWISE, SWIM, MGRID, APPLU) using our compiler and two SPEC OMPM2001 C (ART, EQUAKE) programs by hand. We evaluated the performance on a commodity cluster consisting of PentiumII/Linux nodes, connected via standard Ethernet networks. These benchmark programs are known to exhibit good speedups on shared memory systems [11]. However, their performance on Software DSM systems shows different behavior. For Fortran applications, WUPWISE, and MGRID exhibit speedups, whereas the performance of SWIM and APPLU degrades significantly, as the number of processors increases. Evidently, porting well-performing shared-memory programs to Software DSM does not necessarily lead to uniformly good efficiency. A large part of this performance degradation is due to the fact that real-application benchmarks have complex memory access patterns, which causes expensive shared memory activity on Software DSM. In Section 4, we will analyze the causes for performance degradation further and propose optimization techniques for OpenMP programs on Software DSM systems.

4 Advanced Optimizations

The baseline translation of SPEC OMPM2001 programs, described in Section 3, shows that converting shared-memory programs to Software DSM programs requires optimization. Software DSM implementations have shown acceptable performance on kernel programs [2]. However, kernels differ from realistic shared memory applications in two essential ways: (1) in terms of shared data access patterns and (2) in terms of the size of the shared data space.

A realistic application typically consists of several algorithms that access the shared data in different ways. These access patterns may result in increased message traffic in the underlying Software DSM layer, which is expensive from a performance viewpoint. Kernel programs do not exhibit the complex access patterns of full-sized applications and thus do not bring out these additional costs. Secondly, as the size of shared data is increased, we observed that the coherence and update traffic increased significantly. Typical realistic shared memory applications, such as the SPEC OMPM2001 applications, may have data sets that are in the order of gigabyte.

To address the above issues, we have implemented optimizations that fall into three categories.

- Computation repartitioning for locality enhancement
- Page aware optimization techniques
- Shared data space reduction through privatization

4.1 Computation Repartitioning

```
!$OMP PARALLEL DO
      DO k=2, nz-1
       DO i=ist, iend
        DO j=jst, jend
         DO m=1, 5
          rsd(m,i,j,k)=...
          ...
         ENDDO
        ENDDO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
      DO j=jst, jend
       DO i=ist, iend
        DO k=2, nz-1
         DO m=1, 5
          rtmp(m,k)=rsd(m,i,j,k)-...
          ...
         ENDDO
        ENDDO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL DO
```

(a) original code

```
!$OMP PARALLEL
      DO k=2, nz-1
       DO i=ist, iend
!$OMP DO
        DO j=jst, jend
         DO m=1, 5
          rsd(m,i,j,k)=...
          ...
         ENDDO
        ENDDO
!$OMP ENDDO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
      DO j=jst, jend
       DO i=ist, iend
        DO k=2, nz-1
         DO m=1, 5
          rtmp(m,k)=rsd(m,i,j,k)-...
          ...
         ENDDO
        ENDDO
       ENDDO
      ENDDO
      ...
!$OMP END PARALLEL
```

(b) after computation repartitioning

Fig. 2. Computation repartitioning: Subroutine RHS from APPLU

Page-based Software DSM systems implement consistency by exchanging information at the page level. Between synchronization points, the participating nodes exchange information about which nodes wrote into each page. A node that writes to a page in shared memory thus becomes the temporary *owner* of that particular page. A page could have multiple temporary *owners* if there are multiple nodes writing to the same page between synchronization points. The way this ownership changes during the program may significantly affect the execution time for the application.

For example, the main loop in APPLU contains seven parallel DO-loops. All these parallel DO-loops access a shared array *rsd(m,i,j,k)*. Five of these seven parallel DO-loops partition array *rsd* using the outer most index *k*. One loop does block partitioning, using both *i* and *j* indices and the other partitions using the *j* index. Thus, the main loop of APPLU has four access pattern changes per every loop iteration. Figure 2 illustrates the change in access patterns between two of these loops. This access pattern change will incur a large number of remote node requests in the second parallel loop. To avoid inefficient access patterns, the program needs to be selective about which nodes touch which portions of the

data. For example, the code may have a consistent access pattern across loops, if the inner *j*-loop is partitioned instead of the outermost *k*-loop in the first loop nest. Figure 2 (b) shows the resulting code after *computation repartitioning*.

This optimization requires the compiler's ability to detect further parallelism in the loop nest. We used the Polaris parallelizing compiler for this purpose [12]. However, not all loop nests allow this optimization because some inner loops cannot be parallelized. We applied various techniques, such as adding redundant computation, to enable *computation repartitioning* throughout the whole program.

4.2 Page Aware Optimizations

```
!$OMP PARALLEL DO                    !$OMP PARALLEL DO
      DO 200 J=1,N                         DO 200 J=1,N
      DO 200 I=1,M                         DO 200 I=1,M
      UNEW(I+1,J) = ...                    UNEW(I+1,J) = ...
      VNEW(I,J+1) = ...                    VNEW(I,J+1) = ...
      PNEW(I,J)   = ...                    PNEW(I,J)   = ...
  200 CONTINUE                         200 CONTINUE
!$OMP END PARALLEL DO                !$OMP END PARALLEL DO

                                     !$OMP PARALLEL DO
      DO 210 J=1, N                        DO 210 J=1, N
      UNEW(1,J) = UNEW(M+1,J)              UNEW(1,J) = UNEW(M+1,J)
      VNEW(M+1,J+1) = VNEW(1,J+1)          VNEW(M+1,J+1) = VNEW(1,J+1)
      PNEW(M+1,J) = PNEW(1,J)              PNEW(M+1,J) = PNEW(1,J)
  210 CONTINUE                         210 CONTINUE
                                     !$OMP END PARALLEL DO
      (a)original code               (b)after page aware optimization
```

Fig. 3. Page aware optimization: Subroutine CALC2 from SWIM

Page-aware optimizations use the knowledge that the Software DSM maintains coherence at the page granularity. We will describe two types of *page-aware optimizations*. First, we transform a shared array by *padding*, so that array partitioning across nodes places the partitions at page boundaries. For example, consider an array *U(m,i,j,k)* of size *U(5,61,61,60)*. The array type is double precision (size of a double precision number is 8 bytes in our system). If the compiler partitions this array U using the index j in the parallel region, then padding the first and the second dimension of U will produce *U(8,64,61,60)*. After padding, the size of the shared array U increases slightly. However, the boundaries of partitioned array chunks are now aligned with the page boundaries. This optimization reduces false sharing around the boundaries of partitioned shared data chunks.

The second *page-aware optimization* deals with the page shape. In a column-major language such as Fortran, a process that writes a column in a two dimensional array will touch much fewer pages than a process that writes a row. As

an example, let A be a 2-D array of size 1024x1024 and its elements are 4 bytes integers. If the size of the page in Software DSM is 4 KB, then each column of A can be mapped to a page. Thus, there are 1024 pages occupying corresponding 1024 columns in A. If a node writes a column in A, then only one page is affected. On the other hand, writing a single row in A touches all the pages owned by all the participating nodes. This scenario is illustrated in Figure 3. In this figure, there is an OpenMP parallel loop followed by a serial loop. In the parallel loop, each node writes to its partitioned blocks of the shared arrays and thus temporarily owns the pages in its partition. Then the master node copies a single row to another row for each shared array in the serial loop and in effect, touches all the pages for these shared arrays. Subsequently, when these shared arrays are read by the child nodes, each child node has to request updates from the master node. This incurs substantial overhead, which can be avoided if the second loop is executed in parallel. In the original benchmark code, the second loop is a serial loop, even though it can be parallelized. This is because parallelizing a small loop is not always profitable in shared memory programming. Thus, this optimization highlights a difference of optimization strategy between shared and distributed memory environments.

4.3 Privatization Optimization

This optimization is aimed at reducing the size of the shared data space that must be managed by the Software DSM system. Potentially, all variables that are read and written within parallel regions are shared variables and must be explicitly declared as such. However, we have found that a significant number of such variables are "read-only" within the parallel regions. Furthermore, we have observed that, for certain shared arrays, different nodes read and write disjoint parts of the array. We refer to these variables as *single-owner* data. In the context of the OpenMP program, these are shared variables. However, in the context of a Software DSM implementation, instances of both can be privatized with certain precautions. The first benefit of privatization stems from the fact that access to a private variable is typically faster than access to a shared variable, even if a locally cached copy of the shared variable is available. This is because accesses to shared variables need to trigger certain coherency and bookkeeping actions within the Software DSM. The second important benefit of privatization is the effect on eliminating false sharing. The overall coherency overhead is also reduced because coherency has to be now maintained for a smaller shared data size.

5 Results

We applied the described optimizations by hand to several of the SPEC OMPM2001 benchmarks and achieved marked performance improvements. Figure 4 shows the final performance obtained by the baseline translation and subsequent optimizations. We present the speedups for four Fortran codes (WUPWISE, SWIM, MGRID, APPLU) and two C benchmarks (EQUAKE, ART).

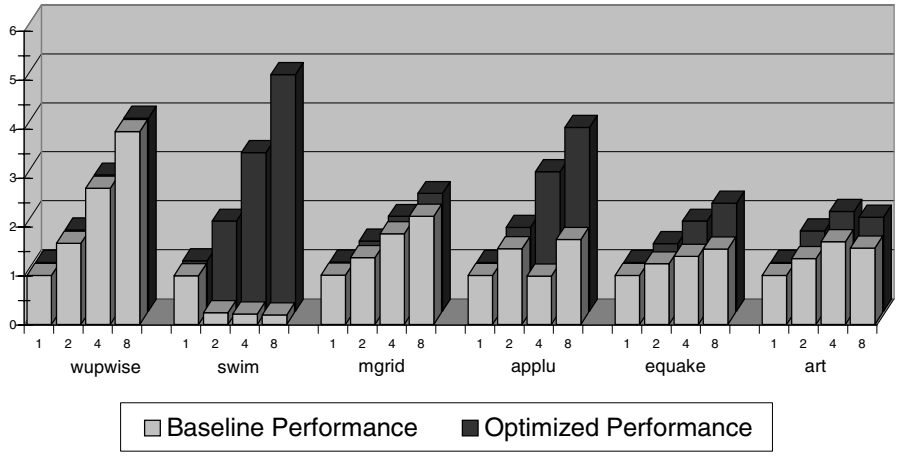


Fig. 4. Performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines

In one of the Fortran codes, WUPWISE, the baseline translation already had acceptable speedup, and so we did not apply further optimizations. For the three other Fortran codes, SWIM, MGRID, and APPLU, we obtained significant performance improvements with *computation repartitioning* and *page-aware optimizations*. In SWIM, application of the *page-aware optimization* shown in figure 3 improved the performance dramatically. We slightly enhanced the performance of MGRID by applying *computation repartitioning*. APPLU, which shows one of the most complex access patterns among the Fortran applications, has been optimized using both *computation repartitioning* and *page-aware optimizations*. The baseline of APPLU performs better with two than with four processors. In this application, the shared array *rsd(m,i,j,k)* is block partitioned using both i and j indices so that *rsd* is partitioned according to the index j on two processors, and is further partitioned using the index i on four processors, and again using the index j on eight processors and so on. Partitioning using the index i results in multiple nodes writing to a single page and thus causes false sharing. Therefore, whenever *rsd* is partitioned according to the inner index i, it suffers a performance drop owing to the increased false sharing. Since the optimized version for APPLU always partitions *rsd* using the index j, it not only avoids this inconsistent speedup, but also shows much better overall performance.

For the C applications, ART showed improved performance after privatizing several arrays that were not declared as private in the original code. In EQUAKE, we derived benefit from making certain parallel loops dynamically scheduled, though the original OpenMP directives specified static scheduling.

The average baseline speedup of six applications is 1.87 and the average optimized performance is 3.17 on 8 processors. Thus, our proposed optimizations, *computation repartitioning*, *page-aware optimizations*, and *access privatization* result in average 70% performance improvement on our SPEC OMPM2001 benchmarks.

6 Conclusions

In this paper, we have described our experiences with the automatic translation and further hand-optimization of realistic OpenMP applications on a commodity cluster of workstations. We have demonstrated that, for these applications, the OpenMP programming paradigm may be extended to distributed systems. We have discussed issues arising in the automatic translation of OpenMP applications for Software DSM. We have then presented several program optimizations for page-based Software DSM systems. In our performance studies, we have found that a baseline compiler translation, similar to the one used for OpenMP on SMP machines, yields speedups for some of the codes but unacceptable performance for others. After applying the proposed optimizations - *computation repartitioning*, *page-aware optimizations*, and *access privatization* - we observed significant improvements in performance. In the next phase of our project, we intend to use explicit message-passing in conjunction with Software DSM and investigate the effects of our optimizations in this hybrid approach.

References

1. OpenMP Forum, "OpenMP: A Proposed Industry Standard API for Shared Memory Programming," Tech. Rep., Oct. 1997.
2. S. Dwarkadas P. Keleher H. Lu R. Rajamony W. Yu C. Amza, A.L. Cox and W. Zwaenepoel, "Treadmarks: Shared memory computing on networks of workstations," *IEEE Computer*, vol. 29, no. 2, pp. 18–28, February 1996.
3. H. Lu, Y. C. Hu, and W. Zwaenepoel, "OpenMP on network of workstations," in *Proc. of Supercomputing'98*, 1998.
4. Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi, "OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP," in *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001)*, July 2001.
5. R. Crowell Z. Cvetanovic J. Harris C. Nelson J. Bircsak, P. Craig and C. Offner, "Extending OpenMP for NUMA Machines," in *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC 2000)*, November 2000.
6. V. Schuster and D. Miles, "Distributed OpenMP, Extensions to OpenMP for SMP Clusters," in *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT'2000)*, July 2000.
7. T.S. Abdelrahman and T.N. Wong, "Compiler support for data distribution on NUMA multiprocessors," *Journal of Supercomputing*, vol. 12, no. 4, pp. 349–371, oct. 1998.

8. High Performance Fortran Forum, "High Performance Fortran language specification, version 1.0," Tech. Rep. CRPC-TR92225, Houston, Tex., 1993.
9. M. Booth and K. Misegades, "Microtasking: A New Way to Harness Multiprocessors," *Cray Channels*, pp. 24–27, 1986.
10. Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann, "Towards OpenMP execution on software distributed shared memory systems," in *Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI'02)*. May 2002, Lecture Notes in Computer Science, 2327, Springer Verlag.
11. Rudolf Eigenmann Greg Gaertner Wesley B. Jones Vishal Aslot, Max Domeika and Bodo Parady, "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in *Proc. of WOMPAT 2001, Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science, 2104*, July 2001, pp. 1–10.
12. William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu, "Parallel programming with Polaris," *IEEE Computer*, pp. 78–82, December 1996.

OpenMP Runtime Support for Clusters of Multiprocessors

Panagiotis E. Hadjidoukas, Eleftherios D. Polychronopoulos, and Theodore S. Papatheodorou

High Performance Information Systems Laboratory (HPCLAB)
Department of Computer Engineering and Informatics
University of Patras, Rio 26500, Patras, Greece
http://www.hpclab.ceid.upatras.gr
{peh,edp,tsp}@hpclab.ceid.upatras.gr

Abstract. This paper presents a prototype runtime system, providing support at the backend of the NANOS OpenMP compiler, that enables the execution of unmodified OpenMP Fortran programs on both SMPs and clusters of multiprocessors, either through the hybrid programming model (MPI+OpenMP) or directly on top of Software Distributed Shared Memory (SDSM). The latter is feasible by adopting a share-everything approach for the generated by the OpenMP compiler code, which corresponds to the "default shared" philosophy of OpenMP. Specifically, the user-level thread stacks and the Fortran common blocks are allocated explicitly, though transparently to the programmer, in shared memory. The management of the internal runtime system structures and of the fork-join multilevel parallelism is based on explicit communication, exploiting however the shared-memory hardware of the available SMP nodes whenever this is possible. The modular design of the runtime system allows the integration of existing unmodified SDSM libraries, despite their design for SPMD execution.

1 Introduction

The OpenMP Application Programming Interface [22] provides a simple and portable model for programming a wide range of parallel applications on parallel platforms. Despite the several implementations of OpenMP on small-scale SMP servers and scalable ccNUMA multiprocessors, only few of them have been presented for clusters of multiprocessors [10,13,16,26]. These implementations, though, target a specific SDSM library, focus on the optimizations to the SDSM library/protocol and translate the OpenMP program into SPMD execution. Most of them do not use native Fortran compilers at the back-end and differentiate the output of the OpenMP compiler for shared and distributed memory machine. Furthermore, these implementations do not support multilevel parallelism and lack of sophisticated runtime systems that implements a two-level thread model on both architectural platforms.

This paper presents a prototype runtime system, providing support at the backend of the NANOS OpenMP compiler [1], that enables the execution of

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 180–194, 2003.

unmodified OpenMP Fortran programs on both SMPs and clusters of multiprocessors, either through the hybrid (MPI+OpenMP) programming model [5] or directly on top of Software Distributed Shared Memory. This is feasible by adopting a share-everything approach for the generated by the OpenMP compiler code, which corresponds to the "default shared" philosophy of OpenMP. Specifically, the user-level thread stacks and the Fortran common blocks are allocated explicitly, though transparently to the programmer, in shared memory. The management of the internal runtime system structures and of the fork-join multilevel parallelism is based on explicit communication, exploiting however the shared-memory hardware of the available SMP nodes whenever this is possible. The runtime system supports OpenMP through the implementation of an appropriate API rather than by targeting a specific DSM system. Its modular design allows the integration of several SDSM libraries, despite their design for SPMD execution. We focus on the functionality of our runtime system, taking into account a portable and modularity design for both architectural platforms, avoiding thus platform specific optimizations.

The rest of this paper is organized as follows: Section 2 outlines the general design and architecture of the proposed runtime system. Section 3 presents our general approach for supporting OpenMP execution on both shared memory multiprocessors and clusters of SMPs. Experimental evaluation with OpenMP programs on both shared and distributed memory machines is reported in Section 4. Related work is presented in Section 5. We discuss our on-going research in Section 6.

2 OpenMP Runtime Library

The proposed runtime library (RTL) implements the Nanothreads programming model [24], a fork-join dependence-driven execution model that allows the efficient exploitation of multiple levels of parallelism. The RTL exports the NANOS API [21], which is targeted by the NanosCompiler, a parallelizing compiler that captures the parallelism expressed by the user through OpenMP directives and the parallelism automatically detected through a detailed analysis of data and control dependences. The front-end of the NanosCompiler converts programs written in Fortran77 that use OpenMP directives to equivalent programs with calls to the RTL. As we demonstrate in this paper, the NanosCompiler can be also used for mixed mode (MPI+OpenMP) programs. Optionally, for execution on top of SDSM, the output code can be passed through a translator that ensures the allocation of common blocks in shared virtual memory. The resulted code is finally compiled by a native compiler and linked with the RTL, the thread package, the SDSM and the communication library.

2.1 Design

Figure 1 illustrates the modular design of our runtime system. The underlying thread package provides to the runtime system a well-defined interface for non-preemptive user-level threads, originally described in [9].

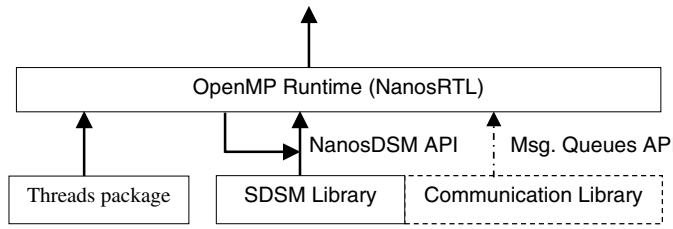


Fig. 1. Modular design

The SDSM library exports the NanosDSM API, which provides mechanisms (e.g. remote kernel thread creation, memory allocation, synchronization) for a fork-join parallel execution model, like OpenMP. If the underlying SDSM library does not provide such functionality, which is the common case, the runtime system implements internally this API on top of the SPMD execution model of the SDSM library. Finally, the communication library exports the Message Queues API [25], which is used by the runtime system for the management of its internal structures. Optionally, this API can be provided by the SDSM library.

2.2 Architecture

Figure 2 illustrates the general architecture of our runtime system on a cluster of multiprocessor nodes. On every node of the cluster, each DSM process consists of one or more virtual processors, which are DSM kernel threads, and a special server thread, the *Listener*, which is a network server that has access to the internal structures of the library on the specific node (process), without requiring access to the shared-data. The Listener is primarily responsible for the dependencies and queue management and actually implements much of the functionality that is required from the SDSM system (NanosDSM API). The virtual processors execute the application code, which may result in invocations of the SDSM protocol. Besides the intra-node queues, there are per-virtual processor inter-node ready queues and per-node global queues. The latter simulate a global distributed queue.

A significant design decision of our runtime system [8], which differs in the initial version of NanosRTL [21,17] and most user-level thread libraries for shared memory, is the adoption of a lazy stack allocation policy. In our case, the work descriptor of the runtime system is separated from its execution vehicle, i.e. an underlying user-level thread. This provides portability, low memory consumption and less data movement. Specifically, the lazy allocation policy is clearly beneficial since we avoid unnecessary invocations of the DSM protocol and thread migrations, caused by the stealing of newly created descriptors. This policy also minimizes the peak number of active stacks and leaves larger portions of the shared address space for the application data. Furthermore, a recycling mechanism is available for both the descriptors and the underlying threads.

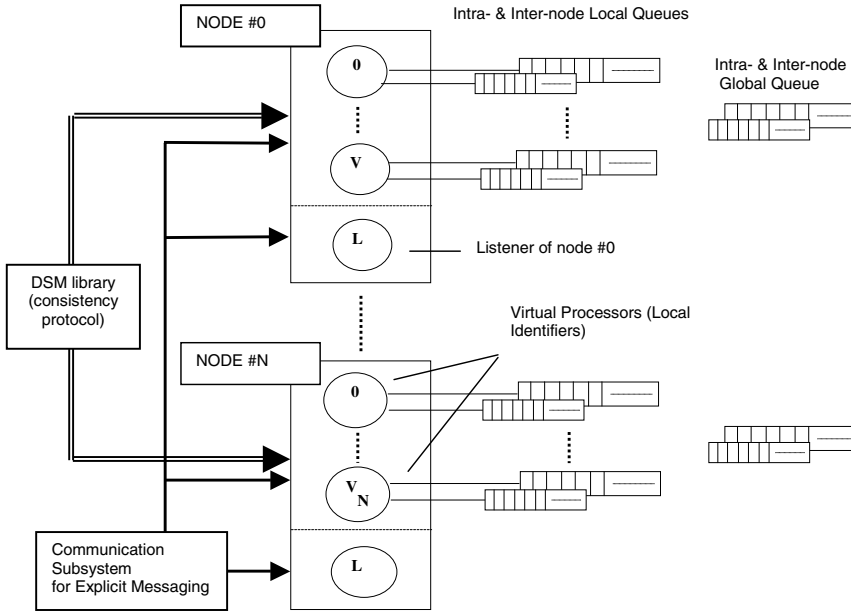


Fig. 2. General architecture

The architecture of queues enables the development of mechanisms that allow good load balancing and exploitation of data locality. There are per-virtual processor inter-node queues, while the global queue is distributed among all nodes and consecutive insertions into it result in the cyclic distribution of the descriptors among the available nodes. The insertion/stealing of a descriptor in/from a remote queue that resides in the same node is performed through hardware shared memory. Otherwise, the operations are performed with explicit messages to the Listener of the remote node.

On clusters of multiprocessors, the coherence of this dependence-driven model can be maintained exclusively through shared-memory, by updating the counter of unresolved dependencies through shared memory. However, this would require the allocation of the descriptors in shared virtual memory, necessary protection with distributed locks and would result in many invocations of the consistency protocol. A more efficient solution is the combination of hardware shared-memory with explicit messages: each descriptor is allocated in private memory and associated with an owner node (creator). If a thread is finished on its owner node, the dependencies of its successors will be updated directly through (hardware) shared memory. Otherwise, its descriptor will return to the owner node and any dependencies will be updated by the Listener.

2.3 Implementation Platforms

The underlying thread package provides very portable non-preemptive user-level threads based on the POSIX `setjmp-longjmp` calls. We have successfully imple-

mented the NanosDSM API and managed to provide OpenMP support on top of several existing software DSM page based systems: SVMLib [23] and Millipede [6] are page-based user-level SDSM libraries that run on Windows NT/2000 platforms, support DSM kernel threads and sequential consistency. Although only their binary release is available, we have managed to provide OpenMP execution on top of them. JIAJIA [11] runs on Unix and supports scope consistency, however it is not multithreaded. Finally Mome [14]runs on Unix, supports kernel threads, sequential and weak consistency, and provides several advanced features, like memory mapping, prefetching, and page manager migration. For JIAJIA and Mome the utilization of SVM stacks required the execution of their signal handlers on alternate signal stacks (`sigaltstack`).

Currently, the explicit communication in the runtime system can be based on UDP, TCP or MPI (compilation option). The latter is very close to the proposed Message Queues API, and its use is possible with the concurrent linking of the SDSM and the MPI libraries and the automatic setup of the appropriate environment variables.

3 OpenMP Execution

As aforementioned, the front-end of the NanosCompiler converts programs written in Fortran77 that use OpenMP directives to equivalent programs with calls to our runtime system. In order to be able to execute these shared-memory codes on software DSM, we must follow a shared-everything approach. One option is to have a SDSM library that provides full address space sharing, where everything, even the application code, is implicitly shared according to the SDSM protocol. On the other hand, existing SDSM libraries support private address spaces and the shared data must be explicitly allocated. We follow the second approach and we apply this level of sharing to the application data, which means the allocation of the user-level thread stacks and common blocks in shared memory.

3.1 Sharing the User-Level Thread Stacks

Current SDSM systems require the arguments of a function that represents inter-node parallelism to be either scalar values or pointers to DSM data. In order to provide true shared-memory functionality on distributed-memory machines the stacks of the user-level threads must be allocated in SVM. This sharing resolves many issues that arise because Fortran passes function arguments by reference.

Sharing the stacks may introduce excessive complexity regarding the false sharing of data and the interaction with relaxed consistency protocols. We believe that either the programmer should explicitly annotate shared data, or the compiler should detect which variables have to be shared. Moreover, the OpenMP programming model provides good hints concerning the privatization of data.

3.2 Sharing the Common Blocks

A Fortran77 programmer uses common blocks to declare global data, which are allocated in the heap. However, the language does not support directly explicit allocation of data. Since most SDSM libraries require an explicit allocation of shared data, common blocks of the output code of the NanosCompiler must be allocated in shared virtual memory. Currently, in order to allocate explicitly, though transparently, the common blocks of the OpenMP programs in shared virtual memory we use the following methods:

- **Pointer Statement**: We have developed a source-to-source translator that takes as input the NanosCompiler output code and associates a POINTER with any variable in a common block, injecting also appropriate memory allocation calls (`ndsmf_malloc`). Fortunately, most compilers support the POINTER statement. This solution can be applied for stack-based variables. Moreover, it provides automatic padding of variables. Complexity can be introduced only if a common variable is used in array declarations.
- **Memory Mapping**: In this case, the common block of the program is mapped in shared virtual memory, an operation that must be supported by the underlying SDSM library. Similar to the previous case, a slight transformation of the NanosCompiler output code is required: the appropriate padding is applied and then a mapping routine (`ndsmf_map_memory`) is injected. This approach can be also applied for mapping global variables of C programs in shared virtual memory. Another advantage of this solution is that it does not depend on the native compiler.
- **Dynamic Allocation of Common Blocks**: The Intel Fortran Compiler [12] provides an optimal solution that allows the direct execution of the NanosCompiler output code on cluster of SMPs, through its support for dynamic allocation of common blocks. On entry to each routine containing a declaration of the dynamic common block, a check is made of whether space for the common block has been allocated. If it is not yet allocated, space is allocated at the check time. The user can supply a memory allocation routine for the common blocks. Additionally, the compiler requires knowing the names of the common blocks that will be dynamically allocated (e.g. `-Qdyncomm"data"`). When a DSM process enters a subroutine that contains a common block that has not been allocated, the allocation routine is automatically invoked. The common block is allocated in shared virtual memory and the name-address pair in stored globally. If the allocation routine is called on another process for an already allocated common block, it simply returns the address of this common block (Figure 3).

In the Treadmarks SDSM library, the global data of a Fortran77 program is declared in a specifically named common block that is loaded in shared virtual memory. Since shared variables must reside on the shared heap, the OpenMP translator gathers all shared global variables in a structure, and allocates that structure on the shared heap. In the Cashmere-2L SDSM system [27], a special

```
void _FTN_ALLOC (void **mem, unsigned long *size, char *name){
   int pos;
   ndsm_lock (cb_table[0].CBLock);
   if (cb_is_allocated(name, &pos))
      *mem = cb_table[pos].cb_address;
   else {
      ndsm_malloc(mem, *size);
      cb_table[pos].cb_address = (unsigned long) *mem;
      strcpy(cb_table[pos].cb_name, name);
   }
   ndsm_unlock(cb_table[0].CBLock);
}
```

Fig. 3. Memory allocation routine

script parses the object files for common blocks and creates a small assembly file with definitions for each common block. These definitions ensure that the common block addresses fall within any address range required by the underlying protocol library. The main disadvantage of these solutions is that are compiler/linker specific.

Figure 4 presents a fragment of code (original) as generated by the NanosCompiler for the EPCC OpenMP synchronization benchmark, and how it is transformed according to the first two methods (POINTER statements and Memory Mapping).

3.3 Relaxing the Protocol

The sequential-consistency SDSM protocol for both the user-level stacks and the global shared data allows us to focus exclusively on the functionality of the OpenMP runtime. In this section, we outline our preliminary effort to use a relaxed consistency protocol in our OpenMP compilation environment. A thorough study of the interaction between the runtime system and the consistency protocol, as presented in [16], is beyond the scope of this paper.

Fortunately, OpenMP assumes a relaxed consistency protocol and determines appropriate rules where the data of the application should be coherent on all processors. This is performed either explicitly by the programmer using the `flush` directive or implicitly with the OpenMP constructs that assume a memory barrier. According to our approach, i.e. unmodified OpenMP directives for clusters of SMPs, the mapping of the consistency protocol of OpenMP on an existing SDSM protocol must be performed transparently, without requiring additional user effort at the application level.

The current weak consistency protocol of Mome is based on the explicit synchronization of shared memory (`MomeSynchronizeRegion`). Before entering and after exiting a parallel region or a critical section we flush the shared data. At the runtime system, an internal `ndsm_flush` operation enforces the synchronization of the shared heap on all nodes. If weak consistency is used for the stacks,

```
/* ORIGINAL CODE */
INTEGER nthreads
DOUBLE PRECISION time(0:50)
COMMON /data/ nthreads, time

/* POINTER STATEMENT */
INTEGER nthreads
DOUBLE PRECISION time(0:50)
COMMON /data/ pnthreads, ptime
POINTER (pnthreads,nthreads)
POINTER (ptime,time)
EXTERNAL ndsmf_malloc

IF (pnthreads.eq.0) THEN
   CALL ndsmf_malloc(pnthreads, 4*1)
ENDIF
IF (ptime.eq.0) THEN
  CALL ndsmf_malloc(ptime, 8*51)
ENDIF

/* MEMORY MAPPING */
INTEGER nthreads
DOUBLE PRECISION time(0:50)
COMMON /data/ pad1, nthreads, time ,pad2
INTEGER pad1(2048),pad2(2048)
EXTERNAL ndsmf_map_memory

CALL ndsmf_map_memory(nthreads,pad2(1))
```

Fig. 4. Allocating common blocks in shared virtual memory

the flush operation is also performed for the currently active user-level stack. To avoid deadlock, the synchronization primitive is called within a different user-level thread with private stack.

On JIAJIA, which supports only scope consistency, every critical section is protected with locks that control the coherence of the data. In this case, the `ndsm_flush` function is called transparently only for parallel regions and corresponds to `jia_barrier` calls on all nodes. Obviously, the barrier must be called from a user-level thread with private stack.

4 Experimental Evaluation

In this section, we present initial experimental results of our OpenMP environment on both target architectural platforms. Our experiments were performed on a quad-processor 200 MHz Pentium Pro system equipped with 512 MB of main memory and on a network of 2 dual-processor 866 MHz Pentium III nodes, with

256 MB of main memory and interconnected with Fast Ethernet. The operating system is Linux 2.4.19 and as native compilers, we have used the Intel C compiler for the runtime system and the Mome DSM library, and the Intel Fortran compiler for the applications. The latter supports both the POINTER statement and dynamic allocation of common blocks. For the experiments with the hybrid programming model, we have used the thread-safe MPIPro 1.6.3 library [19].

The underlying thread package provides very portable non-preemptive user-level threads based on the POSIX setjmp-longjmp calls. The communication subsystem is based on TCP sockets, implementing thus the Listener thread as a select-based server. Mome allows multiple instances of a DSM application on the same node. This feature enable us to perform experiments on the SMP machine, emulating a cluster of 4 uniprocessor nodes or 2 dual-processor nodes, that share however the network interface (loopback) and the main memory of the machine. In all experiments we have used the sequential consistency protocol of the SDSM library, unless otherwise specified.

4.1 Hardware Shared Memory

Figure 5 illustrates the overhead of our runtime system for the basic OpenMP synchronization constructs (**PAR**ALLEL, **DO**, **P**ARALLEL **DO**, **BAR**RIER and **RED**UCTION) on the SMP machine. The observed measurements exhibit similar behavior with those reported in [4]. However, on a single processor the overhead is relatively high because the current version of the NanosCompiler generates threaded code even for a single processor. It is important to mention that currently we have focused on the functionality of our runtime system, tar-

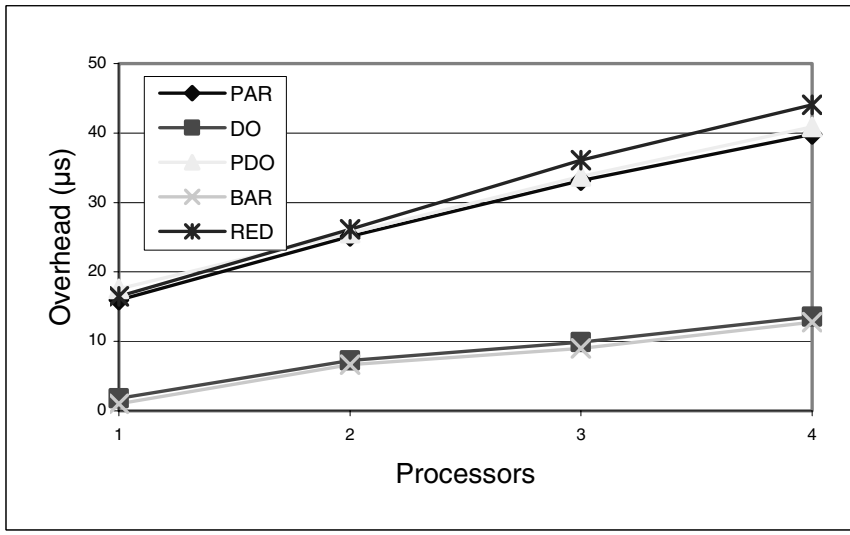


Fig. 5. OpenMP overheads on hardware shared memory

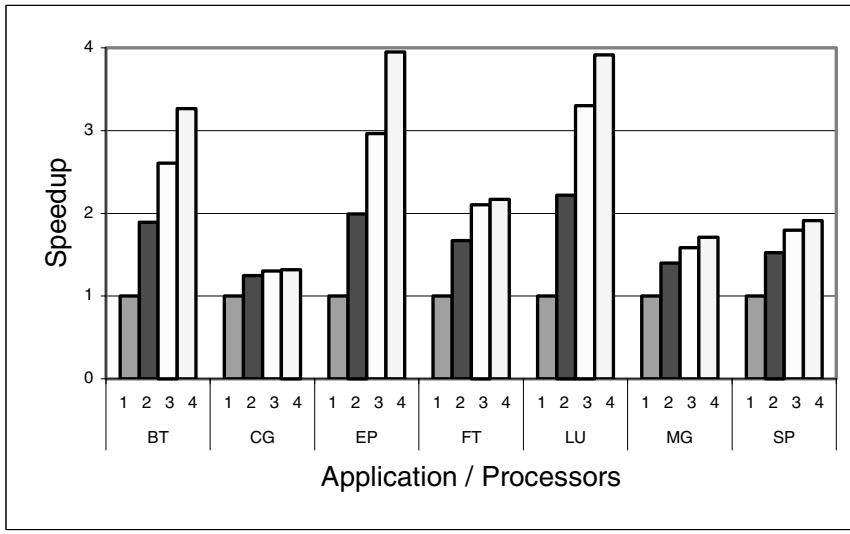


Fig. 6. Speedups on hardware shared memory

geting a portable, modular and general design for both shared and distributed memory machines and thus avoiding platform-specific optimizations.

Figure 6 depicts the measured speedups for the NAS Parallel Benchmarks suite (Class W) [15], parallelized with the OpenMP programming model. Our

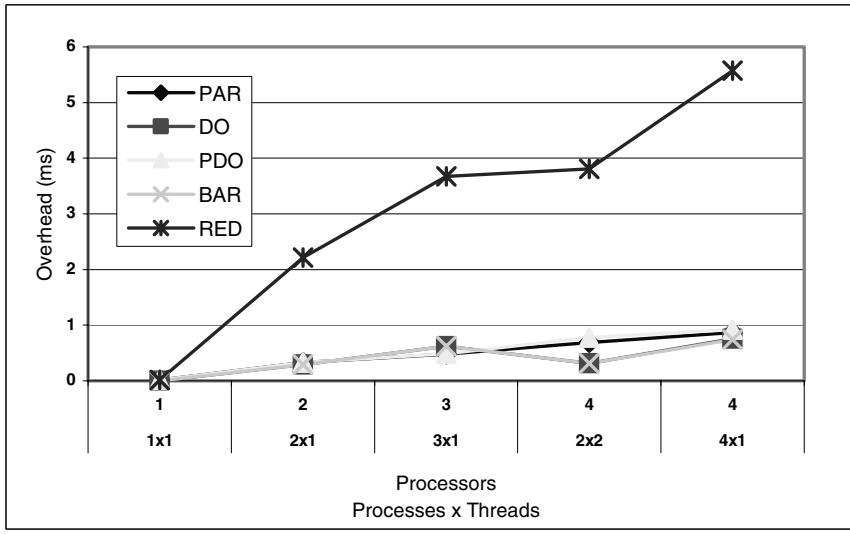


Fig. 7. OpenMP overheads on SDSM (SMP machine)

portable POSIX runtime system performs similarly or even better than our previous machine-dependent and highly optimized library running the NAS benchmarks on the same hardware [2]. The execution times of the applications on a single processor were: `BT:101.34`, `CG:16.56`, `EP:113.08`, `FT:9.68`, `LU:360.53`, `MG:12.79` and `SP:282.72` (in seconds).

4.2 Distributed Memory - SMP Machine

On SDSM, the measured OpenMP overheads (Figure 7) are in the order of milliseconds and agree with the numbers reported in [16] and [3]. Finally, Figure 8 depicts the measured speedups of three applications on top of our OpenMP compilation environment. Each column corresponds to the configuration (Processes x Threads) presented in Figure 7. The first application is the hybrid fine-grained implementation of NAS FT. The rest are the OpenMP version of NAS EP and the MD application from the official site of OpenMP [22], both executed on top of Mome. NAS EP is embarrassingly parallel and thus does not depends on the underlying SDSM protocol. Although MD scales linearly on shared memory, it fails to scale efficiently on SDSM, mainly due to its memory access patterns and the overhead introduced by the implicit movement of data. The degradation in performance is significantly higher when a sequential consistency protocol is used. For this reason, we use the weak memory consistency model of Mome for this application. The execution times on a single processor were: `FT-MIX:17.56`, `EP:113.08` and `MD:959.00` (in seconds).

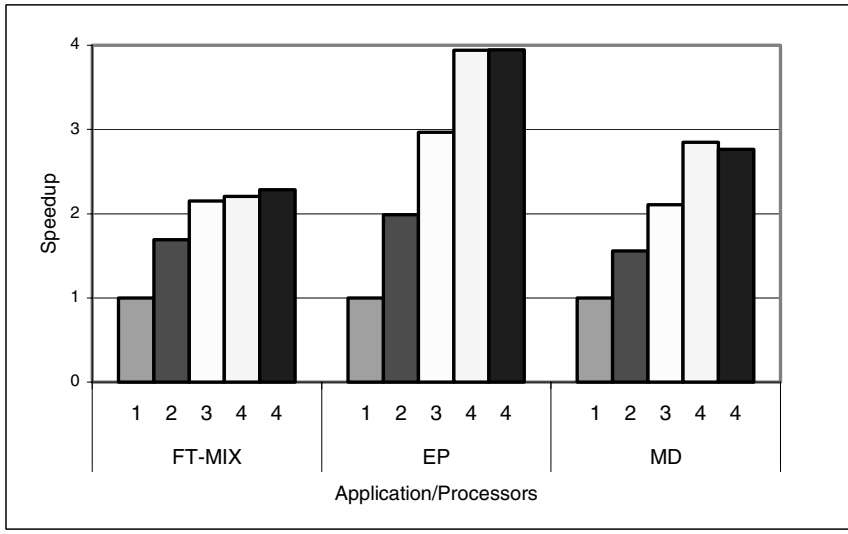


Fig. 8. Speedups on SDSM (SMP machine)

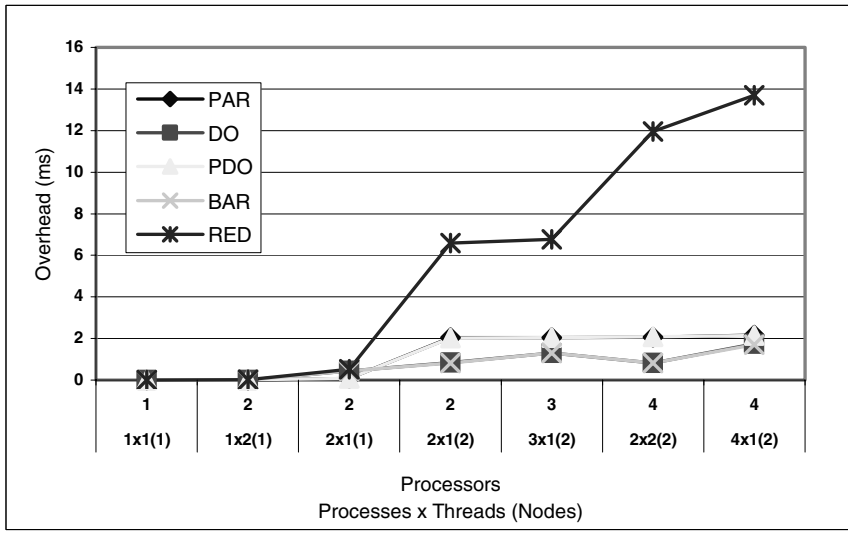


Fig. 9. OpenMP overheads on SDSM (SMP cluster)

4.3 Distributed Memory - Cluster of SMPs

The results presented in Figures 9 and 10 are very similar to the corresponding results on the SMP machine, with main differences the more efficient hardware configuration of the nodes and the higher latency of the interconnection network. The number of processes can differ from the number of nodes because two instances of the same application can be executed on the same node. The execution times on a single processor were: `FT-MIX:4.42`, `EP:25.20` and `MD:218.61` (in seconds).

5 Related Work

The first system that implements OpenMP on a network on SMPs is presented in [10], implemented via a translator that converts OpenMP directives to appropriate calls to a multithreaded version of the TreadMarks SDSM system. Thread's stack is kept in private memory and any potential shared variables are moved in shared heap by the translator, which allocates and frees explicit the necessary storage at the entry and the exit points of a parallel region. In [26], authors present an OpenMP compiler for an SMP cluster. They extend the OpenMP model for an SMP cluster by "compiler-directed" software distributed shared memory system. An OpenMP program is so well structured that it allows the compiler to analyze extent of parallel region for the optimization of efficient communication and synchronization. The OpenMP program is translated into an SPMD execution model. An OpenMP compliant implementation on software DSM is presented in [16]. This work uses the HLRC SDSM library, which does

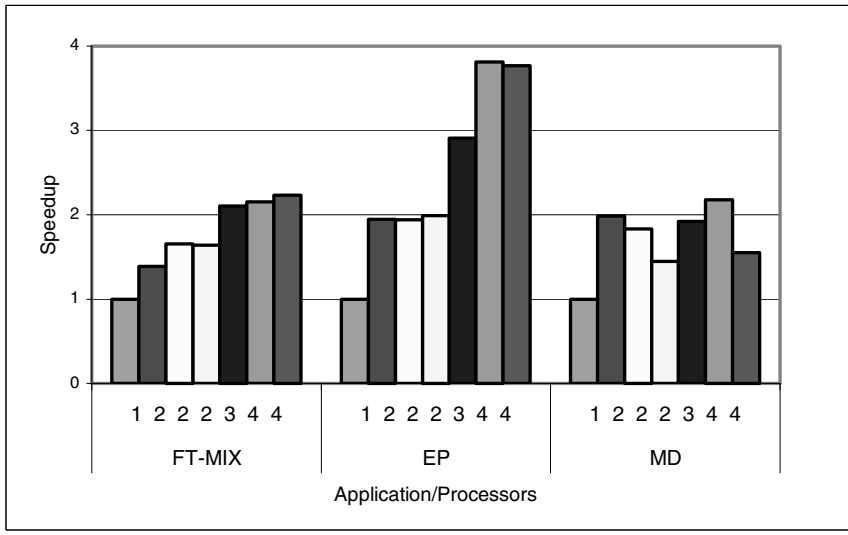


Fig. 10. Speedups on SDSM (SMP cluster)

not support SMP nodes, and focuses on the optimization of the flush directive. CableS [13] is an SDSM library that supports the POSIX threads API on clusters of SMPs. They use a public domain OpenMP compiler (OdinMP) that translates OpenMP C programs to pthreads programs for shared memory multiprocessors and run the translated OpenMP program on their system without modifications to the OpenMP source.

Our runtime system supports OpenMP through the implementation of an appropriate API rather than by targeting a specific DSM system. We dot not bind our work to a specific SDSM library and we do not differentiate the output of the OpenMP compiler for shared memory and clusters of SMPs. Instead, a sophisticated runtime system provides efficient execution of the same OpenMP program (binary) on both architectural platforms.

6 Ongoing Work

Ongoing work includes further implementation improvements of the runtime system, the study of policies for scheduling user-level threads on clusters of SMPs and the evaluation of loop-scheduling policies. We also intend to study further relaxed consistency models and their interaction with the OpenMP programming model and the runtime system, w, and finally to implement the kernel interface of the Nanothreads Programming Model on clusters of SMPs.

Acknowledgments

We would like to thank our partners within the POP (Performance Portability of OpenMP) project. This work was supported by the POP IST/FET project (IST-2001-33071).

References

1. E. Ayguadé, J. Labarta, X. Martorell, N. Navarro, and J. Oliver, *NanosCompiler: A Research Platform for OpenMP Extensions*, In Proceedings of the 1st European Workshop on OpenMP, Lund (Sweden), October 1999.
2. V.K. Barekas, P.E. Hadjidoukas, E.D. Polychronopoulos, and T. S. Papatheodorou, *An OpenMP Implementation for Multiprogrammed SMPs*, In Proceedings of the 3st European Workshop on OpenMP, Barcelona, Spain, August 2001.
3. A. Basumallik, S.J. Min, and R. Eigenmann, *Towards OpenMP Execution on Software Distributed Shared Memory Systems*, In Proceedings of the International Workshop on OpenMP: Experiences and Implementations (WOMPEI'02), Lecture Notes in Computer Science, #2327, Springer Verlag, May 2002.
4. J.M. Bull, *Measuring Synchronization and Scheduling Overheads in OpenMP*, In Proceedings of the 1st European Workshop on OpenMP, Lund, Sweden, October 1999.
5. F. Cappello, O. Richard, and D. Etiemble, *Investigating the performance of two programming models for clusters of SMP PCs*, In Proceedings of the 6th IEEE Symposium On High-Performance Computer Architecture (HPCA-6), Toulouse, France, January 2000.
6. R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster, *Millipede: Easy Parallel Programming in Available Distributed Environments*, Journal of Software: Practice and Experience, 27(8): 929–965, August 1997.
7. P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou, *Integrating MPI and the Nanothreads Programming Model*, In Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2002), Las Palmas, Spain, January 2002.
8. P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou, *Runtime Support for Multigrain and Multiparadigm parallelism*, In Proceedings of the 10th International Conference on High Performance Computing (HIPC' 02), Bangalore, India, December 2002.
9. P.E. Hadjidoukas, E.D. Polychronopoulos, and T.S. Papatheodorou, *Implementing the Nanothreads Programming Model on top of the POSIX Threads API*, In Proceedings of the 20th IASTED Applied Informatics International Conference, Innsburg, Austria, February 2002.
10. C. Hu, H. Lu, A. Cox, and W. Zwaenepoel, *OpenMP for Networks of SMPs*, In Proceedings of the Second Merged Symposium, IPPS/SPDP 99, 1999.
11. W.W. Hu, W.S. Shi, and Z.M. Tang, *JIAJIA: An SVM System Based on A New Cache Coherence Protocol*, In Proceedings of the High Performance Computing and Networking (HPCN'99), April 1999.
12. Intel Corporation, Intel Fortran Compiler, Available at: `http://developer.intel.com`.
13. P. Jamieson, and A. Bilas, *CableS: Thread Control and Memory System Extensions for Shared Virtual Memory Clusters*, In Proceedings of the Workshop on OpenMP Applications and Tools. Purdue University, West Lafayette, Indiana. July 2001.

14. Y. Jegou, *Controlling Distributed Shared Memory Consistency from High Level Programming Languages*, In Proceedings of Parallel and Distributed Processing, IPDPS 2000 Workshops, pages 293-300, May 2000.
15. H. Jin, M. Frumkin, and J. Yan, *The OpenMP Implementation of NAS Parallel Benchmarks and its Performance*, Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
16. S. Karlsson and M. Bronsson, *A Fully Compliant OpenMP implementation on Software Distributed Shared Memory*, In Proceedings of the 10th International Conference on High Performance Computing (HIPC' 02), Bangalore, India, December 2002.
17. X. Martorell, J. Labarta, N. Navarro, and E. Ayguad'e, *A Library Implementation of the Nano-Threads Programming Model*, In Proceedings of the 2nd Euro-Par Conference, Lyon, pp. 644-649, August 1996.
18. Message Passing Interface Forum, *MPI: A message-passing interface standard*, International Journal of Supercomputer Applications and High Performance Computing, Volume 8, Number 3/4, 1994.
19. MPI Software Technology, Inc., `http://www.mpi-softtech.com`.
20. S.J. Min, S.W. Kim, M. Voss, S.I. Lee, and R. Eigenmann, *Portable Compilers for OpenMP*, In Proceedings of WOMPAT 2001, Workshop on OpenMP Applications and Tools, Lecture Notes in Computer Science, 2104, pages 11-19, July 2001.
21. NANOS ESPRIT Project No. 21097, `http://research.ac.upc.es/nanos`.
22. OpenMP Architecture Review Board, *OpenMP Specifications*, Available at: `http://www.openmp.org`.
23. S.M. Paas, M. Dormanns, T. Bemmerl, K. Scholtyssik, and S. Lankes, *Computing on a Cluster of PCs: Project Overview and Early Experiences*, In Proceedings of the 1st Workshop on Cluster-Computing, TU Chemnitz-Zwickau, November 1997.
24. C.D. Polychronopoulos, *Nano-Threads: Compiler Driven Multithreading*, In Proceedings of the 4th International Workshop on Compilers for Parallel Computing CPC'93, Delft (The Netherlands), December 1993.
25. POP (Performance Portability of OpenMP) IST/FET project (IST-2001-33071), `http://www.cepba.upc.es/pop`.
26. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka, *Design of OpenMP compiler for an SMP Cluster*, In Proceedings of the 1st European Workshop on OpenMP, Lund (Sweden), October 1999.
27. R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott, *Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network*, In Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP-16). October 1997.

An Evaluation of MPI and OpenMP Paradigms for Multi-Dimensional Data Remapping

Yun He and Chris H.Q. Ding

CRD Division, Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA

Abstract. We evaluate dynamic data remapping on cluster of SMP architectures under OpenMP, MPI, and hybrid paradigms. Traditional method of multi-dimensional array transpose needs an auxiliary array of the same size and a copy back stage. We recently developed an in-place method using vacancy tracking cycles. The vacancy tracking algorithm outperforms the traditional 2-array method as demonstrated by extensive comparisons. Performance of multi-threaded parallelism using OpenMP are first tested with different scheduling methods and different number of threads. Both methods are then parallelized using several parallel paradigms. At node level, pure OpenMP outperforms pure MPI by a factor of 2.76 for vacancy tracking method. Across entire cluster of SMP nodes, by carefully choosing thread numbers, the hybrid MPI/OpenMP implementation outperforms pure MPI by a factor of 3.79 for traditional method and 4.44 for vacancy tracking method, demonstrating the validity of the parallel paradigm of mixing MPI with OpenMP.

Keywords: Dynamical data remapping, multidimensional arrays, index reshuffle, vacancy tracking cycles, global exchange, MPI, OpenMP, hybrid MPI/OpenMP, SMP cluster.

1 Introduction

Large scale highly parallel systems based on cluster of SMP architectures are today's dominant computing platforms. OpenMP has recently emerged as the definitive standard for parallel programming at the SMP node level[1,2]. Using MPI to handle communications between SMP nodes, the MPI/OpenMP hybrid parallelization paradigm is the emerging trend for parallel programming on cluster of SMP architectures[3,4,5,6,7].

Although the hybrid paradigm is elegant in conceptual and architectural level, in practice, however, performance of many large scale applications indicate otherwise (for example, NAS benchmarks[4], conjugate-gradient algorithms[5] and particle simulations[7]). There are some positive experiences[3,6], but it is often the case that existing applications using pure MPI paradigm over all processors and ignoring the shared memory nature among the processors on the single SMP node outperform the same application codes utilizing the hybrid OpenMP with MPI paradigm.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 195–210, 2003.

In this paper, we provide an in-depth analysis on remapping problem domains on cluster of SMP architectures under several OpenMP, MPI, and hybrid paradigms.

Dynamically remapping problem domains are encountered frequently in many scientific and engineering applications. Instead of fixing the problem decomposition during entire computation, dynamically remapping the problem domains to suit the specific needs at different stages of the computation can often simplify computational tasks significantly, saving coding efforts and reducing total problem solution time.

For example, the 3D fields of an atmosphere (or ocean) model are mapped onto 8 processors, with horizontal dimensions split among the processors. In spectral transform based models, such as the CCM atmospheric model[8] and the shallow water equation[9], one often needs to dynamically remap between the *height-local* domain decomposition and the *longitude-local* decomposition for tasks of distinct nature. In grid-based atmosphere and ocean models, similar remappings are needed for data input/output[10].

To transpose a multidimensional array A, say between 2nd and 3rd indices, the conventional method uses an auxiliary array B of same size of A.

$$B[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3]. \tag{1}$$

In many situations, B is copied back to memory locations of A (denoted as $A \Leftarrow B$), and memory for B is freed. We will call this traditional method as two-array reshuffle method, because of the need of the auxiliary array B. Combining the $B[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3]$ reshuffle phase and the copy-back $A \Leftarrow B$ phase, the net effect of the two-array reshuffle method can be written symbolically as

$$A'[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3] \tag{2}$$

Here A' indicates that reshuffle results are stored at the same location as the original array A.

Recently, a vacancy tracking algorithm for multidimensional array index reshuffle is developed[11] that can perform the transpose in Eq.2 in-place, i.e., without requiring the auxiliary array B. This reduces the memory requirement by half, therefore lifting a severe limitation on memory-bound problems.

Both the conventional two-array method and the new vacancy tracking algorithm can be implemented on cluster of SMP architectures using OpenMP, MPI, and hybrid MPI/OpenMP paradigms. We have investigated them systematically. Some preliminary results were presented at SC2002[12]. Here we perform more systematic studies on both sequential and distributed platforms for both methods. Our main results are that (i) Vacancy tracking algorithm outperforms conventional two-array method in all situations. (ii) OpenMP parallelism performs slightly better than MPI for the traditional two-array method but is substantially faster than MPI parallelism for the vacancy tracking method on a single SMP node. (iii) On up to 128 CPUs, the hybrid paradigm performs about a factor of 4 faster than pure MPI paradigm for both methods. (iv) Contrary to some existing negative experience of developing hybrid programming

applications, our hybrid MPI/OpenMP implementation for the vacancy tracking algorithm outperforms pure MPI by a factor of 4.44.

2 Vacancy Tracking Algorithm

Array transpose can be viewed as a mapping from original memory locations to new target memory locations. The key idea of this algorithm is to move elements from old locations to new locations in a specific memory-saving order, by carefully keeping track of the source and destination memory locations of each array element. When an element is moved from its source to new location, the source location is freed, i.e., a vacancy. This means another appropriate element can be moved to this location without any intermediate buffer. Then the source location of that particular element becomes a vacancy, and yet another element is moved directly from its source to this destination. This is repeated several times, and a closed loop of vacancy tracking cycles is formed.

Consider transposition of a 2D array A(3,2). Six elements of A(3,2) are labeled as A_0,A_1, A_2, A_3, A_4, A_5, and are stored in six consecutive memory locations L_0, L_1, L_2, L_3, L_4, L_5 (shown in the leftmost layout in Figure 1). The transposition is accomplished by moving elements following the vacancy tracking cycle

```
1 - 3 - 4 - 2 - 1
```

Move content in L_1 to a temporary buffer, now L_1 becomes the vacancy; Move content in L_3 to L_1, now L_3 becomes the vacancy; Move content in L_4 to L_3; Move content in L_2 to L_4; Move content in buffer to L_2. Note that contents in L_0 and L_5 are not touched, because they are already in the correct locations. Assume the buffer is a register in CPU, the total memory access is 4 memory writes and 4 memory reads. In contrast, the conventional two-array transpose algorithm will move all 6 elements to array B, and copy them back to A, with a total of 12 reads and 12 writes. The vacancy tracking algorithm achieves the optimal (minimum) number of memory access.

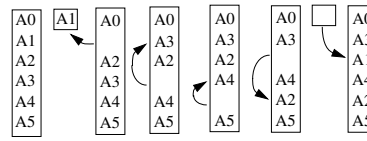


Fig. 1. Transposition for the array A(3,2). L_1, L_3, L_4, L_2 are successive vacancies

Vacancy tracking algorithm applies to many other index operations more complex than two-index reshuffle. For example, the simultaneous transpose of three indexes, the left-circular-shift,

$$A'[k_2, k_3, k_1] \Leftarrow A[k_1, k_2, k_3] \tag{3}$$

can be easily accomplished. For a 3D array $A(4,3,2)$ with 24 elements, the three-index left-shift reshuffle can be achieved by the following two cycles,

```
1 - 4 - 16 - 18 - 3 - 12 - 2 - 8 - 9 - 13 - 6 - 1
5 - 20 - 11 - 21 - 15 - 14 - 10 - 17 - 22 - 19 - 7 - 5
```

A simple algorithm to automatically generate the cycles for 2 indices transpose is

```
! For 2D array A, viewed as A(N1,N2) at input and as A(N2,N1) at output.
! Starting with (i1,i2), find vacancy tracking cycle
       ioffset_start = index_to_offset(N1,N2,i1,i2)
       ioffset_next = -1
       tmp = A(ioffset_start)
       ioffset = ioffset_start                              (C.1)
       do while( ioffset_next .NOT_EQUAL. ioffset_start)
          call offset_to_index(ioffset,N2,N1,j1,j2)  ! N1,N2 exchanged
          ioffset_next = index_to_offset(N1,N2,j2,j1)! j1,j2 exchanged
          if(ioffset .NOT_EQUAL. ioffset_next) then
              A(ioffset) = A(ioffset_next)
              ioffset = ioffset_next
          end if
       end_do_while
       A(ioffset_next) = tmp
```

Here `index_to_offset` and `offset_to_index` are two simple routines that converts two-dimensional *index* from/to one-dimensional *offset*. A slight modification can handle 3 indices operations. The cycle information will be stored in a table first in the actual implementation and the outer do loop (C.1) is performed to move the data from actual memory locations.

We assess the effectiveness of the algorithm by comparing the timing results between the traditional 2-array method and the in-place vacancy tracking method for the index reshuffling of a three-dimensional array $A(N_1, N_2, N_3)$ by moving around the array elements in the block size of the first dimension as shown in Eq.(2). The algorithm is implemented in F90, and tests are carried out on a POWER3 IBM SP. We use compiler option -O5 for highest level of optimization for both methods.

Figure 2 shows the ratio of timing between the two-array method (include array copy back) and in-place algorithm for three array sizes. In-place algorithm performs better for almost all the array sizes except for N ≤ 4. For large array size, vacancy tracking algorithm achieves more than a factor of 3 speedup for $N_3 = 8$, 16, and 64.

The number of memory access and the access pattern could explain the speedup. First, the in-place algorithm eliminates the copy back phase so it reduces memory access by half. Second, for a 2D array $A(N_1, N_2)$, the number of memory access required for the $B[k_2, k_1] \Leftarrow A[k_1, k_2]$ reshuffle phase of a two-array method is $N_1 N_2$. But for the in-place method, the total lengths of all cycles would be $N_1 N_2$. The length-N cycle involves N-1 memory-to-memory copies, one memory-to-`tmp` copy and one `tmp`-to-memory copy. Normally, access to the `tmp` storage is a register or cache, and is much faster than the DRAM access. We could safely count the number of a memory access for the length-N cycle as `N`. Meanwhile, all the length-1 cycles means the memory locations are untouched, thus saves the number of memory access. Third, on cache-based processor architectures, the memory access pattern is as important as the number of memory

access. Though memory access pattern for the in-place method seems more random than traditional method, the number of bytes moved is often large for the problems the method is targeted for. The gap is reduced and the memory access in vacancy tracking algorithm is not irregular at scales relevant to cache performance when the size of the move is larger than cache-line size, which is 128 bytes (16 real*8 data elements). Also, the $B[k_1, k_3, k_2] \Leftarrow A[k_1, k_2, k_3]$ reshuffle phase of the two-array method has the same disadvantage of the in-place method: not efficient memory access due to the large stride.

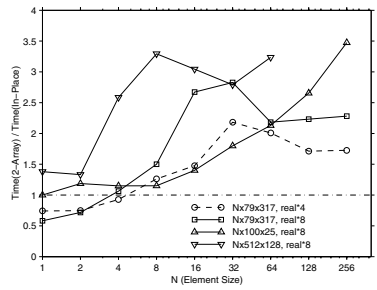


Fig. 2. Timing for a local 3D array index reshuffle on IBM SP. Plotted are the ratio of timing between the two-array method (include array copy back) and in-place algorithm

3 Parallel Paradigms on Cluster SMP Architectures

3.1 Multi-threaded Parallelism

The traditional two-array method adopts a nested do loops to perform index reshuffles. The OpenMP parallelism could be added straightforwardly. The pseudo code is as follows:

```
!$OMP PARALLEL DO DEFAULT (PRIVATE)
!$OMP&         SHARED (N3, N2, A, B)
!$OMP&         SCHEDULE (AFFINITY)
      do i3 = 1, N3
      do i2 = 1, N2
            B(:,i2,i3) = A(:,i3,i2)       (C.2)
      end do
      end do
!$OMP END PARALLEL DO
```

The vacancy tracking algorithm can also be easily parallelized using a multi-threaded approach in a shared-memory multi-processor environment to speed up data reshuffles, e.g., to re-organize a database on a SMP server. As mentioned in Ding[11], the vacancy tracking cycles are non-overlapping. If we assign a thread to each vacancy tracking cycle, they can proceed *independently* and *simultaneously*.

The cycle generation code (C.1) runs first in the initialization phase before the actual data reshuffle, to determine the number of independent vacancy tracking cycles and associated cycle lengths and starting locations. These cycle information can be stored in a table, each cycle entry with a starting location offset and cycle length. The starting offset uniquely determines the cycle, and the cycle length determines the work-load.

The pseudo code for the OpenMP implementation for the main loop for each vacancy tracking cycle is as follows:

```
!$OMP PARALLEL DO DEFAULT (PRIVATE)
!$OMP&         SHARED (N_cycles, info_table, Array)
!$OMP&         SCHEDULE (AFFINITY)
    do k = 1, N_cycles
       an inner loop of memory exchange for each cycle using info_table   (C.3)
    enddo
!$OMP END PARALLEL DO
```

Proper scheduling the independent cycles to threads are important. The workload is based on the cycle lengths. So, it could be done either statically or dynamically. In a static multi-thread implementation, with a given fixed number of threads, an optimization is needed to assign nearly same work-load to each thread. After this assignment, the data reshuffle can be carried out as a regular multi-threaded job.

In a dynamic multi-thread implementation, the next available thread picks up the next independent cycle from the cycle information table and completes the cycle. How to choose the next independent cycle among the remaining cycles in order to minimize the total runtime is a scheduling optimization. For example, a simple and effective method is to choose the task with largest load among the remaining tasks on the queue. Or, if the cycles are short, we could group cycles into chunks so that each thread will pick up a chunk instead of an individual cycle to minimize thread overhead.

3.2 Pure MPI Parallelism

Transposition of a global multi-dimensional array distributed on a distributed-memory system is a remapping of processor subdomains. It involves local array index reshuffles and global data exchanges. The goal is to remap 3D array on processors such that data points along a particular dimension is entirely locally available on the processor, and the data access along this dimension corresponds to the fastest running storage index, just as in the usual array transpose.

Using MPI to communicate data between different processors is the best paradigm for distributed memory architectures such as Cray T3E, where each node has only a single processor. On cluster of SMP architectures, such as IBM SP, each node has, say, 16 processors and they share a global memory space on the node.

Running MPI between different processors on the same node is equivalent to communicating messages using inter-process communication (IPC) between different Unix processes under the control of a single operating system running on the SMP node. Therefore, a simulation code compiled for 64 processors can

successfully run on 4 SMP nodes with 16 processors on each node, since the OS automatically replaces MPI calls by IPC when the relevant inter-processor communications are detected to be within a single SMP node. Communications between processors residing on different SMP nodes will go on to the inter-node communication networks. This "pure" MPI paradigm therefore has the desired portability and flexibility.

Here we outline the algorithm for remapping a 3D array $A(N_1, N_2, N_3)$ regarding the 2nd and 3rd indices using pure MPI (see more details in [11]). For a global multi-dimensional array, it involves local array transpose and global data exchange. Use $A(N_1, N_2, N_3)$ as an example, to transpose it to be $A(N_1, N_3, N_2)$ on P MPI processors, the steps are:

(G1) Do a local transpose on the local array $A(N_1, N_2, N_3/P) \Rightarrow A(N_1, N_3/P, N_2)$
(G2) Do a global all-to-all exchange of data blocks, each of size $N_1(N_3/P)(N_2/P)$
(G3) Do a local transpose on the local array viewed as $A(N_1N_3/P, N_2/P, P) \Rightarrow A(N_1N_3/P, P, N_2/P)$ viewed as $A(N_1, N_3, N_2/P)$.

Local in-place algorithm is used for steps (G1) and (G3). For step (G2) global exchange, the following well known all-to-all communication pattern[9,13,14] is used:

```
! All processors simultaneously do the following:
do q = 1, P - 1
   send a message to destination processor destID       (C.4)
   receive a message from source processor srcID
end do
```

Here we adopt `destID = srcID = (myID XOR q)`, where `myID` is the processor id, and `XOR` is the bit-wise exclusive OR operation. This is a pairwise symmetric exchange communication. As `q` loops through all processors, the `destID` traverses over all other processors.

Communication time can be approximately calculated from a simple *latency + message-size / bandwidth* model. Assuming there are enough communication channels, and no traffic congestion on the network, every processor will spend the same time interval for the global exchange. Adding the local reshuffle time, we have the total global remapping time T_P on P processors:

$$T_P = 2MN_1N_2N_3/P + 2L(P-1) + [2N_1N_3N_2/BP][(P-1)/P] \qquad (4)$$

where M is the average memory access time per element, L is the communication latency including both hardware and software overheads, and B is the point-to-point communication bandwidth.

3.3 Hybrid MPI/OpenMP Parallelism

The emerging programming trend on cluster of SMP is to use MPI between SMP nodes and use multi-threaded OpenMP on the processors within a SMP node. This matches most logically with the underlying system architecture.

A variant of this hybrid parallelism is to create several MPI tasks (Unix processes) on a SMP node and use multi-threaded OpenMP within each such MPI task. For example, on 4 SMP nodes with 16 processors per node, one may create a total of 8 MPI tasks; within each MPI task, one may create 8 threads to match 8 CPUs per MPI task. Therefore, the pure MPI parallelism of Section 3.2 can be viewed as a special case of this hybrid paradigm, where one simply creates 4×16=64 MPI tasks for each of the CPUs on the 4 SMP nodes.

For the remapping problem on cluster SMP architectures, the array is decomposed into subarrays owned by each MPI task. Local transpose will be done by each MPI task, with the choice of either the traditional two-array method or vacancy tracking method. It is parallelized with the multiple threads created by each MPI task. The total number of vacancy tracking cycles shared by the threads are determined by the local array size. After that, global data exchange is done among all the MPI tasks, and another local transpose as in (G.3) is performed. The timing analysis can be approximated by

$$T = 2MN_1N_2N_3/N_{CPU} + 2L(N_{MPI} - 1) + [2N_1N_3N_2/BN_{MPI}][(N_{MPI} - 1)/N_{MPI}] \quad (5)$$

where N_{CPU} is the total number of CPUs in the system and N_{MPI} is number of MPI tasks created.

As the number of MPI tasks increases, the local array size is reduced and so does local reshuffle time (first term in Eqs. 4 and 5). As importantly, as the number of MPI tasks reduces from N_{CPU} to the number of SMP nodes N_{SMP}, the total communication volume decreases, and so does the communication time. (Here we simply assume the communication rates between MPI tasks are the same. In practice, communication between MPI tasks on the same SMP nodes are faster than those between different SMP nodes.) Thus theoretically, we expect the choice $N_{MPI} = N_{SMP}$ is optimal.

A specific issue regarding the speedup on more threads is that vacancy tracking cycles are split among different threads; thus reduce the local reshuffle time as well.

Given the same amount of resources (the total number of CPU), the more MPI tasks we choose, the less available CPUs that OpenMP threads could use, and *vice versa*. No OpenMP parallelism is added in the global exchange stage, the local array size and the number of MPI tasks related to the network communication determine the time spent in this stage. To gain the best performance from the above analysis, we need to utilize an optimal combination of MPI tasks and OpenMP threads on cluster of SMP architectures.

4 Performance

4.1 Scheduling for OpenMP Parallelism

We tested different scheduling methods combined with different number of the threads used on different array sizes within one IBM SP node. The algorithm

is implemented in F90 with OpenMP directives, and tests are carried out on a 16-way IBM SP[15]. Notice, with OpenMP directives (use compiler option -qsmp), optimization level specified as -O5 in compiler option will change the loop structures so that the results are incorrect. We use level -O4 instead. The tested schedules are:

- `Static`: Loops are divided into *N_threads* partitions, each containing *ceiling (N_iterations / N_threads)* iterations. Each thread is responsible for one partition.
- `Affinity`: Loops are divided into *N_threads* partitions, each containing *ceiling (N_iterations / N_threads)* iterations. Then each partition is subdivided into chunks containing n (if specified) or *ceiling (N_remain_iterations_in_partion/2)* (if not specified) iterations. Each thread takes a chunk from its partition first, if none left, then takes a chunk from another thread.
- `Guided`: Loops are divided into progressively smaller chunks until the minimum size of chunk (default 1) is reached. The first chunk contains *ceiling (N_iterations / N_threads)* iterations. Subsequent chunk contains *ceiling (N_remain_iterations / N_threads)* iterations. Threads taking chunks on a first-come-first-serve basis.
- `Dynamic`: Loops are divided into chunks containing n (if specified) or *ceiling (N_iterations / N_threads)* (if not specified) iterations. We choose different chunk sizes. Threads taking chunks on a first-come-first-serve basis.

For the traditional two-array method, the tested array size is 64×512×128. Table 1 lists the timing results obtained from ensemble average of 100 test runs. A star is marked for the fastest timing among all different number of threads. Among all the schedules, `static` and `affinity` have very close timings and are overall the fastest. reasonable speedup is gained for up to 16 threads with these schedules. Although we could set up number of threads as many as we like, there is no gain in performance beyond 16 threads on the 16-way IBM SP. The overhead of threads creation often deteriorates the performance as shown in columns for $N_{threads} = 32$.

For the vacancy tracking method, the tested array sizes are: (i) 64×512×128, $N_cycles = 4114$, *cycle_lengths* = 16; (ii) 16×1024×256, $N_cycles = 29140$, *cycle_lengths* = 9, 3; (iii) 8×1000×500, $N_cycles = 132$, *cycle_lengths* = 8890, 1778, 70, 14, 5; and (iv) 32×100×25, $N_cycles = 42$, *cycle_lengths* = 168, 24, 21, 8, 3.

Tables 2 and 3 list the timing results obtained from ensemble average of 100 test runs. It is shown that with large number of cycles and small cycle lengths, schedule `affinity` is among the best; and with small number of cycles and large or uneven cycle lengths, `dynamic` schedule with small chunk size is preferred. This holds true for almost all number of threads tested. Reasonable speedup is reached with schedules `guided` and `affinity` for relative large arrays up to 16 threads used. There are limited speedup with some of the schedules, even in some cases speed-down, especially for smaller arrays. It is due to the large overhead associated with the creation of the threads, for example, array size 16×1024×256 with `Dynamics,1` and `Dynamics,2` schedules.

Table 1. Timing for Array Size 64×512×128 with Different Schedules and Different Number of Threads Used within One IBM SP Node with Traditional Two-Array Method (Time in seconds)

Schedule	64×512×128				
	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd
Static	44.1*	28.7*	46.1*	90.4*	164.8
Affinity	48.1	27.8*	45.8*	90.6*	164.0
Guided	51.9	45.9	52.6	91.3*	160.8*
Dynamic,1	48.1	50.9	60.5	104.6	191.8
Dynamic,2	50.0	49.6	59.7	99.4	173.7
Dynamic,4	47.6	50.4	55.8	100.0	169.3
Dynamic,8	46.1	48.0	53.7	99.4	173.8
Dynamic,16	56.9	60.3	57.1	97.7	169.1
Dynamic,32	86.9	93.8	87.6	98.7	167.9
Dynamic,64	150.9	163.1	152.5	97.4	162.1
Dynamic,128	296.1	327.0	298.4	328.3	331.9
Dynamic,256	296.3	326.2	301.3	333.0	324.5

Table 2. Timing for Array Sizes 64×512×128 and 16×1024×512 with Different Schedules and Different Number of Threads Used within One IBM SP Node with Vacancy Tracking Method (Time in seconds)

Schedule	64×512×128					16×1024×256				
	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd
Static	34.1	15.3	15.0	25.3	47.4	34.2*	17.8	24.9	42.4	83.6
Affinity	28.8*	10.8*	13.9*	24.7*	47.2*	34.2*	15.5*	23.0*	42.0*	83.1*
Guided	35.3	14.2	20.8	30.4	47.5	38.0	17.6	27.4	46.8	83.3
Dynamic,1	32.3	16.5	22.9	36.1	58.6	358.8	348.7	55.6	68.0	151.6
Dynamic,2	32.6	16.1	22.3	34.7	55.7	180.6	165.8	37.5	61.6	103.3
Dynamic,4	33.8	16.7	22.7	35.6	54.7	39.9	23.3	35.5	58.2	98.8
Dynamic,8	32.4	16.1	21.4	33.5	53.9	39.4	21.4	33.3	54.3	94.8
Dynamic,16	30.3	16.0	22.8	33.6	53.3	36.9	20.6	31.4	52.5	93.0
Dynamic,32	28.9	16.2	21.4	32.4	52.4	38.9	21.2	31.4	62.3	91.5
Dynamic,64	34.9	16.0	20.5	32.8	59.9	38.6	20.2	29.9	50.9	89.9
Dynamic,128	28.9	16.1	20.0	35.8	51.0	33.5	19.2	29.6	64.9	88.9
Dynamic,256	29.5	16.0	20.0	31.8	52.7	34.4	18.9	29.8	50.0	87.6
Dynamic,512	-	-	-	-	-	34.5	19.9	29.9	63.2	87.9
Dynamic,1024	-	-	-	-	-	35.3	19.6	30.7	49.0	87.2

4.2 Pure MPI and Pure OpenMP Parallelisms within One Node

Figure 3 shows the performance of the parallel implementation with different number of threads (pure OpenMP) or processors (pure MPI) within one node on 16-way SMP of IBM SP with each method, respectively. As tested in Ding[11], vacancy tracking method performs better with real*8 than real*4 as compared to two-array method. We use real*8 in this test for all MPI data types. The array sizes are chosen to be fit in local memory of one processor.

Clearly, both parallelisms with both methods have quite good speedup, except that the MPI performance with vacancy tracking method at 2 processors

Table 3. Timing for Array Sizes 8×1000×500 and 32×100×25 with Different Schedules and Different Number of Threads Used within One IBM SP Node with Vacancy Tracking Method (Time in seconds)

Schedule	8×1000×500					32×100×25				
	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd	32 thrd	16 thrd	8 thrd	4 thrd	2 thrd
Static	57.9	58.8	93.3	158.3	261.5	18.7	2.84	1.49	1.34	1.10
Affinity	48.5	38.0	59.3	86.3	158.0	16.6	1.88	1.72	0.95	1.31
Guided	58.0	58.4	92.7	159.9	261.4	15.3*	3.99	1.71	1.93	1.08*
Dynamic,1	47.1*	32.7*	49.7*	84.0	147.2	19.3	0.81*	1.05*	0.94*	1.35
Dynamic,2	50.8	32.7*	51.9	82.5*	145.8	17.0	2.68	1.12	0.97	1.38
Dynamic,4	56.9	37.8	53.9	83.7	144.3	17.0	3.45	2.03	0.98	1.24
Dynamic,8	63.3	52.7	52.3	82.5*	144.1*	16.5	3.29	2.68	1.57	1.28
Dynamic,16	107.9	92.1	92.2	90.2	148.6	18.7	4.28	3.20	2.09	1.33
Dynamic,32	165.1	159.4	158.8	187.8	155.8	-	-	-	-	-

has a drop, which is due to the increased communication overhead compared to the entire local remapping. The vacancy tracking method is about twice faster than the two-array method. Both methods indicate a better OpenMP scaling than MPI scaling. With two-array method, OpenMP is slightly faster than MPI. With vacancy tracking method, OpenMP outperforms MPI substantially at all times. With 16 threads, it is faster than MPI by a factor of 2.76 for array size 64×512×128 and by a factor of 1.99 for array size 16×1024×256. Thus it makes sense to develop a hybrid MPI/OpenMP parallelism, however, experiences of other researchers[3,7] indicate that a better performance is not guaranteed[3].

4.3 Pure MPI and Hybrid MPI/OpenMP Parallelisms Across Nodes

Figure 4 shows the performance of the parallel implementation across several nodes on IBM SP with each method, respectively. (timing with total CPUs = 16 is plotted for comparison) with array size 64×512×128 (results for array size 16×1024×256 are very similar). We use schedule `affinity` for OpenMP parallelism in both methods. The timing for both methods are very close while the speedup values are about twice in the two-array method than those in the vacancy tracking method. This is due to the ratio of sequential running time for both methods used as base to calculate the speedups is about 2 to 1. The closeness of total remapping timing in these methods is due to the fact thay the majority (over 90%) of total timing is spent at the global exchange stage, which is only parallelized by pure MPI.

The maximum number of threads we could efficiently use for one MPI task per node is 16. Pure MPI [NETWORK.MPI=SHARED is already utilized] does not scale above 16 processors. Using one MPI task per node with full 16 threads at each node results even worse performance at 32 total CPUs, although close or better performance than pure MPI are achieved at 64 and 128 total CPUs. The dip at 32 CPUs for both pure MPI and hybrid MPI/OpenMP parallelisms could be explained by the large across-node communication overhead for the global

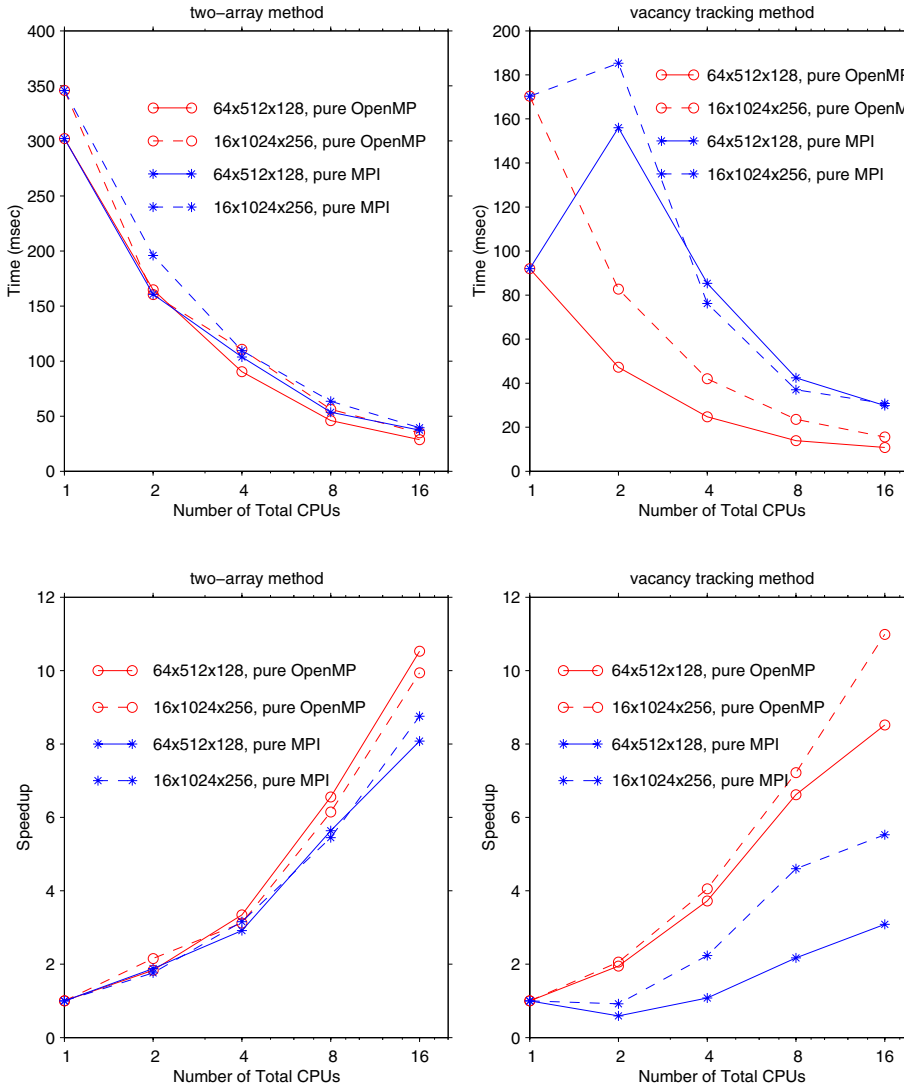


Fig. 3. Time and speedup for global array remapping on different number of processors (pure MPI) or threads (pure OpenMP)

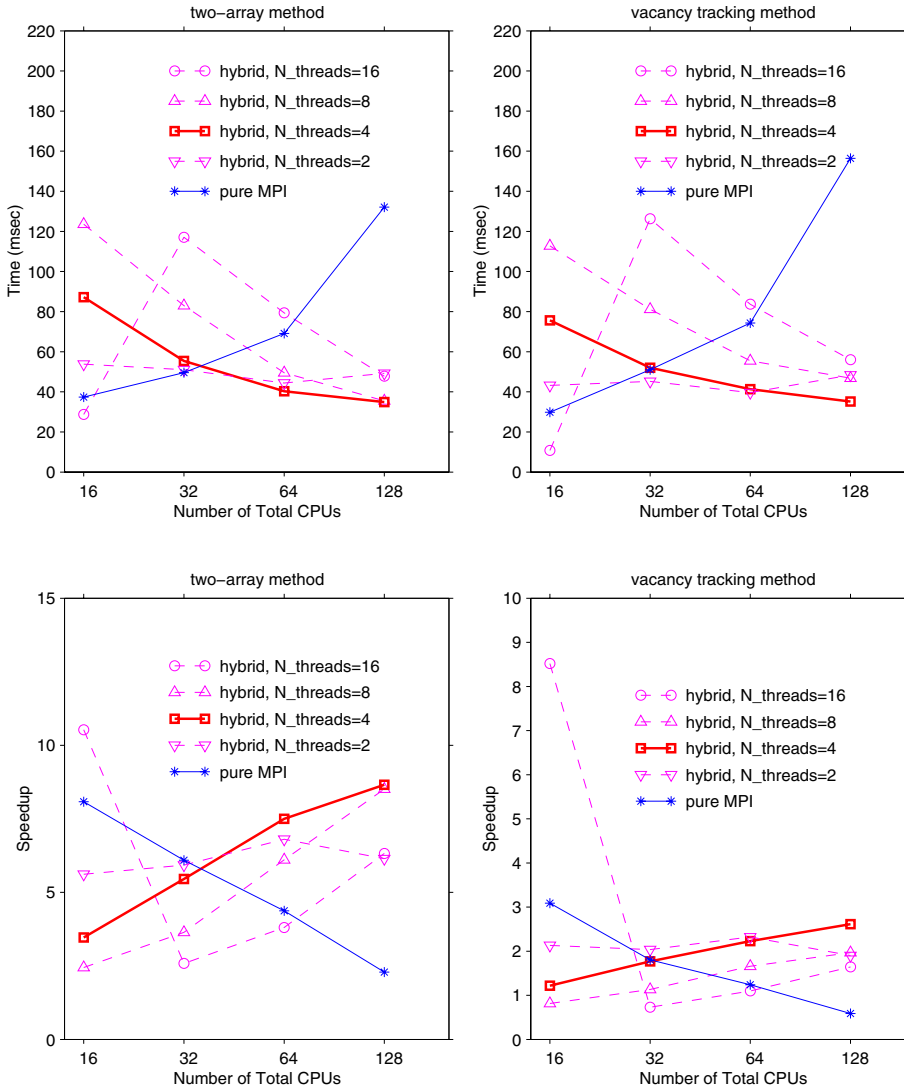


Fig. 4. Time and speedup for global array remapping with different combinations of MPI tasks and OpenMP threads for array size 64×512×128

exchange stage (last term in Eq.5) which accounts for more than 90% of total remapping time.

Given total number of CPUs (N_{CPU}), we could adjust the number of processors to be used as MPI tasks (N_{MPI}) and number of threads per MPI task ($N_{threads}$). An example of choices for N_{MPI} and $N_{threads}$ with $N_{CPU} = 64$ would be:

```
N_CPU = N_MPI * N_threads
  64       4       16
  64       8        8
  64      16        4
```

The communication overhead is reduced when MPI tasks within same node utilize the in-node MPI network. The different subarray size each MPI task owns also contributes to the timing difference. Although we use the same number of total CPUs, the subarray sizes are determined by the number of MPI tasks; thus the numbers of vacancy tracking cycles in the loop for different N_{MPI} are different. Since we only have OpenMP directives for the local reshuffles, we need to find an optimal combination of MPI tasks and OpenMP threads to achieve the overall best performance.

From our experiments, $N_{threads} = 4$ gives an overall best performance among all other number of threads. At 128 total CPUs, the hybrid MPI/OpenMP parallelism with $N_{threads} = 4$ performs faster than with $N_{threads} = 16$ by a factor of 1.37 (two-array method) and 1.59 (vacancy tracking method), respectively; and it performs faster than pure MPI parallelism by a factor of 3.79 (two-array method) and 4.44 (vacancy tracking method), respectively. Although it still does not have better performance than with $N\_threads = 16$, the hybrid MPI/OpenMP parallelism scales up from 32 CPUs to 128 CPUs, compared to scales down with pure MPI.

5 Conclusions

In this paper, we first assess the effectiveness of an in-place vacancy tracking algorithm for multi-dimensional data remapping by comparing the timing results with traditional 2-array methods. We tested with different array sizes on IBM SP. The in-place method outperforms the traditional method for all the array sizes when the data block to move is not too small. The speedup we gain for a big array is 3.24. This could be explained by two factors: 1) the elimination of the auxiliary array thus the copy back; 2) the memory access volume and pattern.

The vacancy tracking algorithm is efficient and easy to parallelize with OpenMP in a shared programming model. The independency of the vacancy tracking cycles allows us to parallelize the in-place method using a multi-threaded approach in a shared-memory processor environment to speed up data reshuffles. The vacancy tracking cycles are non-overlapped so multiple threads can process each tracking cycles simultaneously. We extensively tested the different thread

scheduling methods on 16-way IBM SP and found that schedule *affinity* optimizes the performance for arrays with large number of cycles and small cycle lengths, while *dynamic* schedule with small chunk size is preferred for arrays with small number of cycles and large or uneven cycle lengths. Meanwhile, the OpenMP parallelism could be used directly upon the nested loops of the memory copy for the traditional two-array method. The timing results show that schedules `static` and `affinity` are among the best.

On distributed memory architectures, both methods could be parallelized using pure MPI with the combination of local array transpose method and an existing global exchange method. Based on the fact that pure OpenMP performs faster and has better scaling than pure MPI on a single node of IBM SP, we are encouraged to develop a hybrid MPI/OpenMP approach on cluster SMP architectures for an efficient global data remapping algorithm.

We discussed the algorithms of the hybrid approach, advantages and disadvantages of choosing the number of MPI tasks and OpenMP threads if the total number of nodes is given, and carried out systematic tests on IBM SP. For both array sizes we tested, pure MPI does not scale beyond total 16 processors, in fact, scales down from 32 processors to 128 processors. But by using hybrid MPI/OpenMP approach, and by carefully choosing the number of threads per MPI task, we gained about a factor of 4 speedup for both the two-array method and the vacancy tracking method from pure MPI.

In a distributed memory model, the local transpose is only applied within each MPI task and a standard all-to-all communication algorithm is used across the nodes. Thus the OpenMP parallelization is more efficient than the MPI parallelization within one SMP node. As expected, the hybrid OpenMP/MPI parallel performance is in between.

Contrary to some existing negative experience in hybrid programming applications, this paper gives a positive aspect of developing hybrid MPI and OpenMP parallel paradigms for real applications. The vacancy tracking algorithm itself also eliminates an important memory limitation for multi-dimensional data remapping on sequential, distributed memory and cluster SMP computer architectures while improving performance significantly at same time.

Acknowledgment

This work is supported by Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences, and Office of Biological and Environmental Research, Climate Change Prediction Program, of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

References

1. OpenMP: Simple, Portable, Scalable SMP Programming. `http://www.openmp.org`
2. COMPunity - The Community for OpenMP Users. `http://www.comunity.org`
3. L. Smith and M. Bull, Development of hybrid mode MPI/OpenMP applications, Scientific Programming, Vol. 9, No 2-3, 83–98, 2001.

4. F. Cappello and D. Etiemble,"MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks", SC 2000, Dallas, Texas, Nov 4–10, 2000.
5. P. Lanucara and S. Rovida, "Conjugate-Gradient Algorithms: An MPI Open-MP Implementation on Distributed Shared Memory Systems". EWOMP 1999. Lund University, Sweden, Sept.30–Oct.1, 1999.
6. A. Kneer, "Industrial Hybrid OpenMP/MPI CFD application for Practical Use in Free-surface Flow Calculations", WOMPAT 2000: Workshop on OpenMP Applications and Tools, San Diego, July 6–7, 2000.
7. D. S. Henty,"Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling", SC 2000, Dallas, Texas, Nov 4–10, 2000.
8. J. Drake, I. Foster, J. Michalakes, B. Toonen and P. Worley, "Design and performance of a scalable parallel community climate model", Parallel Computing, v.21, pp.1571–1581, 1995.
9. I. T. Foster and P. H. Worley. "Parallel algorithms for the spectral transform method," SIAM J. Sci. Stat. Comput., v.18, pp. 806–837. 1997.
10. C. H.Q. Ding and Y. He, "Data Organization and I/O in a parallel ocean circulation model", Lawrence Berkeley National Lab Tech Report 43384. Proceedings of Supercomputing'99, Nov 1999.
11. C. H.Q. Ding, "An Optimal Index Reshuffle Algorithm for Multidimensional Arrays and Its Applications for Parallel Architectures", IEEE Transactions on Parallel and Distributed Systems, V.12, No.3, pp.306–315, 2001.
12. Y. He and C. H.Q. Ding, "MPI and OpenMP Paradiagms on Cluster of SMP Architectures: the Vacancy Tracking Algorithm for Multi-Dimensional Array Transpose", SC2002, Baltimore, Maryland, Nov 15–19, 2002.
13. S. L. Johnsson and C.-T. Ho, "Matrix transposition on boolen n-cube configured ensemble architectures", SIAM J. Matrix Anal. Appl. v.9. pp.419–454, 1988.
14. S. H. Bokhari. "Complete Exchange on the Intel iPSC-860 hypercube", Technical Report 91-4, ICASE, 1991.
15. Using OpenMP on seaborg. `http://hpcf.nersc.gov/computers/SP/openmp.html`

Experiences Using OpenMP Based on Compiler Directed Software DSM on a PC Cluster

Matthias Hess[1], Gabriele Jost*[2], Matthias Müller[1], and Roland Rühle[1]

[1] HLRS, Allmandring 30,70550 Stuttgart, Germany
[2] NASA Ames Research Center, Moffett Field, CA 94035-1000, USA

Abstract. In this work we report on our experiences running OpenMP programs on a commodity cluster of PCs running a software distributed shared memory (DSM) system. We describe our test environment and report on the performance of a subset of the NAS Parallel Benchmarks that have been automatically parallelized for OpenMP. We compare the performance of the OpenMP implementations with that of their message passing counterparts and discuss performance differences.

1 Introduction

Computer architectures using clusters of PCs with commodity networking have become a low cost alternative for high end scientific computing. Currently message passing is the dominating programming model for such clusters. The development of a parallel program based on message passing adds a new level of complexity to the software engineering process since not only computation, but also the explicit movement of data between the processes must be specified.

Shared memory parallel processors (SMP) provide a user friendlier programming model. The use of globally addressable memory allows users to exploit parallelism while avoiding the difficulties of explicit data distribution on parallel machines. Parallelism is commonly achieved by multi-threading the execution of loops. Compiler directives to support multi-threaded execution of loops are supported on most shared memory parallel platforms. In addition, many compilers provide an automatic parallelization feature taking all the burden of code analysis off the user. Efficiency of compiler parallelized code is often limited, since a thorough dependence analysis is not possible without user information. Alternatively, there are parallelization support tools available which take the tedious work of dependence analysis and generation of directives off the user but allow user guidance for critical parts of the code. An example of such a tool is CAPO [10].

While shared memory architectures provide a convenient programming model for the user, a drawback is that they are expensive. During recent years there have been considerable efforts to develop system software to support DSM (Distributed Shared Memory) programming which enables the user to employ the convenient shared memory programming model on a network of processors,

* Employee of Computer Sciences Corporation.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 211–226, 2003.

thereby maintaining the ease of use as well as the low cost of hardware. Examples of such systems are TreadMarks [2] and SCASH [13]. These systems allow the support of OpenMP parallelization on clusters of processors, thereby removing the major impediment to their usage which is the high effort to develop a message passing version from a sequential program. We have installed publicly available DSM software on a commodity cluster of PCs and tested its performance on a set of benchmark kernels. The paper seeks to address the issue of evaluating the efficiency of DSM without explicit hardware support. The rest of the paper is structured as follows: In section 2 we discuss the message passing and the shared address space programming models. In section 3 we describe the hardware platform and system software of our test environment. In section 4 we describe our evaluation strategy and discuss the performance of the individual benchmark kernels. In section 5 we discuss some of the problems we encountered. In section 6 we briefly examine some related work and in section 7 we summarize our conclusions and discuss future work.

2 Programming Models

Currently message passing and the use of shared address space are the two leading parallel programming models.

2.1 Message Passing

Message passing is a well understood programming paradigm. The computational work and the associated data are distributed between a number of processes. If a process needs to access data located in the memory of another process, it has to be communicated via the exchange of messages. The data transfer requires cooperative operations to be performed by each process, that is, every send must have a matching receive. The regular message passing communication achieves two effects: communication of data from sender to receiver and synchronization of sender with receiver.

MPI (Message Passing Interface) [12] is a widely accepted standard for writing message passing programs. It is a standard programming interface for the construction of a portable, parallel application in Fortran or in C/C++, which is commonly used when the application can be decomposed into a fixed number of processes operating in a fixed topology (for example, a pipeline, grid, or tree). MPI provides the user with a programming model where processes communicate by calling library routines to send and receive messages. Pairs of processes can perform point-to-point communication to exchange messages. For increased convenience and performance a group of processes can also call collective communication routines to implement global operations such as broadcasting values or calculating global sums. Global synchronization can be implemented by calls to barrier routines. Asynchronous communication is supported by providing calls for probing and waiting for certain messages. In MPI-1 [12], all communication operations require the sending as well as the receiving side to issue calls to the message-passing library.

2.2 Shared Address Space

Parallel programming on a shared memory machine can take advantage of the globally shared address space. Compilers for shared memory architectures usually support multi-threaded execution of a program. Loop level parallelism can be exploited by using compiler directives such as those defined in the OpenMP standard [14]. Multiple execution threads are automatically created for performing the work in parallel. Data transfer between threads is done by direct memory references. OpenMP provides a fork/join execution model in which a program begins execution as a single process or thread. This thread executes sequentially until a PARALLEL construct is found. At this time, the thread creates a team of threads and it becomes its master thread. All threads execute the statements lexically enclosed by the parallel construct. Work-sharing constructs (DO, SECTIONS and SINGLE) are provided to divide the execution of the enclosed code region among the members of a team. All threads are independent and may synchronize at the end of each work-sharing construct or at specific points either implicitly or explicitly (specified by the BARRIER directive). Exclusive execution mode is also possible through the definition of CRITICAL regions.

This approach provides a relatively easy way to develop parallel programs but has disadvantages. It is often difficult to achieve scalability of the code for a large number of processors due to a lack of data locality and possibly large synchronization costs.

3 Hardware Platform and Software Description

Our test environment consists of a cluster of commodity PCs at the High Performance Computing Center of the University of Stuttgart (HLRS). In the following we give some details about hardware and system software.

3.1 Platform Description

We have used a cluster at HLRS consisting of 8 NEC 120Ed server nodes as the test platform. The nodes are dual processor systems with two 1 GHz Pentium III CPUs and 2 GB of main memory. Each node is equipped with a Myrinet 2000 NIC in a fast 64 bit / 66 MHz PCI slot. The nodes are based on the ServerSet III HE chip set and have a good communication performance to the Myrinet cards. The bandwidth from memory to the card is 409 MB/s for read operations and 480 MB/s for write operations. These data have been acquired with the program 'gm_debug' provided by Myricom. A collection of data for other motherboards and chip sets can be found at [1]. For the current study we used only one CPU per node.

In order to compare the performance of the DSM software with a true shared memory system, we used a 16-way NEC Azusa. The NEC Azusa system is a shared architecture system with IA-64 processors. Both systems, the distributed memory cluster and the shared memory Azusa, were running Linux in its 2.4 version. This reduces effects due to different memory managements of different

operating systems on the distributed and the shared memory architecture. The performance impact of different memory management systems is discussed in [5]. We did not have a four or eight processor IA-32 system available for the tests.

3.2 SCore

The PC cluster used for our study is running SCore [13] software. SCore is a parallel programming environment for workstations and PC clusters, developed by the Real World Computing Partnership (RWCP). The project has now been transferred to the PC Cluster Consortium [13]. Among other features, SCore provides its own communication layer called PM [20,21]. It aims at providing a uniform interface to different communication devices like Fast Ethernet, Gigabit Ethernet or Myrinet.

SCore also supports different parallel programming paradigms like message passing or shared memory. On the message passing side there is a MPI-implementation based on MPICH with an additional device specifically designed for the PM layer. Shared memory is supported in two ways. The PM layer has a shared memory device that is intended for SMP systems. It uses memory-mapped shared segments for the communication between processes on a true shared memory system. Additionally, the SCore architecture has a software distributed shared memory system called SCASH [6], that we employed to obtain the results of the tests we present in this paper.

3.3 SCASH

SCASH [6] is a page-based software distributed shared memory system. It is implemented as a user-space runtime library which uses the PM layer for communicating pages between cluster nodes.

It employs an eager release consistency model to ensure the consistency of shared memory on a per-page basis. This means that at memory synchronization points only modified parts of memory are updated, which usually requires exchange of data between nodes.

The home node of a page is the node that keeps the latest data of the page. If other nodes change the data within a page it must be updated on the home node. To reduce memory transfer, SCASH also provides the possibility to change the home node of a page. It is possible to use two page consistency protocols, an invalidate and an update protocol, which can be chosen dynamically.

To reduce memory transfer between nodes, the nodes use cached copies of requested pages. Only on write operations to the memory can these copies become inconsistent. The update protocol specifies that all copies of a particular page be updated once one node changes its contents.

In the invalidate protocol, the home node of a page notifies all nodes which share that page when a page has been altered and cached copies of that page on other nodes become invalid.

3.4 Omni OpenMP

Omni OpenMP [13] is a collection of programs and libraries that enable OpenMP for back-end compilers that do not support it natively. The front-end to these compilers translates C or Fortran77 OpenMP source texts into multi-threaded C with calls to a runtime library.

One of the main goals of Omni OpenMP is portability, so the translation pass from an OpenMP program to the target code is written in JAVA. The target code is – in turn – compiled by the back-end C compiler on the target platform. For the tests presented here we used the GNU C Compiler as the back-end compiler.

The Omni compiler suite can be configured to use several different underlying libraries. For the thread system Solaris Threads or pthreads are supported, but there is also support for StackThreads [19] developed by Real World Computing Project (RWCP). In addition to the support of threads there is support for several shared memory implementations, like UNIX shmem. In our tests we used the support for the SCASH distributed shared memory system which has been described above.

The Omni OpenMP compiler suite is also available for IA-64. For tests on the shared memory Azusa system (see 3.1) we used the Omni compiler, too, again in order to minimize the influence of different software. This way we can attribute certain observations to either the DSM system or the Omni OpenMP compiler.

4 Case Studies

For our evaluation we selected a subset of the NAS Parallel Benchmarks [3]. They were designed to compare the performance of parallel computers for computational fluid dynamics (CFD) applications. The full suite consists of five benchmark kernels and three simulated CFD applications. We selected three of the five benchmark kernels for our study.

4.1 Evaluation Strategy

To evaluate the performance of our test environment we compare the timings of OpenMP implementations of the benchmark kernels to:

1. Timings of their message passing counterparts on the same system.
2. Timings obtained on a true shared memory system but with a similar operating system and therefore a comparable memory management system.

The comparison of OpenMP versus MPI will give us some means to determine how well the DSM software handles memory coherency and synchronization. In the MPI implementation access to remote data is achieved by calls to the message passing library. The user has control over data locality and decides when and how much data to communicate. This provides the opportunity to minimize

communication during program execution. Another aspect of the message passing approach is that data communication and synchronization are integrated. The send and receive operations not only exchange data, but also regulate the progress of the processes. In the OpenMP implementation the location of the data, the amount of data to be communicated, and the synchronization among the threads depends on the DSM system and the compiler. As explained in section 3, the DSM system detects the necessity of communicating data when a page of memory is accessed that has been marked as updated by another process. We will use the number of page requests as an indicator for the amount of communication in the DSM system. Even in the case where a hand-optimized message passing implementation outperforms the DSM system, the ease of application porting may compensate for a certain loss of performance.

The comparison of OpenMP on a cluster versus OpenMP on a shared memory node gives us some estimate of the speedup that can be expected from the OpenMP programming paradigm on a true shared memory architecture. Our test platforms are described in section 3. We use the Omni compiler on both platforms.

The benchmarks come in different classes determined by the problem size. We ran only the small problems of class S,W, and A. We encountered some problems with the larger sizes which will be discussed in section 5. Since our system is small, consisting of only 8 nodes, it is hazardous to extrapolate the scalability studies to larger systems. However, running the very small benchmark classes allows us to gain some insight into how the computation to communication ratio impacts the performance.

Since the ease of application porting is an important factor in favor of the DSM system, we started out with a sequential version of our benchmark kernels and used the automatic parallelization support tool CAPO [10] to insert OpenMP directives, thereby minimizing the parallelization effort. CAPO was developed at the NASA Ames Research Center. It takes as input a sequential Fortran program. It then performs an extensive dependence analysis over statements, loop iterations, and subroutine calls and generates Fortran code containing OpenMP directives. CAPO is based on the dependence analysis module of the CAPTools [8] parallelization tool. Our starting point for the message passing version of the benchmark kernels was the NPB2.3 [4] release of the NAS Parallel benchmarks. For the OpenMP implementations we started with an optimized serial implementation of the same benchmarks as described in [9]. The structure of the serial code is kept very close to the message passing code. Only slight modifications were applied to the kernels considered in our study and we will describe them in the sections below. A good description about how to use CAPO for the OpenMP parallelization of the benchmarks is given in [10],

4.2 The EP Benchmark Kernel

EP stands for embarrassingly parallel. The kernel generates pairs of Gaussian random deviates according to a specific scheme. As the name suggests, the iterations of the main loop can be executed in parallel. Tool based OpenMP par-

allelization of the kernel was possible without user interaction. Once the data is distributed, the main loop which generates the Gaussian pairs and tallies the counts does not require access to remote data except for several global sum reductions. In the MPI implementation the global sum is achieved by calls to `mpi_allreduce`. The OpenMP implementation uses the OMP PARALLEL REDUCTION directive. The MPI implementation shows a very low communication overhead, which is less than 1 % even for the smallest benchmark class on 8 nodes. If m denotes the log2 of the number of complex pairs of uniform (0, 1) random numbers, then the problem size of the benchmark classes under consideration is:

Class S: m = 24
Class W: m = 25
Class A: m = 28

The OpenMP/DSM implementation shows a very low number of page requests to the DSM system. As expected, the message passing as well as the OpenMP/DSM implementation show an almost linear speedup for all benchmark classes. For 8 nodes the OpenMP/DSM performance ranges within 97 % to 102% of that of MPI, depending on the benchmarks class. As an example we show the speedup for class A in Fig. 1.

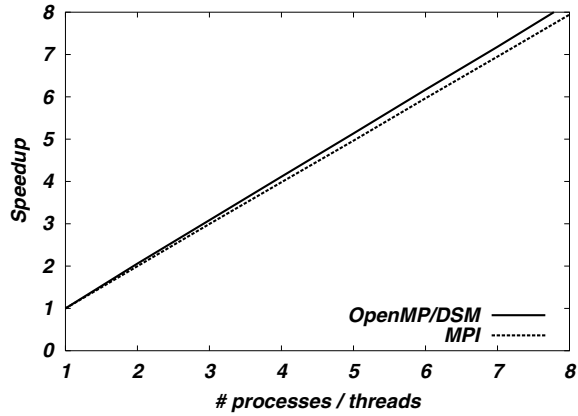


Fig. 1. Speedups for class A of the EP benchmark for OpenMP/DSM and MPI

4.3 The CG Benchmark Kernel

The CG benchmarks kernel uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. The kernel is useful for testing unstructured grid computations and communications since the underlying matrix has randomly generated locations of entries.

Parallelization for message passing and directive based versions occur on the same level within the conjugate gradient algorithm. The basic parallel operations are: sparse matrix vector multiply, AXPY operations, and sum reductions. The code was parallelized using CAPO without any user interaction. If na denotes the number of rows of the sparse matrix and nz the number of non-zero elements per row, then the problem size of the benchmark classes under consideration are:

Class S: na = 1400, nz = 7
Class W: na = 7000, nz = 8
Class A: na = 14000, nz = 11

In Fig. 2 we show the speedup for the three benchmark classes. For class A, the MPI as well as the OpenMP/DSM and OpenMP/SMP implementations show reasonable speedup. The OpenMP/SMP version shows occasional superlinear speedup due to cache effects. For 8 nodes, the OpenMP/DSM efficiency reaches about 75% of that of MPI. The MPI version maintains this speedup for the smaller problem sizes but the performance of the OpenMP/DSM version decreases drastically. For 8 nodes and class W the OpenMP/DSM efficiency is only 35% and for class S is goes down to 6% yielding a speedup of less than 1. The class S problem size is far too small to serve as a realistic example. However, we have a closer look at the performance differences for this class to get an idea about potential scalability issues related to the DSM system.

Our first observation is that the Omni compiler and its runtime library introduce additional overhead which decreases performance even on a shared memory system. This is demonstrated in Fig. 2d, where we compare the speedup of class S for the Omni compiler with that of the Intel compiler and Guide, which is part of the KAP/Pro ToolSet of Kuck & Associates/Intel.

To analyze the DSM performance we examine the three major time consuming loops within one conjugate gradient iteration. These loops are the same in the MPI and the OpenMP/DSM implementation. They implement a sparse matrix-vector multiplication (MVM), a dot-product (DOT) , and a loop combining two AXPY operations and a dot-product. Code examples are shown in Fig. 3

The sparse matrix A is stored in packed format such that indirect addressing is required for matrix operations. The sparse matrix-vector multiply is a double-nested loop requiring indirect addressing. For OpenMP, it is parallelized by using an `OMP PARALLEL DO` on the outer loop across the rows of the sparse matrix. The dot-product as well as the AXPY's combined with a dot-product are single loop nests, using the OpenMP REDUCTION clause to build the global sum.

The speedups for class S for the three major loops are shown in Fig. 4. Both implementations suffer from a large communication to computation ratio for the single nested loops. However, the effect is far more severe for the DSM system. In the MPI version the communication required for the global reduction operations is highly optimized by using non-blocking send and receive to minimize synchronization overhead. The set of processes that communicate with each other is determined in advance. This allows the reduction of the amount of

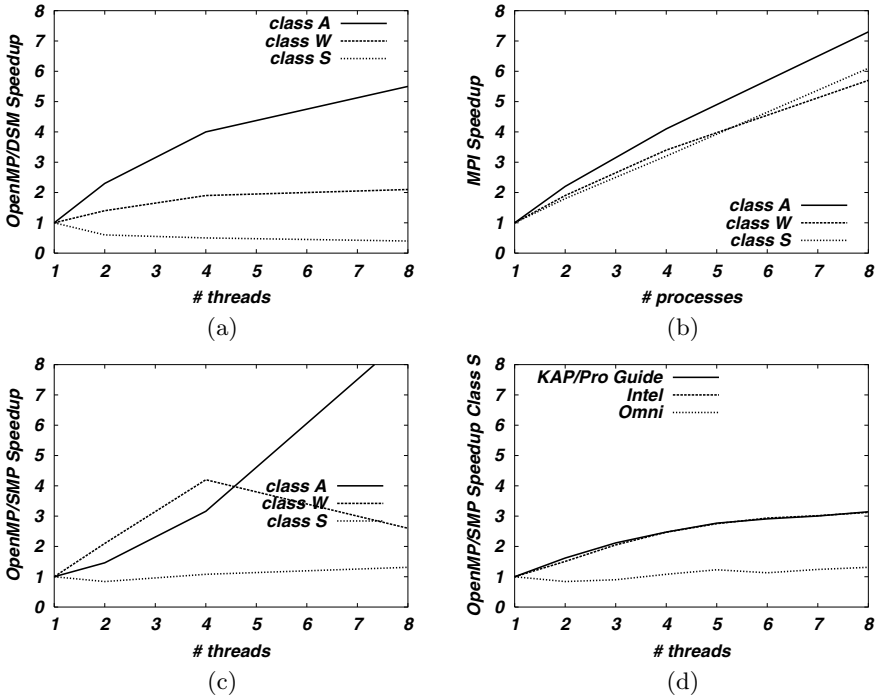


Fig. 2. Speedups for different classes of the CG benchmarks. In (a) the speedup for OpenMP/DSM is shown for classes A, W and S. The MPI speedup for the same classes is given in (b). The speedup for a true shared memory system is presented in (c). (d) shows a comparison of the speedup for class S for different compilers on a shared memory platform. The Guide and the Intel compiler both support OpenMP natively

communication within the iteration loop. In the OpenMP/DSM implementation, processing the OpenMP `REDUCTION` clause by the DSM system generates a large communication overhead which is indicated by a high number of page requests and manifests itself by poor speedup as can be seen in Fig. 4. The parallel efficiency is bad for the matrix-vector-multiply and disastrous for the dot-product and AXPY operations. We conclude that the performance loss for the small size problems is due to:

1. Additional overhead due to the Omni compiler,
2. A high communication to computation ratio which results from short loops and global communication operations.

For the more realistic benchmark class A the performance of the DSM system is acceptable.

Matrix-Vector Product:

```
!$omp parallel do private(j,k,sum)
         do j=1,lastrow-firstrow+1
            sum = 0.d0
            do k=rowstr(j),rowstr(j+1)-1
               sum = sum + a(k)*p(colidx(k))
            enddo
            q(j) = sum
         enddo
```

Dot-Product

```
         d = 0.0d0
!$omp parallel do private(j) reduction(+:d)
         do j=1, lastcol-firstcol+1
            d = d + p(j)*q(j)
         enddo
```

AXPY/Dot-Product Combination

```
         rho = 0.0d0
!$omp parallel do private(j) reduction(+:rho)
         do j=1, lastcol-firstcol+1
            z(j) = z(j) + alpha*p(j)
            r(j) = r(j) - alpha*q(j)
            rho = rho + r(j)*r(j)
         enddo
```

Fig. 3. Code examples for multiplication, a dot-product, and a loop combining two AXPY operations and a dot-product

4.4 The FT Kernel Benchmark

The FT benchmark is the computational kernel of a spectral method based on a 3-D Fast Fourier Transform (FFT). During the setup phase the 3-D array is filled with random numbers. Unlike in the other benchmarks, the setup phase is part of the timed code. The serial implementation of FT code was changed to pre-calculate the values for the loop that initializes each data plane. This enables the directive based parallelization of the loop. The main loop in FT could not be parallelized completely automatically. Due to the complicated structure of the loop CAPO had to assume data dependencies that prevented parallelization. In contrast to a compiler CAPO allows interactive user guidance during the parallelization process. Parallelization could be achieved by privatizing certain arrays through the CAPO user interface.

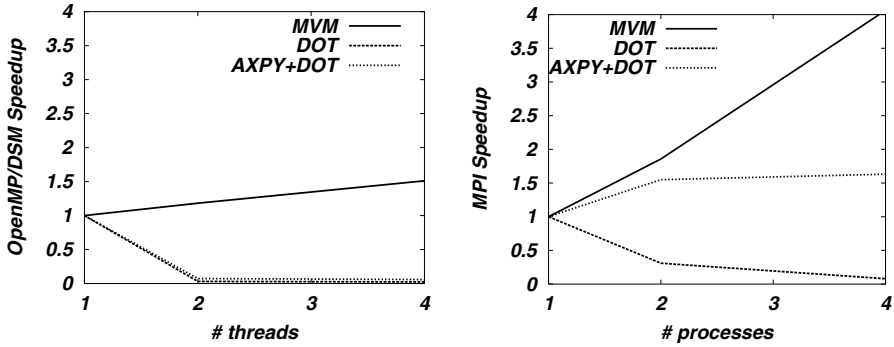


Fig. 4. Details of CG benchmark's class S. Speedups are shown for the matrix-vector multiplication (MVM), for the dot-product (DOT) and AXPY+dot-product (AXPY+DOT). (a) results for DSM, (b) results for MPI

If nx, ny, and nz denote the number of grid points in each of the spatial dimensions,the sizes of the benchmark classes under consideration are given as:

Class S: nx= 64, ny= 64, nz= 64
Class W: nx=128, ny=128, nz= 32
Class A: nx=256, ny=256, nz=128

The speedup for OpenMP/DSM, MPI, and OpenMP/SMP versions for our three benchmark classes is shown in Fig. 5. For 8 nodes the OpenMP/DSM implementation achieves about 70% of the MPI speedup, for class W 65% and for class S 50%. The OpenMP/DSM speedup is limited to about 4 out of 8 processes compared to 6 out of 8 for the MPI implementation. To understand the performance difference we examine the different steps of the FT benchmarks in detail. In both implementations, the 3-D FFT is accomplished by performing a 1-D FFT in each of the three spatial dimensions. For each spatial dimension the three-dimensional array is copied into a one-dimensional array, the FFT is performed on the one-dimensional array, and the result is copied back. A code fragment for the first dimension is shown Fig. 6.

The OpenMP parallelization is achieved by inserting an OMP PARALLEL DO on the outermost loop. This results in a distribution of the data in dimension of K corresponding to the z-direction. The speedup for the individual three spatial dimensions for the OpenMP implementation on the class A benchmark is shown in Fig. 7. While the FFT in x and y dimension reach a speedup of 6 out of 8, the speedup in z-dimension is only 2 out of 8. The performance loss in X and Y dimension is mostly due to communication caused by writing to the shared array U which is indicated by page requests within this loop. Logically there is no communication required for this loop, since only the local part of the array is accessed. The performance decrease for the z-dimension is due to

the fact that here the outermost loop of the loop nest from Fig. 6 runs in J and not in K dimension. Since the data was distributed in K dimension, parallel execution of the loop requires access to remote data and causes a large number of page requests. The MPI implementation performs a transpose of the three-dimensional array in z-dimension, which is achieved by a call to `MPI_ALLTOALL`. This causes some decrease in performance, but not as severe as in the DSM system.

5 Problems Encountered

The installation of SCore, SCASH and Omni OpenMP was rather straightforward. For the basic SCore installation we tried to use aggressive compiler optimizations whenever possible and we went through an iterative process to find a stable configuration in terms of compiler settings. The SCASH and Omni OpenMP configurations were based on the one found for the basic SCore system. We were able to run all tests and examples delivered with either SCASH or the Omni OpenMP compiler suite successfully.

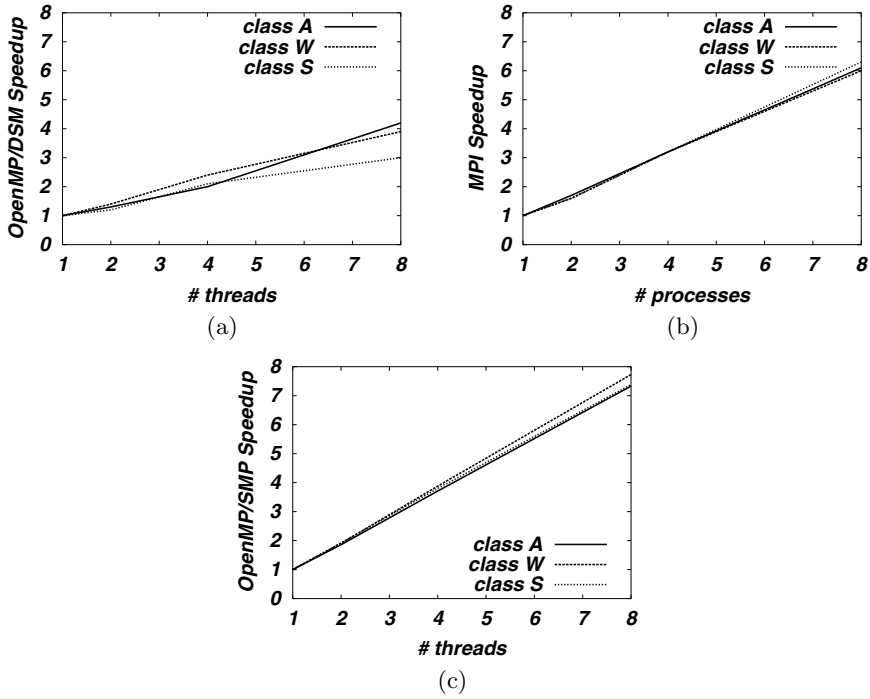


Fig. 5. Comparison of MPI and OpenMP/DSM speedups for classes A, W and S of the FT benchmark. (a) Speedup for OpenMP/DSM, (b) MPI Speedup, (c) Speedup on the SMP system

```
do k = 1, n3
      do j = 1, n2
         do i = 1, n1
            w(i) = u(i,j,k)
         enddo

         call fft (w,...)

         do i = 1, d(1)
            u(i,j,k) = w(i)
         enddo
    enddo
enddo
```

Fig. 6. Code fragment for the first dimension of FFT

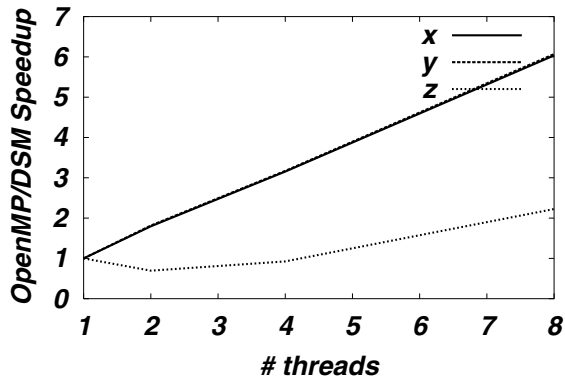


Fig. 7. Speedup for different directions of the FFT Class A benchmark on the DSM system

We ran into problems when trying to run the three kernel benchmarks EP, CG, and FT for larger problem sizes such as they are given class B or C. We also could not run any of the simulated CFD applications BT, SP, and LU that are part of the benchmark suite, even for the small problem size given in class A. The problems we encountered were due to the fact that SCASH was not able to allocate enough of virtual memory. The SCASH system itself uses a large amount of memory for its own memory management on top of the one provided by the operating system. To improve data exchange performance (i.e. bandwidth and latency) SCASH specifically allocates pin-down memory [22]. For larger benchmark classes it seems that there is not enough pin-able memory available.

Another severe restriction is the 32 bit address-space of the IA32 architecture. With 32 bit addresses the address-space is restricted to at most 2^{32} addresses, equal to 4 GB of main memory. Usually the memory management of operating systems like Linux or Windows [1] allows a process to use only part of this address-space for its private data. The operating system uses the rest to mirror some internal data structures into the process' virtual address-space. Under Linux a process can only use 2 GB of the theoretical maximum of 4 GB for its private data, because Linux maps kernel structures and data into the upper 2 GB.

Without additional effort, the kernel itself would suffer from this 4 GB barrier. To enable the use of more main memory, IA32 Linux uses Physical Address Extension (PAE) to access up to 64 GB. This is achieved by having a three stage page address translation mechanism. But even with this system, only the kernel can handle more than 4 GB. A single process is still restricted to 2 GB of private memory.

A software distributed shared memory system like SCASH that runs in user-mode and uses a 32 bit global address-space will therefore be restricted to a maximum of 4 GB global shared memory.

6 Related Work

Another system supporting the OpenMP paradigm on distributed memory systems is TreadMarks [2]. Comparisons of the TreadMarks systems with message passing programming are given in [7] and [11]. Other systems that support software DSM programming are Cashmere [18] and SMP-Shasta [15] There are a number of papers reporting on comparisons of different programming paradigms. As an example we name [16] and [17] where message passing and shared memory programming are compared on shared memory architectures.

7 Conclusions and Future Work

We have evaluated the performance of OpenMP/DSM implementations of three of the NAS Parallel Benchmarks on a commodity cluster of PCs. We compared the speedup to the speedup obtained with MPI implementations of the same algorithms. The difference in performance depends on the structure of the application and the problem size. For the largest problem sizes under consideration the observed OpenMP/DSM speedups range between 100% and 70% of the MPI speedup for all benchmarks. In cases with an extremely high communication to computation ratio the OpenMP/DSM speedup is significantly lower than that obtained by MPI. This occurs in the smallest class of the CG benchmark, where AXPY and dot-product operations for short vector lengths are being parallelized. We have noticed that in this extreme case part of the performance decrease was due to compiler deficiencies which also show on a shared memory system. The memory problems described in section 5 are implementation dependent and we

[1] Windows is a registered trademark of Microsoft Corp.

expect them to be resolved in commercial software. Usage of 64 bit system software and kernel enhancements to support DSM on a system level will improve the general usability of DSM systems.

All in all we are encouraged by the results we obtained considering the fact that we were using public domain software. The DSM system allowed us to exploit parallelism over all nodes of the cluster by using automatically parallelized code based on OpenMP. We find the performance differences when compared with hand-optimized MPI code acceptable when we take into account the extremely short development time of the parallel code. Our future plan is to run full size applications in our testbed environment.

Acknowledgments

We would like to thank Jahed Djohmeri and Rob van der Wijngaart of NAS for reviewing the paper and the suggestions they made for improving it. The authors also wish to thank Rob van der Wijngaart for many helpful discussions about the NAS Parallel Benchmarks and DSM systems. Part of this work was supported by NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D with Computer Sciences Corporation.

References

1. `http://www.conservativecomputer.com/myrinet/perf.html`.
2. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.
3. D. Bailey, J. Barton, T Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA, 1991.
4. D. Bailey, T. Harris, W. Saphir, R van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995. `http://www.nas.nasa.gov/Software/NPB`.
5. Phillip Ezolt. A Study in Malloc: A Case of Excessive Minor Faults. In *Proceedings of the 5th Annual Linux Showcase & Conference, November 5–10*, 2001.
6. H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic Home Node Reallocation on Software Distributed Shared Memory. In *Proceedings of HPC Asia 2000, Beijing, China*, pages 158–163, May 2000.
7. Y. C. Hu, H. Lu A. L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. In *Proceedings of the Thirteenth International Parallel Processing Symposium*, pages 302–310, 1999.
8. C. S. Ierotheou, S. P. Johnson, M. Cross, and P. F. Leggett. Computer Aided Parallelisation Tools (CAPTools)-Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes. *Parallel Computing*, 22:163–195, 1996.
9. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NAS, 1999.
10. H. Jin, M. Frumkin, and J. Yan. Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes. In *Proceedings of Third International Symposium on High Performance Computing (ISHPC2000), Tokyo, Japan, October 16-18*, 2000.

11. H. Lu, S. Dwarkdadas, A. L. Cox, and W. Zwaenepoel. Quantifying the Performance Differences Between PVM and TreadMarks. *Journal of Parallel and Distributed Computation*, 43(2):65–78, June 1997.
12. *MPI 1.1 Standard*. `http://www-unix.mcs.anl.gov/mpi/mpich`.
13. *Omni OpenMP and SCASH*. `http://www.pccluster.org`.
14. *OpenMP Fortran Application Program Interface*. `http://www.openmp.org`.
15. D. Scales, K. Gharachorloo, and A. Aggarwal. Finegran software distributed shared memory on SMP clusters. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 125–136, February 1998.
16. H. Shan and J. Pal Singh. A comparison of MPI,SHMEM, and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessor. *International Journal of Parallel Programming*, 29(3), 2001.
17. H. Shan and J. Pal Singh. Comparison of Three Programming Models for Adaptive Applications on the Origin 2000. *Journal of Parallel and Distributed Computing*, 62:241–266, 2002.
18. R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L:Software coherent shared memory on a clustered remote write network. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 170–183, October 1997.
19. K. Taura, S. Matsuoka, and A. Yonezawa. StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Proceedings of Workshop on Theory and Practice of Parallel Programming*, pages 121–136, 1994.
20. H. Tezuka, A. Hori, and Y. Ishikawa. Design and Implementation of PM: a Communication Library for Workstation Cluster. In *JSPP'96, IPSJ*, pages 41–48, June 1996. (In Japanese).
21. H. Tezuka, A. Hori, and Y. Ishikawa. PM: A High-Performance Communication Library for Multi-user Parallel Environments. Technical Report TR-96015, RWC, November 1996.
22. H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Managment Technique for Zero-copy Communication. Technical Report TR 97006, Tsukuba Research Center, Real World Computing Partnership, 1997.

Managing C++ OpenMP Code and Its Exception Handling

Shi-Jung Kao

Hewlett-Packard Company
shi-jung.kao@hp.com

Abstract. This paper discusses the issue of C++ exception handling in C++ OpenMP Programs. Two possible implementation techniques are described and contrasted. This paper also suggests ways to synchronize the execution of C++ OpenMP programs in the event of uncaught exceptions.

1 Introduction

The OpenMP Standard states that a program that throws an exception within a parallel region and fails to catch that exception within that parallel region is non-conforming. Because the standard does not address the behavior of non-conforming programs, the compiler is free to choose any behavior. Users, however, can have certain expectations, and for this reason, behavior consistent with the C++ standard exception handling mechanism and other non-OpenMP code would be desirable.

1.1 Current Standards Requirements

The "OpenMP C and C++ Application Program Interface", Version 2.0, March 2002, specifies that "A throw executed inside a parallel region must cause execution to resume within the dynamic extent of the same structured block, and it must be caught by the same thread that threw the exception."

The C++ ISO Standard states "when an exception is thrown, control is transferred to the nearest handler with a matching type [15.1 -2-]" and "…exception handling must be abandoned for less subtle error handling techniques: … when the exception handling mechanism cannot find a handler for a thrown exception … [15.5.1 -1-]".

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 227–243, 2003.

1.2 Examples

Consider the following examples:

Example 1.

```
try {
      #pragma omp parallel
      { // region A
            …
            throw 5;
      }
      …
  } catch (…) {
      … // handler
  }
```

(An OpenMP parallel construct within a C++ try block.)

Example 2.

```
try {
      #pragma omp parallel
      {
            #pragma omp parallel
            {// region A
                  …
                  throw 5;
            }
            …
      }
  } catch (…) {
      … // handler.
  }
```

(Nested OpenMP parallel constructs within a C++ try block. It has more threads then the other example)

#pragma omp parallel defines a parallel region, which is a region of the program to be executed by multiple threads in parallel.

When a thread encounters an OpenMP parallel construct, a team of threads is created. This thread becomes the master thread of the team, and all threads, including the master thread, execute the region in parallel.

In both examples, the execution of the master thread transfers control to its handler while all other threads created for the parallel region are terminated after the "throw" statement. The execution of the handler may not be finished before the whole execution is terminated (aborted).

While both examples can be exposed by modern compilers which can then issue warnings about applications throwing an exception inside a parallel region without a handler, more complex cases can be difficult to detect.

1.3 Expensive Runtime Diagnostic

What we need is a runtime implementation that can do either of the following:

- Detect such an uncaught C++ exception from an OpenMP parallel region and issue a diagnostic.
- Treat the exception like a C++ uncaught exception.

Although issuing a runtime diagnostic seems to be a straightforward way to deal with an uncaught C++ exception from an OpenMP parallel region, it is expensive. The C++ standard does not require issuing such a diagnostic on an uncaught exception. This would be an add-on feature to our C++ compiler that would not only require changes in our C++ exception runtime library but would also incur a performance penalty for all non-OpenMP applications and all OpenMP applications that have no exceptions due to a common code path during execution.

Would it not be better to make the runtime mechanism for uncaught exceptions within a parallel region similar to that for uncaught exceptions in the C++ standard?

In Appendix A at the end of this paper, a method of analyzing runtime uncaught exception is suggested.

2 Implementation Possibilities

There are three possible implementations:

1. Make the parallel region a function call with a throw( ); that is, use a C++ exception specification.
2. Associate the parallel region through try block.
    a. Put a try block around the parallel region with a catch-all handler to terminate the execution when necessary.
    b. Reset the try block when entering a parallel region and let C++ standard behavior take care of uncaught exceptions.

Note that none of this is necessary if the parallel code cannot throw an exception. (Such behavior is not unusual and could be implemented.)

2.1 Using an Exception Specification

The set of exceptions that might be thrown can be specified in the C++ function declaration as follows:

void f( ) throw ( ); // exception specification with empty list.

Function f throws no exception, and during execution, any uncaught exception inside f would be transformed into a call of std::unexpected. The std::unexpected( ) call would then call std::terminate( ).

It seems reasonable to mark the parallel region with an empty exception-specification (throw( ), for example), considering that the behavior of uncaught exceptions within a parallel region and the behavior of uncaught exceptions within function f are similar.

We implemented a parallel region as a function. Although labeling such a function with "throw ( )" would be easy, we rejected this approach for two reasons:

1. Enabling the master thread to continue to search for its handler beyond the parallel region would require changes in the runtime library.
2. It might have been necessary to add the exception specification to the "pragma omp parallel" in the future.

In a similar situation, marking the function that contains the parallel region with an empty exception specification would also not achieve the desired behavior. This approach would complicate the situation and interfere with the user's exception specification, making it difficult to separate exceptions from parallel regions and other places.

2.2 Using a Try Block

Two implementations should be considered for a compiler to generate code to manage uncaught OpenMP exceptions during runtime:

1. Use the existing internal structure layout[1] of a try block, and emit a catch-all handler for the entire parallel region. This implementation would cause the runtime to search for the handler and stop at the catch-all handler, after which execution can be terminated. The implementation must find a way to let the master thread continue its execution beyond such a parallel region if this behavior matches its default behavior.

[1] Internal structure layout of "try block" – To ensure that make sure the C++ exception is be handled properly and all objects associated with to the try block are destroyed/destructed correctly when necessary, the compiler must maintain has to keep some internal information structures about the layout of the try block available at both compiler time and run time.

2. Reset the internal layout of the try block to no nested try block outside the parallel region when the compiler encounters the first try block inside this parallel region. The search for the handler inside the function f that contains this parallel region stops within the parallel region, even if the parallel region is nested inside a try block of function f. If there is no stack unwinding, everything should terminate correctly.

2.2.1 Emitting a Catch-All Handler for the Entire Parallel Region

```
#pragma omp parallel
{
      // Body of parallel region
      …
}
```

This approach can be implemented semantically in two ways, as shown in the following code fragments. Note that the semantic meanings can vary according to the compiler's implementation.

a)
```
#pragma omp parallel
try {
      // Body of parallel region
      …
} catch (…) {
      std::unexpected();
}
```

b)
```
try {
      #pragma omp parallel
      {
            // Body of parallel region
            …
      }
} catch (…) {
      std::unexpected();
}
```

The advantage of this implementation is that the compiler needs to emit only an extra try block along with its catch-all handler for the OpenMP parallel region. The compiler therefore has less chance of breaking existing non-OpenMP applications. The obvious disadvantage is that an extra try block per OpenMP parallel region can be expensive, especially in a small parallel region. Because this is an add-on feature, it is important that the behavior be consistent with the behaviors of existing C++ exception applications.

The following example shows how the output size of an assembly listing increases with this implementation.

Example 3.

```
int main ( ) {
      int i = 0;
      #pragma omp parallel
      {
            i = 5;
      }
      return 0;
}
```

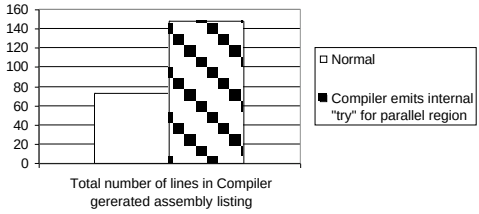


Fig. 1. After the compiler emits the try..catch(…) handler, its assembly listing size doubles

Consider also the case of a parallel region within a user-declared try block, where the new behavior of uncaught C++ exceptions in a parallel region meets the existing behavior of C++ exceptions in a nested try block.

User's code

```
try {
      …
      #pragma omp parallel
      {
            // Body of parallel region
            …
      }
} catch (…) {
      // Handler code.
}
```

Compiler emits equivalent code

```
try {
      …
      #pragma omp parallel
      try {
            // Body of parallel region
            …
      } catch (…) {
            std::unexpected();
      }
} catch (…) {
      // Handler code.
}
```

This code, an OpenMP parallel region within a try block, semantically directs mapping to code with nested try blocks. When a thread from the inner try block terminates, its cleanup process can cause the handler of the outer try block to be executed. It can also lead to the destruction of all objects constructed in the outer try block, even though they do not exist for some threads from the inner try block. That is, threads are created in a parallel region other than the master one. Attempted destruction of non-existing objects can then cause serious problems.
The following example illustrates the correct and incorrect number of objects to be destroyed.

Example 4 (Assuming OMP_NUM_THREADS is 4).

```
struct Object {
      int count;
      Object ( ) { }
      ~Object ( ) { }
}

extern void foo(Object obj);

int main ( ) {
      try {
            Object obj1;
            #pragma omp parallel
            {
                  foo(obj1);
            }
      } catch (…) {
            // Handler code.
      }
}
```

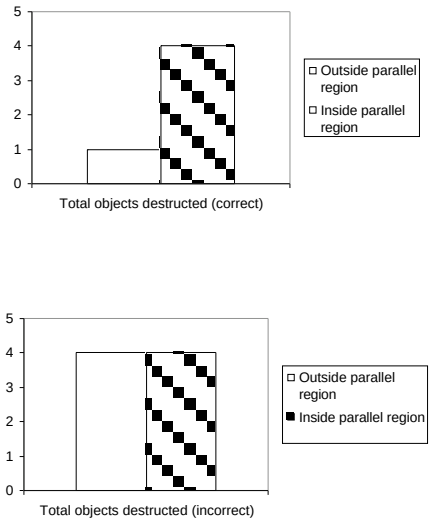


Fig. 2. A total of four destructions should occur within the parallel region (that is, one copy of obj1 per thread is destroyed) and only one destruction should occur outside the parallel region (the *obj1* of the master thread)

Because the C++ standard states that the mechanism for unwinding the stack after the function terminate( ) is called is implementation-defined (see following), and because the "unwind" can lead to improper destruction of objects, a reset of the try block when entering an OpenMP parallel region could work well if this behavior matches the existing implementation-defined behavior.

The C++ ISO Standard states "… If no matching handler is found in a program, the function terminate( ) is called; whether or not the stack is unwound before this call to terminate( ) is implementation-defined.[C++ ISO Standard, 15.3]"

In this example, to prevent threads from destroying the obj1 illegally, we must reset the internal layout of the try block when execution enters the parallel region. The master thread, however, could be a problem.

2.2.2 Resetting the Internal Layout of the Try Block

To obtain correct behavior in some environments, the compiler must modify the layout of the try block to prevent the thread from accessing the environment of a try block that does not belong to it. In other words, the compiler must reset the layout of the try block when entering the parallel region. This behavior recognizes that each thread within the parallel region has no pre-existing try block.

In the previous example, if all four threads were trying to destroy obj1 (continue their cleanup outside the parallel region), the only way to prevent this destruction would be to reset the internal layout of the try block when execution enters the parallel region. This reset ensures that the handler search stops at the parallel region. For example:

Example 5.
User's code

```
try {
      …
      #pragma omp parallel
      {
            // Body of parallel region
            …
      }
} catch (…) {
      // Handler code.
}
```

Compiler emits equivalent code

```
      #pragma omp parallel
      { // Not within a try block
            // Body of parallel region
            …
      }
```

Advantages: If no additional code is added, there is no performance penalty.
Disadvantages: Because the compiler must change or reset some structure contents of the try block's internal layout, chances of breaking the compiler's existing behavior increase.

Consider these two cases:

- A parallel region within a try block.
- A parallel region within the dynamic extent of a try block. The "reset" of the internal layout of a try block is addressed in the first section. The runtime library must ensure that only the master thread crosses the parallel region during the unwind process.

3 Comparison of Results on EPCC OpenMP Microbenchmarks (http://www.epcc.ed.ac.uk/research/openmpbench)

EPCC tests results ran on an ES40 AlphaServer with four processors in HP Tru64 UNIX V5.0a, showing no performance difference between these two implementations under the test of Synchronization Overhead (All four C programs were compiled with the C++ Compiler. The EPCC contains two sets of tests: Synchronization Overhead and Scheduling Overhead.)

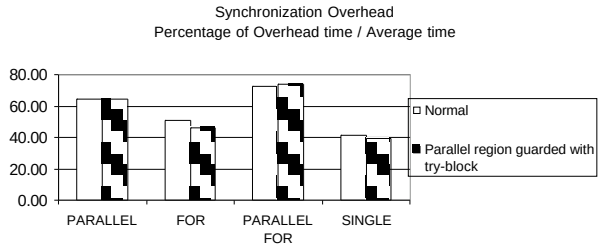


Fig. 3. Synchronization overhead

On the other hand, the results for the Scheduling benchmark, which compares the overhead of the DO directive using different scheduling options with chunk sizes, did incur penalties on the parallel region that was guarded with additional try blocks.

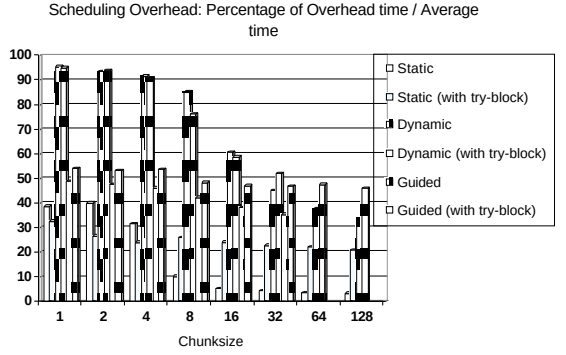


Fig. 4. Scheduling overhead: Percentage of overhead time / average time

The Scheduling Overhead showed that parallel regions guarded with a try block had a higher overhead time in general. Data was collected with the following characteristics:

Running OpenMP benchmark on 4 thread(s)
Assumed clock rate = 667 MHz
Delay length = 14
Delay time = 65.165900 cycles

Note that this data was collected from a prototype HP C++ Compiler on Tru64 UNIX, and is used only for comparison of the two experimental implementations discussed in this document.

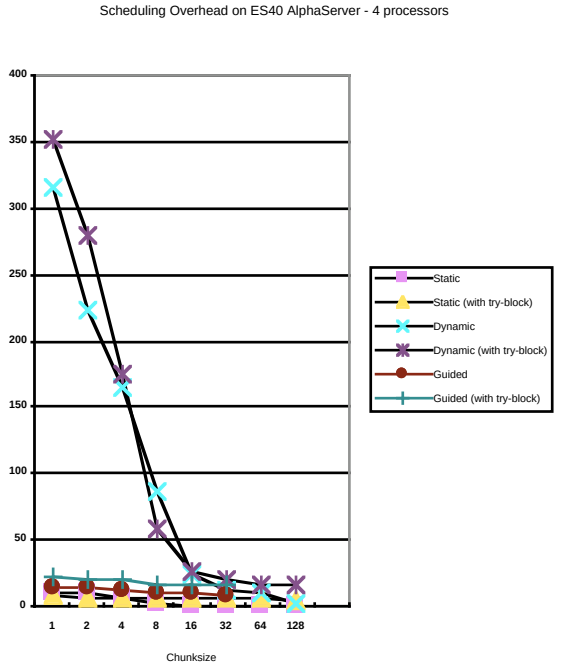


Fig. 5. Scheduling overheads (microseconds)

4 Conclusions

Adding a new feature to an existing compiler with minimal work is good practice, and avoiding runtime library changes at the same time is even better — unless functionality, compatibility and performance become issues. An uncaught C++ exception within a parallel region can make cleanup complicated during the unwind process. Such an uncaught exception can require a runtime check to identify the master thread from a parallel region inside the dynamic extent of a try block and to let that thread continue its execution and comply with the C++ standard.

It is fair to conclude that resetting the internal layout of try block is a better solution if no changes are needed in the runtime library for the cleanup and unwind processes.

While epcc benchmark tests show some performance degradation when the compiler emits an extra try block to guard the parallel region, a similar study at Digital Equipment Corporation (now Hewlett-Packard Company) in 1998 showed that optimizing away unneeded C++ exception handling (based on the paper "Optimizing Away C++ Exception Handling" by Jonathan L. Schilling from SCO, Inc.) would reduce object sizes in C++ applications from between 3.5% to 10%.

The following chart compares the object size of the C++ Standard Library before and after such optimization:

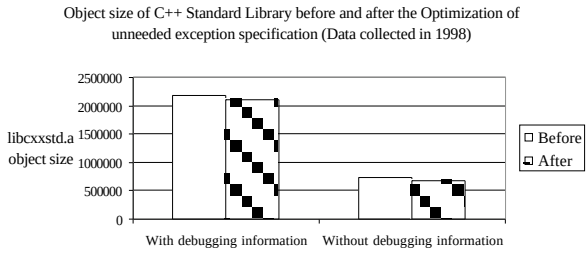


Fig. 6. Optimization on unneeded exception specification

5 Future Direction

We might want to consider adding an exception specification to the syntax of "omp parallel". The OpenMP applications could then tell compilers whether and which kind of C++ exception can occur. If the compiler knows that the parallel region can have

no C++ exceptions, it can generate more efficient code. A C++ exception specification added to the "clause" of an OpenMP parallel construct would have the following syntax:

#pragma omp parallel throw( )

Acknowledgments

Larry Weissman, August Reinig, Kenneth Block, Lazarus Guisso, Kevin Harris, Premanand Rao, Bob Morgan, John Paolillo, and Kristine Kao contributed useful comments from both technical and editorial review of this paper.

References

OpenMP C and C++ Application Program Interface, Version 2.0, March 2002

C++ Front End Internal Documentation, January 19,1996, Edison Design Group, Inc.

The C++ Programming Language Third Edition, Bjarne Stroustrup. Published by Addison Wesley

STL Tutorial and Reference Guide, David R. Musser. Published by Addison Wesley

Tru64 UNIX Guide to DECthreads, July 1999, Compaq Computer Corporation

Tru64 UNIX Version 5.1B, Online document, Hewlett-Packard Company

Digital C++ Exception Handling Implementation, September 18, 1998, Coleen Phillimore, Digital Equipment Corporation

DEC C Design Note DN043, Microsoft Structured Exception Handling in DEC C Alpha VMS, August 17 1992, Digital Equipment Corporation

Optimizing Away C++ Exception Handling, Jonathan L. Schilling, SCO Inc.

Internal C and C++ Notes, December 1998, Digital Equipment Corporation

The EPCC Microbenchmarks, http://www.epcc.ed.ac.uk/research/openmpbench

C++ ISO Standard Paper, http://www.iso.ch/iso/en/ISOOnline

Appendix: Debugging Uncaught Exception from a Parallel Region

Function set_unexpected( ) and set_terminate( )

Debugging a parallel application is difficult. Debugging a parallel application that causes "uncaught" C++ exceptions is even more difficult. Two functions that can be helpful in analyzing the problem are

set_unexpected [ISO 18.6.2.3 lib.set.unexpected]
unexpected_handler set_unexpected(unexpected_handler *f*)throw();

This function sets f as the current unexpected handler. When a function exits via an exception not allowed by its *exception-specification,* the unexpected( ) function is called, which then calls unexpected_handler.

set_terminate [ISO 18.6.3.2 lib.set.terminate]
terminate_handler set_terminate(terminate_handler *f*) throw();

This function sets f as the current handler function for terminating exception processing. When exception handling must be abandoned, the terminate( ) function is called, which then calls the terminate_handler.
The following example illustrates the use of these two functions.

Example 6 (Assuming OMP_NUM_THREADS is 4).

```
#include "cxx_exception.h"
extern "C" int printf(const char *,…);
extern "C" int exit(int);

void announce( ) {
      printf("Unexpected!\n");
      throw;
}

void my_terminate( ) {
      printf("Terminated!\n");
      exit(0);
}

void foo( ) throw( );
void foo( ) throw( ) {
      static int x = 44;
      throw &x;
}
```

```
int main( ) {
#pragma omp parallel
{
      set_unexpected(announce);
      set_terminate(my_terminate);
      try {
            foo( );
      } catch (int *p) {
            printf("Caught %d\n", *p);
      }
}
return 0;
} // main
```

Runtime output:

```
      Unexpected!
      Terminated!
```

If we rewrite my_terminate( ) and announce( ) into following:

```
void my_terminate( ) {
      printf("Terminated!\n");
#pragma omp barrier
      exit(0);
}// See output1 and output2
void announce( ) {
      printf("Unexpected!\n");
#pragma omp barrier
      throw;
}// See output2
```

Output1:

```
      Unexpected!
      Unexpected!
      Terminated!
      Terminated!
      Unexpected!
      Terminated!
      Unexpected!
      Terminated!
```

Output2:

```
      Unexpected!
      Unexpected!
      Unexpected!
      Unexpected!
      Terminated!
      Terminated!
      Terminated!
      Terminated!
```

Sometimes it is still difficult to debug an unexpected behavior even with all the debugger tools.
The following test was converted from an original non-OpenMP test in our compiler development testing environment. Considerable effort was required to determine why it caused an unexpected runtime hang.

Example 7 (Assuming OMP_NUM_THREADS is 4).

```
#include "cxx_exception.h"
extern "C" int printf(const char *,…);
extern "C" int exit(int);

void announce( ) {
      printf("Terminated!\n");
#pragma omp barrier
      exit(0);
}

struct Object {
      static int static_count;
      static int threshold;
      int count;
      Object ( ) {
            if (static_count >= threshold) {
                  threshold += 10;
                  throw static_count;
            }
            count = ++ static_count;
            printf("Making: %d\n", count);
       }
      ~Object ( ) {
            printf("Killed %d\n", count);
            throw count;
      }
};

int object::static_count = 0;
int object::threshold = 2;

struct derived {
      object obj[3];
      derived ( ) {  }
};

int main ( ) {
#pragma omp parallel
      set_terminate(announce);
```

```
#pragma omp parallel
      {
            try {
                  derived local;
            } catch (…) {
                  printf("Caught explicit\n");
            }
      }
      return 0;
}
```

The execution hung at the end because not every thread reached the barrier in the function announce( ).

Its runtime output showed something was missing.

```
Making: 1
Making: 4
Making: 5
Killed 5
Making: 3
Making: 6
Making: 7
Killed 7
Making: 2
Making: 8
Making: 9
Killed 6
Killed 4
Terminated!
Terminated!
Killed 9
Caught explicit
Killed 8
Terminated!
```

Improving the Performance of OpenMP by Array Privatization*

Zhenying Liu, Barbara Chapman, Tien-Hsiung Weng, and Oscar Hernandez

Department of Computer Science, University of Houston
{zliu,chapman,thweng,oscar}@cs.uh.edu

Abstract. The scalability of an OpenMP program in a ccNUMA system with a large number of processors suffers from remote memory accesses, cache misses and false sharing. Good data locality is needed to overcome these problems whereas OpenMP offers limited capabilities to control it on ccNUMA architecture. A so-called SPMD style OpenMP program can achieve data locality by means of array privatization, and this approach has shown good performance in previous research. It is hard to write SPMD OpenMP code; therefore we are building a tool to relieve users from this task by automatically converting OpenMP programs into equivalent SPMD style OpenMP. We show the process of the translation by considering how to modify array declarations, parallel loops, and showing how to handle a variety of OpenMP constructs including REDUCTION, ORDERED clauses and synchronization. We are currently implementing these translations in an interactive tool based on the Open64 compiler.

1 Introduction

OpenMP has emerged as a popular parallel programming interface for medium scale high performance applications on shared memory platforms. Strong points are its ability to support incremental parallelization, portability, and ease of use. However, the obstacles to scale an OpenMP code to hundreds or thousands of processors, as may be configured in ccNUMA systems, are latency of remote memory access, poor cache memory reuse, a large number of barriers and false sharing of data in cache. Good data locality is needed to overcome these performance problems.

It is possible to obtain data locality, as well as to minimize false sharing between threads on a ccNUMA system, via manual privatization of the arrays in an OpenMP program. However, a systematic use of array privatization that separates array elements that are shared by multiple threads from those elements of the same arrays that are not shared, requires extensive program modification. The so-called SPMD style of OpenMP programming is a systematic realization of this approach and it has been

*This work was partially supported by the DOE under contract DE-FC03-01ER25502 and by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 244–259, 2003.

shown to provide scalable performance [2, 3, 20] that is superior to a straightforward parallelization of loops for ccNUMA systems.

SPMD style OpenMP code is distinct from ordinary OpenMP code: In most OpenMP programs, shared arrays are declared and PARALLEL DO (or FOR) directives are used to realize a distribution of work among threads possibly via explicit loop scheduling. In the SPMD style, a systematic privatization of arrays, by creating private instances of (sub-)arrays, provides opportunities to spread computation among threads in a manner that ensures data locality. When data that have been privatized need to be shared by two or more threads, the programmer must insert additional data structures and code to achieve this. This is because in OpenMP programs, interactions between threads occur via shared variables only; each thread may moreover access its own private variables which are not visible to other threads. Thus a number of non-trivial program modifications are required to convert a program to the SPMD style.

It is thus hard for a user to write SPMD style OpenMP code, especially for a large application. Ease of program development is a major motivation for adopting OpenMP and it is important to provide some help for users who wish to improve the performance of a rapidly created program by adopting SPMD style. One approach is to provide a tool that supports the generation of SPMD style OpenMP code, either from a sequential program or from an OpenMP code with loop-level parallelism. We are developing just such a tool. It is being implemented using the open source Open64 compiler [14] infrastructure. The tool is interactive, because we believe that user involvement is essential for good results.

This paper is organized as follows. In the next section, we describe the basic translation to SPMD style, using a simple Jacobi code fragment to show the process step by step. The subsequent sections discuss the translation of OpenMP directives and clauses that may be encountered in the original OpenMP program. After this, an overview of related work is given and some conclusions are reached.

2 Overview of SPMD Translation

Programs written using SPMD style with array privatization have shown superior performance [2, 3] on the SGI Origin 2000, a typical ccNUMA architecture. Indeed, their superior data locality has even been shown to improve performance on small SMPs [2]. Our goal is to help the user translate OpenMP programs into equivalent SPMD ones, which are expected to achieve data locality and high performance in ccNUMA systems, without introducing extensions to OpenMP. We follow a translation strategy that privatizes a program's arrays automatically or semi-automatically by following the OpenMP semantics of loop scheduling. The work described here assumes we know which arrays are to be privatized and the manner of privatization.

SPMD (Single Program Multiple Data) fashion code is parameterized by the thread number and loaded by all the target processors. Arrays are generally distributed among threads in such a manner that the region of the array assigned to a thread becomes a private data structure and is allocated locally. If appropriately carried out, this permits a high degree of data locality. This implies that additional work is needed when-

ever privatized array elements are used by other threads. In the following, we outline the general techniques required to translate OpenMP code into SMPD style for execution on a ccNUMA machine. While doing so, we point out the difference between the array privatization required by this style and the privatization commonly used in OpenMP codes.

2.1 SPMD Style with Array Privatization

Our privatization approach in support of SPMD translation differs from other work on array privatization [19]. First, in the previous research, the following condition must be satisfied in order to privatize an array: every fetch to an array element in a loop must be preceded by a store to the element in the same iteration of the loop. In other words, there may be no dependences between loop iterations involving this array. This condition does not need to hold in order for us to privatize an array to generate SPMD style code. Second, a thread may access array elements that are private to other threads in SPMD style code, while this is not allowed in previous array privatization. We must naturally perform additional code modifications in order to realize these accesses.

Basic Translation. In this section, we describe the process of translating loop-parallel OpenMP programs into SPMD style code. We indicate the modifications needed for array declarations when privatizing arrays, the basic modification of loop nests, and a strategy for sharing the privatized data between threads when it is required by the code. We show a simple example of a Jacobi code which approximates the solution of a partial differential equation discretized on a grid, and translate this straightforward OpenMP program in Fig. 1 into corresponding SPMD style in Fig. 2. We discuss the translation of this code fragment to illustrate our approach in the following subsections.

The Declaration of Private Arrays. In order to obtain an SPMD program, the major arrays must be distributed among the threads. Within the context of OpenMP, data distribution is achieved via privatization. The approach is similar to that of MPI in the sense that a data distribution is chosen by the user or a tool and modified data structures are declared that have a shape and size corresponding to the (thread-) local portion of the array. In general, our translation strategy will choose to create threadprivate data structures, as threadprivate variables are globally accessible within the code. As a consequence of the above, the size of the threadprivate arrays at run time will depend on the manner in which we have partitioned and distributed the arrays, as well as their original size and the number of threads. The thread that declares and defines a section of a privatized array is called the owner of that section.
In our example code fragment, we privatize arrays A and B. The manner in which we do so is not arbitrary, but depends on the approach taken by the user to parallelize the original OpenMP code. Here, the j-loop has been parallelized (how to achieve the loop bounds *start_y* and *end_y* of the j-loop in Fig. 2 is further explained below) and the threads each access a contiguous segment of the second dimension of arrays A and B,

whereas all elements in the first dimension are accessed. (In the current OpenMP definition, only one loop nest may be parallelized and the parallelization of two or more loops, which will often correspond to distribution of two or more array dimensions, is only possible via collapsing of loops. We expect this to change in the future.) Array section analysis computes the regions of *A* and *B* that are read and written by individual threads. We use the array sections that are written by threads to decide how to distribute the arrays. For example, in *S1* statement of Fig. 2, the array section *A*[1:1000,*id*_1000/*numthreads*:(id+1)_1000/numthreads] is defined by the thread with identifier *id*, where *numthreads* is the total number of threads, since the default block schedule is used. Therefore we distribute array *A* by block in the second dimension according to the loop scheduling in the OpenMP program. In this case, the parallelization strategy enables us to distribute data in a manner that privatizes almost all references to both arrays. In a more realistic program, this may not be the case. Since OpenMP by its nature does not expect the user to compare the data usage implied by the parallelization strategy, it is possible that the loops chosen for parallelization imply different data distributions in different parts of the program. Since this will result in many nonlocal accesses, no matter which distribution we base our translation on, it will seriously detract from the performance benefits of our approach. This is a challenge for our translation process and is one of the reasons why we do not expect to fully automate this translation. We return to this topic below. But note that achieving consistency in thread data usage is likely to benefit even small programs executed on just a few CPUs [2, 3].

```
double precision A(1000,1000)
double precision B(1000,1000)
   call omp_set_num_threads(4)
!$omp parallel default(shared)
!$omp& private(i,j)
   do k=1, 20
!$omp do
     do j = 2 ,999
        do i = 2 , 999
S1:        A(i,j) = ( B(i-1,j)+B(i+1,j)
               + B(i,j-1)+B(i,j+1)) * c
        end do
      end do
!$omp do
      do j=1, 1000
        do i= 1, 1000
S2:        B(i,j) = A(i,j)
        end do
       end do
     end do
!$omp end parallel
```

Fig.1. Jacobi kernel in OpenMP

```
double precision Aloc(1000,0:251),Bloc(1000,0:251)
double precision buf_upper(1000,-1:5)
double precision buf_lower(1000,-1:5)
!$omp threadprivate(Aloc, Bloc)
!$omp parallel default(shared)
!$omp& private(shtart_y,end_y,i,j,k,id)
       id = omp_get_thread_num()
       do k = 1, 20
 S1':    buf_upper(1:1000, id)=Bloc(1:1000,250)
 S2':    buf_lower(1:1000, id) = Bloc(1:1000, 1)
!$omp barrier
 S3':    Bloc(1:1000,0)=buf_upper(1:1000,id -1)
 S4':    Bloc(1:1000,251)=buf_lower(1:1000,id+1)
         do j=start_y, end_y
           do i=2, 999
 S5':          Aloc(i,j) = ((Bloc(i-1,j) + Bloc(i+1,j)
                      +  Bloc(i,j-1) + Bloc(i,j+1)) * c
           end do
         end do
         do j=1, 250
           do i=1, 1000
 S6':            Bloc(i,j) = Aloc(i,j)
           end do
         end do
       end do
!$omp end parallel
```

Fig. 2. Jacobi kernel in OpenMP SPMD

In our example translation (Figures 1 and 2), we have assumed that the program is executed by four threads for the sake of simplicity. The parallelization strategy assigns each thread the task of updating a contiguous block of columns of each of arrays *A* and *B*. We accordingly distribute a contiguous block of columns of *A* and *B* to each thread; the corresponding private arrays have been renamed *Aloc* and Bloc. Note that the user has not specified a loop schedule; with such a schedule, the distribution of loop iterations, and hence of array updates, would change. We are able to declare the threadprivate arrays statically in this case, since the number of threads at run time is fixed (Fig. 1). Dynamic arrays have to be used if the user does not explicitly specify how many threads will execute the code. It is obviously more flexible to compile the program for an unspecified number of threads. We must use the relationship between an original globally shared array and the privatized array to translate array references. For example, *Aloc*(1:1000,1:250) for thread 0, 1, 2, and 3 in Fig. 2 represents *A*(1:1000,1:250), *A*(1:1000:1,251:500), *A*(1:1000,501:750) and *A*(1:1000,751:1000) in the original OpenMP program in Fig. 1 respectively. The translation process actually introduces a common block, since it is needed to pass privatized arrays across the procedures in SPMD code. In the examples of this paper, we simplify our discussion by only declaring the privatized array in a threadprivate directive without introducing a new common block.

```
!$omp parallel private(start_y, end_y, id, numthreads)
      numthreads = omp_get_num_threads()
      if( id .eq.0) then
        start_y = 2;     end_y   = 250
      else if (id .eq. (numthreads-1)) then
        start_y = 1;     end_y   = 249
      else
        start_y = 1;      end_y   = 250
      end if
      do j = start_y,  end_y
            …
      end do
!$omp end parallel
```

Fig. 3. Translation of loop bounds (start_y, end_y) of the first j-loop in Fig. 1 into SPMD style

Loop Translation. Parallel loops are the major source of parallelism in most programs and they drive our translation into SPMD style. The translation process must not only translate array references, but also deal with DO directives, the do loop control statement and the loop body. Some DO and END DO directives can be removed if a privatized array is inside a parallel region. In general, assignment statements are surrounded by guards (if-constructs) in SPMD code [18] so that they are only executed by the thread owning the privatized array element on the left hand side. In the case of statements within parallel loops, these loop bounds are modified so that the guards are redundant. We adapt the loop bounds so that each thread executes the portion assigned to it by the loop schedule. We may choose to retain global coordinates and use the thread id to parameterize it, or we may translate the array bounds and

hence all references to the loop variable in the original loop (as in our code). In each case, the loop control variable can be directly used as subscript for privatized arrays without modification. The original loop bounds of *j*, 2 to 999, are reduced to *start_y* and *end_y* in Fig. 2, whose calculation is demonstrated in Fig. 3.

Sharing Elements of Privatized Data between Threads. In OpenMP, threads interact via references to shared data. In the original OpenMP program, multiple threads may reference part of an array that has been privatized in our SPMD translation process. Since entire loop iterations are executed by a thread, it may occur that a thread needs to read data that have been privatized and are not local; it may also need to write non-local data as in an LBE program we previously studied [3]. For example, in our Jacobi code fragment, some elements of array *B* are accessed by more than one thread. Since *B* has been privatized in the SPMD version, some array elements private to a thread must be accessed (read, in this case) by another thread. We thus have the task of identifying such array elements and of modifying the code to ensure that the data are available: the owner thread must explicitly "export" this data. In our example, the privatized array *Bloc* on thread 1 will correspond to the original *B*(1:1000,250:500). This is equivalent to the region of the array that is written by it. However, thread 1 also needs to read the array section *B*(1:1000,249:501). When we compare this with the privatized region, we observe that two columns of the original array are used by thread 1 but are not immediately accessible to it: the non-local array elements *B*(1:1000,249) and *B*(1:1000,501) must be shared between threads as a result.

Wherever privatized data must be shared between threads, we require a translation from the original OpenMP program into SPMD style in the following steps:

(1) Shared buffers must be declared to store the non-local array references. The size of the shared buffers depends on the total number of threads, the number of array sections of non-local elements and the dimensions that are not privatized. For instance, in Fig. 2, *buf_lower* and *buf_upper* are declared for this purpose. Their declaration is double precision *buf_lower*(1000,4), *buf_upper*(1000,4). Because each of four threads accesses one column of B from neighbor threads, the total size of the buffer is one column multiplied by total number of threads. If the number of threads is not fixed, two dynamic arrays have to be declared to hold the data shared with neighbor threads at left and right boundaries.

(2) The size of the privatized array must be increased to create space to store the non-local data accessed. The adjusted size of the privatized array is obtained from the non-local access pattern. For example, *Aloc* and *Bloc* must be expanded by one column at both sides. The declarations of arrays Aloc and Bloc are thus modified to double precision *Aloc*(1:1000, 0:251), *Bloc*(1:1000,0:251). The additional memory spaces (1:1000, 0) and (1:1000, 251) are referred to as the shadow area [259], or ghost area. It is also common to append this storage directly to the local array in MPI SPMD programs, since it simplifies the translation process and is likely to provide better performance than using separate data structures.

(3) Executable instructions are generated to implement the sharing of private data between threads. Statements are inserted into the code to ensure that the owner thread writes data that are needed by another thread into shared buffers. In our example, *S1'* and *S2'* in Fig. 2 are inserted to copy two columns from array Bloc to *buf_upper* and *buf_lower*.
(4) Synchronization directives are inserted to ensure correct order of accesses to data in the shared buffers. In Fig. 2, data are written to, and should subsequently be retrieved from, shared buffers *buf_upper* and *buf_lower*; we need to insert a barrier to enforce this order of access. In general, we have several alternatives for implementing this, including barriers or a more loosely synchronous code.
(5) Executable instructions are generated to store the contents of shared buffers into the shadow area of the reading thread. Hence in Fig. 2, statements *S3'* and *S4'* are inserted. Since the local copies are part of a private data structure, we can benefit from its data locality in subsequent use. Note that these steps are only slightly modified if a thread writes non-local data.

2.2 General Strategy for SPMD Translation

In the previous section, we outlined the basic translation steps using our Jacobi example. Now we discuss a more general translation technique for SPMD style code, because we need to apply this strategy not only when the program is much more complex, but also in the presence of other OpenMP constructs.

The Transformation between Global and Local Representations. For a given parallel loop in an OpenMP program, we may need to translate references to both privatized arrays as well as arrays that remain shared. Basically, we follow the method of [18] that we will first remove the DO directive and reduce the loop bounds. The resulting array subscript is local to each thread. This method is described in the translation of Jacobi program. However, whenever the loop control variable appears outside a reference to a privatized array, we have to recover its original "global" value to ensure correct translation. This requires a local to global conversion (see) of subscripts. To show this, we choose to privatize array *A* only in the example of Fig. 3(a). The code is transformed into the SPMD style in Fig. 3(b) using the index conversion method; the global loop index *i* is transformed into local *iloc*, where *i* is equal to *iloc+id*chunk*. Thus we must replace the loop variable by the latter expression wherever it is not referred to in a privatized data structure.

In a few cases, the above method does not work. Hence an alternative translation is required. In this approach, we retain the original loop bounds and thus it is the subscript of the privatized array that is modified. The loop schedule ensures that threads execute the required iterations. Although the global *i* is untouched, we need the local index to access privatized arrays. The global to local conversion as described in Table 1 is thus required to deal with references to the privatized array *Aloc*; we substitute i-*id*chunk* for global *i*. Code generated using this method is displayed in Fig. 3(c). We can see the difference between these translations by comparing Fig. 3(b) and

(c). This second strategy is used to translate the OpenMP ORDERED directive in Section 3.1 below since we must keep the DO directive.

```
double precision A(200)
double precision B(200)
call omp_set_num_threads(10)
!$omp parallel do shared(A,B)
    do i=1,200
      A(i) = i + B(i)
    end do
!$omp end parallel do
…
```

(a) OpenMP example code

```
double precision Aloc(20),B(200)
!$omp threadprivate(Aloc)
   call omp_set_num_threads(10)
!$omp parallel shared(B)
   id = omp_get_thread_num()
   chunk = 200/omp_get_num_threads()
   do iloc=1,chunk
     Aloc(iloc)=iloc + id*chunk
              +B(iloc+id*chunk)
    end do
!$omp end parallel
…
```

(b) SPMD style with loop bounds reduction

```
double precision Aloc(10), B(200)
!$omp threadprivate(Aloc)
     call omp_set_num_threads(10)
!$omp parallel shared(B)
     id = omp_get_thread_num()
   chunk=200/omp_get_num_threads()
!$omp do schedule(static, chunk)
    do i=1,200
       Aloc(i-id*chunk) = i + B(i)
     end do
!$omp end do
!$omp end parallel
…
```

(c) SPMD style with transformed subscripts for privatized array

Fig. 4. An OpenMP example code, and its SPMD style code with loop bounds reduction and reduced array size. Array *A* is privatized into *Aloc*, while *B* is not to be privatized

Table 1 gives formulae used to adapt loop lower and upper bounds, and transform between local and global address spaces for the most common loop schedules. It can be used as a reference to translate parallel loops into SPMD style when the loop scheduling of OpenMP programs is the straightforward default chunk size and we follow this loop scheduling to privatize arrays.

Table 1. Creation of loop bounds, local and global indices [18]

Transformation	Block	Cyclic
Global to local lower loop bounds	*MAX((id_chunk)+1, L) –id_chunk*	*lb=((L -1)/numthreads + 1 If (id < MOD(L-1, numthreads)) lb = lb +1*
Global to local upper loop bounds	*MIN((id+1)_chunk,U) –id_chunk*	*ub = ((U-1)/numthreads) + 1 If (id > MOD(U-1, numthreads)) ub = ub – 1*
Global to local	*i - id _ chunk*	*(i – 1)/chunk + 1*
Local to global	*i+ id _ chunk*	*((i-1)_chunk) + 1 + id*
Global index to owned thread	*CEIL(N/numthreads) – 1*	*MOD(i-1,numthreads)*

numthreads = total number of threads; *id* = the thread number;
chunk = chunk size of local iterations; *L*, *U*: original lower, upper bounds;
lb, *ub*: transformed loop lower and upper bound

3 Translating OpenMP Constructs

In the previous section, we focused on the basic SPMD translation that permits us to convert an OpenMP program with some shared arrays into an equivalent SPMD style OpenMP program with threadprivate arrays. In this section, we discuss how efforts have to be exerted to translate other OpenMP directives and clauses that may appear in the code. We consider several DO directive clauses and other OpenMP constructs including synchronization directives.

3.1 Other Issues in Loop Translation

We have discussed the basic loop translation in Section 2. However, more work has to be done when some clauses, including REDUCTION and ORDERED of parallel do loops, are encountered.

```
    S= 0.0
!$omp parallel do default(shared)
!$omp& reduction(+:S)
      do i = 1, N
        S = S + A(i) * B(i)
      end do
!$omp end parallel do
```

(a) An OpenMP code with REDUCTION

```
!$omp threadprivate(Aloc, Bloc)
      S = 0.0
!$omp parallel reduction(+:S) default(shared)
      Sloc = 0.0
       do iloc=1,ub
         Sloc=Sloc + Aloc(i)*Bloc(i)
       end do
       S = S+ Sloc
!$omp end parallel
```

(b) The SMPD code

Fig. 5. Translation of reduction operations in OpenMP

REDUCTION Clause. If the arrays to be privatized are encountered in a parallel do loop with a REDUCTION clause, a new private variable has to be introduced to save the local result of the current thread; we must perform a global reduction operation among all the threads to combine the local results. The REDUCTION clause will be moved so that it is associated with the parallel region although it may be associated with a DO directive in original OpenMP program. The SPMD transformations for the do loops and privatized array are still needed. Fig. 4(a) and (b) depict the translation of a REDUCTION clause. Sloc, which is a private scalar, is introduced to accumulate the local sum for each thread taking advantage of privatized arrays *Aloc* and *Bloc*, and *S* is the result of a global sum of the local *Sloc* among all the threads. Variable *ub* has to be calculated according to Table 1.

ORDERED in a Parallel Loop. The ORDERED directive is special in OpenMP as neither CRITICAL nor ATOMIC can ensure the sequential execution order of loop iterations. So we have to retain not only the ORDERED directive, but also the corresponding PARALLEL and DO directives in the generated code due to the semantics of ORDERED. The translation of the parallel do loop follows the second method introduced in section 2.3, where the DO directive and loop control statement remain the same, while the index of the privatized array is converted from global to local format. A schedule that permits threads to access private array elements as far as possible is declared explicitly. The example in Fig. 5 shows an OpenMP program with the ORDERED directive and corresponding SPMD code. The generated loop in Fig. 5(b) has to sweep from 1 to *N* to ensure the correct distribution of iterations to threads. We use an IF construct inside the do loop to isolate the wasteful first and last iteration.

```
    integer N
     parameter(N=1000)
    double precision A(N)
    call omp_set_num_threads(10)
!$omp parallel do ordered
   do i=2, N-1
      …
!$omp ordered
      write (4,*) A(i)
!$omp end ordered
      …
   end do
!$omp end parallel do
```

(a) An OpenMP code with ORDERED

```
  double precision Aloc(100)
!$omp threadprivate(Aloc)
!$omp parallel
    id = omp_get_thread_num()
    chunk = N/omp_get_num_threads()
!$omp do ordered schedule ( static, chunk )
   do i=1, N
     if ( i .ge. 2 .and. i .le. N-1 ) then
          …
!$omp ordered
       write (4,*) Aloc(i-id*chunk)
!$omp end ordered
       …
      end if
    end do
!$omp end do
!$omp end parallel
```

(b) The SPMD code

Fig. 6. Translation of ORDERED directive into SPMD style

Different Loop Scheduling and Different Number of Threads. In an OpenMP program, we may encounter parallel loops whose execution requires that each thread access arrays in a pattern that is quite different from the array elements that they will need to access during the execution of other parallel loops. This is generally detrimental to performance, since it will not permit the reuse of data in cache. However, it may be a suitable way to exploit parallelism inherent in the code. The ADI (Alternating Direction Implicit) kernel provides a common example of this problem. Since this could also introduce substantial inefficiencies into an SPMD program, we have considered two solutions to this problem. In one of these, we create two private arrays, each of which is privatized in first and second dimension separately. When the loop scheduling is changed between these two dimensions, we have to transfer the content of the first privatized array to the second through a shared buffer. This is similar to performing data redistribution. In the other solution, a private array is used for one kind of loop scheduling; whenever the loop scheduling changes, the contents of the private array are transferred to a shared buffer, which is subsequently referenced. Both methods require a good deal of data motion. From previous experiments [9], we know that such sharing is very expensive unless there is a great deal of computation to amortize the overheads or a large number of processors are involved. In practice we may ask the user to decide how to privatize.

In our outline of the basic translation strategy, we discussed the case of static scheduling and a fixed number of threads. In our SPMD translation, we disable the dynamic, guided scheduling and dynamic number of threads including NUM_THREADS clause in order not to degrade the performance. It is hard to determine which iterations are executed by a thread under these scheduling methods and when there is a dynamic number of threads. The size of privatized arrays may change and a large number of privatized array elements may need to be shared between threads. All of these will prevent us from generating efficient code.

3.2 Translation of Other OpenMP Constructs

Parallel regions must replace serial regions to enable access to privatized arrays within them, since the thread number and threadprivate arrays that are essential for SPMD style code can only be obtained in a parallel region. After constructing the parallel region, we need to decide two things: which thread executes each statement of the parallel region, and whether we need to share the privatized data between threads.

During the translation, only assignment statements may be modified, while control statements are kept without modifications in the generated code. If a privatized array appears on the left hand side of an assignment statement, then the owner thread of the privatized array element executes this statement. If the private array is accessed on both sides of a statement, the owner thread of the privatized array element on the right hand side will execute this statement; data sharing statements are inserted if non-local array elements are referenced.

```
   double precision A(100), B(200)
   call omp_set_num_threads(4)
!$omp parallel shared(A,B)
…
!$omp master
      B(120) = A(50)
!$omp end master
…
```

(a) An OpenMP code with MASTER

```
   double precision A(100), Bloc(50)
!$omp threadprivate (Bloc)
   call omp_set_num_threads(4)
!$omp parallel shared(A)
…
      if ( id .eq. 2 ) then
         Bloc(20) = A(50)
      end if
…
```

(b) The SPMD code

Fig. 6. Translation of MASTER directive into SPMD style

MASTER and SINGLE Directives. The semantics of the MASTER directive and sequential regions is that the master thread performs the enclosed computation. We can follow the OpenMP semantics in such program regions by modifying the code so that privatized array elements are shared between the master thread and the owner thread, or we can let the owner thread of a privatized array element compute instead. Since both of these methods ensure correctness, we select the latter in our implementation for performance reasons, although it does not entirely follow the semantics of the original OpenMP program. For example, we elect to privatize *B* but not *A* in Fig. 6(a). The translated code for the MASTER directive is shown in Fig. 6(b). We can see that B(120) is mapped to Bloc(20) for thread 2, so thread 2 executes the code enclosed within the MASTER directive. More conditionals may have to be introduced for each statement of the code enclosed by the MASTER directive; these IF constructs may be merged during the subsequent optimization process. The code enclosed within a SINGLE directive could be translated in a similar way to that of the MASTER directive. The only difference between translating SINGLE and MASTER directives is that we have to replace END SINGLE with a BARRIER if NOWAIT is not specified.

Synchronization Directives. If a private array appears in the code enclosed by CRITICAL, ATOMIC and FLUSH directives, the enclosed code is translated according to the strategy outlined in Section 2. Since the owner of a privatized array element on the left hand side usually executes a statement, accesses to privatized arrays are serialized. This may allow us to move the statements associated with private arrays outside the synchronization directives, and improve the performance to some extent. Unfortunately, sometimes the control flow and data flow of the program will not allow us to carry out this improvement.

4 Current Implementation

The performance of the generated SPMD style OpenMP code will be greatly enhanced by precise analyses and advanced optimizations [16]. Many compiler analyses are required for the SPMD translation, for instance, dependence analysis in a do loop, array section analysis within and between the loops and between the threads, parallel

data flow analysis for OpenMP programs, the array access pattern analysis, affinity analysis between the arrays [7], etc. Interprocedural analysis is important to determine the array regions read and written in loop iterations. We need to calculate accesses to array elements accurately for each thread in order to determine array privatization and shared data elements, even in the presence of function calls within a parallel loop. Interprocedural constant propagation can increase our precision. For example, if we do not know whether a reference (for example, $A(i,j+k)$), is local, we have to make a conservative assumption that they are not, and share it between the owner thread and accessing thread, although this is not efficient. $A(i,j+k)$ may be known to be local to the current thread at compile time if the value of variable k can be fixed by interprocedural constant propagation.

Our translation from OpenMP into SPMD style OpenMP is being realized within our Dragon tool based on the Open64 compiler [14], a suite of optimizing compilers for Intel Itanium systems running on Linux that is a continuation of the SGI Pro64 compiler. Our choice of this system was motivated by a desire to create a robust, deployable tool. It is a well-structured code that contains state-of-the-art analyses including interprocedural analysis [8], array region analysis, pointer and equivalence analyses, advanced data dependence analysis based on the Omega test, and a variety of traditional data flow analyses and optimization. It also has a sophisticated loop nest optimization package. A version of the Dragon tool with limited functionality has been made widely available [5]. This extended version is aimed at helping the user generate SPMD code interactively, by providing them with on-demand analysis and assisting in the code generation/optimization process. The high-level forms of WHIRL, the intermediate representation (IR) in the Open64 compiler, are able to explicitly represent OpenMP constructs. We transform a standard OpenMP program into an equivalent SPMD style one at the high level, i.e. before the IR constructs corresponding to OpenMP directives and clauses are lowered. At that point, we may either unparse the resulting WHIRL into source OpenMP code, or we may continue by lowering the IR for the SPMD style code into low-level WHIRL and then object code for the IA-64. We will discuss our implementation, including the analyses and optimizations required for SPMD translation, in more detail in future publications.

5 Related Work

The idea of data privatization can be dated to the first version of OpenMP language. However, it is still challenging to write an OpenMP program with scalable performance for ccNUMA architectures due to the data locality problem. In fact, data locality issues and the strongly synchronous nature of most OpenMP programs pose problems even on small SMP systems architectures. Most related work to date has addressed the problem of obtaining performance under OpenMP on ccNUMA platforms.

Researchers have presented several strategies to solve this problem automatically, taking advantage of the first-touch policy and page migration policy provided either by some operating systems such as those on SGI [17] and Compaq systems [1], or by the user-level run-time libraries [12]. Hence program changes are minor. Besides, the

SCHEDULE clause in OpenMP may be extended to schedule the computation in a cache friendly way for regular and irregular applications [13]. These means attempt to minimize the intervention from users. But the operating system or run-time libraries are not able to know precisely when to migrate the page. Besides, the users are not able to fully control the behavior of operating systems and tune the performance.

Data distribution directives give users facilities for determining how to allocate data and arrange the computation locally. Decomposition of data in terms of page granularity and element granularity is supported by SGI and Compaq. Data and thread affinity are enabled by SGI to ensure the mapping of data with the corresponding computation; similar functionality is provided by Compaq's NUMA directive. There are some differences; SGI maps data to processors, whereas Compaq maps data to node memories. Extending OpenMP with data distribution directives may improve the data locality and decrease the remote memory access, but important features of OpenMP - incremental parallelism, and portability and ease for programmer – will suffer as a result. Our method is different from the data distribution method since we attempt to achieve the same result, data locality, without introducing more directives. The work distribution already implies a strategy for data distribution. Our goal is to permit the user to retain full control of the translation process by making them aware of the performance implications of their parallelization strategy and helping them change it as desired.

Some translation techniques used in this paper are similar to those developed for HPF compilation [18, 21], as both are targeted to generating SPMD code. For instance, HPF compilers also perform loop bounds reduction, global to local and local to global conversion. There are several major differences between this approach and HPF compilation, however. First, our SPMD code is not based on message passing, but on the use of shared data to exchange values. A pair of assignment statements with synchronization between them is enough to affect this exchange. Also the further optimizations of data sharing in OpenMP and optimizations of communication in MPI are different. Second, in HPF compilation, all the variables in the generated code must be local or private, whereas some variables could possibly remain shared in the translated SPMD style OpenMP code. This means that the user or tool may have several alternative strategies available when adapting a program in this way.

6 Conclusion and Future Work

In this paper, we have described a basic strategy for translating OpenMP code into SPMD style with array privatization for ccNUMA systems. Privatization is achieved via the threadprivate feature of OpenMP, and may introduce the need to account for sharing of privatized array elements between the threads. The translation of several OpenMP constructs, directives, clauses and their enclosed code into SPMD style is presented.

Our method of array privatization is based on mapping the data to threads. A disadvantage of this method is that we cannot ensure that logical neighbor threads are physically near to each other. On the other hand, if we want to share the private data among (say) two threads, we require global synchronization between the copies to and

from a shared buffer and will have to deal with false sharing between them. Without local synchronization, the generated code will be much less efficient. Unstructured or irregular problems are not covered in this paper.

Our translated codes are primarily targeted to ccNUMA platforms. On the other hand, we also plan to integrate SPMD OpenMP style code with Global Arrays [11], a portable efficient shared-memory programming model for distributed memory systems, so that we can implement OpenMP for distributed memory computers.

References

1. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson C.A., and Offner, C.D.: Extending OpenMP for NUMA machines. Scientific programming. Vol. 8, No. 3, (2000)
2. Chapman, B., Bregier, F., Patil, A., and Prabhakar, A.: Achieving High Performance under OpenMP on ccNUMA and Software Distributed Share Memory Systems. Currency and Computation Practice and Experience. Vol. 14, (2002) 1-17
3. Chapman, B., Patil, A., and Prabhakar, A.: Performance Oriented Programming for NUMA Architectures. Workshop on OpenMP Applications and Tools (WOMPACT'01), Purdue University, West Lafayette, Indiana. July 30-31 (2001)
4. Chapman, B., Weng, T.-H., Hernandez, O., Liu, Z., Huang, L., Wen, Y., and Adhianto, L.: Cougar: An Interactive Tool for Cluster Computing. 6th World Multiconference on Systemics, Cybernetics and Informatics. Orlando, Florida, July 14-18, (2002)
5. The Dragon analysis tool. http://www.cs.uh.edu/~dragon
6. Eggers, S.J., Emer, J.S., Lo, J.L., Stamm, R.L., and Tullsen, D.M.: Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, Vol. 17, No. 5, (1997) 12-19
7. Frumkin, M., and Yan, J.: Automatic Data Distribution for CFD Applications on Structured Grids. The 3rd Annual HPF User Group Meeting, Redondo Beach, CA, August 1-2, 1999. Full version: NAS Technical report NAS-99-012, (1999)
8. Hall, M.W., Hiranandani, S., Kennedy, K., and Tseng, C.-W.: Interprocedural Compilation of FORTRAN D for MIMD Distributed-Memory Machines. Proceedings of Supercomputing 92', Nov. (1992) 522-534.
9. Marowka, A., Liu, Z., and Chapman, B.: OpenMP-Oriented Applications for Distributed Shared Memory. In the Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems. November 4-6, 2002, Cambridge, (2002)
10. Muller, M.: OpenMP Optimization Techniques: Comparison of FORTRAN and C Compilers. Third European Workshop on OpenMP (EWOMP 2001), (2001)
11. Nieplocha, J., Harrison, R.J., and Littlefield, R.J.: Global Arrays: A portable `shared-memory' programming model for distributed memory computers. Proceedings of Supercomputing, (1994) 340-349
12. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., and Ayguade, E.: Is data distribution necessary in OpenMP. Proceedings of Supercomputing, Dallas, TX, (2000)
13. Nicolopoulos, D.S., Ayguadé, E.: Scaling Irregular Parallel Codes with Minimal Programming Effort. Proceedings of Supercomputing 2001 (SC'01), the International Conference for High Performance Computing and Communications, Denver, Colorado, November 10-16, (2001)

14. The Open64 compiler. http://open64.sourceforge.net/
15. Sato, M., Harada, H., Hasegawa A., and Ishikawa Y.: Cluster-Enabled OpenMP: An OpenMP Compiler for SCASH Software Distributed Share Memory System. Scientific Programming Vol. 9, No. 2-3, Special Issue: OpenMP, (2001): 123-130
16. Satoh, S., Ksano K., and Sato, M.: Compiler Optimization Techniques for OpenMP Programs. Scientific Programming Vol. 9, No. 2-3, Special Issue: OpenMP, (2001) 131-142
17. Silicon Graphics Inc. MIPSpro 7 FORTRAN 90 Commands and Directives Reference Manual, Chapter 5: Parallel Processing on Origin Series Systems. Documentation number 007-3696-003. http://techpubs.sgi.com/
18. Tseng, C.-W.: An Optimizing FORTRAN D Compiler for MIMD Distributed-Memory Machines. PhD thesis, Dept. of Computer Science, Rice University, January (1993)
19. Tu, P., and Padua, D.: Automatic Array Privatization. Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science., Vol. 768, August 12-14, (1993) 500–521
20. Wallcraft, A.J.: SPMD OpenMP vs. MPI for Ocean Models. Proceedings of First European Workshops on OpenMP (EWOMP'99), Lund, Sweden, (1999)
21. Zima, H., and Chapman, B.: Compiling for Distributed Memory Systems, Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines, Vol. 81, No. 2, Feb. (1993) 264-287

OpenMP Application Tuning Using Hardware Performance Counters

Nils Smeds

Parallelldatorcentrum (PDC)
Royal Institute of Technology, SE100 44 Stockholm, Sweden
smeds@pdc.kth.se
http://www.pdc.kth.se/

Abstract. Hardware counter events on some popular architectures were investigated with the purpose of detecting bottle-necks of particular interest to shared memory programming, such as OpenMP. A fully portable test suite was written in OpenMP, accessing the hardware performance counters be means of PAPI. Relevant events for the intended purpose were shown to exist on the investigated platforms. Further, these events could in most cases be accessed directly through their platform independent, PAPI pre-defined, names. In some cases suggestions for improvement in the pre-defined mapping were made based on the experiments.

1 Introduction

Today's dominating computing platform for scientific analysis is the super-scalar CPU architecture. This architecture is characterized by multiple units in the CPU capable of executing concurrent tasks within a serial workflow. To fully utilize the computing capacity, all available units should be scheduled with useful work. In particular, if a unit gets stalled because of an event such as a read from system memory, there is a high risk that other units experience stalls due to the complex dependencies between the concurrent tasks in the work flow of the CPU.

One of the most critical components in minimizing the amount of CPU stall is the hierarchical memory system. Ideally, this cache system is able to replace loads and stores to system memory with loads and stores to a much faster CPU cache. In reality, only some of the memory accesses can be fulfilled by the cache hierarchy. The overall efficiency of the application is usually closely related to the degree by which the caches can be used to satisfy memory accesses.

The cache coherency protocol present in close to all HPC systems today assures that each local cache copy is either up to date or marked invalid. A modification of a memory location in a cache line copy by one CPU will invalidate the copies of the same cache line in the other caches. This coherency protocol becomes an important issue in shared memory programming, such as OpenMP. The availability of data in the local cache is then not only dependent on the memory access pattern of the executing thread. It will also dependent on invalidations caused by other threads.

M.J. Voss (Ed.): WOMPAT 2003, LNCS 2716, pp. 260–270, 2003.

The time to access system memory is typically in the order of 10 to 100 times longer than to access local cache for the type of systems studied here. A cache-to-cache refill due to an earlier cache line invalidation is comparative in performance to a system memory access. That is, the possible performance penalty of a cache miss due to an invalidation is of the same magnitude as the number of CPUs available, if not larger. The potential speed-up by a shared memory parallelization is thus at risk if the memory access pattern in the algorithm introduces cache misses by cache invalidations. The effect usually becomes more pronounced as more CPUs are used, resulting in poor, if indeed any, speed-up.

By inspection of the code and detailed knowledge of the algorithms used, experienced programmers may often be able to determine if a section of code will experience poor scalability. However, this process of examining code is slow and the complexity of modern computer systems makes it hard to predict to what degree a particular stretch of code will affect the over-all performance. For these reasons, it is desirable to use tools to measure the actual behavior of the program on the computer platform, directing the programmers' efforts to the most critical regions of the code.

The CPUs of most modern computer architectures have a set of programmable counters available that can be used to count the occurrence of pre-defined states or state changes. Examples of such events are CPU clock cycles passed and the number of floating point operations performed. The application programmers interface (API) to these counters have often been proprietary and not publicly available. The rising need to measure detailed application performance have forced manufacturers to publish their calling interfaces and provide proper operating system support in order to access the counters from user level code.

However, even with a documented API it is still a demanding task for the developer to take benefit from the performance counters. The API and the events available tend to differ between CPU models and are usually drastically different between CPU and operating system vendors. This problem of in-portability is eliminated by the *Performance counters API (PAPI)*[2]. A computer platform with PAPI support gives the programmer access to a standardized, efficient and well documented way to access the counters in the CPU.

With PAPI, the performance tool developer need only to implement code for one unified counter API. With full access to the hardware performance counters, performance analysis tools can develop in the direction where they assist the programmer in diagnosing why a particular section of code is a potential bottle-neck instead of just locating code sections that consume the most CPU time. This paper demonstrates techniques that can be used in performance tools to discover and diagnose hot-spots due to mutual cache pollution by different threads. The methods for diagnosing OpenMP bottle-necks due to cache invalidations presented here investigate the applicability of PAPI and its set of pre-defined events for this purpose.

2 The Experimental Setup

2.1 Test Cases Used

The most important potential bottle-neck in shared memory parallel programs such as OpenMP programs, is here considered to be *invalidations* of shared cache lines. For this reason the test cases were designed to verify if the available events were able to discriminate between cache line sharing and cache line invalidation. The former case is not considered to be a performance problem while the latter is to be avoided by application programs if possible.

The cases used in this study are pictured in Fig. 1. Each test case consist of an initialization phase and two measurement sections. In the initialization phase the caches are put into a known state and the level 2 cache of the CPU executing thread 0 gets filled with valid copies of the data array M. Any data of the array present in the cache of the CPU executing thread 1 will at this point become invalid.

In the first measurement section, thread 1 (TID 1) accesses the data of the array in a strided fashion. The stride is a variable parameter of the experiment. In a MESI cache coherency protocol, these accesses result in either an Invalid (Test 2) or a Shared (Test 3) state of the cache line in the cache of the CPU that just executed thread 0. The cache line in the CPU executing thread 1 will correspondingly be put in state Exclusive or Shared. For other coherency protocols the general behavior is similar although the details are modified according to the protocol used.

In the second measurement section, thread 0 traverses through the array and increments each item. In test 2 this will induce a cache-to-cache fill from the CPU that executed thread 1, followed by an invalidation of the cache line when the updated value is written by thread 0. In test 3 there is no cache-to-cache fill, but only the invalidation caused by the write to the shared cache line by thread 0.

Each of the two test cases comes in two versions. In the base version the counters are started and stopped separately in each thread so that counts are only registered during the section where the thread is "active". In the second version, B, the counters are started and stopped simultaneously in both threads and the counting of events covers both measurement sections. The extent of the measured region for each thread in the two versions is illustrated with brackets in Fig. 1. The two versions of each experiment makes it possible to deduce in which of the measured sections the counts are being detected. The count registered in the B-tests should always be larger than in the base test as the B-test includes both the "active" phase measured in the base test, but also the counts registered in the "passive" part where the thread is waiting in a barrier for the completion of the "active" section of the other thread.

The array used in the experiments is an array of 32768 double precision elements (8 bytes) of a total size of 256 KiBytes (1 KiByte=1024 bytes). The size was chosen to be significantly larger than the level 1 data cache on all tested platforms, while still only occupying at most half of the available level 2 cache. The results thus pertain to the level 2 cache behavior of the systems. For each

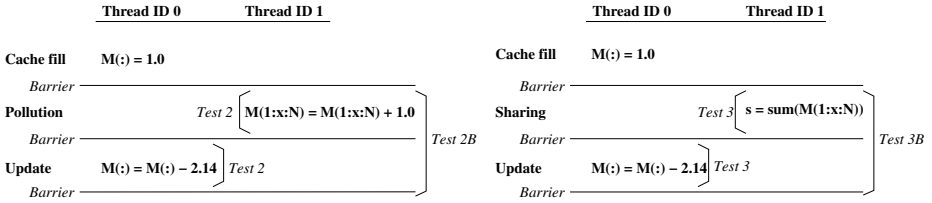


Fig. 1. Pseudo-code describing the two test cases used. To the left the write-invalidate/update-shared/write-shared case (test 2) and to the right the write-invalidate/share/write-shared case (test 3). The extent of the measurement for the two versions of each case is marked with brackets. The variable stride is indicated by the variable x

stride the test was repeated 16 times. The average number of counts registered together with the minimum and maximum counts are reported here. Further, each set of experiments was repeated at different times of day and on repeated days to ensure that the results obtained were representative and not biased by a particular system load. No experiments were conducted under high system load.

2.2 Computer Architectures and PAPI Implementations

Four different shared memory parallel computer systems were used in this investigation.

1. **Intel PentiumII**, a dual CPU shared memory bus system implementing a MESI cache coherency protocol.
2. **IBM Power3**, Nighthawk nodes with a switched memory bus and a MESI protocol.
3. **SGI R10K**, ccNUMA system with IP27 MIPS CPUs, IRIX 6.5 and SGI IRIX compilers. This system implements a directory based modified DASH coherency protocol.
4. **SGI R14K**, ccNUMA system of a similar design as the previous system, but with IP35 MIPS CPUs.

The systems are described in more detail in Tab. 1. Two different versions of PAPI was used. The CVS repository development head of August 2002 and a pre-release of version 2.3.4 available in April 2003.

3 Experimental Findings

3.1 Intel PentiumII

The Intel PentiumII system and later models implement a snoop based MESI cache coherency protocol. The protocol is implemented by two separate bus signals, HIT# and HITM#, used by the CPUs to signal that they have detected

Table 1. Systems used. Type of CPUs and number of CPUs in a node, clock speeds and cache characteristics including level 1 data cache sizes, level 2 unified cache sizes and line length. 1 Ki is $2^{10} = 1024$

.

CPU type	Clock MHz	CPUs	L1D KiByte	L2 KiByte	Line length Byte
Intel PII	350	2	16	512	32
IBM pwr3	222	8	64	4096	128
SGI R10K	195	12	32	4096	128
SGI R14K	500	128	32	8192	128

a load of a memory address that the CPU has a copy of in its local cache [4]. Further, a write to a shared cache line needs to be announced to the other CPUs on the bus to invalidate their copies of the cache line. In Fig. 2, the frequency of the event BUS_TRAN_INVAL available on this platform is shown. Only results from test 2 and test 3 are reproduced. Test 2B and 3B show identical behavior to the base version, which means that the events occur in the "active" section of the tests for both threads.

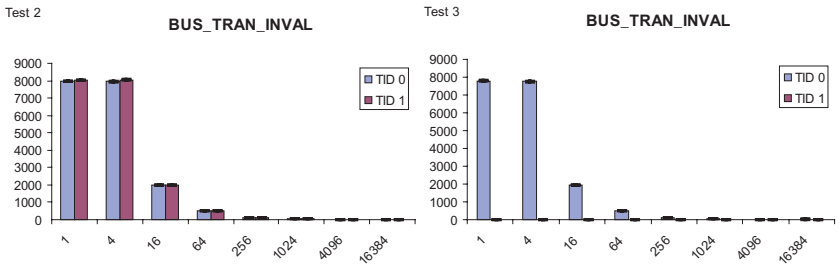


Fig. 2. Number of invalidation requests registered in an Intel PII dual CPU system. The array access in thread 1 (TID 1) is varied from linear access (stride 1) to stride 16384. The bars represent the average of 16 repetitions, the max and min count detected are indicated with error bars (barely noticeable here). The data cache line length is 32 bytes (i.e. 4 elements) which is clearly reflected in the results

From Fig. 2 it is clear that the event BUS_TRAN_INVAL, available to the PAPI user as PAPI_CA_INV, can be used to detect the invalidation of the shared cache lines. The complete data array occupies 8192 cache lines, which is approximately the number of counts registered for strides up to the cache line length. Important here is the difference in the counts registered by thread 1 in the two test cases reflecting the different invalidation patterns of the tests. The variation

in the results is very low as indicated by the barely noticeable error bars. As a contrast, the results for some other events are shown in Fig. 3 to illustrate the situation where the events registered are not correlated to the memory access pattern of the specific test.

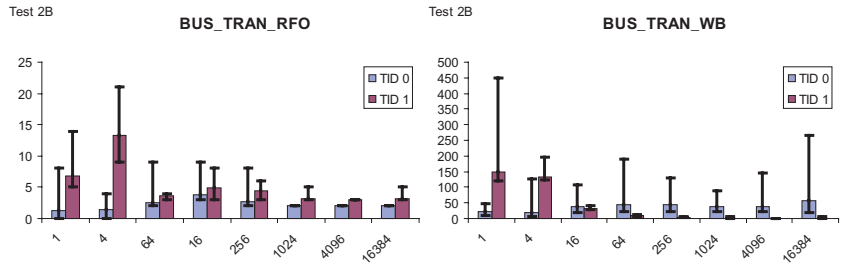


Fig. 3. Example of read-for-ownership (RFO) and cache write-back (WB) events registered on an Intel PII dual CPU system for test 2B. The registered counts appear uncorrelated to the access pattern of the test

Two events that ought to have been relevant to the test kernels used here were the BUS_HIT_DRV and BUS_HITM_DRV events. These events count the number of cycles the CPU is driving the corresponding bus line. The PII/PIII CPU also offers a feature called edge detection [5], which could have been used with these events to estimate the number of cache loads that resulted in a shared state and cache-to-cache fills. Unfortunately, experiments indicated that edge detection did not work in conjunction with these particular events. Further investigation is needed to determine why these events could not be applied to the test cases.

3.2 IBM Power3

A slightly different behavior from the Intel PII results was found on the IBM Power3 platform. This architecture offers a large number of events related to the cache coherency protocol, which is a bus-based snoopy MESI state protocol [3]. The two events shown in Figs. 4-5 are cache line transitions E‖S → I and M → E‖S. The graphs clearly illustrate that – for this platform and these metrics – it is the "passive" thread waiting at the barrier that registers the counts.

A noteworthy characteristic of the results is that the variation in the number of registered events is slightly larger than for the Intel experiments above. This could be due to the more complex nature of the system. There is a larger number of CPUs available and the memory bus is more complex than the bus architecture of the PII/PIII Intel system. The tests were performed using 1:1 thread scheduling (*system scope*) and spin waits (*XLSMPOPTS="spins=0:yields=0"*).

The expected number of counts for strides in the range 1–16 is 2048 due to the cache line length on this platform. Figure 4 shows that the present version (April 2003) of PAPI is able to produce values close to this. This is an improvement

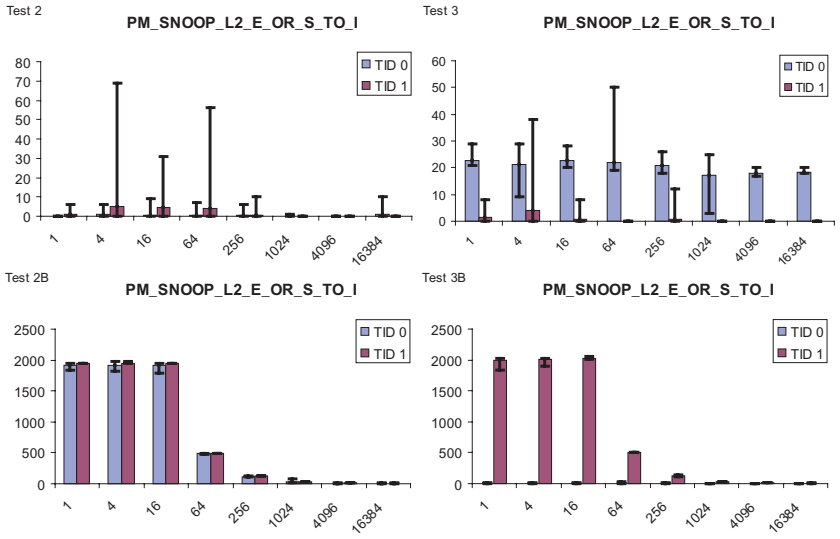


Fig. 4. Registered events as a function of access stride on an 8-CPU IBM Power3 system. The level 2 cache line length corresponds to 16 elements (128 bytes) in the array, which is reflected in the counts for the test cases 2B and 3B. Noticeable is that it is the thread experiencing invalidations (the passive thread) rather than the thread causing the invalidations that can be used for detecting the condition

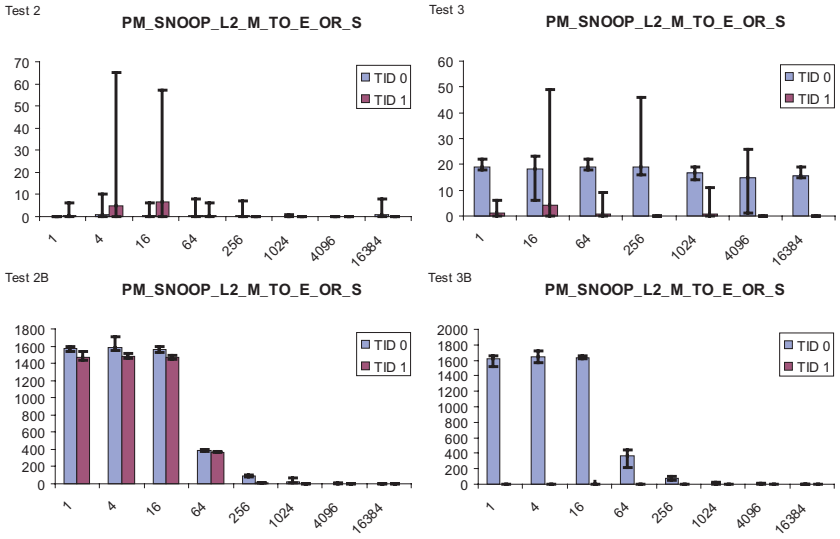


Fig. 5. Intervention events detected on IBM power3 CPUs. The counts refer to transitions where a modified cache line goes into a shared state

from earlier experiences (August 2002) where an undercount of a factor $\frac{1}{4}$ to $\frac{1}{3}$ was present and the variance in the results was larger. In the case of Fig. 5 there is an apparent undercount of events. However, this is compensated by events registered in the event PM_SNOOP_L1_M_TO_E_OR_S not included in the figure.

Some other events available on this hardware platform are shown in Fig. 6. The PM_SNOOP and PM_SNOOP_L2HIT events shown are a factor 2 larger than the results in Fig. 4. This reflects that in test 2B, the cache lines accessed by thread 0 is first shared and invalidated by thread 1, then shared again between thread 0 and 1 and finally invalidated as thread 0 updates its value. The corresponding counts for PM_SNOOP and PM_SNOOP_L2HIT for test 3B were found to be half the number of counts as for test 2B. Each passive thread thus registered counts for both of the transitions Modified to Shared and Shared to Invalid. No counts were registered in the active section for any thread in either test.

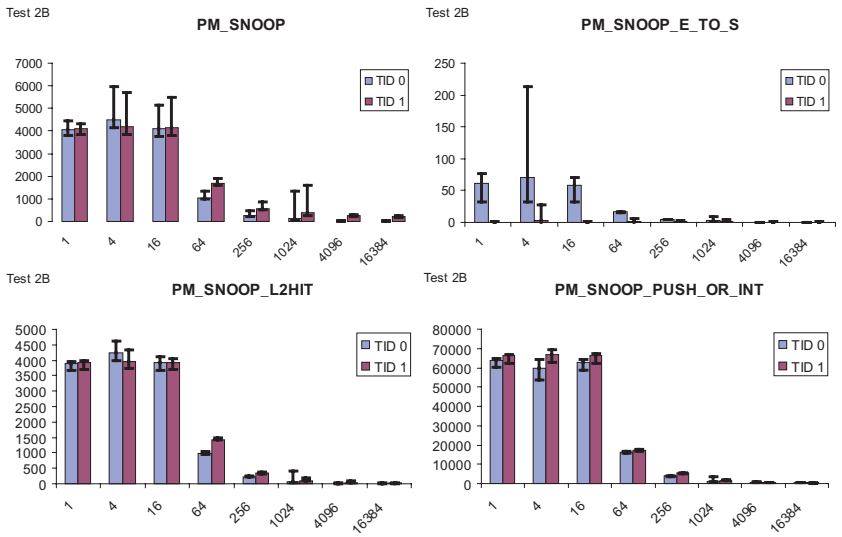


Fig. 6. Some other available events on the IBM power3 architecture shown for the invalidating test case including passive counts (Test 2B). The corresponding sharing benchmark showed similar behavior, but with approximately half the number of counts for events PM_SNOOP and PM_SNOOP_L2_HIT for both threads. Also, for TID 1 the counts for PM_SNOOP_PUSH_INT were negligible indicating that thread 1 do not need to push modified cache lines to thread 0

Further, in Fig. 6 it can be noticed that there were no transitions from Exclusive to Shared detected. This could be due to the type of memory access exercised by the particular tests used in this study. This behavior could also occur if the

event used refers to level 1 cache activity only, as the tests are constructed to investigate level 2 cache behavior.

The last event illustrated in Fig. 6 is the PM_SNOOP_PUSH_INT event. This event is in this test most likely registering cache-to-cache fills. This assumption is supported by the behavior of the same event for test 3B, although not shown in the figure. In test 3B only thread 0 is showing counts of the same magnitude as both threads do in test 2B. A particularity of this event is the high number of counts which could indicate that it is not the number of occurrences that is counted, but instead the number of CPU or bus cycles this state is active. Still, the overall characteristics of the counts indicate that the registered number of counts is closely related to the work load caused by the test kernel.

3.3 SGI R10K/R14K

The are four counting events available on the SGI platform that are directly related to the cache coherency protocol [6]. Similar to the IBM AIX platform, a significant improvement has been made in the ability of PAPI to accurately measure the cache events in this experiment. In August 2002, only the general trend of the expected behavior could be recovered from the counts registered. Repeated measurements in April 2003 show the excellent agreement to theory and low variation depicted in Fig. 7. This improvement applies to the R14K platform as well as the R10K. As in the case of IBM Power3 the counts are registered in the "passive" section of the tests and for this reason only the B versions of the test cases are included in this paper.

4 Discussion

The experiments conducted show that bottle-necks typical of shared memory parallel programs may be detected using hardware performance counters. The different hardware platforms provide events that can be used for this purpose and PAPI provides a means to access these events in a portable and platform independent way.

In the present investigation the counters were programmed using the native event support of PAPI. Native events removes the restriction on the programmer to use only the pre-defined events in the PAPI event map. Instead, the full set of events available on the platform can be accessed. In this study, native events allowed for experimentation with events outside the PAPI map and also to vary the accompanying bit masks of the events.

When writing performance analysis tools, using native events may be cumbersome. The tools need to provide users with an easy interface to the events, and the tools need to be maintainable as new processor versions, with modified event sets, reach the market. For this reason it is important that the pre-defined event names in the PAPI event map are relevant. The current (April 2003) map for the cache coherency related events is shown in Tab. 2 for the platforms in this study. The methodology in this paper provides a framework to investigate

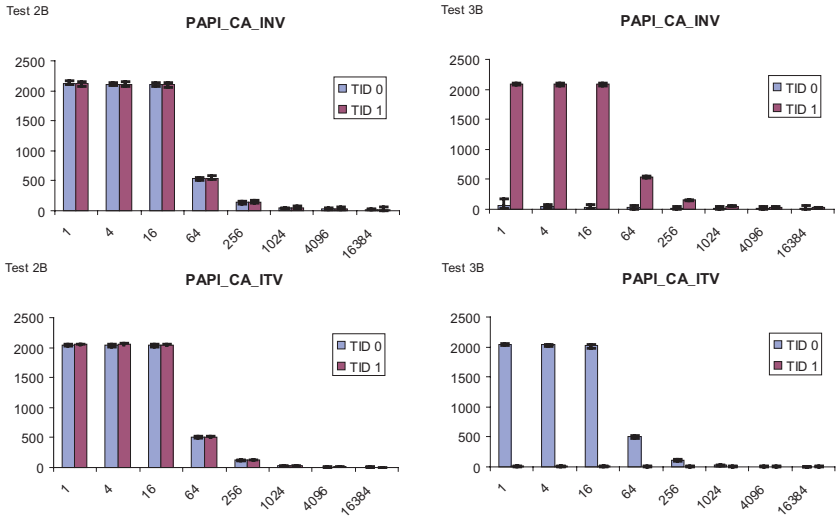


Fig. 7. Cache coherency events measured on a SGI R10K system with 12 CPUs. The results for the R14K system were close to identical. The cache line length corresponds to 16 array elements (128 Bytes)

how the events in the mapping correlate to events needed for application performance optimization. In the light of the experiments performed here it is possible to comment on the current mapping.

Table 2. Current (April 2003) mapping of cache coherency protocol related events in PAPI. For the Intel events the bit mask used with the event is indicated in parenthesis

PAPI event	Intel PII/PIII	IBM Power3	SGI R10K/R14K
PAPI_CA_SNP	—	PM_SNOOP	—
PAPI_CA_SHR	L2_RQSTS (meSi)	PM_SNOOP_E_TO_S	Store/Fetch excl. to shared block (L2)
PAPI_CA_CLN	BUS_TRAN_RFO (self)	—	Store/Fetch excl. to clean block (L2)
PAPI_CA_INV	BUS_TRAN_INVAL (self)	PM_SNOOP_PUSH_INT	Ext. invalidations
PAPI_CA_ITV	BUS_HITM_DRV (self, edge)	—	Ext. interventions

In the case of the Intel PII/PIII mapping it is clear from Fig. 2 that PAPI_CA_INV is mapped to a suitable CPU event. The PAPI_CA_ITV event needs some further investigation as there appeared to be some issues with the

edge detection mechanism either in PAPI or in the CPU architecture. The relevance of the mappings of the remaining events in Tab. 2 could not be examined by the test cases in this study.

The rich number of events available on the IBM Power3 platform calls for a rigorous investigation to find a suitable subset to use in the PAPI event name mapping. In the light of the current results PAPI_CA_ITV could perhaps be better mapped to the sum of PM_SNOOP_L2_M_TO_E_OR_S and PM_SNOOP_L1_M_TO_E_OR_S. Likewise, PAPI_CA_INV could perhaps better be mapped to PM_SNOOP_L2_E_OR_S_TO_I. The remaining events in the current map need further investigation to assure that they detect relevant events in all cache levels. Also, the methodology used here for the Power3 CPU needs to be repeated in a similar study for the Power4 CPU with its more complex memory system [1].

The PAPI event map on the SGI platform does not call for modifications. The tests made in this work demonstrate that the available events can be effectively used for the intended purpose as regards the events PAPI_CA_ITV and PAPI_CA_INV. The other two events in the map on this platform were not investigated here. The improved accuracy of the results measured by PAPI on this platform and on the IBM Power3 is believed to be due to a revision of the thread handling in PAPI.

5 Conclusion

The present work demonstrates that it is possible to use PAPI to instrument parallel applications written in OpenMP. Further, as OpenMP and PAPI are available on most current computer systems this provides a portable implementation that can be used on different hardware. Still, more work is called for in the mapping of available events to pre-defined PAPI names to match the measurement needs of application developers writing shared memory parallel programs.

References

1. Andersson, S., Bell, R., Hague, J., Holthoff, H., Mayes, P., Nakano, J., Shieh, D., Tuccillo, J.: POWER3 Introduction and Tuning Guide. IBM RedBook, `http://www.redbooks.ibm.com` (1998)
2. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. Int. J. High Perf. Comput. Appl., **14**(3) (2000) 189–204
3. Papermaster, M., Dinkjian, R., Mayfield, M., Lenk, P., Ciarfella, B., Connell, F.O., DuPont, R.: POWER3: Next Generation 64-bit PowerPC Processor Design. (1998)
4. Intel® Architecture Optimization Reference Manual. Intel Corporation. (1999)
5. Intel Architecture Software Developer's Manual Volume 3: System Programming. Intel Corporation. (1999)
6. MIPS R10000 Microprocessor User's Manual. MIPS Technologies (1996)

Author Index