

**PURDUE UNIVERSITY**  
**GRADUATE SCHOOL**  
**Thesis Acceptance**

This is to certify that the thesis prepared

By Ayon Basumallik

Entitled

Compiling Shared-Memory Applications for Distributed-Memory Systems

Complies with University regulations and meets the standards of the Graduate School for originality and quality

For the degree of Doctor of Philosophy

Final examining committee members

R. Eigenmann

, Chair

S. P. Midkiff

T. N. Vijaykumar

A. Y. Grama

Approved by Major Professor(s): R. Eigenmann

Approved by Head of Graduate Program: M. R. Melloch

Date of Graduate Program Head's Approval: 9/25/07



COMPILING SHARED-MEMORY APPLICATIONS FOR  
DISTRIBUTED-MEMORY SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ayon Basumallik

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2007

Purdue University

West Lafayette, Indiana

UMI Number: 3307404



---

UMI Microform 3307404

Copyright 2008 by ProQuest Information and Learning Company.  
All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.

---

ProQuest Information and Learning Company  
300 North Zeeb Road  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

Dedicated to my parents and grandparents.

## ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Rudolf Eigenmann, for his assistance and direction during my doctoral studies at Purdue. I appreciate his thoroughness and the emphasis he places on quantitative evaluation of research ideas. I would also like to thank my committee members, Prof. T.N. Vijaykumar, Prof. S. Midkiff and Prof. A. Grama for the feedback they have provided throughout my doctoral studies. I am indebted to Prof. Midkiff for the patient hearings he has given me whenever I barged into his office. I enjoyed all the spirited discussions that I have had with Prof. Vijaykumar and I appreciate the zest with which he approaches research. I am also grateful to other faculty members at Purdue who have helped me through their courses, seminars and direct involvement in the graduate program.

I am thankful to the members of the ParaMount Research Group at Purdue ECE, especially Seung-Jai Min who has collaborated with me over a substantial part of my research and Troy Johnson and Sang-Ik Lee who helped create the Cetus compiler infrastructure that I have used to implement my ideas.

I would like to thank Prof. S. Sengupta, Prof. B.N. Chatterjee and Prof. P.P. Chakrabarti of I.I.T Kharagpur for introducing me to the world of parallel computing. I am also grateful to Jay Hoefflinger and others at the Intel PAT Lab in Champaign, Illinois, for giving me an opportunity to explore the impact of my research on a production system with real-world applications.

A Ph.D. student, owing to the very nature of research and doctoral studies, has his fair share of difficult and uncertain times. I am thankful to my family for sustaining me through those phases. Most of all, I am grateful for the friends I have found at Purdue. Because of them, I have enough happy and fond memories of Purdue to last a lifetime.

## TABLE OF CONTENTS

	Page
LIST OF FIGURES . . . . .	vii
ABSTRACT . . . . .	x
1 INTRODUCTION . . . . .	1
1.1 Motivation . . . . .	1
1.2 A Brief Overview of OpenMP Shared-Memory Programming . . . . .	6
1.3 Contributions of This Dissertation . . . . .	7
1.4 Thesis Organization . . . . .	8
2 RELATED WORK . . . . .	9
2.1 Shared-Memory Abstractions on Distributed Memory Systems . . . . .	10
2.2 Languages for Shared-Memory Programming on Distributed-Memory Systems . . . . .	10
3 COMPILING OPENMP FOR SOFTWARE DISTRIBUTED SHARED MEM- ORY SYSTEMS . . . . .	13
3.1 Introduction . . . . .	13
3.2 Automatic Translation of OpenMP Applications into Software Dis- tributed Shared Memory Applications . . . . .	14
3.3 Baseline Performance Evaluation . . . . .	16
3.4 Advanced Optimizations . . . . .	18
3.4.1 Computation Repartitioning . . . . .	20
3.4.2 Page Aware Optimizations . . . . .	21
3.4.3 Privatization Optimization . . . . .	23
3.5 Performance of Optimized Translation of OpenMP to Software DSM .	24
4 TRANSLATION OF OPENMP APPLICATIONS TO MESSAGE-PASSING APPLICATIONS . . . . .	27
4.1 Motivation . . . . .	27

	Page
4.2 Baseline Translation of OpenMP Applications to MPI . . . . .	30
4.2.1 Translation of OpenMP Directives . . . . .	31
4.2.2 Incorporation of OpenMP Memory Consistency into the Dataflow Analysis . . . . .	34
4.2.3 Computation of Message Sets . . . . .	45
4.2.4 Placement of Messages in the Translated Program . . . . .	49
4.3 Optimizations . . . . .	52
4.3.1 Computation Alignment and Repartitioning . . . . .	53
4.3.2 Recognition of Reduction Idioms . . . . .	57
4.3.3 Optimization of Sends and Receives for Shared Data . . . . .	57
4.4 Performance Evaluation . . . . .	58
4.4.1 Brief Description of the Benchmarks . . . . .	59
4.4.2 Comparison with Hand-Tuned MPI . . . . .	60
4.4.3 Comparison with HPF . . . . .	66
4.4.4 Comparison with OpenMP deployed using Software Distributed Shared Memory . . . . .	69
5 OPTIMIZING IRREGULAR SHARED-MEMORY APPLICATIONS FOR DISTRIBUTED-MEMORY SYSTEMS . . . . .	72
5.1 Introduction . . . . .	72
5.2 Need for Optimization in the Baseline OpenMP to MPI Translation Scheme for Irregular OpenMP Applications . . . . .	75
5.3 Compile-Time Analysis of Irregular Accesses . . . . .	76
5.3.1 Irregular Reads . . . . .	77
5.3.2 Irregular Writes . . . . .	77
5.4 Inspection and Loop Restructuring for Computation-Communication Overlap . . . . .	78
5.5 Ensuring Computation-Communication Overlap . . . . .	87
5.6 Performance Evaluation . . . . .	89
5.6.1 Performance of <i>Equake</i> . . . . .	92
5.6.2 Performance of <i>CG</i> . . . . .	94



	Page
5.6.3 Performance of <i>MOLDYN</i> . . . . .	97
5.7 Related Work . . . . .	99
5.8 Conclusion . . . . .	102
6 EPILOGUE . . . . .	103
6.1 Conclusions . . . . .	103
6.2 Future Work . . . . .	105
6.2.1 Shared-Memory Programming for Hybrid Platforms . . . . .	105
6.2.2 Runtime and Interactive Optimizations . . . . .	105
LIST OF REFERENCES . . . . .	107
VITA . . . . .	115

## LIST OF FIGURES

Figure	Page
1.1 Serial version. . . . .	2
1.2 OpenMP version. . . . .	3
1.3 Basic Message-Passing version. . . . .	4
1.4 Optimized Message-Passing version. . . . .	5
3.1 Inter-Procedural Analysis to find the Shared Variables in a Program . . .	16
3.2 Execution Time and Speedup of the <i>PI</i> kernel on 1,2,4 and 8 processors.	17
3.3 Overheads of OpenMP Synchronization constructs, as measured by the OpenMP Synchronization Microbenchmark. The overheads have been measured on a system of 1,2,4 and 8 processors. . . . .	18
3.4 Baseline performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines. . . . .	19
3.5 Computation Repartitioning: subroutine RHS from APPLU . . . . .	20
3.6 Page Aware Optimization: subroutine CALC2 from SWIM . . . . .	22
3.7 Performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines. . . . .	24
4.1 Translation of OpenMP loop with a <i>reduction</i> clause. . . . .	33
4.2 Basic Translation scheme for converting OpenMP applications to MPI. Shared data is allocated by all processes. All processes redundantly exe- cute serial regions. At the end of a parallel work-sharing construct, each processes communicates those writes that will potentially be read by other processes. . . . .	35
4.3 Algorithm to create list of <i>Shared Variables</i> in an OpenMP Program. . .	36
4.4 Incorporation of OpenMP Memory Consistency Semantics into the Pro- gram Flow Graph: Additional dependencies are introduced by the <i>flush</i> directive. Existing dependencies are relaxed by the <i>nowait</i> clause. . . . .	38
4.5 Algorithm to adjust the Control Flow Graph to remove dependencies ac- cording to OpenMP's Memory Consistency specifications . . . . .	40

Figure	Page
4.6 Algorithm to adjust the Control Flow Graph to incorporate dependencies created by explicit <i>flushes</i> . . . . .	41
4.7 Algorithm to Compute Message Sets for Resolving Remote Accesses . . .	47
4.8 OpenMP program snippet with a pair of <i>flush</i> directives. . . . .	51
4.9 Algorithm to generate messages for explicit <i>OMP FLUSH</i> directives . . .	52
4.10 Summary of the Baseline Translation of OpenMP applications to MPI. .	53
4.11 OpenMP program snippet for Computation Alignment Example . . . . .	54
4.12 MPI version without Computation Alignment . . . . .	55
4.13 MPI version with Computation Alignment . . . . .	56
4.14 Performance of EP (CLASS B). . . . .	61
4.15 Performance of CG (CLASS B). . . . .	61
4.16 Performance of FT (CLASS A on Ethernet Cluster and CLASS B on IBM SP2 ). . . . .	62
4.17 Performance of IS (CLASS B). . . . .	62
4.18 Performance of LU (CLASS B). . . . .	62
4.19 Performance of EQUAKE (REF Dataset). . . . .	63
4.20 Performance of ART (REF Dataset). . . . .	63
4.21 Scalability comparison on the Linux Cluster. The input data set used is CLASS A for FT, CLASS B for the other NAS benchmarks and <i>train</i> for the SPECOMPM2001 benchmarks. . . . .	64
4.22 Scalability comparison on the IBM SP2 nodes. The input data set used is CLASS B for all the NAS benchmarks and <i>train</i> for the SPECOMPM2001 benchmarks. . . . .	64
4.23 Comparison of the Performance of MPI, OpenMP and HPF versions of CG (CLASS B) and LU (CLASS A) on a Linux Cluster. . . . .	67
4.24 Performance Comparison of the OpenMP versions translated to MPI with the OpenMP versions deployed on the TreadMarks DSM system. . . . .	70
5.1 Sparse Matrix-Vector Multiplication Kernel . . . . .	80
5.2 Sparse Matrix-Vector Multiplication Kernel with Loop Distribution of loop L2 . . . . .	80
5.3 Restructuring of Sparse Matrix-Vector Multiplication Loop . . . . .	80

Figure	Page
5.4 Compile-time Algorithm to create inspector and executor versions of a loop containing irregular accesses . . . . .	82
5.5 Runtime Algorithm to reorder executor loop iterations and produce computation and communication blocks . . . . .	83
5.6 Runtime Algorithm to Schedule Overlapping Computation and Communication . . . . .	84
5.7 Measurement of Computation-Communication Overlap : A scenario with three processes - $p, q$ and $r$ . All processes send data computed locally to all other processes that need these data in the next step. Measured on process $p$ , the time spent in sends/recvs is $(t_{send} + t_{recv-q} + t_{recv-r})$ . The computation available for overlap is $(t_{comp-p} + t_{comp-q})$ . The actual wait time (time spent by the computation thread on $p$ waiting for sends/receives to complete) is $(t_{wait-q} + t_{wait-r})$ . . . . .	91
5.8 Performance of <i>Equake</i> . . . . .	94
5.9 Computation-Communication Overlap in <i>Equake</i> . . . . .	95
5.10 Performance of <i>CG</i> . . . . .	96
5.11 Computation-Communication overlap in <i>CG</i> . . . . .	97
5.12 Performance of <i>MOLDYN</i> . . . . .	99
5.13 Computation-Communication overlap in <i>MOLDYN</i> . . . . .	100

## ABSTRACT

Basumallik, Ayon Ph.D., Purdue University, December, 2007. Compiling Shared-Memory Applications for Distributed-Memory Systems. Major Professor: Rudolf Eigenmann.

OpenMP has established itself as the de facto standard for parallel programming on shared-memory platforms. OpenMP provides a simpler approach to parallel programming, allowing programs to be parallelized incrementally by the insertion of directives. In contrast, parallel programming on distributed-memory systems using message-passing is effort intensive. It requires the programmer to parallelize programs as a whole and to explicitly manage data transfers.

This dissertation aims to extend the ease of parallel programming in OpenMP to distributed-memory systems as well. To that end, we propose two approaches. In the first approach, we use an underlying layer of Software Distributed Shared Memory (SDSM). However, SDSM systems have some inherent performance limitations. Therefore, this dissertation explores a second approach - direct translation of OpenMP applications to MPI. We present the basic translation scheme and optimizations for both regular and irregular OpenMP applications.

Experiments with seven OpenMP benchmarks indicate that OpenMP applications translated directly to MPI using our techniques achieve average scalability within 12% of their hand-coded MPI counterparts, a 30 % higher average scalability than corresponding SDSM applications and between 12% to 89% higher scalability than corresponding applications written in High Performance Fortran (HPF). A combined compile-time/runtime scheme, evaluated using three representative irregular OpenMP applications, achieves performance to within 10% of its MPI counterpart in one case and outperforms its hand-coded MPI counterparts in the other two cases.

# 1. INTRODUCTION

## 1.1 Motivation

OpenMP [1] has established itself as an important method and language extension for programming shared-memory parallel computers. On these platforms, OpenMP offers an easier programming model than the currently widely-used message passing paradigm. Programs written in OpenMP can usually be parallelized stepwise, starting from a sequential program. OpenMP programs often resemble their original, sequential versions, the main difference being the inserted OpenMP directives.

This approach contrasts with programs written using the Message Passing Interface (MPI) [2], for example, which generally need to be translated into parallel form as a whole, and which can look drastically different from their sequential versions. As an example, consider a program segment that performs a matrix-multiplication followed by a transpose within a loop. Figure 1.1 shows the serial version of this program segment. To create a shared-memory parallel version of this using OpenMP, the programmer simply inserts the appropriate OpenMP directives specifying the parallel loops as shown in Figure 1.2. However, in order to convert this program segment to a parallel distributed-memory version using MPI, the programmer has to be aware of where data is being produced and consumed. A simple message-passing version is shown in Figure 1.3, where the transpose operation necessitates the intercommunication of chunks of the matrix C. This may be done by first creating locally transposed submatrices and then sending and receiving them. In order to ensure scalability for the distributed-memory version, the user may need a detailed understanding of the algorithms used in the program and perform significant restructuring of the code. In this example, the programmer may (with additional effort) produce the code shown

```

for(outer=0;outer<NO OUTER;outer++){
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            for(k=0;k<N;k++){
                C[i][j] += A[outer][i][k]*B[outer][k][j] ;
            }

    for(i=0;i<N;i++)
        for(j=0;j<=i;j++){
            temp = C[i][j]; C[i][j]=C[j][i]; C[j][i] = temp ;
        }
}

```

Fig. 1.1. Serial version.

in Figure 1.4 where the communication occurs only once, instead of occurring in every iteration of the outer loop.

While OpenMP has clear advantages on shared-memory platforms, message passing is today still the most widely used programming paradigm for distributed-memory computers, such as clusters and highly parallel systems. In this dissertation, we explore the suitability of OpenMP for distributed systems as well. To that end, we examine two approaches.

Our first 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. We present compiler techniques for translating OpenMP applications into this model. We measure the resulting performance and present optimizations that significantly improve performance.

Our second approach is to translate OpenMP applications directly to message-passing. We analyze the challenges associated with such a translation. We present

```

for(outer=0;outer<NO OUTER;outer++){
#pragma omp parallel for private(i,j,k)
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            for(k=0;k<N;k++){
                C[i][j] += A[outer][i][k]*B[outer][k][j] ;
            }

#pragma omp parallel for private(temp,i,j,k)
    for(i=0;i<N;i++)
        for(j=0;j<=i;j++) {
            temp = C[i][j]; C[i][j]=C[j][i]; C[j][i] = temp;
        }
}

```

Fig. 1.2. OpenMP version.



```

for(outer=0;outer<NO OUTER;outer++){
    lower = my_proc_id*N/num_of_procs;
    upper = (my_proc_id+1)*N/num_of_procs;
    for(i=lower;i<upper;i++)
        for(j=0;j<N;j++)
            for(k=0;k<N;k++){
                C[i][j] += A[outer][i][k]*B[outer][k][j] ;
            }

    //Replace transpose computation with communication
    create_local_transpose(C);
    send_rows_to_other_procs();
    receive_columns_from_other_procs();
}

```

Fig. 1.3. Basic Message-Passing version.

```

for(outer=0;outer<NO OUTER;outer++){
    for(i=0;i<N;i++)
        for(j=0;j<N;j++)
            for(k=0;k<N;k++){
                if(outer%2) { a=i;b=j;}
                else        { a=j;b=i;}
                C[a][b] += A[outer][i][k]*B[outer][k][j] ;
            }
        //Eliminate the transpose computation here
    }
    //Transpose C if NO OUTER is odd
    if(NO OUTER % 2) {
        create_local_transpose(C);
        send_rows_to_other_procs();
        receive_columns_from_other_procs();
    }
}

```

Fig. 1.4. Optimized Message-Passing version.

compiler techniques to perform this translation and optimizations to improve performance. We present measurements of the resulting performance on a set of representative applications.

For our measurements, we used two types of distributed-memory systems. The first is a commodity cluster architecture consisting of 16 PentiumIII/Linux processors connected via standard 100 Mbps Ethernet network. The second is a cluster of 16 IBM SP2 375 MHz POWER3-II nodes connected by a high-speed switch. We expect the scaling behavior of these architectures to be representative of that of common cluster systems with modest network connectivity. We have measured basic OpenMP low-level performance attributes via the kernel benchmarks introduced in [3] as well as the application-level performance behavior of several SPEC OMP benchmarks [4, 5].

## 1.2 A Brief Overview of OpenMP Shared-Memory Programming

OpenMP provides a *fork-join* model of parallelism, where the programmer specifies parallelism in the code using *OpenMP directives*. The OpenMP directives are embedded in the code either as special comments (in FORTRAN) or as *pragmas* (in C and C++). At the core of OpenMP are *parallel regions* and *work sharing constructs*, which enable the programmer to express parallelism at the level of structured blocks within the program, such as loops and program sections. Work sharing constructs in OpenMP include the *OMP DO* loops in Fortran (correspondingly, in C/C++ there are *omp for* loops) and OpenMP *sections*. OpenMP directives for work-sharing constructs may include a scheduling clause, defining the way work will be mapped onto the parallel threads. OpenMP loops may also include a reduction clause.

OpenMP has a notion of implicit and explicit barriers. These are the points in the program where all participating threads are guaranteed to have a consistent view of the shared address space. Updates to shared memory are not guaranteed to be complete till such an implicit or explicit barrier is reached. Examples of such explicit barriers are the *omp barrier* and *omp flush* directives. Implicit barriers are present at

the end of regions demarcated using OpenMP parallel and work-sharing directives, unless a *nowait* clause is explicitly specified for them.

Currently, most OpenMP implementations use the Microtasking Model [6] which fits in well with the fork-join type of parallelism expressed by OpenMP. In this model, the *master processor* begins program execution as a sequential process. During initialization, the master creates *helper threads* on the participating processors. The helpers *sleep* until needed. When a parallel construct is encountered, the master wakes up the helpers and informs them about the parallel code to be executed and the environment to be setup for this execution. An important OpenMP property is that data is shared by default, including data with global scope and local data of subroutines called from within sequential regions. This can be implemented efficiently on today's *Symmetric Multi-Processors*, which support fully-shared address spaces and low communication latencies. The overheads introduced by the OpenMP constructs have been measured and discussed on many of the state-of-the-art Shared Memory platforms [3].

### 1.3 Contributions of This Dissertation

This dissertation makes the following contributions -

- It presents translation and optimization techniques for deploying OpenMP programs on distributed systems using Software Distributed Shared Memory. It also measures and evaluates the performance achieved by this translation on several realistic OpenMP benchmarks. In doing so, it goes beyond previous work in this area that has only evaluated the translation of kernel OpenMP programs where data is private by default.
- It presents translation and optimization techniques for converting OpenMP applications directly to message-passing applications that use MPI. A key difference between our scheme and related work (such as High Performance Fortran) is our use of *partial replication* and *redundant execution of serial regions*. As

part of these translation techniques, this dissertation makes the following contributions -

- Presents and evaluates a translation scheme for translating OpenMP programs with regular accesses using compile-time techniques.
- Presents and evaluates a scheme for handling irregular accesses in the direct translation of OpenMP to MPI, using combined compile-time and run-time mechanisms.

## 1.4 Thesis Organization

The rest of this dissertation is organized as follows. Chapter 2 will place this dissertation in the context of related work. Related work is also referred to in each chapter. Chapter 3 will present the compiler techniques for translating and optimizing OpenMP applications on Software Distributed Shared Memory systems. Chapter 4 will discuss techniques and optimizations for the direct translation of OpenMP applications to MPI applications. Chapter 5 will discuss techniques for the translation of OpenMP programs, containing irregular accesses to MPI. Chapter 6 will summarize this report and briefly discuss potential directions for future research based on this work.

## 2. RELATED WORK

Approaches for Parallel and High Performance Computing broadly fall into two categories, both in terms of architecture and programming paradigms – Shared-Memory and Distributed-Memory. The tradeoffs between Shared-Memory and Distributed-Memory programming have been examined often [7–9]. Most of this work has dealt with the criteria for selecting one paradigm as opposed to the other in designing multiprocessors and writing parallel programs. Along with the development of architectures and platforms for each of these two categories, several programming models have also evolved for both of them. For distributed-memory systems, most of these programming models have been based on a notion of message-passing. Programming models based on architectural support such as Active Messages [10] and use of messaging libraries such as PVM [11] have been used for programming distributed-memory systems. Currently, the Message Passing Interface (MPI) [12] is widely acknowledged as the de facto standard for programming distributed-memory systems. For shared-memory systems, several APIs like Pthreads [13] and Cilk [14] have evolved for multi-threaded programming. However, OpenMP [1] is currently acknowledged as the de facto standard for parallel programming on shared-memory systems.

A premise of this dissertation, that shared-memory parallel programming is usually less effort-intensive than distributed-memory programming, is supported by several past efforts that have sought to extend the ease of shared-memory programming to distributed-memory systems. Some of these efforts focussed on providing software-based support of a shared memory abstraction on distributed-memory architectures. Others have focussed on language, compiler and run-time issues to provide shared-memory style programming on distributed memory systems. In the rest of this chapter, we discuss related work in these two categories.

## 2.1 Shared-Memory Abstractions on Distributed Memory Systems

Several researchers have proposed software techniques to provide a shared-memory abstraction on distributed memory systems. This has been implemented in Software Distributed Shared Memory (SDSM) Systems such as TreadMarks [15], Midway [16] and Munin [17]. A common scheme suggested for deploying OpenMP on clusters has been to use this underlying Software SDSM system, an idea that was probably first proposed by Hu et al [18]. That effort presented a preliminary view of the translation of some simple OpenMP kernels for execution on SDSM systems. However, they dealt with OpenMP programs where data was *private* by default and they did not present or evaluate any scheme for deploying general OpenMP programs on distributed-memory systems. We believe our work is the first to present a full compiler for translating OpenMP applications on software DSM. We are also the first to evaluate the performance of a set of realistic OpenMP benchmarks deployed on SDSM.

## 2.2 Languages for Shared-Memory Programming on Distributed-Memory Systems

Some parallelizing compilers have targeted distributed-memory systems and have attempted to directly generate parallel message-passing versions of serial applications [19,20]. By contrast, our starting point is code parallelized by the programmer. Li et al [21,22] addressed the issue of compiling shared memory programs for a specific class of distributed-memory machines. They presented a general framework for matching syntactic reference patterns with appropriate aggregate communication routines. To the best of our knowledge, ours is the only effort that comprehensively addresses the direct, source-level translation of shared-memory programs(in OpenMP) to a portable message-passing form(in MPI).

An important contribution towards a higher level programming paradigm for distributed-memory machines was the development of High Performance Fortran

(HPF) [23, 24]. HPF extended Fortran with directives to specify data distribution and data alignment. HPF support was provided by commercial compiler such as those from IBM [25] and The Portland Group [26].

There are important differences between our approach and that taken by HPF. HPF allows the programmer to specify parallelism either explicitly through an *independent* directive or implicitly by specifying the data distribution. For the user, deducing an optimal data distribution can be a tedious task and Kremer [27] has suggested compiler techniques for assisting in the data distribution task. Furthermore HPF does not specify any scheme of handling irregular accesses. Suggested extensions to HPF used inspector-executor schemes [28] for handling irregular accesses. Frumkin et al [29] evaluate the implementation of the NAS Parallel Benchmarks [30] in HPF. Chamberlain et al [31] examine the implementation of the NAS MG benchmark across different parallel programming languages and evaluate the languages in terms of its portability, performance and the ability to concisely express the algorithms. Both conclude that the HPF versions perform poorly compared to the MPI versions. In terms of absolute performance measured with the NAS benchmarks [29], the HPF versions are more than twice as slow as the corresponding MPI versions. In terms speedup on 16 nodes of an SGI Origin 2000, for the six benchmarks measured, the HPF versions were on the average 9.14 times faster compared to the HPF single processor version, while the MPI versions were on the average 16.84 times faster compared to the MPI single processor version. By contrast, our starting point is an OpenMP application, where the programmer uses OpenMP directives to specify parallelism and work sharing explicitly. Our OpenMP to MPI translation scheme does not need to decide the mapping of iterations to processes. This makes our task more tractable. We only need to focus on resolving data accesses. A key difference between our approach and HPF is that we do not physically distribute data across nodes. Instead, we consider replicating shared-data on all participating nodes, which simplifies the resolution of accesses. We shall revisit these differences when we examine our translation scheme in detail in Section 4.4.3.



Split-C [32], Unified Parallel C(UPC) [33], Co-Array Fortran [34], Titanium [35] and Global Arrays [36] are other programming paradigms that have been proposed to ease programming effort by providing a global address space, that is logically partitioned between threads. The programmer specifies affinity between threads and data which, for the programmer, may be as difficult as the task of data distribution in HPF. Ghazawi and Cantonnet have compared hand optimized UPC versions (hand optimized to privatize local accesses) of the NAS benchmarks with MPI versions [37]. Their results indicate that the UPC versions achieve speedups that are 25% to 50% less than the speedups of the MPI versions. They claim that an important reason for the better scalability of the MPI versions is the optimized implementation of collective operations provided by MPI vendors. In our case, final product of our translation is MPI and we thus leverage the availability of efficient MPI libraries on multiple HPC platforms.

Recently, several other researchers have suggested the use of OpenMP on clusters. In [38] remote method invocation and remote procedure calls are used to implement OpenMP on a distributed system. Another related approach is to use explicit message passing for communication across distributed systems and OpenMP within shared-memory multiprocessor nodes [39]. This approach deals with hybrid applications whereas we deal with pure OpenMP applications.

Our work complements related efforts to implement OpenMP programs on distributed systems. Both language extensions and architecture support have been considered. Several recent papers have proposed language extensions. For example, in [40–43], the authors describe data distribution directives similar to the ones designed for High-Performance Fortran (HPF) [23]. Other researchers have proposed page placement techniques to map data to the most suitable processing nodes [40].

While most of these related efforts have considered language extensions for OpenMP on distributed systems, our goal is to quantify the efficiency at which standard OpenMP programs can be implemented using advanced compiler techniques.

### 3. COMPILING OPENMP FOR SOFTWARE DISTRIBUTED SHARED MEMORY SYSTEMS

#### 3.1 Introduction

Software Distributed Shared Memory(DSM) systems [17, 44] provide the illusion of shared-memory on a distributed memory system by using software mechanisms to trap access to remote data and perform communication. Usually, these systems support some form of *weak consistency* [45] to improve performance and may use the virtual memory functionality supported by operating systems to trap remote accesses. In our work, we have used the TreadMarks [44] SDSM system to provide a shared-memory layer on which we deploy OpenMP applications.

TreadMarks is a page-based SDSM system that implements the *Lazy Release with Multiple Writers(LRMW)* [46] protocol. The release consistency model fits the consistency requirements of the OpenMP programming model, since the implicit and explicit synchronization points in OpenMP correspond to a release-acquire point in an LRMW protocol.

In this chapter, we describe the compiler techniques for deploying OpenMP applications on TreadMarks. We then evaluate the performance of our translation on several realistic application benchmarks. The distributed-memory platform that we have used for the experiments in this chapter is a commodity cluster of eight PII nodes, running RedHat Linux 2.7, connected by 100 Mbps Ethernet.

### 3.2 Automatic Translation of OpenMP Applications into Software Distributed Shared Memory Applications

In the transition from shared-memory to distributed-memory systems, the first major change is that each participating node now has its own private address space, which is not visible to other nodes. The Software DSM layer provides the abstraction of a shared data space. So, an OpenMP application needs to be translated to a SDSM program and it can then be deployed on distributed-memory systems. We have developed a compiler infrastructure that performs the following translations.

- First, our compiler converts the OpenMP application into a micro-tasking form [6].
- Second, it converts OpenMP shared data into the form necessary for Software DSM systems.

In the *micro-tasking form*, parallel regions are present as separate functions. The program starts with a number of threads executing in parallel, but all other threads except the master thread are put to sleep. The master thread then executes the serial region. When it reaches the beginning of a parallel region, it wakes up the helper threads and provides them the function pointer for the function corresponding to that parallel region. For parallel loops, the master thread also provides the helper threads informations about which iterations they should each execute.

To convert the OpenMP application into a micro-tasking form, parallel regions and work-sharing constructs are extracted into *micro-tasking subroutines*. For OpenMP loops, static block scheduling is done to divide iterations between participating processes. At the start of a parallel region, the master process invokes these micro-tasking subroutines on all processes.

Though the SDSM system provides a shared-memory abstraction, shared data in an SDSM program has to be explicitly allocated using the API provided by the SDSM system. This presents a challenge in translating OpenMP programs to SDSM programs. OpenMP shared data includes, by default, all data in the program except

- Data that is *declared* inside an OpenMP parallel region but not explicitly shared by an OpenMP *shared* clause.
- Data that is explicitly declared to be private to each thread with an OpenMP *private* directive.

Previous work in the translation of OpenMP programs to SDSM programs [18] solved this problem by imposing the restriction that the OpenMP programs would have data *private* by default and shared data would have to be explicitly declared with the OpenMP *shared* directive. We have removed this restriction by implementing a compiler algorithm that identifies OpenMP shared data and allocating this data explicitly using the API provided by TreadMarks. This algorithm was developed in collaboration with Seung-Jai Min [47].

We implemented the inter-procedural compiler algorithm, shown in Figure 3.1 to identify OpenMP shared data. This shared data may include data residing on the process stack(subroutine local variable explicitly shared by a *omp shared* clause or shared-variables passed in as subroutine parameters). Since process stacks are not visible to all processes in TreadMarks, all declarations for shared data is hoisted to global scope. The data is explicitly allocated in shared space using the API provided by TreadMarks.

After the explicit allocation of shared data, our compiler maps OpenMP synchronization constructs to synchronization calls provided by the TreadMarks API. The three steps of (1) Creation of micro-tasking subroutines, (2) Explicit allocation of shared data and (3) translation of synchronization constructs completes the transformation of an OpenMP program to a TreadMarks program. We refer to this translation as a *baseline translation*.

In the next section, we examine the performance of this baseline translation scheme. Section 3.4 will present some optimizations to improve the performance of the translated applications beyond what is achieved by this baseline translation.

```

// UseSet(PU) : the set of variables used in the program unit PU
// CommonBlkVariableSet(PU) : the set of common block variables defined in the program unit PU
// Microtasking subroutine : When the OpenMP compiler translates the OpenMP parallel construct, it creates
// a microtasking subroutine that contains the parallel code section of the parallel construct and replaces
// the parallel construct by the call to this microtasking subroutine.

IPA_Driver
  initialize G, the set of all global, shared variables
  Foreach ProgramUnit PU
    call IPA_ProgramUnit(PU);

IPA_ProgramUnit (ProgramUnit PU)
  if ( PU is a microtasking subroutine ) then
    initialize a shared variable set, A;
    Foreach Call Statement S
      // PU_Called is the ProgramUnit called by S
      T = UseSet(PU_Called)  $\cap$  CommonBlkVariablesSet(PU_Called);
      A = A  $\cup$  T;
    find the private variable set P, defined by the OpenMP directive for PU;
    A = A - P;
    G = G  $\cup$  A;
  else
    Foreach Call Statement S to a microtasking subroutine
      Foreach parameter p of S
        if ( p belongs to the parameter list of PU ) then
          let param_loc be the position of p in PU's parameter list;
          call IPA_CallStmt(PU, param_loc);

IPA_CallStmt(ProgramUnit PU, int param_loc)
  foreach ProgramUnit Caller_PU that calls PU
    foreach Call Statement S that calls PU
      let p be the parameter at position param_loc in the parameter list of S;
      if ( p belongs to the parameter list of the Caller_PU ) then
        let param_loc be the position of p in Caller_PU's parameter list;
        call IPA_CallStmt(Caller_PU, param_loc);
      else
        G = G  $\cup$  {p};

```

Fig. 3.1. Inter-Procedural Analysis to find the Shared Variables in a Program .

### 3.3 Baseline Performance Evaluation

We first used microbenchmarks [3] to quantify the performance of specific OpenMP synchronization constructs implemented using *barrier* and *lock* functions in TreadMarks. Figure 3.3 shows the measured performance. We then measured the performance of a kernel program to estimate an upper bound for the performance of OpenMP applications on our system. The *PI* kernel computes the value of  $\pi$  using

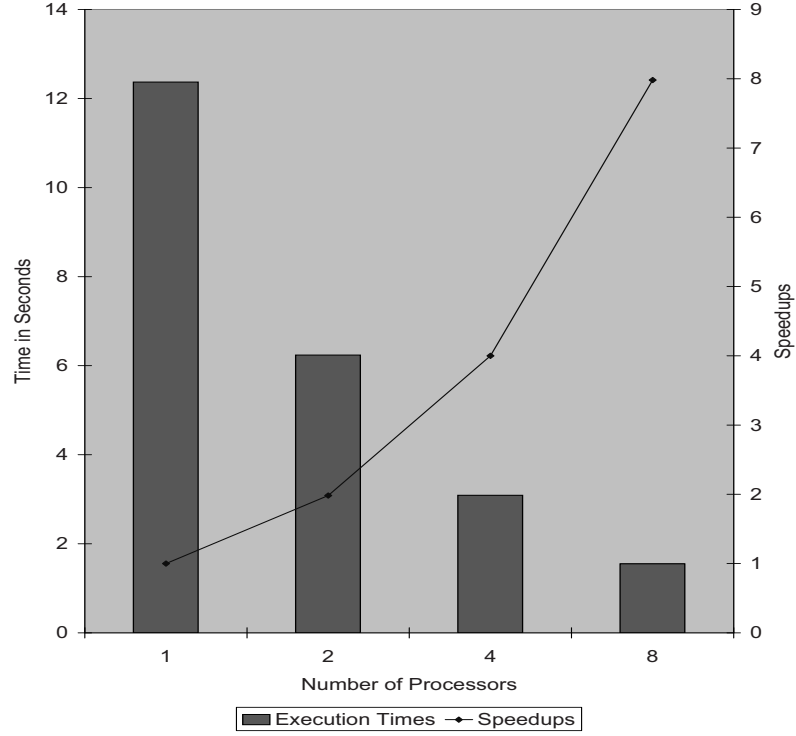


Fig. 3.2. Execution Time and Speedup of the *PI* kernel on 1,2,4 and 8 processors.

the approximation integral  $\int_0^1 4/(1+x^2)dx$ . The kernel has one OpenMP for loop with a reduction clause. Figure 3.2 shows the performance of this kernel.

Next, we measured the performance of our baseline translation scheme using more realistic applications from the SPEC OMPM2001 suite. The SPEC OMPM2001 suite of benchmarks [5] consists of realistic OpenMP C and Fortran applications. Figure 3.4 shows the performance achieved by the baseline translation of four Fortran and two C applications from this suite. The observed speedups range from a moderate 3.9 on 8 processors in case of WUPWISE to a slowdown in case of SWIM.

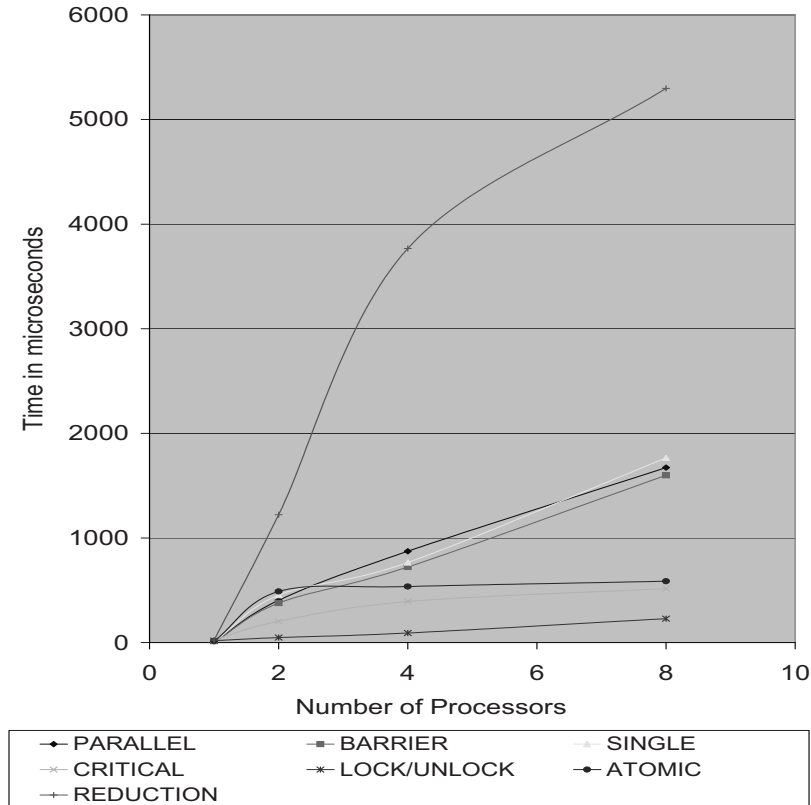


Fig. 3.3. Overheads of OpenMP Synchronization constructs, as measured by the OpenMP Synchronization Microbenchmark. The overheads have been measured on a system of 1,2,4 and 8 processors.

### 3.4 Advanced Optimizations

The performance of the baseline translation of SPEC OMPM2001 programs, described in Section 3.3, indicates that a naive translation is not sufficient to achieve desired speedups for realistic applications. We have performed detailed measurements per program region of a realistic benchmark to identify performance bottlenecks [48]. Software DSM implementations have shown acceptable performance on kernel programs [44]. 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.

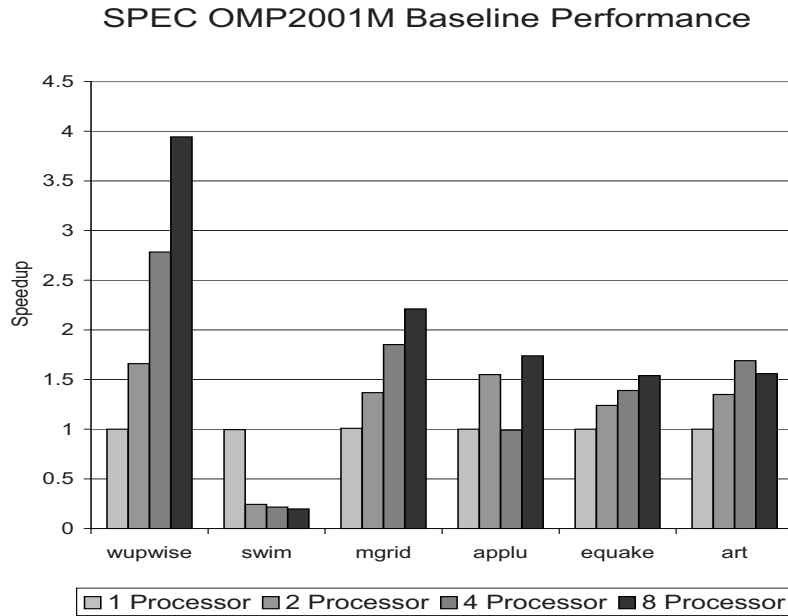


Fig. 3.4. Baseline performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines.

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 [49] applications, may have data sets that are in the order of gigabytes.

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.

### 3.4.1 Computation Repartitioning

<pre> !\$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 </pre>	<pre> !\$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 j=jst, jend     DO i=ist, iend       !\$OMP DO         DO k=2, nz-1           DO m=1, 5             rtmp(m,k)=rsd(m,i,j,k)-...             ...           ENDDO         ENDDO       !\$OMP END DO     ENDDO   ENDDO   ... !\$OMP END PARALLEL </pre>
(a) original code	(b) after selective touch optimization

Fig. 3.5. 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 3.5 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 3.5 (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 [50]. 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.

### 3.4.2 Page Aware Optimizations

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

<pre> !\$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 !\$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 !\$OMP END PARALLEL DO </pre>	<pre> !\$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 !\$OMP END PARALLEL DO  !\$OMP PARALLEL   DO j=jst, jend     DO i=ist, iend       !\$OMP DO         DO k=2, nz-1           DO m=1, 5             rtmp(m,k)=rsd(m,i,j,k)-...           ...         ENDDO       ENDDO     ENDDO   ENDDO !\$OMP END PARALLEL </pre>
(a) original code	(b) after selective touch optimization

Fig. 3.6. Page Aware Optimization: subroutine CALC2 from SWIM

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 3.6. 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.

### 3.4.3 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.

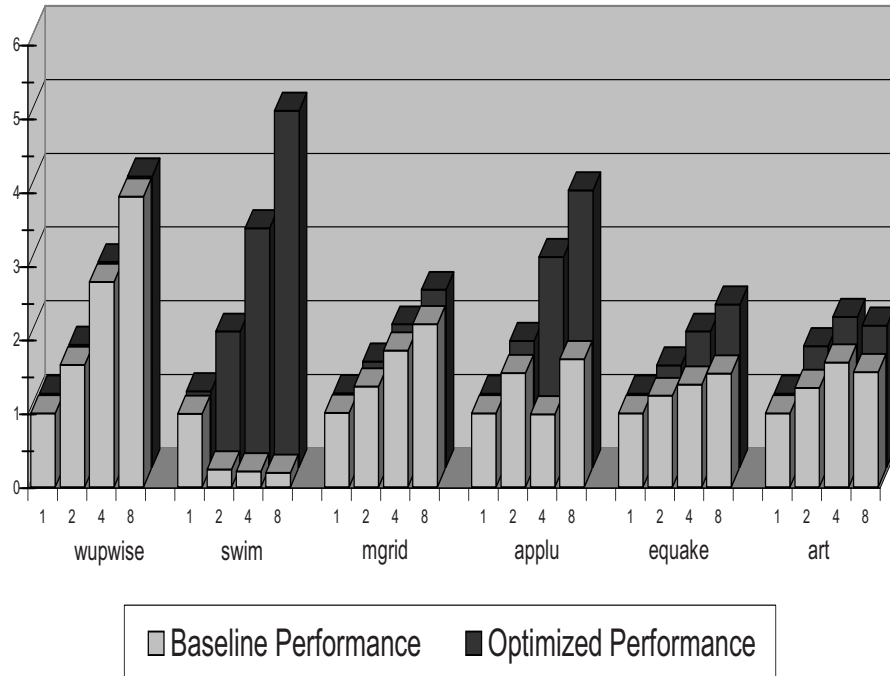


Fig. 3.7. Performance of four Fortran 77 and two C benchmarks from the SPEC OMPM2001 benchmark suite on 1, 2, 4, and 8 machines.

### 3.5 Performance of Optimized Translation of OpenMP to Software DSM

We applied the described optimizations described in Section 3.4 by hand to several of the SPEC OMPM2001 benchmarks and achieved marked performance improvements. Figure 3.7 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).

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 perfor-

mance improvements with *computation repartitioning* and *page-aware optimizations*. In SWIM, application of the *page-aware optimization* shown in figure 3.6 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.

In this chapter, we have examined a method of deploying OpenMP applications on distributed-memory systems by using an underlying layer of Software DSM. We have briefly described our baseline translation scheme. Our evaluation showed that the baseline translation scheme has some performance limitations. To improve the per-

formance of the baseline scheme, this chapter proposed and evaluated certain performance optimizations.

The level of compiler analysis required for the optimizations opens up an interesting prospect – direct translation of the OpenMP application to a message-passing form, which would eliminate some performance limitations inherent to SDSM systems. In the next chapter, we examine this alternative approach.

## 4. TRANSLATION OF OPENMP APPLICATIONS TO MESSAGE-PASSING APPLICATIONS

### 4.1 Motivation

In the previous chapter, we have evaluated the deployment of OpenMP applications on distributed-memory systems using an underlying layer of Software DSM. The Software DSM layer maintains a shared-memory abstraction and thus this deployment requires relatively low compiler complexity for a baseline translation. However, Software DSM systems suffer from some inherent performance limitations. A comparative study of Message-Passing(using PVM [11]) and TreadMarks applications [51] concluded that message-passing applications have two basic advantages over Software DSM applications. The first advantage is that message-passing applications can piggyback synchronization with sends and receives of data whereas Software DSMs incur separate overheads for synchronization and data transfer. The second advantage is that message-passing applications implicitly perform aggregation for transferring data and while Software DSMs are limited by the granularity at which they maintain coherence for shared data(for example, page based SDSMs perform shared data transfers on a per page basis). Techniques using prefetch [52] and compiler assisted analysis of future accesses for aggregation [53] have been proposed to mitigate these performance limitations.

In order to avoid the performance limitations imposed by a Software DSM system, we explore the possibility of translating OpenMP applications directly to message passing applications that use MPI. Essentially, an SDSM layer performs two functions:

- It traps accesses to *remote data* – data written on a thread or process and read by another thread or process.



- It provides a mechanism for communicating data between threads (or processes) on demand.

For the first function, we now use a combination of compile-time analysis and runtime methods to resolve remote accesses. For the second function, we use MPI libraries to communicate data. The direct use of message passing provides the compiler greater control of when and how processes intercommunicate their data and thus makes it easier to optimize this communication as well as to implement aggregation and prefetching. Additionally, robust and optimized MPI libraries are available for almost all types of parallel processing environments.

To achieve good performance, our translation scheme includes efficient management of shared data as well as advanced handling of irregular accesses. These issues were also considered by related approaches to compile shared-address-space languages onto distributed memory machines. Compiler techniques for High Performance Fortran (HPF) and those for shared-memory programming on clusters using Software Distributed Shared Memory (DSM) Systems are important representatives. A key difference that distinguishes our approach from these others is the concept of *partial replication*. In HPF-like approaches, all data have a single owner. The data distribution defined by the user defines which parts of large arrays are owned by which processor; computation partitioning schemes (e.g. owner computes) define which computations are executed by which processor. By contrast, software DSM systems, such as TreadMarks [15], replicate shared program data as well as management data structures (such as shadow copies of shared data) on all processors. Data replication simplifies the management of global data, as we will show throughout this chapter. However, it limits the data scalability of programs – that is, programs can no longer process  $n$ -times larger data sets on  $n$  processors. By partially replicating data, our approach combines the advantages of both approaches. Data management is simplified to the point where our techniques outperform HPF programs, even without the help of user directives for data distribution. At the same time, we can run the large

data sets of our benchmarks, which was not possible on software DSMs due to the aggressive replication.

In this chapter, we examine the techniques and optimizations required to translate OpenMP applications directly to MPI. We then present an evaluation of the performance of these translation techniques by comparing the performance of the translated applications with that of their hand-tuned MPI counterparts.

Specifically, this chapter makes the following contributions:

- We present compiler techniques for translating OpenMP programs to MPI.
- We present a detailed performance evaluation of the translated versions of seven representative OpenMP programs translated to MPI on two platforms. We also compare our performance with those of corresponding hand-tuned MPI and Software Distributed Shared Memory versions.

In our measurements, we used two different platforms. The first is a set of sixteen PIII-850MHz/Linux nodes, connected via commodity 100 Mbps Ethernet. The second is a cluster of 16 WinterHawk-II 375 MHz POWER3-II IBM-SP2 nodes. We expect the scaling behavior of these architectures to be representative of a broad range of common cluster systems. A goal of this dissertation is to demonstrate that realistic OpenMP applications can be automatically translated and further optimized for execution with acceptable performance levels on commodity clusters. To that end, we have selected seven OpenMP benchmarks for our experiments, two from the SPEC OMPM2001 suite [49] and five from the OpenMP versions of the NAS Parallel Benchmarks [30], which are representative of moderate-sized applications for shared-memory systems.

The rest of this chapter is organized as follows - Section 4.2 presents the basic translation scheme for translating OpenMP applications to MPI, Section 4.3 presents optimizations to improve the performance of the translated OpenMP applications and Section 4.4 presents a detailed performance analysis of the techniques presented in this chapter.

## 4.2 Baseline Translation of OpenMP Applications to MPI

The objective of the OpenMP to MPI translation scheme is perform a source-to-source translation of a shared-memory OpenMP program to an MPI program. While there are several shared-memory parallel programming paradigms, there are two specific reasons why we selected OpenMP -

- OpenMP has gained widespread acceptance in industry and academia and has become the *de facto* standard for shared-memory parallel programming.
- OpenMP programs have an analyzable structure in terms of the parts of the program that may execute concurrently. While the determination of concurrency in general parallel programs is NP-hard [54], the OpenMP specification of data and control concurrency makes OpenMP programs amenable to the types of analysis required to transform them to MPI programs.

This section presents the compiler techniques required to transform OpenMP programs to MPI programs. We refer to these as our baseline translation scheme. Section 4.3 will present some optimizations on top of this baseline scheme. The baseline translation scheme involves four steps -

1. Interpretation of OpenMP directives.
2. Incorporation of OpenMP memory consistency semantics.
3. Summarization of array accesses and computation of message sets.
4. Generation of MPI Messages.

The rest of this section discusses these four steps in detail. Subsection 4.2.1 describes how our OpenMP to MPI translator interprets OpenMP directives. Subsection 4.2.2 presents a technique for incorporating OpenMP memory consistency semantics into the program's dataflow analysis through the creation of a *Producer-Consumer Flow Graph*. Subsection 4.2.3 describes the summarization of array accesses and computation of message sets to satisfy producer-consumer relationships

for shared data in the program. Subsection 4.2.4 describes the generation and placement of MPI messages to move the computed message sets from producers to remote consumers. We have implemented these compiler transformations using the Cetus [55] compiler infrastructure.

#### 4.2.1 Translation of OpenMP Directives

OpenMP directives fall into three categories - (1) directives that specify work-sharing (*omp parallel for*, *omp parallel sections*), (2) directives that specify data properties (*omp shared*, *omp private* etc.) and (3) synchronization directives (*omp barrier*, *omp critical*, *omp flush* etc.). We now describe the transformation steps to handle each of these categories.

A common method of compiling OpenMP applications is to convert them to a microtasking form [6]. However, for translation of OpenMP applications to MPI, our compiler converts OpenMP applications to SPMD [56] form with the following characteristics -

- All participating processes redundantly execute serial regions. For work sharing constructs, such as OpenMP for loops, the work is partitioned between participating processes.
- Shared data, including shared arrays, are allocated on all processes. There is no concept of an *owner* for any shared data item. There are only producers and consumers.
- At the end of each parallel construct, each participating process communicates any shared data that it has written, to other processes that *may* use this data in the future, using MPI messages.

The redundant execution of serial regions does not degrade performance (since other processes would be idling during the execution of serial regions if serial regions

were executed on a single processor). It has the advantage of substituting communication with computation. Shared data produced in serial regions need not be communicated since redundant execution ensures that it is produced on all participating processes.

A common scalability feature of MPI applications is the ability to handle larger data sets as more processes are added to the system. It may seem that by allocating all shared data on all processes, we lose this ability. However, that is not the case. Replication applies only to *shared* data and entails that virtual memory be allocated for shared data on each node. The physical memory requirement depends on the application. An OpenMP programmer who wants to handle larger arrays than what can be allocated on a single process can do this by allocating private copies on each process, corresponding to only the parts that would be allocated on a single process in an MPI application. These private data can then be used in conjunction with shared data to affect the necessary computation. Since our starting point is OpenMP and we replicate only the shared data, our scheme can handle increasing dataset sizes just like OpenMP programs do. Such a way of handling large datasets in OpenMP in a scalable manner may seem somewhat inconvenient. So this is a cost we pay for using OpenMP on clusters. Nonetheless, the programmer still has a way of handling scalable data.

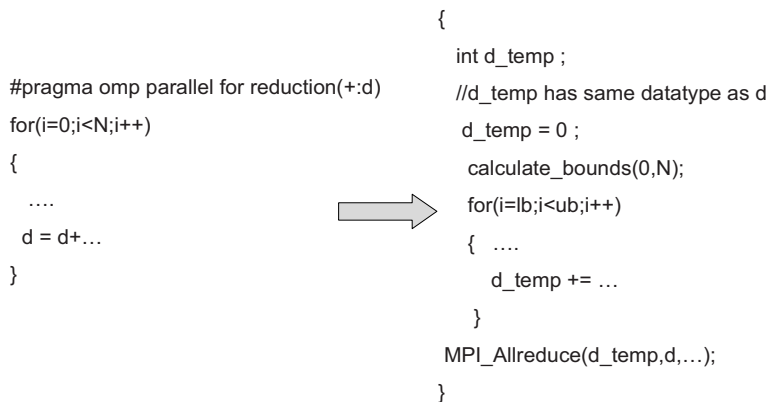
In tandem, shared data replication and redundant execution have some major advantages. Firstly, by eliminating the concept of an *owner* for shared data, we greatly simplify the manner in which accesses to shared data is handled. In previous efforts such as High Performance Fortran, compilers had to resort to elaborate analysis to resolve which nodes owned the data for a particular computation and allocate temporary place-holders for remote data which had to be used in a computation. In our case, for every stage of computation, irrespective of the access pattern, each process already has the data that it needs since data producers propagate their writes to all possible future consumers at the end of parallel constructs. Additionally, replication

ensures that a future consumer already has a placeholder to which a producer can communicate its writes.

Redundant execution of serial regions and replication of the data space affects the semantics of how certain OpenMP directives are handled. An exception to redundant execution of the serial regions is file I/O. Reading from files is redundantly done by all processes, but writing to file is done by only one process (the process with the smallest MPI rank). The same redundant execution strategy is followed for constructs in the parallel regions demarcated by *omp master* and *omp single* directives.

## Interpretation of Work-Sharing Directives

Our compiler converts OpenMP *section* constructs to OpenMP loops using a *switch-case* construct so that each iteration executes a different OpenMP section. Henceforth, the term "work-sharing construct" will only refer to OpenMP loops. OpenMP has four scheduling strategies that may be specified for parallel loops - static, dynamic, guided and runtime. Currently, we support only static scheduling. Additionally, at this step, if the OpenMP loop contains a *reduction* clause, then the compiler inserts an MPI\_Allreduce clause as shown in Figure 4.1.



```

#pragma omp parallel for reduction(+:d)
for(i=0;i<N;i++)
{
    ....
    d = d+...
}

{
    int d_temp ;
    //d_temp has same datatype as d
    d_temp = 0 ;
    calculate_bounds(0,N);
    for(i=lb;i<ub;i++)
    {
        ....
        d_temp += ...
    }
    MPI_Allreduce(d_temp,d,...);
}

```

Fig. 4.1. Translation of OpenMP loop with a *reduction* clause.

## Interpretation of Synchronization Directives

Synchronization directives in OpenMP may specify control synchronization or data synchronization (or both). We shall describe how our compiler interprets data synchronization directives (such as *flush*) later in this chapter in Sections 4.2.2 and 4.2.4.

Control synchronization directives such as *OMP CRITICAL* and synchronization functions such as *omp\_set\_lock* are implemented in the translated MPI program using MPI one-sided messaging. MPI one-sided messaging is described in detail in the specifications for MPI-2 [57]. For every OpenMP critical region in the program, our compiler initializes an MPI one sided communication “window”. The compiler inserts an `MPI_WIN_LOCK` function at the entry to a critical section and an `MPI_WIN_UNLOCK` function at the exit. The same mechanism is used to implement OpenMP lock functions.

## Interpretation of Data Directives

OpenMP data directives (such as *shared*, *private*, *threadprivate*) are interpreted by the compiler in finding the set of shared variables in the program. This step is described in the next subsection.

Thus, in the first translation step, our compiler converts the OpenMP application to an SPMD form, accordingly translating the OpenMP directives and performing the requisite partitioning for work sharing constructs. This step is illustrated with a simple example in Figure 4.2.

### 4.2.2 Incorporation of OpenMP Memory Consistency into the Dataflow Analysis

The next step in the translation of OpenMP applications to MPI involves the analysis of accesses to *shared data* in the OpenMP program. *Shared data* is defined

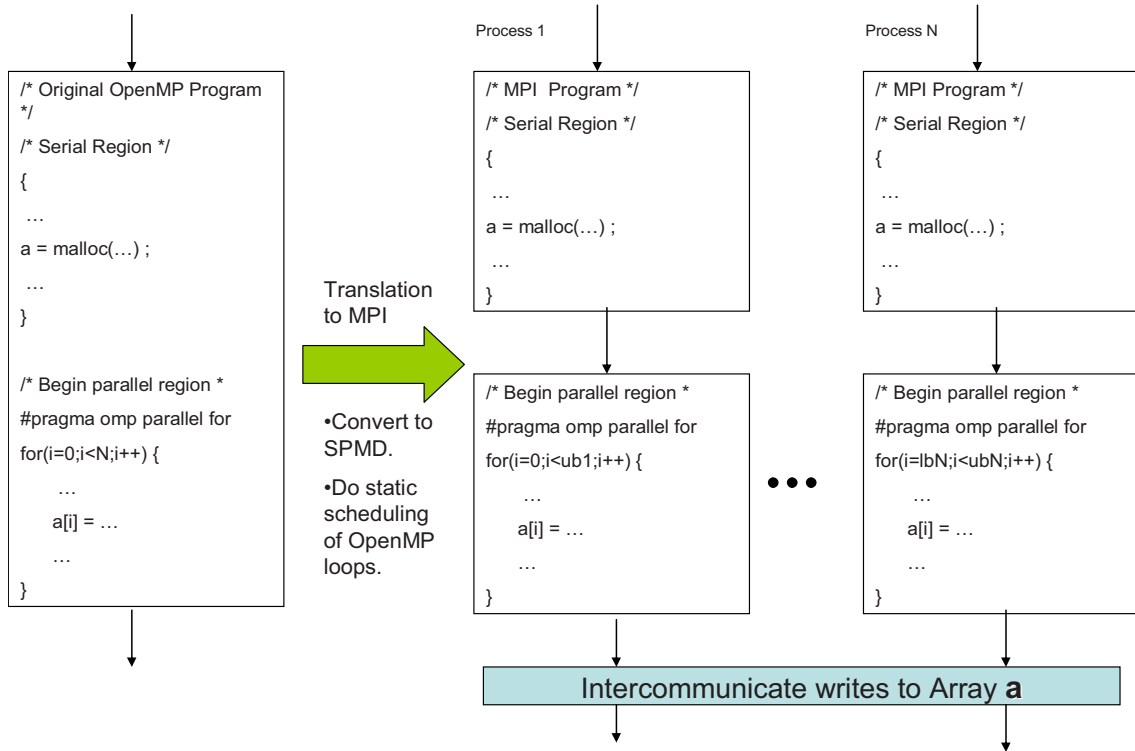


Fig. 4.2. Basic Translation scheme for converting OpenMP applications to MPI. Shared data is allocated by all processes. All processes redundantly execute serial regions. At the end of a parallel work-sharing construct, each processes communicates those writes that will potentially be read by other processes.

as data in the program that may be read or written by more than one thread in the OpenMP program. The compiler identifies shared-data using the algorithm that we have described in Figure 4.3.

There are two challenges involved in analyzing accesses to shared data in an OpenMP program. The first challenge is that OpenMP memory consistency specifications need to be incorporated into the dataflow analysis in resolving *def-use* or *producer-consumer* relationships between writes and reads to shared data. The second challenge is to perform data flow analysis for array variables in addition to scalar data. This section presents a technique for incorporating OpenMP memory consistency specifications into the data flow analysis for the program.



**Algorithm *list\_shared\_variables*****Input :**  $A$  - An OpenMP program. **Output :**  $S$  - A List of *Shared Variables* in  $A$ .**Start *list\_shared\_variables***

1. Set  $S = \Phi$
2. **do**  $\forall R$ ,  $R$  is an OpenMP *parallel region* in  $A$
3.     Set  $V$  = Set of all variables used in  $R$
4.     Set  $L$  = Set of all variables declared locally within  $R$
5.     Set  $PV$  = Set of all variables explicitly declared *private* for  $R$
6.     Set  $SV$  = Set of all variables explicitly declared *shared* for  $R$
7.     Set  $S = S \cup (V - L - PV) \cup SV$
8. **end do**
9. **if** ( $S = \Phi$ ) , *exit*, **endif**
10. **do**  $\forall Pr$ ,  $Pr$  is a *Procedure* defined within  $A$
11.     **do**  $\forall F$ ,  $F$  is *function call* within  $Pr$
12.         **do**  $\forall Pa$  ,  $Pa$  is a parameter of  $F$
13.             **if** ( $Pa \in S$ )
14.                 Let  $FP$  be the *Procedure* that defines  $F$
15.                 Let  $PA$  be the *Procedure Parameter* of  $FP$   
corresponding to function parameter  $Pa$
16.                 Set  $S = S \cup PA$
17.             **end if**
18.         **end do**
19.     **end do**
20. **end do**
21. **if** (Steps 10 through 20 have added new elements to  $S$ )
22.     Go to Step 10
23. **end if**

**End *list\_shared\_variables***Fig. 4.3. Algorithm to create list of *Shared Variables* in an OpenMP Program.

After interpreting the OpenMP directives, as discussed in the previous subsection, our compiler creates a control flow graph  $G = \langle V, E \rangle$  where a vertex  $V$  represents a basic block in the program and an edge  $E = V_1 \rightarrow V_2$  exists if the basic block denoted by  $V_2$  is a successor of the basic block denoted by  $V_1$ . This control flow graph  $G$  forms the basis for the dataflow discussed subsequently in this chapter. To incorporate OpenMP memory consistency semantics into the dataflow, this graph is adjusted to create an *OpenMP producer-consumer* graph.

The OpenMP memory consistency model is roughly equivalent to *Weak Consistency* [45]. Writes to shared data by one thread are not guaranteed to be visible to another thread till a synchronization point is reached. OpenMP has both implicit and explicit memory synchronization points. Examples of explicit synchronization points include *barrier* and *flush* directives. Implicitly, there are memory synchronization points at the end of work sharing constructs (unless they have explicit *nowait* clauses) and at the end of synchronization directives like *master* and *critical*. This means, for example, that writes to shared data in one iteration of an OpenMP *for* loop by one thread are not guaranteed to be visible to another thread executing a different iteration of the same loop till the implicit synchronization at the end of the loop is reached.

Figure 4.4 illustrates some other ramifications of the OpenMP consistency model. The *nowait* clause in loop  $L1$  denotes that writes to the array  $A$  by one thread in loop  $L1$  are not guaranteed to be visible to another thread executing iterations of loop  $L2$ . On the other hand, the flush directives denote that the update to the scalar *mydo* after loop  $L2$  by one thread may be visible to another thread reading *mydo* in loop  $L1$ . The control flow graph  $G$  now needs to be adjusted so that there is a path from the write to a shared variable to a read if the write by one thread is visible to the read by another thread as per OpenMP specifications. A way of doing this for the graph shown in Figure 4.4 would be to add the edge  $e1$  to account for the flush directives, delete the edge  $e2$  to account for the *nowait* clauses and to add edges  $e3$  and  $e4$  to keep other paths in the graph unbroken even though the edge  $e2$  has

been deleted. By doing these edge additions and deletions, we create an *OpenMP producer-consumer flow graph* where there is a path from a write  $W$  to a read  $R$  in the program if and only if the write  $W$  occurring on one thread can be visible to the read  $R$  occurring on another thread. In certain cases, like the reads and writes connected by edge  $e1$  in Figure 4.4, the read may be before the write in program order.

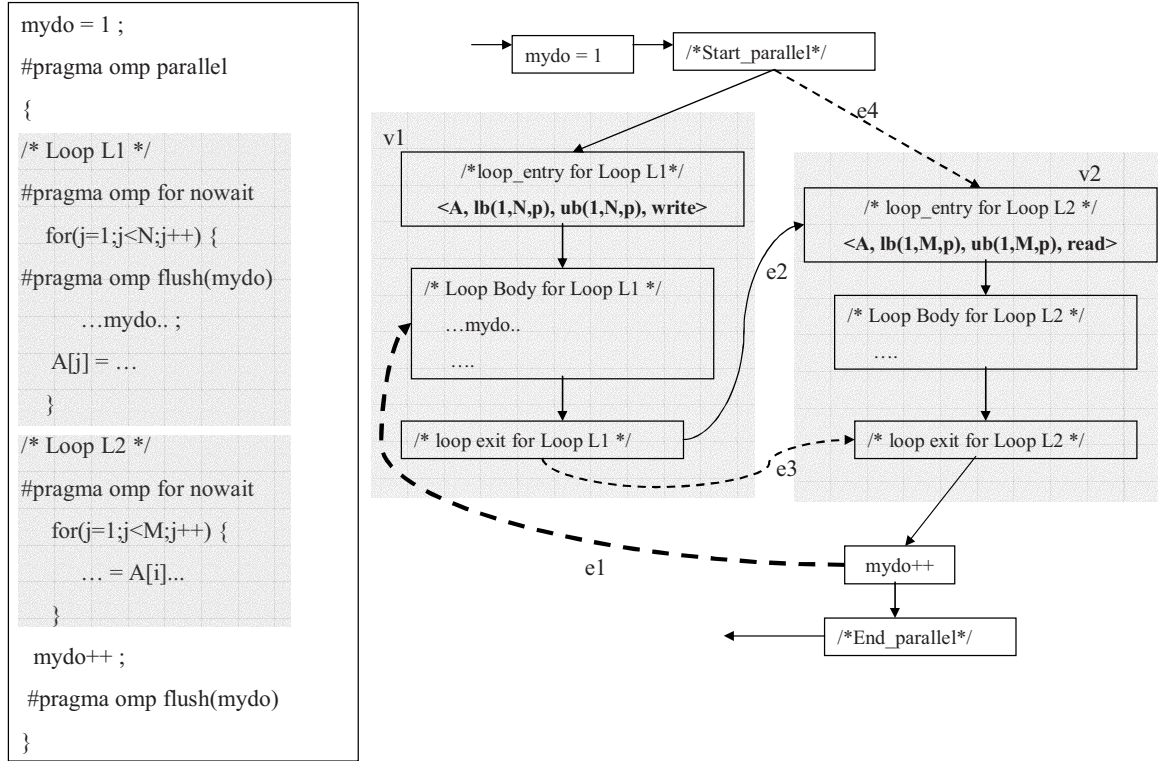


Fig. 4.4. Incorporation of OpenMP Memory Consistency Semantics into the Program Flow Graph: Additional dependencies are introduced by the *flush* directive. Existing dependencies are relaxed by the *nowait* clause.

We now present a formal algorithm to create the *OpenMP producer-consumer flow graph* from the control-flow graph  $G$ . This algorithm has three steps –

1. Make synchronization explicit in the program.
2. Relax Sequential Consistency.

### 3. Adjust for flushes.

In the control flow graph  $G$  for the OpenMP program, OpenMP directives are represented with special vertices. Directives that refer to a set of statements in the program code, are represented by *entry* and *exit* vertex pairs. For example, for each OpenMP parallel region in the program, there is a *parallel region entry* and a *parallel region exit* vertex. For each OpenMP critical section in the program, there is a *critical section entry* and a *critical section exit* vertex.

Stand-alone directives such as the *flush* and *barrier* directives are represented with a single vertex in the program flow graph. OpenMP *flush* directives are associated with a *flush set*, which is a list of all shared variables that need to be *flushed* at that point. When the flush set is explicitly specified in the program, the corresponding flush vertex is annotated with this flush set. The *atomic* directive is represented with a pair of *atomic entry* and *atomic exit* vertices around the atomic statement.

The first step, making synchronization explicit adds a *barrier* directive wherever control synchronization is implicit in an OpenMP directive. Thus, *barrier* vertices are added to the control flow graph  $G$  at the entry to an exit from parallel regions and worksharing regions that do not have *nowait* clauses. *flush* vertices are inserted where a flush is implicit without a barrier, such as at entry to and exit from *critical*, *ordered* and *atomic* regions. Flush sets are derived by the compiler for these inserted *flush* vertices. For example, for *critical* and *atomic* regions, flush sets include the shared variables accessed in these regions. For shared variables that have a *volatile* type, pairs of *flush* vertices enclose every access to these variables.

Thus, at the end of this first step, the control flow graph contains *barrier* vertices where control synchronization is implied in the program and *flush* vertices where a data coherence is implied in the OpenMP program.

With the graph  $G$  now containing explicit synchronization vertices and vertices corresponding to OpenMP directives, our compiler relaxes sequential consistency using the algorithm *relax\_sequential\_consistency* shown in Figure 4.5. In this algorithm,

**Algorithm *relax\_sequential\_consistency***

**Input :** 1. The Control-Flow Graph  $G$  for the OpenMP program containing explicit synchronization vertices for *barrier* and *flush* and *entry* and *exit* vertices for OpenMP directives.

**Output :** 1. A Control-Flow Graph  $G'$  for the program with a Relaxed Memory Consistency Model.

**Start *relax\_sequential\_consistency***

1. **do**  $\forall L$ ,  $L$  is an OpenMP loop,
2.     Remove the back edge from loop entry to loop exit for  $L$ .
3. **end do**
4. **do**  $\forall V_x$ ,  $V_x$  is an OpenMP *exit* vertex in  $G$
5.     **if** (  $G$  contains an edge  $V_x \rightarrow V_y$  where
6.          $V_y$  is not an OpenMP *barrier* vertex ) then
7.         Delete edge  $V_x \rightarrow V_y$
8.         Let  $V_{ey}$  be a *barrier* vertex reachable from  $V_y$  without intervening barriers
9.         Let  $V_{dx}$  be a *barrier* vertex that strictly dominates  $V_x$
10.        Add edge  $V_x \rightarrow V_{ey}$  to  $G$
11.        Add edge  $V_{dx} \rightarrow V_y$  to  $G$
12.     **end if**
11. **end do**

**End *relax\_sequential\_consistency***

Fig. 4.5. Algorithm to adjust the Control Flow Graph to remove dependencies according to OpenMP's Memory Consistency specifications

**Algorithm *Adjust\_for\_Flushes***

**Input :** 1. The Control-Flow Graph  $G$  for the OpenMP program  
created by algorithm *relax\_sequential\_consistency*.

**Output :** 1. An OpenMP Producer-Consumer Flow Graph  $G'$  for the program.

**Start *Adjust\_for\_Flushes***

1. **do**  $\forall V_f, V_f$  is a *flush* vertex in  $G$ ,
2.     **do**  $\forall V'_f, V'_f$  is a *flush* vertex in  $G, V_f \neq V'_f$
3.         **if** ( $V_f \neq V'_f$  and  $V'_f$  is not reachable from  $V_f$ ) **then**
4.             Let  $S$  be the *flush-set* of  $V_f$
5.             Let  $S'$  be the *flush-set* of  $V'_f$
6.             **if** ( $V_f$  and  $V'_f$  can be *concurrent* [58]
7.                 and  $S \cap S' \neq \Phi$  ) **then**
8.                 Add edge  $V_f \rightarrow V'_f$  to  $G$
9.             **end if**
10.         **end if**
11.     **end do**
12. **end do**

**End *Adjust\_for\_Flushes***

Fig. 4.6. Algorithm to adjust the Control Flow Graph to incorporate dependencies created by explicit *flushes*

the compiler deletes edges from the program’s control flow graph, to break paths from writes to subsequent reads in the program where the weak consistency model of OpenMP specifies that the write by one thread may not be visible to the read on another thread. Then the compiler adds edges from the previous synchronization points in the program to ensure that the paths to the read from writes before the previous synchronization point still exist.

Finally, our compiler uses the algorithm *Adjust\_for\_Flushes* shown in Figure 4.6 to adjust for explicit flushes in the program. For line 6 of this algorithm, the compiler uses a concurrency analysis for OpenMP [58] which has been used by other researchers as part of static race detection in OpenMP programs.

At this point, the compiler has a control flow graph that reflects the OpenMP memory consistency model. In this graph, there is a path from a write statement  $S1$  to a future read statement  $S2$  if and only if the execution of  $S1$  by one thread produces an update to memory that is visible to the execution of  $S2$  by another thread, as per OpenMP specifications. We refer to this adjusted control flow graph as the *OpenMP Producer-Consumer Flow Graph*.

Previous research into the compiler analysis for programs with relaxed consistency models have focussed on *delay set analysis* [59–61] to ensure correct execution of programs. Others have proposed techniques to use the compiler to hide or abstract the effects of the memory model from the programmer [62,63] and in doing so have relied on specialized graph representations such as the *Concurrent Static Single Assignment* form to represent parallel programs. Our techniques do not have to incorporate delay set analysis since our goal is to incorporate the relaxed semantics of OpenMP and we are performing source-to-source compilation without any reordering of loads and stores. The requisite fences or barriers in our program are already present in the form of OpenMP synchronization statements. Also, rather than incorporating any specialized representation, our technique adds and deletes edges to the program’s control-flow graph to incorporate the effects of weak memory consistency and the additional constraints introduced by OpenMP flushes.

We now present a formal proof of the correctness of the two algorithms presented above.

**Lemma 1** *A Read statement vertex  $R$  is reachable from a Write statement vertex  $W$  in the OpenMP Producer Consumer Flow Graph  $\Rightarrow$  the execution of  $W$  by one thread is guaranteed to be visible to the execution of  $R$  by another thread according to OpenMP specifications.*

**Proof** Consider two cases.

Case 1 –  $R$  occurs after  $W$  in program order.

In this case, there can be three scenarios – (i) both  $R$  and  $W$  are in serial regions, (ii) either  $R$  or the  $W$  is in a serial region and (iii) both  $R$  and  $W$  are in a parallel regions.

If  $R$  and  $W$  are both in serial sections and there is a path from  $W$  to  $R$ , then the statement is trivially true.

If  $W$  is in a serial region and  $R$  is in a parallel region, let  $E_R$  be the entry vertex for the parallel region that  $R$  is in.  $E_R$  dominates  $R$  and so any path from  $W$  to  $R$  must contain  $E_R$ . Since there is an implicit synchronization at the beginning and end of each parallel region,  $E_R$  is dominated by a *barrier* vertex which must also be in the path from  $W$  to  $R$  and thus, the execution of  $W$  will be visible to an execution of  $R$  on any thread. Similarly, if  $W$  is in a parallel region, the barrier at the end of this parallel region must be in the path from  $W$  to  $R$  and thus the execution of  $W$  will be visible to an execution of  $R$  on any thread.

If  $W$  and  $R$  are both in a parallel regions then let  $E_W$  be the exit vertex for the OpenMP construct that  $W$  is within. Since Algorithm 4.5 ensures that the successor of  $E_W$  is always a *barrier* vertex, there is always a *barrier* vertex in the path from  $W$  to  $R$  and thus the execution of  $W$  will be visible to an execution of  $R$  on any thread.

Case 2 –  $R$  occurs before  $W$  in program order.

In this case, the path from  $W$  to  $R$  must contain an edge not present in the original control flow graph of the program. Additional edges are introduced by line 8



in algorithm *Adjust\_for\_Flushes* in Figure 4.6 and by lines 10-11 in algorithm *relax\_sequential\_consistency* in Figure 4.5. Since these edges contain vertices which are either barriers or flush pairs, the execution of W must be visible to an execution of R on any thread. ■

**Lemma 2** *The execution of a Write statement W by one thread is guaranteed to be visible to the execution of a Read statement R by another thread according to OpenMP specifications  $\Rightarrow$  R is reachable from W in the OpenMP Control Flow Graph G.*

**Proof** If W is visible to R on all threads as per OpenMP specifications, then one of two cases must be true –

Case 1 – R is reachable from W in the original program flow graph.

In this case, there must be an intervening *barrier* vertex in the path from W to R since W is guaranteed to be complete before R is started on any thread. We call this *barrier* vertex  $V_b$ . The only transformation that deletes edges from the original control flow graph is line 7 in algorithm *relax\_sequential\_consistency* in Figure 4.5. However, the additional edges introduced in lines 10 and 11 of this algorithm ensure that paths from OpenMP *entry* and *exit* vertices to preceding and succeeding barriers are not broken. Thus, a path from W to  $V_b$  and from  $V_b$  to R is unbroken by the two algorithms. Thus, R is reachable from W in the OpenMP Producer Consumer Flow Graph G.

Case 2 – R is not reachable from W in the original program flow graph.

In this case, W must become visible to R because of OpenMP *flush* directives. Thus, there must be a *flush* in the program after W that is dominated by W in the original program flow graph. There must also be a *flush* in the program that dominates R in the original program flow graph. Additionally, these two flushes must be *concurrent*. However, if these flushes are concurrent, then line 8 in algorithm *Adjust\_for\_Flushes* in Figure 4.6 will create an edge between them. Thus, R will be reachable from W in the OpenMP Producer Consumer Flow Graph G. ■

**Theorem 4.2.1** *For an OpenMP Producer Consumer Flow Graph  $G$ , a Read statement  $R$  is reachable from a Write statement  $W \Leftrightarrow$  the execution of  $W$  by one thread is guaranteed to be visible to the execution of  $R$  by another thread according to OpenMP specifications.*

**Proof** By Lemma 1, A *Read* statement  $R$  is reachable from a *Write* statement  $W \Rightarrow$  the execution of  $W$  by one thread is guaranteed to be visible to the execution of  $R$  by another thread according to OpenMP specifications.

By Lemma 2, the execution of  $W$  by one thread is guaranteed to be visible to the execution of  $R$  by another thread according to OpenMP specifications  $\Rightarrow$  A *Read* statement  $R$  is reachable from a *Write* statement  $W$ .

Combining the two, we get “A *Read* statement  $R$  is reachable from a *Write* statement  $W \Leftrightarrow$  the execution of  $W$  by one thread is guaranteed to be visible to the execution of  $R$  by another thread according to OpenMP specifications.” ■

### 4.2.3 Computation of Message Sets

In the next step, our compiler computes the message-sets – data that must be sent from producers to remote consumers to resolve remote accesses. Remote accesses may be to scalars or arrays. First, we present the computation of message sets to satisfy remote array accesses.

Precise analysis of array accesses is an essential part of our translation from OpenMP to MPI and our dataflow analysis must deal with array-dataflow. Several schemes such as Linearized Memory Access Descriptors(LMAD) [64] and Regular Section Descriptors(RSD) [65] have been proposed to characterize array accesses. Our compiler characterizes and aggregates accesses to shared arrays using Regular Section Descriptors. As a first step in building a dataflow graph, the compiler creates RSDs for the shared arrays. For arrays accessed in a loop, dataflow nodes are inserted before the loop body, containing RSDs that characterize the accesses. For OpenMP

parallel loops that contain inner loops, the RSD is formulated at the level of the outer parallel loop.

The objective of this pass is to identify the minimal subset of shared data that a producer  $p$  needs to communicate to a potential future remote consumer  $q$  and then to efficiently communicate this data. To do this, starting from dataflow nodes corresponding to the Regular Section Descriptors that specify the *def* sets of OpenMP loops, the dataflow graph is traversed in the forward direction to identify elements that are used before being overwritten. This pass identifies the set of shared-data produced in the loop for which future remote accesses may occur. This algorithm shown in Figure 4.7 computes the producer consumer relationships in terms of message sets. Generation of communication using array regions has been used in several compiler efforts [66,67]. While being similar in essence, our method differs in that we focus on the point at which data is produced and then do a forward dataflow analysis.

For affine accesses, the intersection operation  $WA_i \cap U_j$  produces affine expressions for  $UW_{ij}$ . If the accesses are regular but not affine,  $UW_{ij}$  is expressed symbolically and gets resolved during program execution.

In case of irregular accesses, the compiler may not be able to precisely determine the sets  $WA_i$  in line 10 and  $U_j$  in line 8. The compiler may then conservatively treat reads and set  $U_j$  to be the entire array  $A$  in line 8. Methods for handling irregular writes are discussed in Section 5.3.

Next, the compiler resolves remote accesses to scalars. Scalars produced in the serial regions never need to be communicated since the serial region is redundantly executed by all processes. A scalar produced inside parallel regions need to be communicated in two cases - when there is an explicit synchronization directive (such as a flush) after the scalar is written or if the write to the scalar is not reachable along all paths in the loop. In this case, the dataflow graph is traversed starting from the write to the scalar and if there are any reachable remote reads, a single-element message set (containing only the scalar) is computed for the scalar.

**Algorithm *COMPUTE\_MSG\_SETS***

$N$  is the number of participating processes

**Inputs :**

The Dataflow graph for the program.

Regular Section Analysis(RSA) output that provides a summary of array accesses for loops.

$S_W$  - The set of arrays written in work-sharing construct  $W$

**Outputs:**

$UW_{ij}$  - The set of data elements that must be communicated from process  $i$  to process  $j$  at the end of work-sharing construct  $W$

***Start COMPUTE\_MSG\_SETS***

1.  $\forall$  Work-sharing construct  $W$
2. do
3.   Initialize  $UW_{ij} = \phi, 1 \leq i, j \leq N$
4.   Let  $S_W =$  Set of arrays written to in  $W$  (this is provided by the RSA pass)
5.    $\forall$  array  $A, A \in S_W$ , find
6.    $FU_A =$  Set of future uses of  $A$  (done by traversing the dataflow graph
7.   and using RSA output for accesses within loops).
8.    $\forall U \in FU_A$  create  $U_j$ , the set of elements of  $A$  used by process  $j$ .
9.   do  $i=1, N$
10.     Find  $WA_i$ , the set of elements of  $A$  written in  $W$  such that
11.      $WA = WA_1 \cup WA_2 \dots \cup WA_N$
12.     do  $j = 1, N$
13.        $UW_{ij} = U_{Wij} \cup (WA_i \cap U_j)$
14.     enddo
15.   enddo
16. enddo

***End COMPUTE\_MSG\_SETS***

Fig. 4.7. Algorithm to Compute Message Sets for Resolving Remote Accesses

## Creation of Datatypes for Non-Contiguous Accesses

MPI send and receive messages specify a starting address, a datatype and the number of contiguous elements to send. Each element is of the specified datatype and the first element resides at the starting address. If the set  $UW_{ij}$  in Figure 4.7 consists of a set of non-contiguous elements, then they cannot be sent in a simple message. This is frequently the case where  $D$  is a multi-dimensional array. As an example, consider a 8X8 matrix  $A$ , which is written to in a parallel loop, row-wise, by 4 processes. Thus process 1 produces rows 0 and 1; process 2 produces rows 2 and 3 and so on. In a later phase of the program,  $A$  is consumed column-wise. So process 1 reads columns 0 and 1; process 2 reads columns 2 and 3 and so on. So, process 1 must send  $A[0][2], A[0][3], A[1][2]$  and  $A[1][3]$  to process 2. Of these 4 elements, the first two and the second two reside contiguously in memory, but there is a gap of 6 elements between  $A[0][3]$  and  $A[1][2]$ . Multiple short messages may be used to communicate such non-contiguous elements. However, this would result in poor performance as there is a per message overhead. Instead, our compiler solves this problem by registering a new datatype for every non-contiguous dataset that must be communicated. This, in effect, specifies to the MPI library how non-contiguous data must be packed at the sender and unpacked at the receiver. The send/receive is then carried out with just a single message, by using this new datatype.

The formulation of the datatype is done as follows. Consider a  $k$ -dimensional array  $A$  defined as  $A[N_0][N_1] \dots [N_{k-1}]$ , that is written along the  $x^{th}$  dimension in a parallel loop and read along the  $y^{th}$  dimension in a later parallel loop. For simplicity, assume that  $N_t$  is divisible by  $p$  (the number of processes)  $\forall 1 \leq t \leq k$ . Then, for the array  $A$ , the set  $UW_{ij}$  for the algorithm shown in Figure 4.7 is given by  $A < [0, N_0), [0, N_1), \dots, [N_x * i/p, N_x * (i+1)/p), \dots, [N_y * j/p, N_y * (j+1)/p), \dots, [0, N_{k-1}) >$ . Thus, to send these elements from process  $i$  to process  $j$ , a *vector* datatype is declared in MPI [12] using the following attributes-

$$count = N_0 * N_1 * \dots * N_{x-1} * (N_x/p) * N_{x+1} * \dots * N_{y-1}$$

$$stride = N_y * N_{y+1} * \dots * N_{k-1}$$

$$blocklength = (N_y/p) * N_{y+1} * \dots * N_{k-1}$$

Two additional steps are performed here. Firstly, if possible, the creation of the datatype is hoisted so that it does not need to be done for every invocation of the send/receive pair. Secondly, the same datatype is used for every send-receive that has the same pattern (that is, the same number of non-contiguous sets with the same number of contiguous elements in the same set).

#### 4.2.4 Placement of Messages in the Translated Program

Section 4.2.3 described the computation of the message sets for satisfying producer-consumer dependencies in the program. The next step is the actual placement of messages in the translated MPI code.

The movement of data in the message sets computed in Section 4.2.3 is accomplished using MPI two-sided messaging – using a *non-blocking send* and a *blocking receive* pair of operations. Data produced in a work-sharing construct or a parallel region is sent at the end of the construct or region where it is produced. The corresponding receive for the data sent is inserted before the region or work sharing construct where any of this data is *potentially consumed for the first time*. Referring to Figure 4.7, this means that for the message set  $UW_{ij}$  calculated for a construct  $W$ , the non-blocking send from process  $i$  to process  $j$  is started on process  $i$  at the end of the construct  $W$ . The corresponding blocking receive is posted on process  $j$  before the first construct where any part of  $UW_{ij}$  is used. If the set  $U = \{U_1, \dots, U_N\}$  represent the set of constructs reachable from  $W$  where part of  $UW_{ij}$  is consumed and there is a single construct  $U_k$  that dominates all other constructs in the set  $U$ , then the blocking receive for  $UW_{ij}$  is posted on process  $j$  just before the construct  $U_k$ . If there is no such dominator construct  $U_k$ , then the receive is posted on process  $j$  right after construct  $W$  itself.

In matching a send with a receive, the compiler uses two types of parameters embedded within the MPI send and receive calls. MPI non-blocking send calls have the prototype *MPI\_Isend(void \*buf, int count, MPI\_Datatype datatype, int destination, int tag, MPI\_Comm comm, MPI\_Request \*request)* and MPI blocking receives have the prototype *MPI\_Recv(void \*buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Status \*status)*. To match a send and a receive, the compiler makes use of the *source/destination* parameters as well as the *tag* parameter. Thus, for communicating the message set  $UW_{ij}$ , the non-blocking send on process  $i$  sets its destination parameter to  $j$  and the blocking receive on process  $j$  sets its source parameter to  $i$ . Additionally, each message set  $UW$  communicated in Algorithm *COMPUTE\_MESSAGE\_SETS* in Figure 4.7 is assigned an unique identifier  $M$ . The *tag* parameter in both the send and the receive is set to this identifier  $M$ . Thus, the compiler uses the combination of source, destination and tag parameters to match the sends and receives for a message set  $UW_{ij}$ .

### Message Generation for Flushes

The implicit barriers at the end of parallel regions and work sharing constructs (that do not have *nowait* clauses) represent well-defined *join* points in the fork-join structure of OpenMP parallelism. These join points are positions in the program where the send and receive messages discussed above may be placed. However, there are certain scenarios where there may be no such join point between the part of the program where shared data is produced and where it is consumed.

Consider the OpenMP code snippet shown in Figure 4.2.4. In this code, because of the OpenMP semantics regarding flush, there is a producer-consumer relationship between the flush in statement  $S2$  and the flush in statement  $S1$ . Thus, in the translated MPI code, there must be a send posted by the process that updates  $njob$  and reaches statement  $S2$  and a corresponding receive posted by the process executing

```

njob = 0 ;
#pragma omp parallel
{
    while (myid > njob) {
S1: #pragma omp flush(njob)
        ; // Wait for lower rank to increment njob
    }

    ...

    njob++ ;
S2: #pragma omp flush(njob)
}

```

Fig. 4.8. OpenMP program snippet with a pair of *flush* directives.

statement *S1*. However, there are no intervening explicit or implicit barriers between *S2* and *S1* where such messages may be placed.

To satisfy producer-consumer relationships produced by flushes, our compiler uses a specific message generation algorithm for flushes shown in Figure 4.9. Each element in a flush list in the program is assigned an unique identifier *M*. For each explicit OpenMP flush directive, as well as implicit flushes associated with critical sections and OpenMP *atomic* directives, the Algorithm shown in Figure 4.9 is used to generate MPI messages.

The four transformations done by the compiler for the baseline translation of an OpenMP application to an MPI application are summarized in Figure 4.10. In the next section, we describe optimizations to improve the performance of the translated application.



**Algorithm *Generate\_Messages\_for\_Flushes***

**Input :**

1. An explicit OpenMP Flush Directive  $F$ .

**Output :**

1. MPI Messages to be generated for  $F$ .

**Start *Generate\_Messages\_for\_Flushes***

1. **do**  $\forall V_F$ ,  $V_F$  is an element in the *flush-set* of  $F$ ,
2.     Let  $M$  be the unique identifier corresponding to  $V_F$ .
3.     **if** ( $F$  is dominated by an update to  $V_F$  without intervening flushes)
4.         Generate non-blocking sends of  $V_F$  to all  
            other processes with message tag set to  $M$ .
5.     **else**
6.         Generate an *MPI\_IProbe* message with
7.          $source = MPI\_ANY\_SOURCE$  and  $tag = M$
8.         **if** (*MPI\_Iprobe* returns status as *received*)
9.             Generate a blocking receive for  $V_F$ .
10.        **end if**
11.     **end if**
12. **end do**

**End *Adjust\_for\_Flushes***

Fig. 4.9. Algorithm to generate messages for explicit *OMP FLUSH* directives

### 4.3 Optimizations

In Section 4.2, we have described techniques for the baseline translation of OpenMP applications to MPI. In this section, we describe three compile-time optimizations

1. Interpret OpenMP Directives.
  - Convert program to *SPMD* form with serial regions redundantly executed on all processes.
  - Do *static scheduling* of OpenMP work-sharing constructs.
  - Generate MPI code for OpenMP synchronization constructs.
2. Incorporate OpenMP Memory Consistency Model to Create OpenMP Producer-Consumer Flow Graph.
  - Make synchronization explicit in the program's control flow graph.
  - Relax sequential consistency.
  - Adjust for flushes.
3. Compute Message Sets.
  - Summarize array accesses using *Regular Section Descriptors*.
  - Use inter-procedural *def-use* analysis to find potential future uses of shared data produced in parallel regions.
  - Create datatypes for non-contiguous accesses.
4. Generate Messages.
  - Use non-blocking sends and blocking receives to communicate message sets.
  - Generate messages for flushes.

Fig. 4.10. Summary of the Baseline Translation of OpenMP applications to MPI.

which have been found to improve performance considerably. While a detailed evaluation of the impact of each optimization is beyond the scope of this chapter, in Section 4.4 we shall indicate the applicability of these optimization techniques for the benchmarks we evaluated.

#### 4.3.1 Computation Alignment and Repartitioning

In chapter 3, we discussed computation repartitioning in the context of deploying OpenMP applications on Software DSM systems. The same optimization is useful

even in translating OpenMP applications to MPI. It addresses the case where the OpenMP application has loops partitioned in such a way that multi-dimensional arrays are accessed along different indices in different loops. Computation repartitioning then changes the level at which parallelism is specified and partitions work using the parallelism of the inner loops. Essentially, computation repartitioning aims to maintain locality of access. In doing so, it may eliminate the need to communicate data. An important point is that on shared-memory platforms, OpenMP applications have a fork-join overhead for every parallel region. To minimize this fork-join overhead, in the case of nested parallel loops where some of the inner loops are parallel as well, the outermost loop is marked for work-sharing. While this reduces fork-join overheads, this may be harmful from a locality perspective. In our translation to MPI, there is no fork-join overhead since the OpenMP application is translated to SPMD form. The entire overhead in our case comes from the communication of data and computation repartitioning may reduce this overhead substantially.

```
#pragma omp for
for(i=lx;i<ux;i++) {
    ...=A[i];
    A[i] = ...;
}
...
#pragma omp for
for(i=ly;i<uy;i++) {
    ...=A[i];
    A[i]= ...;
}
```

Fig. 4.11. OpenMP program snippet for Computation Alignment Example

```

    calculate_bounds(lx,ux,&l1,&u1,my_proc_num);
for(i=l1;i<u1;i++) {
    ...=A[i];
    A[i] = ...;
}
calculate_bounds(ly,uy,&l2,&u2,my_proc_num);
if(l1 != l2 || u1 != u2)
    // Send and receive elements outside
    // overlap of old and new partition
    ...
for(i=l2;i<u2;i++) {
    ...=A[i];
    A[i]= ...;
}

```

Fig. 4.12. MPI version without Computation Alignment

A related optimization is Computation Alignment. Repartitioning addresses the issue of accesses to multi-dimensional arrays along different axes. Alignment addresses the case where in spite of access along the same direction, a difference in loop bounds may cause iterations to be allocated to processors in a way that necessitates communication. For example, consider the program segment in Figure 4.11. A simple translation to MPI is shown in Figure 4.12. This necessitates the communication between the loops. An MPI version is shown in Figure 4.13. Here, the alignment of access regions eliminated the need for communication. The flip side is that this may introduce some load imbalance. However, the gain from eliminating the communication outweighs the disadvantage of the load imbalance introduced.

```

lz = MIN(lx,ly); uz = MAX(ux,uy) ;
calculate_bounds(lz,uz,&l1,&u1,my_proc_num);
if (lz < lx) l1=lx ;
if (uz > ux) u1=ux ;
for(i=l1;i<u1;i++) {
    ...=A[i];
    A[i] = ...;
}

// No communication necessary
...
if (lz < ly) l1=ly ;
if (uz > uy) u1=uy ;
for(i=l1;i<u1;i++) {
    ...=A[i];
    A[i]= ...;
}

```

Fig. 4.13. MPI version with Computation Alignment

### 4.3.2 Recognition of Reduction Idioms

The objective of this optimization is to recognize that the programmer is using a series of OpenMP constructs to perform a reduction operation and to substitute it with a MPI collective reduction operation. Recognition of reduction idioms have been discussed in the context of parallelizing compilers [68]. We apply this technique in a way that is the converse of the transformations done by parallelizing compilers. OpenMP does have a reduction clause which can be specified for OpenMP loops. However, this is most commonly used for scalar reductions. Array reduction and other complex forms of reduction operations are usually implemented in OpenMP using an OpenMP for loop followed by a critical section in which threads update global memory. In the baseline translation from OpenMP to MPI, a critical section is implemented as a section which processes enter in order. Consider that there are  $N$  processes, numbered  $1 \dots N$ , that are going to execute a critical region.  $\forall i > 1$ , before entering the critical region, process  $i$  waits for a message from process  $i - 1$  which notifies process  $i$  that process  $i - 1$  has completed the critical region and also communicates the updates made by process  $i - 1$  to process  $i$ .  $\forall j < N$ , process  $j$  sends a message to process  $j + 1$  communicating its updates when it has completed the critical region. Finally, when process  $N$  completes the critical region, it broadcasts its update to process  $1$  through  $N - 1$ . This whole process results in a large performance overhead which grows linearly with the number of processes involved. It can be eliminated if the combination of the parallel loop and the critical section can be recognized as implementing reductions, in which case simple MPI reduction functions can be used.

### 4.3.3 Optimization of Sends and Receives for Shared Data

The message-generation described in Section 4.2.4, performed as part of the baseline translation scheme described in Section 4.2, identifies the production and consumption points of the message sets. To overlap computation with communication,

messages are initiated early using *non-blocking sends/receives* and completed just before the consumption point at the receiver with a *wait*. A similar scheme has been used by the IBM HPF compiler [67].

However, this scheme produces limited improvement when the distance between the production and consumption points is insufficient. In such a case, if the message-generation algorithm places the corresponding receives immediately after the sends and the producer-consumer relationships are of the many-to-many type, then our compiler casts the sends and receives as collective communications using `MPI_Alltoallv` calls. Most MPI implementations use advanced algorithms to optimize collective communications and this optimization achieves significantly better performance than that of pairwise send/receives between processes.

#### 4.4 Performance Evaluation

We have implemented the translation steps, described in Section 4.2, in the Cetus compiler infrastructure [55]. We then manually applied the techniques described in Section 4.3 to seven representative OpenMP applications.

As our benchmarks, we have selected five out of the eight benchmarks from the NAS Parallel Benchmarks(NPB3.0) suite [30,69], namely CG,IS,EP,FT and LU, and two benchmarks from the SPEC OMPM2001 suite [49], namely ART and EQUAKE. Our compiler infrastructure handles only C programs. Thus for SPEC OMPM2001 we have used two out of the three available C benchmarks and for the NAS benchmarks, we have used the OpenMP C versions created by the Real World Computing Partnership(<http://phase.hpcc.jp/Omni/benchmarks/NPB/>). These versions have been found to be comparable to the OpenMP versions included as part of the NPB3.0 suite [70]. Currently, we manually apply the optimization techniques described in Section 4.3. Four of the NAS application benchmarks (LU, MG, BT and SP) have similar characteristics. So, we have included only one of them (LU) in our evaluation. Of the three SPEC OMPM2001 C applications, AMMP has been found to show lim-

ited scalability even on shared-memory platforms owing to fine-grained locking used in the code. Therefore, we have excluded that benchmark from our evaluation as well.

In this section, we shall present a performance evaluation of these translated OpenMP versions (henceforth referred to simply as the OpenMP versions).

#### 4.4.1 Brief Description of the Benchmarks

- **ART** is an application for Image Recognition using the Adaptive Resonance Technique, in the SPEC OMPM2001 benchmark suite. The most time consuming part in this application is the *scan\_recognize* subroutine. It mostly contains regular accesses, but has dynamic allocation of shared data inside the parallel region.
- **EQUAKE** is an application for Earthquake Simulation in the SPEC OMPM2001 suite. This application spends a major part of its time performing sparse matrix-vector multiplication. It contains irregular reads and writes for shared-data.
- **IS** does Integer Sorting and is a part of the NAS Parallel Benchmarks. The timed section of this benchmark performs a ranking of the elements to be sorted. This benchmark contains irregular reads and writes.
- **CG** is a NAS Parallel Benchmark that implements the *conjugate gradient* algorithm. The timed section iteratively computes the conjugate gradient and contains irregular read accesses.
- **EP** is an Embarrassingly Parallel benchmark from the NAS Parallel benchmarks suite. It generates pairs of Gaussian random deviates according to a specific scheme. The time consuming part of this application is a parallel loop that has a reduction clause.
- **FT**, from the NAS Parallel Benchmarks suite, contains the computational kernel of a 3-D fast Fourier Transform (FFT)-based spectral method. FT per-



forms three one-dimensional (1-D) FFTs, one for each dimension. It has multi-dimensional arrays that are accessed along different directions.

- **LU**, from the NAS Parallel Benchmarks suite, is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems. The most time consuming part of this application is the *ssor()* subroutine that implements the SSOR method.

#### 4.4.2 Comparison with Hand-Tuned MPI

The main objective of this performance evaluation is to compare the scalability achieved by the OpenMP versions with the scalability of their hand-coded MPI counterparts. The hand-coded MPI versions for CG, IS, EP, FT and LU are part of the NAS Benchmarks NPB2.4 suite. They have been used to evaluate MPI implementations on several platforms and represent an estimated upper bound on performance. The SPEC OMPM2001 benchmarks EQUAKE and ART do not have any such existing MPI versions. We created the MPI versions, using reasonable programming efforts.

For our experiments, we have selected two different platforms – (1) a cluster of 16 PIII 800 MHz Linux nodes, with 256 MB memory per node, connected by a commodity 100 Mbps Ethernet network and (2) sixteen IBM SP2 WinterHawk-II nodes connected by a high-performance switch. We expect the scaling behaviour of the benchmarks on these systems to be representative of a wide class of high-performance computing platforms. The MPI libraries used for these platforms is MPICH version 1.2.5 on Linux and IBM MPI libraries on the IBM SP2. The back end compilers used are gcc-3.3 on Linux and xlc version 6 on IBM, both at optimization level O3.

Figures 4.14,4.15,4.16,4.17,4.18,4.19 and 4.20 show the execution time and speedups of the translated OpenMP and hand-coded MPI versions of these benchmarks. The speedups shown in these figures is calculated as  $speedup = \frac{t_{serial}}{t_{parallel}}$  where  $t_{serial}$  is the execution time of best serial version (NAS serial versions in case of the NAS benchmarks and the OpenMP version compiled with the directives ignored for the SPEC OMP benchmarks) and  $t_{parallel}$  is the execution time of the relevant parallel version.

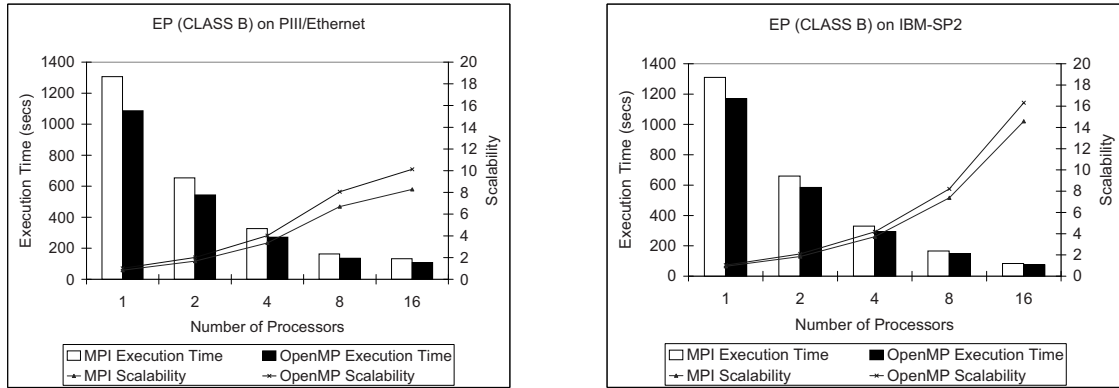


Fig. 4.14. Performance of EP (CLASS B).

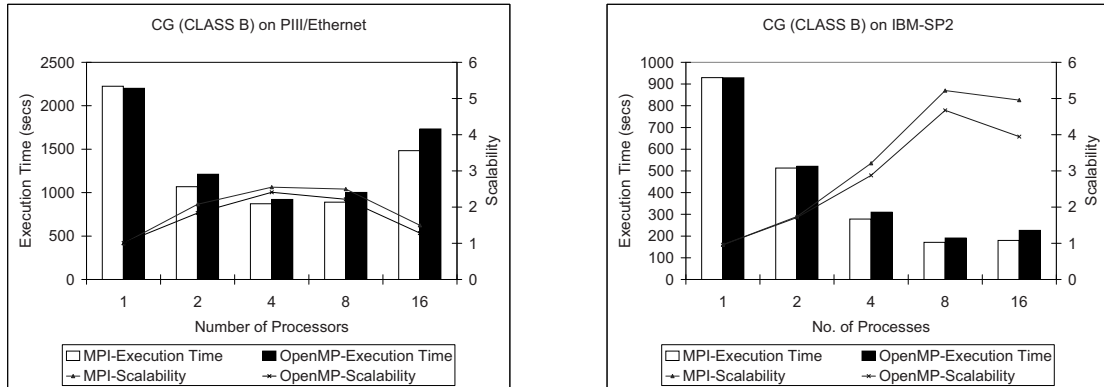


Fig. 4.15. Performance of CG (CLASS B).

For four out of the five NAS benchmarks examined, the original MPI and serial versions are in Fortran. Owing to the difference in the quality of code produced by C and Fortran compilers, there are considerable variations in the execution times on

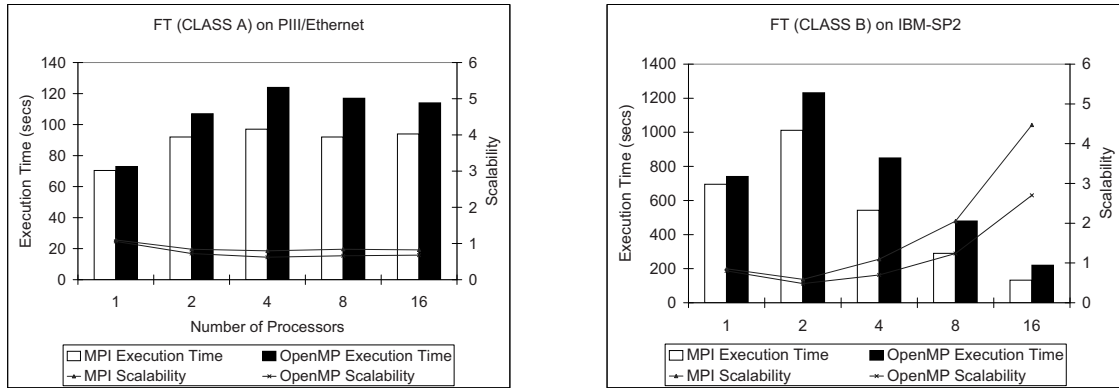


Fig. 4.16. Performance of FT (CLASS A on Ethernet Cluster and CLASS B on IBM SP2 ).

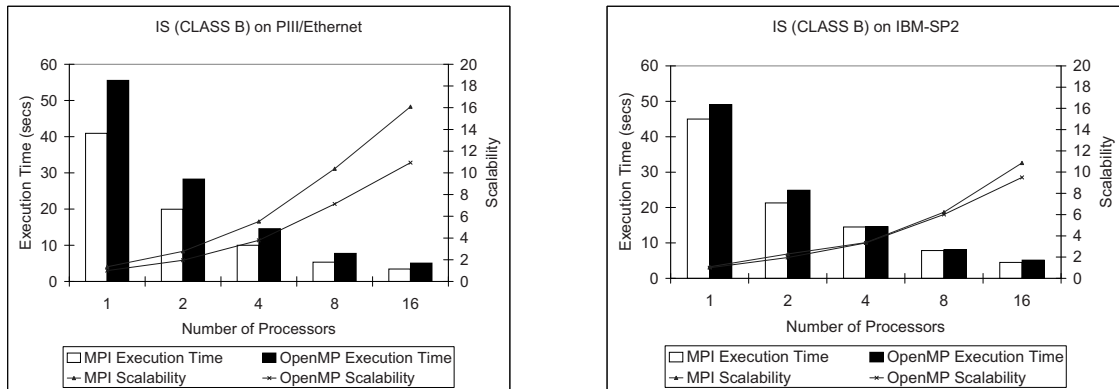


Fig. 4.17. Performance of IS (CLASS B).

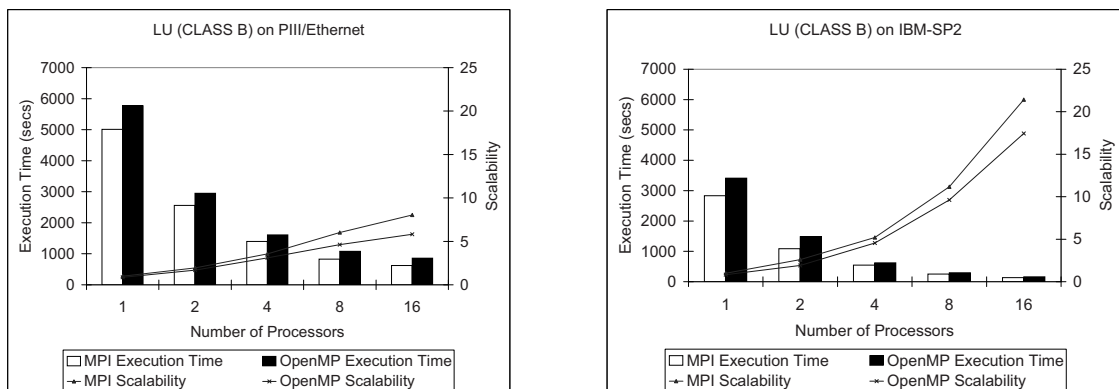


Fig. 4.18. Performance of LU (CLASS B).

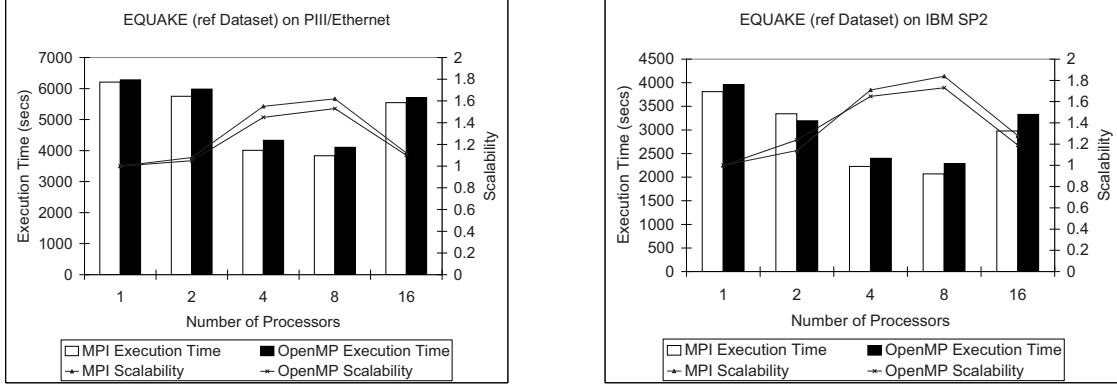


Fig. 4.19. Performance of EQUAKE (REF Dataset).

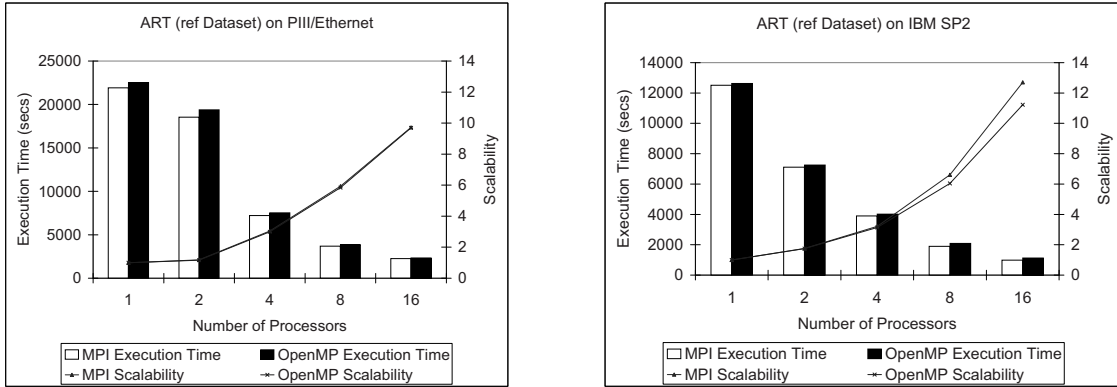


Fig. 4.20. Performance of ART (REF Dataset).

one processor from the serial and MPI to the translated OpenMP version on one processor for these benchmarks.

Figures 4.21 and 4.22 summarizes the speedups of the OpenMP and the MPI versions on one, two, four, eight and sixteen nodes on the Linux cluster and the IBM SP2 respectively, calculated by comparing with their respective serial versions as  $Speedup_{MPI-N-processes} = \frac{t_{MPI-one-process}}{t_{MPI-N-process}}$  and  $Speedup_{OpenMP-N-processes} = \frac{t_{OpenMP-one-process}}{t_{OpenMP-N-process}}$ .

The average speedups of our translated OpenMP versions for the seven benchmarks is about 10% less than the average speedups of their hand-coded MPI counterparts. We now individually discuss the performance of each benchmark.

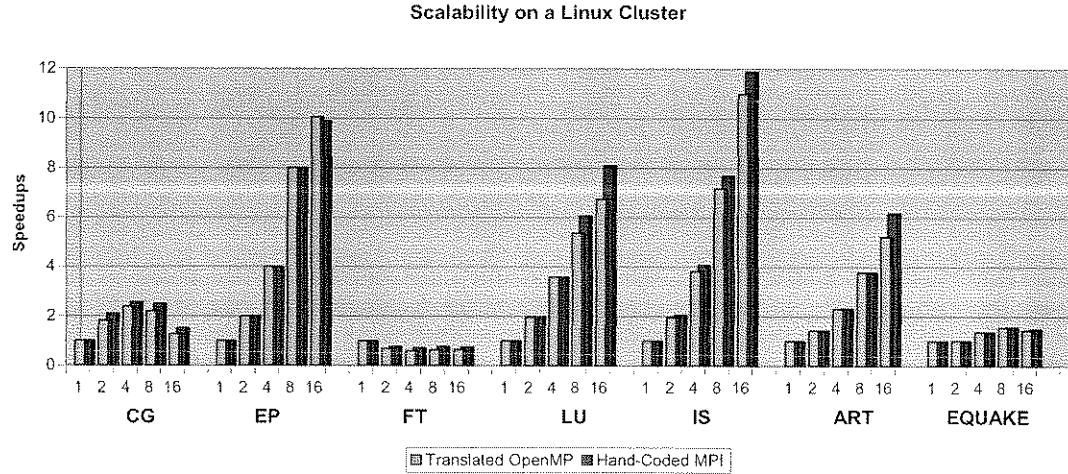


Fig. 4.21. Scalability comparison on the Linux Cluster. The input data set used is CLASS A for FT, CLASS B for the other NAS benchmarks and *train* for the SPECOMPM2001 benchmarks.

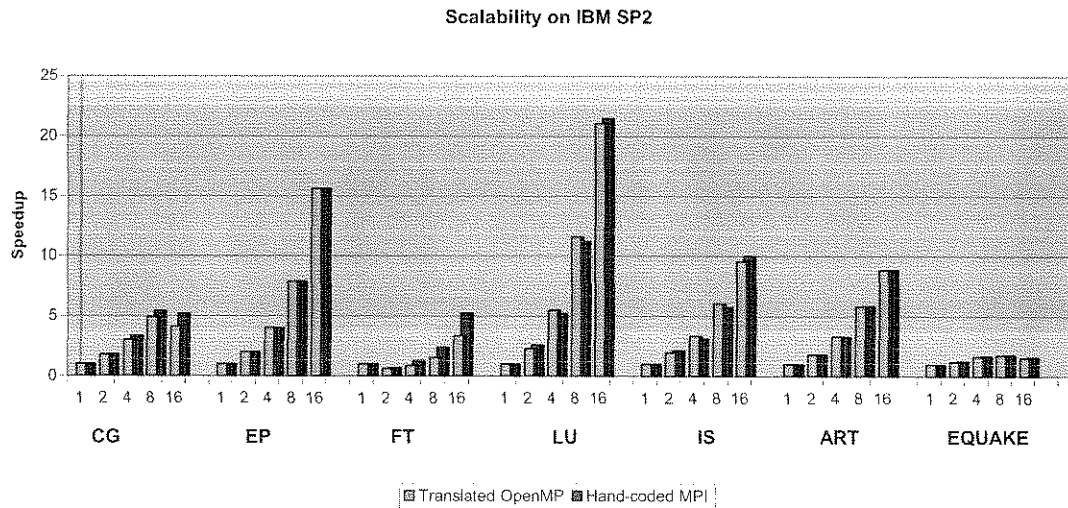


Fig. 4.22. Scalability comparison on the IBM SP2 nodes. The input data set used is CLASS B for all the NAS benchmarks and *train* for the SPECOMPM2001 benchmarks.

The hand-tuned MPI version of CG benefits from the programmer's knowledge of exact transpose, reduction and data exchange patterns in the algorithm. In our translation of the OpenMP version, the message set computation in Figure 4.7 rec-

ognizes the access affinity in the regular accesses for several arrays in the *conj\_grad* subroutine. However, there are irregular reads to two arrays. Analysis of the indirection array provides very loose bounds for these irregular reads. For writes to these arrays, each process has to broadcast its writes to all other processes. In spite of this difference, our version achieves speedups close to those achieved by the hand-coded MPI version.

In case of EP, the MPI version uses several large arrays, which were eliminated while creating the serial version (discussed in [4]). Since the OpenMP version is derived from this serial version, our translated version actually performs marginally better than hand-tuned MPI version. EP also benefited from the recognition of an array reduction.

For FT, the hand-tuned MPI version employs a sophisticated strategy to map the problem into specific network topologies, depending on the problem set size and the number of processes. Furthermore, the resulting transpose communication patterns are also hand-optimized. Owing to these differences, the translated OpenMP version is about 15% slower than its hand-coded MPI counterpart.

LU benefited greatly from the computation repartitioning optimization discussed in Section 4.3. The optimized nearest neighbour communication strategy in the hand-tuned MPI version still enables it to achieve a marginally better performance on the Linux cluster (about 9%). The two versions achieve similar performance on the IBM SP2.

IS is a NAS benchmark that performs integer sorting. The hand-tuned MPI version of IS uses a *bucket-sort* algorithm with 1024 buckets (which has been found to be the optimal number of buckets for this data set). The OpenMP version does not use bucket-sort, in effect performing the ranking using only one *bucket*. This is an example of an application where the hand-coded MPI version actually makes use of a different algorithm to obtain performance. However, the OpenMP version still achieves speedups to within 8% of the MPI version on both platforms and actually does slightly better than the MPI version on four and eight process runs on the IBM

SP2. The translation of IS benefited from the recognition of reduction patterns that we discussed in Section 4.3.

For EQUAKE and ART, the translated versions do not differ significantly from the hand-crafted versions and thus achieve similar performance.

#### 4.4.3 Comparison with HPF

In Section 2.2, we initiated a discussion of HPF. HPF was an important contribution towards a higher level of abstraction in programming distributed-memory machines. In this section, we compare the performance obtained by HPF applications with the performance obtained by corresponding MPI versions and OpenMP versions translated to MPI using the techniques presented in this chapter.

HPF versions of some the NAS benchmarks are available as part of the NPB3.0-HPF suite. This suite was developed by the developers of the original NAS benchmarks. A detailed discussion of these HPF versions was presented in a report from the developers [29]. IS and EP are not included in this suite of HPF benchmarks. Therefore, we confined our comparison study to the CG, LU and FT benchmarks. We compiled the HPF benchmarks using the *pghpf* compiler [26]. The compiler version was 5.2.4 and the benchmarks were compiled at optimization level -O3. The *pghpf* compiler we used fails to compile the HPF version of FT in NPB3.0-HPF. Therefore, our study is further restricted to CG and LU.

Figure 4.23 shows the execution times for the MPI, OpenMP translated to MPI and HPF versions of CG(CLASS B) and LU(CLASS A) on one, two, four, eight and sixteen nodes of a linux cluster. The hardware configuration is the same as that used for the results presented in the previous section. For LU, we could not use the CLASS B dataset since the HPF version runs out of memory for this dataset.

In CG, the OpenMP version translated to MPI performs about 15% better than the HPF versions on the average. For LU, the HPF version is ten times slower than the OpenMP version on the average. For CG, the HPF version has a similar

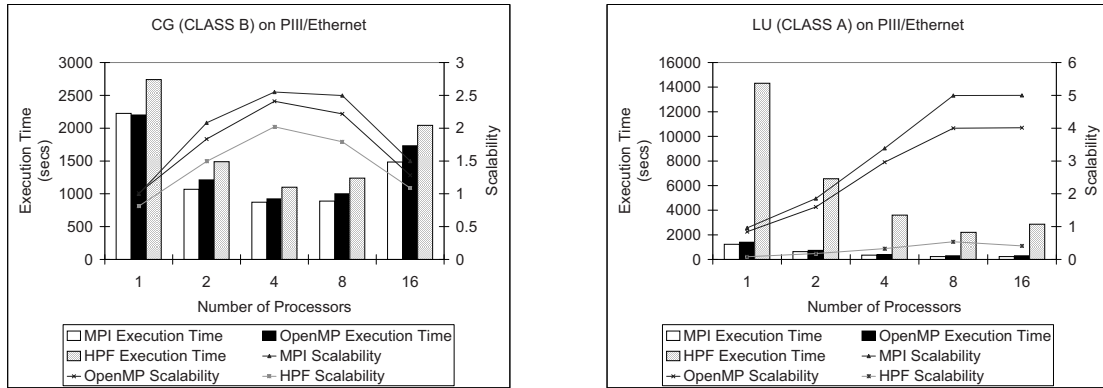


Fig. 4.23. Comparison of the Performance of MPI, OpenMP and HPF versions of CG (CLASS B) and LU (CLASS A) on a Linux Cluster.

structure. The most time consuming part for both of these is the sparse matrix-vector multiplication step  $q = A.p$ . In HPF, the sparse matrix  $A$  is divided blockwise between processors. The vector  $q$  is also divided blockwise between processors but the vector  $p$  is replicated on all processors to avoid communication during the sparse matrix-vector product (SMVP) step. However, the vector  $q$  becomes  $p$  in the next iteration and thus, there is an array copying and communication step in every iteration in copying  $q$  to  $p$ . In case of the OpenMP version, each processor needs to communicate its updates to  $q$  to every other processor as well. However, an array copying step is not required because of replication and this gives a performance advantage.

In case of LU, the HPF version adopts a slightly different algorithm for subroutines *blts* and *buts* which solve a lower and upper triangular system of matrices respectively. The MPI and OpenMP versions implement a pipelined-parallelization scheme. However, since such a scheme cannot be efficiently implemented in a data-parallel paradigm, the HPF version employs a Hyperplane Algorithm [71, 72]. However, it still suffers from load imbalance. Furthermore, it also has the overhead of array copying for several of the main arrays. Thus, overall, it is about ten times slower than the corresponding OpenMP version translated to MPI.

In investigating the performance bottlenecks for HPF programs, we identified certain aspects in which our OpenMP to MPI translation scheme has an advantage over



HPF compilation schemes that follow the *owner-computes* rule. We now enumerate some of these.

- *Slowdown of Serial Regions* - Owing to the physical distribution of arrays, computation in the serial region may need to be done on different processors when the owner-computes rule is followed. In this case, synchronization overhead is often introduced as well. In our translation scheme, serial regions are redundantly executed on all processors. That may require processors to inter-communicate data produced in previous parallel regions. However, in practice, we find that this introduces limited overheads in our case.
- *Complexity in Scheduling Loop Iterations* - To schedule iterations in parallel loops in HPF, HPF compilers usually try to identify a *home array* for a loop nest. The owner-computes rule is applied to schedule iterations according to the distribution of the home array. Often, if there are multiple arrays accessed in a loop, each with different subscript expressions, or subscript expressions in the left-hand sides (LHS) of assignment expressions are complicated, then HPF compilers often fail to identify home arrays and serialize such loops. An example of this is the shuffling operation in the projection and interpolation subroutines in the NAS benchmark MG [29]. These have operations of the form
 
$$u(2*i1-1, 2*i2-1, 2*i3-1) = u(2*i1-1, 2*i2-1, 2*i3-1) + z(i1, i2, i3)$$

$$u(2*i1, 2*i2-1, 2*i3-1) = u(2*i1, 2*i2-1, 2*i3-1) + z(i1+1, i2, i3) + z(i1, i2, i3)$$
 and the compiler is unable to schedule these loops as parallel loops because of the complicated LHS of these array expressions. In case of the NAS benchmark, the authors thus have to specify the home-arrays for these loops. Furthermore, the analysis complexity is compounded in case of HPF compilers when the LHS and RHS of array expressions both have complex subscripts. As an example, consider an expression of the form  $A[f(i)] = B[g(i)] + c$ . The scheduling function  $S(i) : i \rightarrow Processor$ , will now be a function of both  $f(i)$  and the distribution of  $A$ . Then, to calculate which elements of array  $B$  will be communicated on

which processor, the compiler will have to take into account both functions  $g(i)$  and  $S(i)$  as well as the distribution of  $B$ . Even if  $A$  and  $B$  have simple block distributions, the relevant compiler analysis is complicated and the compiler will often fail to schedule such loops as parallel. In our case, the scheduling function is always simple block scheduling. So, our regular section analysis for writes to  $A$  needs to consider only the function  $f(i)$  and the regular section analysis for array  $B$  needs to consider only the function  $g(i)$ . Thus, in our case, the analysis may still be tractable while in case of HPF it may not be so.

- *Maintaining Replicated Copies of Non-Distributed Data* - In HPF, arrays that are not distributed using any HPF distribution or alignment directive and scalars that are used outside parallel regions are replicated on all processors. These replicated data items need to be kept consistent on all processors at all times during the program. This introduces overheads whenever these arrays are copied and communicated between processors. In our case, the shared variable analysis pass identifies the set of shared variables and the producer-consumer analysis with the incorporation of OpenMP's weak memory consistency semantics (described in section 4.2) reduces unnecessary communication for this shared data.

There is currently no suitable HPF source-to-source compiler available to us which would allow us to explicitly examine the code generated by a HPF compiler. However, performance measurements with the NAS HPF benchmarks compiled with pghpf indicate that these are the most important sources of overheads introduced by a typical HPF compiler.

#### 4.4.4 Comparison with OpenMP deployed using Software Distributed Shared Memory

As discussed in Section 5.1, recent schemes proposed for the deployment of OpenMP utilize an underlying layer of Software Distributed Shared Memory (SDSM). In [47,73],

we have evaluated the performance of OpenMP applications deployed on clusters using the TreadMarks [15] SDSM system, using the techniques and optimizations proposed in chapter 3. In this section, we compare the performance of the OpenMP versions (translated to MPI using our proposed techniques) of our benchmarks with the performance of the OpenMP versions deployed using the TreadMarks SDSM system. For this set of experiments, we have used the sixteen node Linux cluster. Figure 4.24 shows the performance of the two schemes.

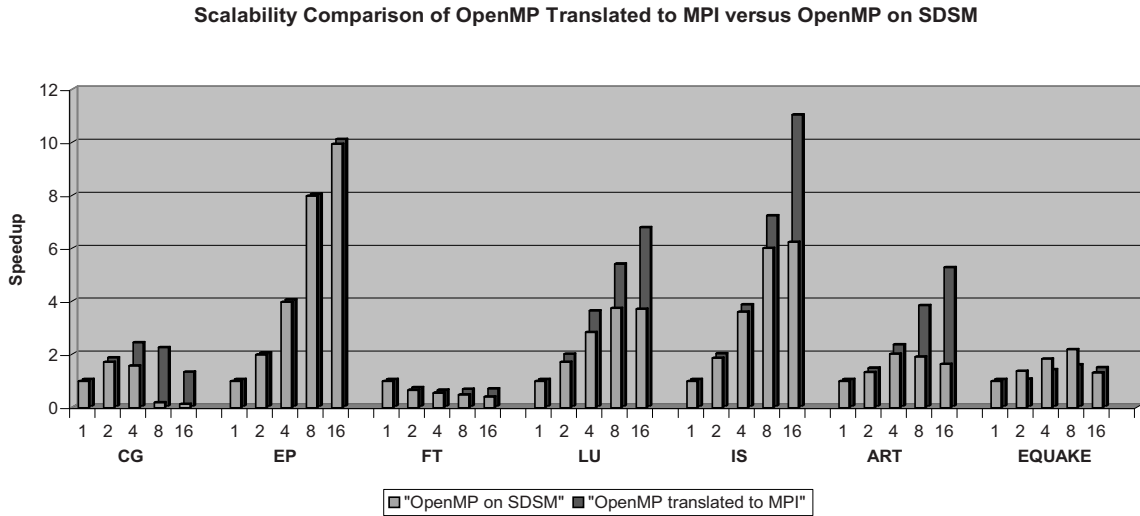


Fig. 4.24. Performance Comparison of the OpenMP versions translated to MPI with the OpenMP versions deployed on the TreadMarks SDSM system.

Apart from EQUAKE, the versions translated to MPI always perform better compared to the corresponding SDSM versions. EQUAKE has two large arrays that are accessed along different subscripts in the original partitioning scheme. Furthermore, the sparse matrix vector product expands the shared data size by allocation work-arrays whose size is proportional to the number of threads. An MPI implementation would need to heavily restructure the existing application to achieve performance. In the SDSM version of EQUAKE, an additional optimization was that iterations for the main loop in the *smvp* (sparse-matrix-vector-product) subroutine were dynamically

scheduled, which serialized most accesses and eliminated communication. But such a scheme would not be very scalable either.

For the other benchmarks, the MPI versions profit from the implicit aggregation of messages and the piggy-backing of synchronization with message-exchange. Both these issues were investigated in [51] as possible advantages of message-passing systems. An important shortcoming of the SDSM versions not evident from the speedup comparison is the constraint on the size of shared data that an SDSM system like TreadMarks can allocate. TreadMarks creates a *twin* for every page of shared data. At runtime, writes to a page of shared data is tracked by comparing the current pages with its original *twin*. Furthermore, memory also needs to be allocated for storing the list of writers to each page of shared data in a program interval. As a result, the maximum size of shared data that can be allocated is constrained to be a fraction of the total available address space. Due to this, one cannot use large datasets such as the *ref* datasets of ART and EQUAKE for the SDSM versions of these benchmarks and for this experiment, we had to use *train* data sets for these applications. By contrast, our translation to MPI does not introduce such size constraints, despite the replication of shared data on all nodes, and we can run the largest problem sizes allowed by the architecture.

In this chapter, we discussed compile-time techniques for translating OpenMP applications directly to MPI. To evaluate our techniques, we compared the performance of the translated versions of seven OpenMP benchmarks with their hand-coded MPI counterparts. We found that in almost all the benchmarks, the performance achieved is close to that achieved by hand-coded MPI. However, two applications that showed limited scalability are CG and EQUAKE. This performance limitation is largely owing to the fact that these applications have irregular array accesses, which must be treated conservatively by compiler analyses. In the next chapter, we present optimization techniques that improve performance in the case of irregular OpenMP applications.

## 5. OPTIMIZING IRREGULAR SHARED-MEMORY APPLICATIONS FOR DISTRIBUTED-MEMORY SYSTEMS

### 5.1 Introduction

In Chapter 4, we have introduced techniques to extend the ease of OpenMP shared-memory programming to distributed memory systems by automatically translating *standard* OpenMP programs directly to MPI programs. For applications that contain regular data access patterns, our scheme has achieved performance close to that of hand-tuned MPI applications.

Irregular applications, where the actual pattern of data accesses are not analyzable at compile-time, are often difficult to parallelize. Converting irregular, serial applications directly to parallel message passing form poses an even greater challenge. Along with parallelization, programmers must perform data-to-process mapping and judiciously insert messages to satisfy remote accesses, even though the actual access patterns are not known at compile-time. This chapter aims at making easier such programming efforts by introducing compiler and runtime techniques that directly translate irregular OpenMP applications to MPI.

Specifically, this chapter presents two approaches for translating irregular OpenMP applications – OpenMP programs that contain irregular accesses – to MPI. The first technique relies on deducing certain properties of irregular accesses at compile-time. The second is a more general combined compile-time/runtime technique, that is applicable even if the irregular accesses do not exhibit such properties. One observation is that irregular applications are often iterative, whereby the same access pattern (known only at run-time), is repeated over several iterations. The key idea of our compile-time/runtime approach is to overlap computation and communication by re-

structuring the iteration spaces of irregular loops. This is done in a way that leads to computation on locally available data first, while simultaneously communicating remote data.

The second technique makes use of runtime analysis, or *inspection*. A key insight is that inspecting irregular accesses not only resolves them accurately but also can analyze the precise mapping of accesses to iterations. On each process, this information allows us to partition irregular loop iterations based on the location of the shared data accessed. Computation on local data partitions will then be performed first, overlapped by communication of remote partitions. Next, computation on the closest remote partition will be performed, and so on. When irregular accesses occur within nested loops, the loop may need to be restructured to derive the maximum possible overlap of computation and communication. This chapter also describes such restructuring techniques.

Similar issues were considered by proposals to extend High Performance Fortran [23] to handle irregular applications [28]. However, there are several key differences. At a high level, HPF considered the use of data distribution directives, whereas such information is not given explicitly in OpenMP programs. Also, most HPF execution models followed an *owner computes* rule, necessitating inspection for correct execution when the irregular access occurs on the left-hand side of expressions. In our case, inspection is done as an optimization. Our scheme does not have *owners* of shared data. Furthermore, our partial replication scheme allows shared data to be forwarded conservatively in our baseline translation scheme, without allocating extra storage. It also means that the runtime analysis in our case must resolve accesses in terms of producer-consumer relationships derived by the OpenMP-to-MPI compiler whereas in HPF, inspectors must resolve irregular accesses in the context of the specified data distribution. Perhaps most importantly, previous approaches to optimizing irregular accesses on distributed memory machines have not considered loop restructuring for achieving computation-communication overlap.

Restructuring programs to identify potential overlap of computation with communication and inserting non-blocking message passing calls does not yet guarantee improved performance. Current message passing libraries may not ensure that the overlap actually happens. To provide this guarantee, we have implemented a runtime scheme with an explicit communication thread. In our implementation, this thread executes on a separate processor on an architecture with dual-cpu nodes.

The proposed new optimization techniques led to improved performance in *all* our irregular programs where the baseline techniques had limited yield. These baseline techniques came close to the performance achieved by hand-tuned MPI programs, in most of our benchmarks discussed in Chapter 4. In Section 5.6 we will evaluate all three irregular applications where this was not the case. They include the SPEC OMPM2001 program Equake, the NAS benchmark CG, and the molecular dynamics application MOLDYN.

This chapter makes the following contributions:

- We present compiler techniques for optimizing the performance of irregular OpenMP programs translated to MPI, based on runtime analysis and iteration reordering for overlapping computation with communication.
- We present runtime techniques for ensuring true computation-communication overlap. They are based on separate threads for computation and communication.
- We evaluate three representative irregular OpenMP programs from two important classes of irregular applications – sparse matrix computations and molecular dynamics. We compare the performance of our new translation scheme with corresponding hand-tuned MPI program versions, with programs versions translated from OpenMP using the baseline OpenMP-to-MPI translation and with versions that perform runtime analysis without reordering iterations.

The rest of the chapter is organized as follows. Section 5.3 discusses a compile time approach to translate irregular OpenMP applications to MPI. Section 5.4

presents techniques for inspection and iteration reordering of irregular applications to accomplish computation-communication overlap. Section 5.5 presents techniques for creating separate threads for computation and communication, ensuring true overlap. Section 5.6 evaluates the performance of these techniques. Section 5.7 places this chapter in the context of related work. Section 5.8 concludes the chapter.

## 5.2 Need for Optimization in the Baseline OpenMP to MPI Translation Scheme for Irregular OpenMP Applications

Chapter 4 presented the basic translation scheme for translating regular OpenMP applications to MPI. The objective of this translation scheme is to perform a source-to-source translation of a shared-memory OpenMP program to an MPI program. The OpenMP-to-MPI translator interprets OpenMP directives to (1) partition iterations of parallel loops between processes using block scheduling, and (2) identifies all the shared data in the program [47]. It then builds an OpenMP Producer-Consumer Flow Graph and summarizes array accesses using Regular Section Descriptors (RSD) [74] of the form *jprocess-rank,read/write,array,dimension,lowerbound,upperbound,...j*. The translator then traverses this producer-consumer flow graph, starting from each point where shared data is produced and derives the requisite communication sets for communicating shared data to potential consumers. These translation steps have been incorporated into the Cetus [55] compiler infrastructure to build an OpenMP-to-MPI compiler. The idea of communicating data sections from producers to future consumers, rather than maintaining fixed data distributions, has also been pursued in the context of non-uniform shared-memory machines [75].

Irregular applications pose a challenge, because the compiler cannot accurately analyze the accesses at compile time. Instead, it must conservatively over-estimate data consumption. Consider the following code.

```
L1 : #pragma omp parallel for
      for(i=0;i<10;i++)
```



```
A[i] = ...
```

```
L2 : #pragma omp parallel for
    for(j=0;j<20;j++)
        B[j] = A[C[j]] + ...
```

Considering parallel execution on 2 processes (numbered 0 and 1), the compiler summarizes the writes in Loop L1 using an RSD of the form  $\langle p, write, A, 1, 5*p, 5*p+5 \rangle$ . For L2, the compiler produces the two RSDs  $\langle p, write, B, 1, 10*p, 10*p+5 \rangle$  and  $\langle p, read, A, 1, undefined, undefined \rangle$ . In L2, array A is accessed using the indirection array C and thus, the accesses to A cannot be resolved at compile time. In such cases, the baseline translation scheme presented in Chapter 4 will treat the access to array A in loop L2 as an access to the entire array. Thus, at the end of loop L1, each process will communicate the part of array A it has produced to all other processes. This may result in expensive redundant communication.

In the next two sections, we discuss techniques to refine the compiler's analysis of the part of array A accessed in loop L2. The first technique attempts to deduce certain properties of the indirection array C. This information serves to obtain bounds on the region of array A accessed by each process. The second technique deduces the actual access pattern at runtime using inspection and performs loop restructuring based on the inspected access pattern.

### 5.3 Compile-Time Analysis of Irregular Accesses

To refine the analysis of such irregular accesses at compile-time, we make use of the property of *monotonicity*. Monotonicity is a property which states  $\forall i, j, i \leq j \Leftrightarrow A[i] \leq A[j]$ . Testing for monotonicity in irregular accesses have been discussed in the context of parallelizing compilers [76, 77] for analyzing the monotonicity properties of index arrays. We have manually applied these techniques to our benchmarks. We

now separately consider the two forms of irregular accesses, namely irregular reads and irregular writes.

### 5.3.1 Irregular Reads

Irregular reads refer to the case where, for a future read, it is not possible at compile-time to exactly determine which process reads which element of an array. In the baseline case, for an array that is irregularly read in the future, all processes can conservatively communicate their writes. This can be done efficiently in our translation scheme since the entire shared data space is allocated on all processes and thus, there already exist place holders on potential consumers to which producers can conservatively forward data. This is different from HPF, where temporary storage has to be allocated for shared data and this temporary allocation and deallocation may introduce considerable overheads.

However, conservatively forwarding all writes may result in a lot of redundant communication. So, as a compile-time optimization, the compiler checks if the future irregular read has a monotonicity property and accordingly refines the accesses. Consider a one-dimensional array A. In a loop, process  $p$  writes elements  $A[l_p]$  to  $A[u_p]$ . In a future irregular read, process  $q$  reads from array A using an indirection array B. Thus, accesses to A are of the form  $A[B[i]]$  where  $i$  is the loop index. Say process  $q$  executes iterations  $l_q$  through  $u_q$  of that loop. Then, we try to trace the evolution of array B and check if it has the monotonicity property. If B can be proved to be monotonic, what  $p$  should send to  $q$  are the elements of A with subscripts in the range  $[l_p, u_p] \cap [B[l_q], B[u_q]]$ . The sends and receives are now formulated accordingly.

### 5.3.2 Irregular Writes

Irregular writes refer to the case where, for the current write, it is not possible to exactly determine which process writes which element of an array. In such a case it may actually be incorrect to indiscriminately communicate a range of elements.

As an example, consider a parallel loop executed by two processes  $p_1$  and  $p_2$ , in which they write to a shared array  $S$ .  $p_1$  writes the elements  $S[1], S[3]$  to  $S[6]$  and  $S[12]$ .  $p_2$  writes the elements  $S[2]$  and  $S[7]$  to  $S[11]$ . In a future read, both processes access the entire array and processes must thus intercommunicate all their writes. Then, if  $p_1$  sends  $p_2$  the entire contiguous sets of elements from  $S[1]$  to  $S[12]$ ,  $p_2$  will lose the elements it has written. It may be possible to use the monotonicity property to refine the analysis of irregular writes. As an example, consider an array  $A$ , written using an indirection array  $B$ . Process  $p$  executes iterations  $l_p$  through  $u_p$  and process  $q$  executes iterations  $l_q$  through  $u_q$  for the loop. Since parallel loops are statically scheduled with block scheduling, the two iteration spaces cover disjoint intervals, that is,  $[l_p, u_p] \cap [l_q, u_q] = \phi$ . If the indirection array  $B$  can be proved to be monotonic, then we can say that  $p$  and  $q$  write to the disjoint intervals  $[B[l_p], B[u_p]]$  and  $[B[l_q], B[u_q]]$ . We can now create the RSD for the write to array  $A$  using these bounds. An additional point is that the parallel region may contain shared data allocation that occurs on some processes and not on others. In this case, if writes to these locations need to be communicated, the allocation is replicated on all processes.

Our compiler first tries to prove the monotonicity property for indirection arrays(or functions) that are used in the irregular write. If that fails, either because the indirection function is not provably monotonic or because the irregular write is not in the form of `ARRAY[indirection function]` form, we can use the runtime mechanism described in the next section. For the applications that we have encountered, all instances of irregular writes have been analyzable using the monotonicity property.

#### 5.4 Inspection and Loop Restructuring for Computation-Communication Overlap

In Section 5.2, we examined a scenario where run-time inspection may help reduce unnecessary communication in translating irregular OpenMP applications to MPI. Continuing with the example code of Section 5.2, we see how iteration reordering may

allow computation-communication overlap. For the code shown, block scheduling for loop L2 results in processor 0 executing the first 10 iterations, with  $j = 0, 1, 2, \dots, 9$ . If the array C is  $C=2,3,5,9,7,1,4,2,3,5,9,8,2,0,7,6,3,4,5,1$ , then runtime inspection reveals that process 0 needs to receive  $A[5], A[9]$  and  $A[7]$  from processor 1 (and not all 5 elements  $A[5]$  through  $A[9]$ ). Now, if process 0 reorders its iterations and the new iteration order for loop L2 on process 0 is  $j=0,1,5,6,7,8,2,3,4,9$  instead of  $j=0,1..9$ , then, in the first six iterations, process 0 will access locally available elements. This computation can overlap in time with the receipt of  $A[5], A[9]$  and  $A[7]$  from process 1. Here, inspection helps to reduce redundant communication and also reveals which iterations access which data elements. Using this information, iteration reordering can be done to overlap communication of remotely produced data with computation that accesses locally available data. Now, consider what happens on process 1. Process 1 performs iterations  $j=10,11,\dots,19$  of loop L2. Thus, it needs to receive all five elements  $A[0]$  through  $A[4]$  from process 0. Thus, inspection does not reduce the amount of data process 0 must send to process 1. However, inspection does reveal that if process 1 reorders its iterations of loop L2 to be  $j=10,11,14,15,18,12,13,16,17,19$  then it can execute the first five iterations with locally available data, while it receives the elements  $A[0]$  through  $A[4]$  from process 0. This is an example where inspection does not reduce communication but it still enables simple iteration re-ordering that exposes available opportunity for computation-communication overlap.

The above iteration re-ordering scheme may not yet expose the maximum available opportunity for computation-communication overlap. Consider a common sparse matrix-vector product shown in Figure 5.1, taken from the NAS Conjugate Gradient benchmark. The irregular access here occurs in statement S2 inside the nested loop L2. Each outer  $j$  iteration in loop L2 now contains multiple irregular accesses to the vector  $p$ , which is produced blockwise in loop L1. Therefore, simply reordering the outer  $j$  iterations may not suffice to partition accesses into accesses of local and remote data.

```

L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
        for(k=rowstr[j];k<rowstr[j+1];k++)
S2:      w[j] = w[j] + a[k]*p[col[k]] ;
    }

```

Fig. 5.1. Sparse Matrix-Vector Multiplication Kernel

```

L1 : #pragma omp parallel for
    for(i=0;i<N;i++)
        p[i] = x[i] + alpha*r[i] ;

L2-1 : #pragma omp parallel for
    for(j=0;j<N;j++) {
        w[j] = 0 ;
    }

L2-2: #pragma omp parallel for
    for(j=0;j<N;j++) {
        for(k=rowstr[j];k<rowstr[j+1];k++)
S2:      w[j] = w[j] + a[k]*p[col[k]] ;
    }

```

Fig. 5.2. Sparse Matrix-Vector Multiplication Kernel with Loop Distribution of loop L2

```

L3: for(i=0;i<num_iter;i++)
    w[T[i].j] = w[T[i].j] +
        a[T[i].k]*p[T[i].col] ;

```

Fig. 5.3. Restructuring of Sparse Matrix-Vector Multiplication Loop

To expose the maximum available opportunity for computation-communication overlap, the loop L2 in Figure 5.1 is restructured to the form shown in Figure 5.2. Loop L2 is distributed into the loop L2-1 and the perfectly nested loop L2-2. On each process, at run-time, inspection is done for statement S2 and in every execution  $i$  of statement S2, the loop indices  $j$  and  $k$  as well as the corresponding value of  $col[k]$  are recorded in an inspection structure  $T$ .  $T$  here can be considered a mapping function  $T : [j, k, col[k]] \rightarrow [i]$ . This mapping can then be used to transform loop L2-2 in Figure 5.2 to loop L3 in Figure 5.3. Iterations of loop L3 can now be re-ordered to achieve maximum overlap of computation and communication in our sparse matrix-vector multiplication example.

The OpenMP-to-MPI compiler, in coordination with run-time inspection must affect three transformations of the parallel loops that contain irregular accesses to achieve the described computation-communication overlap. The goal of these transformations is to “conceptually coalesce” the loop nest, followed by the techniques described in the previous paragraph. The three transformations are –

- (1) Conversion of loop nests containing irregular accesses to perfectly nested loops.
- (2) Inspection of irregular accesses, restructuring of n-dimensional loop nests to one-dimensional loops and reordering of a one-dimensional loop on the basis of inspected data accesses.
- (3) Scheduling of overlapping communication with the reordered computation.

#### **(I) Conversion of Loop Nests with Irregular Accesses to Perfectly Nested Form**

**Form** This step is a compile-time transformation. The goal of this step is twofold -

- (1) convert the irregular parallel loop into multiple loops such that in each loop nest, there is either only a single statement that contains irregular accesses or if there are multiple statements containing irregular accesses, they are at the same loop depth; and
- (2) nested loops containing irregular accesses are perfectly nested up to the depth

**Algorithm *create\_inspector\_executor***

**Input :** 1. A perfectly nested loop L containing irregular accesses of shared arrays.

**Output :** 1. An inspector version L' of loop L.

2. An executor version L'' of loop L.

3. Linked list of runtime inspection results for each array  $a$

accessed in loop L. Each inspection structure instance has the form

$T_a : \{i_1, \dots, i_n : \text{the index vector of the loop nest L,}$

$x_1, \dots, x_k : \text{values of irregular subscripts for accesses to array } a.\}$

**Start *create\_inspector\_executor***

1. Set iteration space of L' identical to iteration space of L.

2. Set loop body of L' identical to loop body of L.

3. *do* for each array  $a$  accessed irregularly in L

4. In the loop body of L', add statements to

5. (a) Allocate a new instance of  $T_a$

6. (b) Assign  $T_a.i_1 \leftarrow i_1, \dots, T_a.i_n \leftarrow i_n$

7. (c) Increment  $length(T_a)$

8. (d)  $\forall idx_k, idx_k$  is an irregular subscript of the array  $a$ , Assign  $T_a.x_k \leftarrow idx_k$

9. (e) Call the runtime library for iteration reordering (algorithm in Figure 5.5)

10. *end do*

11. Create loop L'' with single-dimension iteration space  $[1, length(T_a)]$

12. Set loop body of L'' identical to loop body of L.

13. In the loop body of L'', substitute

14. (a) index variables  $i_1, \dots, i_I$  with  $T_a.i_1, \dots, T_a.i_I$ , and

15. (b) array subscripts  $x_1, \dots, x_j$  with  $T_a.x_1, \dots, T_a.x_j$ .

**End *create\_inspector\_executor***

Fig. 5.4. Compile-time Algorithm to create inspector and executor versions of a loop containing irregular accesses

**Algorithm *restructure\_and\_sort***

**Input :** 1. Linked list of runtime inspection results  $T_a$  for each array  $a$  irregularly accessed in the loop L.  
 2.  $N$  - the total number of processes. 3.  $pid$  - the local process rank.  
 4.  $P_{a1}, \dots, P_{aN}$  - the regular section descriptors of reaching definitions for each array  $a$  irregularly accessed in the loop L. (See Section 5.2)

**Effect :** 1. Creation of Computation blocks  $S_1, \dots, S_N$  (each containing a chunk of executor loop iterations)  
 2. Creation of Communication blocks  $C_1, \dots, C_{N-1}$

**Start *restructure\_and\_sort***

1. Allocate sets  $prod_1, \dots, prod_{length(T_a)}$  and initialize them to  $\phi$
2.  $\forall i \in [1, N]$  initialize  $S_i = \phi, C_i = \phi$
3. *do*  $\forall$  iteration  $i$  of inspector loop L'
4.     *do*  $\forall$  array  $a$  irregularly accessed in iteration  $i$
5.         *do*  $\forall$  irregular subscripts  $T_a.x_{t_i}$  of access to  $a$
6.             *if*  $T_a.x_{t_i} \in P_{aK}$  *then*
7.                 Set  $C_K = C_K \cup a[T_a.x_{t_i}]$ ,  $prod_i = prod_i \cup K$
8.             *endif*
9.         *end do*
10.     *end do*
11. *end do*
12. Sort iterations  $j$  of L" lexicographically on tuples  $\langle prod_i, T_a.i_{1j}, \dots, T_a.i_{Ij} \rangle$ .
13. Partition the sorted iteration space  $I'$  of executor loop L" into the computation sets  $S_1, \dots, S_N$  such that  
 $\forall$  iteration  $i, i \in S_k \Rightarrow prod_i \subseteq \{pid, \dots, (pid + k - 1) \text{ modulo } N\} \wedge k \in prod_i$

**End *restructure\_and\_sort***

Fig. 5.5. Runtime Algorithm to reorder executor loop iterations and produce computation and communication blocks



**Algorithm *schedule\_comp\_comm*****Input :**

1.  $N$  - the total number of processes
2. Computation blocks  $S_1, \dots, S_N$
3. Communication blocks  $C_1, \dots, C_{N-1}$
4.  $myid$  - the local process rank

**Effect :**

1. Scheduling of computation and communication

**Start *schedule\_comp\_comm***

1. begin non-blocking receives of Communication blocks  $C_1, \dots, C_{N-1}$   
from processes  $(myid + 1) \bmod N, \dots, (myid + N - 1) \bmod N$ .
2. *for*  $j = 1$  to  $N - 1$
3.     schedule computation block  $S_j$
4.     complete non-blocking receive of communication block  $C_j$
5. *end for*
6. schedule computation block  $S_N$

**End *schedule\_comp\_comm***

Fig. 5.6. Runtime Algorithm to Schedule Overlapping Computation and Communication

where the irregular access occurs. Several techniques are available for converting imperfectly nested loops to perfectly nested loops [78]. We found that array privatization [79] followed by loop distribution [80] is sufficient to achieve the above two conditions for all parallel irregular loops that we encountered. In our example, this step would convert loop L2 in Figure 5.1 to loops L2-1 and L2-2 in Figure 5.2. Additionally, at this step, the compiler verifies the legality of restructuring the loop and re-ordering the loop iterations. Restructuring and reordering are legal only if there are no loop-carried dependencies in the nested loop. For the irregular applications we encountered, recognition of reductions followed by simple dependence tests was sufficient to prove the absence of any loop-carried dependencies.

**(II) Inspection, Loop Re-structuring and Iteration Reordering** The second step is a combined compile-time/run-time transformation. The input to this step is a set of perfectly nested loops that contain the irregular accesses. From these loops, at compile-time, the compiler generates inspector and executor loops using the algorithm presented in Figure 5.4. To create the inspector, the compiler inserts code in the original loop to record the values of loop indices and irregular subscripts. To create the executor loop, the compiler first coalesces the original nested loops to one-dimensional loops. Note that by one-dimensional loop, we imply that this transformation is aimed only to lower the nest dimension to the extent required to put the irregular access in the outermost loop. (There may be a  $k$ -deep loop at a further depth which does not contain irregular accesses. For our purpose, we can consider this pattern as a single statement rather than a loop nest). The inspector uses an inspection structure  $T$  to record the values of the array subscripts for irregular accesses as well as the values of the loop indices. Recording array subscripts is essential in mapping iterations to array accesses. Recording values of loop indices is essential for coalescing the nested loop to a one dimensional loop, especially if the loop bounds are not affine expressions of the surrounding loops.

At runtime, the inspector loop creates the inspection results  $T$  for each array accessed irregularly and calls a library for reordering iterations of the executor loop and creation of computation and communication sets. The effect of this sequence of calls to the library is shown in the algorithm in Figure 5.5. A key input for this step is the Regular Section Descriptor  $P_{an}$  for the reaching definitions of array  $a$  from each producer  $n$  (see Section 5.2). Our OpenMP-to-MPI compiler uses Regular Section Descriptors (RSDs) to summarize production and consumption information for shared arrays. By comparing the inspection structure  $T_A$  for accesses to  $A$  with the RSDs of the reaching definitions, this algorithm deduces - (1) the source of data required by a particular iteration and (2) the exact array elements required by a particular iteration. This is a key difference between inspection in our scheme and inspection in the context of HPF. In HPF, the inspector resolves the array sources by comparing accesses with the data distribution specified in the program. In our case, sources are resolved in terms of the regular section descriptors constructed by the OpenMP-to-MPI compiler, to create the producer sets  $prod_i$  in Figure 5.5. The iterations of the one-dimensional executor loop created by the compiler are then lexicographically sorted based on these producer sets as well as the values of the index variables  $\langle i_1, \dots, i_n \rangle$ . Using the producer sets as the primary key results in contiguous blocks of iterations having the same data source. The compiler then forms computation sets  $S_k$  using such contiguous blocks of iterations. Using the index variables  $\langle i_1, \dots, i_n \rangle$  as a secondary key in the lexicographical sorting maintains, to the extent possible, the cache affinity of any regular array accesses present in the loops.

At this point, we make two observations regarding the overheads of the algorithm described in Figure 5.5. For each irregular access, the inspector must determine the data source by comparing the inspected value of the subscript with a Regular Section Descriptor (RSD). Since the RSDs are expressed as ranges, this can be done efficiently. The main overhead of this algorithm involves the lexicographical sorting. However, in our execution model, the irregular parallel loop is distributed blockwise between

processes [81] and each process has to perform these steps for only the iteration block assigned to it. Thus, the total iteration space that must be inspected and restructured decreases proportionally with increasing number of processes. Thus, we found that the overhead of this algorithm also decreases proportionally with increasing number of processes.

**(III) Scheduling of Overlapping Communication with the Reordered Computation** The algorithm described in Figure 5.5 creates computation and communication blocks that are scheduled by the algorithm described in Figure 5.6. The fact that the iteration space is already lexicographically sorted in terms of the array accesses results in each computation partition  $S_k$  having a set of contiguous iterations. This algorithm simply overlaps the communication for a data block with computation on data that is either locally available or has already been received.

## 5.5 Ensuring Computation-Communication Overlap

The inspection technique described in the previous section maps iterations to shared data accesses. The iteration reordering scheme then creates computation and communication sets that can be overlapped with each other. In principle, we could overlap computation and communication by using non-blocking send/receive calls in MPI. However, in practice, we find that this does not necessarily guarantee that background MPI communication will progress while the computation runs in the foreground. As far as the semantics of non-blocking calls is concerned, the MPI specification [12] only states that a non-blocking MPI send call will return irrespective of whether a matching MPI receive call has been posted, and a non-blocking MPI receive call will return irrespective of whether a matching MPI send call has been posted. If a process has called `MPI_Irecv` (non-blocking receive) there is no guarantee that communication will start and progress as soon as another process posts a matching `MPI_Isend` (non-blocking send) call. In fact, depending upon the MPI implementation and the amount of data being communicated, this point-to-point communication may

not progress to completion till the receiver posts a matching `MPI_Wait` or `MPI_Test` call. Many current vendor implementations do ensure progress of non-blocking communication in the background. However, target platforms for our translation scheme also include commodity systems such as network of workstations connected by regular ethernet and running publicly available MPI libraries such as MPICH [82] and LAM [83], where background progress of non-blocking communication is not automatically guaranteed. In the experiments discussed in Section 5.6, our platform is a set of sixteen dual-processor WinterHawk nodes of an IBM SP2 system, connected by a high-performance switch. Current IBM SP switches have a Remote Direct Memory Access (RDMA) capability, whereby non-blocking MPI communication can progress concurrently with computation. However, the switch used on our platform is less recent and lacks RDMA capability.

To address this lack of progress guarantee, our approach is to split the program into separate computation and communication threads. For each dual processor node in our system, one processor is dedicated for running the communication thread. This approach is also advocated in some current message-passing architectures like Blue-Gene [84,85], where the recommended use of one processor on a multi-processor node is as a communication co-processor. A communication thread is spawned per process at the very beginning of the program. Each process routes subsequent MPI calls through this communication thread. When a process needs to perform a non-blocking receive operation overlapped with computation, the computation thread queues a special non-blocking receive call with the communication thread. The communication thread actually performs this non-blocking receive using a blocking receive operation to ensure progress. In this way, true overlap of computation and communication is achieved irrespective of whether the underlying MPI implementation guarantees progress for non-blocking operations.

We have implemented a Pthreads-based library of MPI calls for this purpose. The library contains wrapper functions for most basic MPI calls. For example, for send, there are two wrapper functions. One of these is called by the computation thread.

It allocates space to hold the parameters for the send call, copies the parameters into this allocated space and signals the communication thread, passing to it pointers to the allocated space and the second wrapper function. The communication thread executes the second wrapper function, which performs the actual MPI send call.

A novel feature of our system is that, in addition to the basic wrapper functions in this library, the OpenMP-to-MPI compiler also generates specialized “pack-and-send” and “receive-and-unpack” subroutines per instance of overlapping computation and communication where required. Irregular accesses often result in access to non-contiguous array elements for a contiguous set of iterations, even though they access data from the same source process. In such a case, the non-contiguous elements need to be packed into a buffer by the sender before being sent and need to be unpacked at the receiver before being used. By generating the specialized subroutines, the compiler assigns the packing and unpacking tasks to the communication thread and thus, manages to hide the packing and unpacking latencies at run-time. From a performance viewpoint, this generating specialized “pack-and-send” and “receive-and-unpack” subroutines is a more efficient alternative to having generalized wrappers for the MPI functions. It also helps maintain cleaner semantics regarding which non-blocking MPI calls should be actually executed by the communication thread in a blocking manner.

## 5.6 Performance Evaluation

To evaluate the effectiveness of our reordering scheme, we have manually applied the transformations described in Section 5.4 to three representative OpenMP benchmarks from two important classes of irregular applications – sparse matrix computations and molecular dynamics. For sparse matrix computations, we have selected Equake from the SPEC OMPM2001 benchmarks [49] and CG from the NAS Parallel Benchmarks 2.3 suite [30]. For molecular dynamics, we have selected the MOLDYN kernel from CHARMM [86] application. Our compiler infrastructure handles C pro-

grams. Thus for SPEC OMPM2001 we have used the only available irregular C benchmark. For the NAS benchmarks, we have used the NPB-2.3 OpenMP C version of CG created by RWCP (<http://phase.hpcc.jp/Omni/benchmarks/NPB/>).

CG and IS are the two NAS benchmarks that have irregular accesses. In previous work [81] we had already achieved satisfactory scalability for IS. Therefore, for this study, we have selected CG. For each of these applications, we evaluate the effects of our transformation using two metrics – (1) in terms of the reduction in execution time for the whole application and (2) in terms of the actual computation-communication overlap achieved by the reordering.

For evaluating the reduction in execution time for the whole application, we compare the execution times of

- (a) A hand-coded MPI version, that does not use computation-communication overlap. In case of CG, this version is the official NPB2.3-MPI CG version. For MOLDYN and Equake, we have created these versions ourselves with modest programming effort.
- (b) A baseline version translated from OpenMP to MPI [81] using only compile-time analysis. This version does not incorporate any run-time inspection.
- (c) A version translated from OpenMP to MPI that uses a run-time inspector, uses non-blocking communication to overlap communication between processes, but does not perform iteration reordering for computation-communication overlap.
- (d) A version translated from OpenMP to MPI that uses a run-time inspector and also reorders iterations for computation-communication overlap.

For measuring the actual computation-communication overlap achieved by iteration reordering, we individually measure the actual time spent in sends/receives by the communication thread, the overlapping computation and the actual time spent waiting by the computation thread for the required communication to complete. Figure 7 depicts a typical scenario that may occur in an execution with three involved processes  $p, q$  and  $r$ .

The total time spent in MPI Send/Recvs by the communication thread of process  $p$  is  $t_{send-recv} = t_{send} + t_{recv-q} + t_{recv-r}$  and the computation time available for overlap

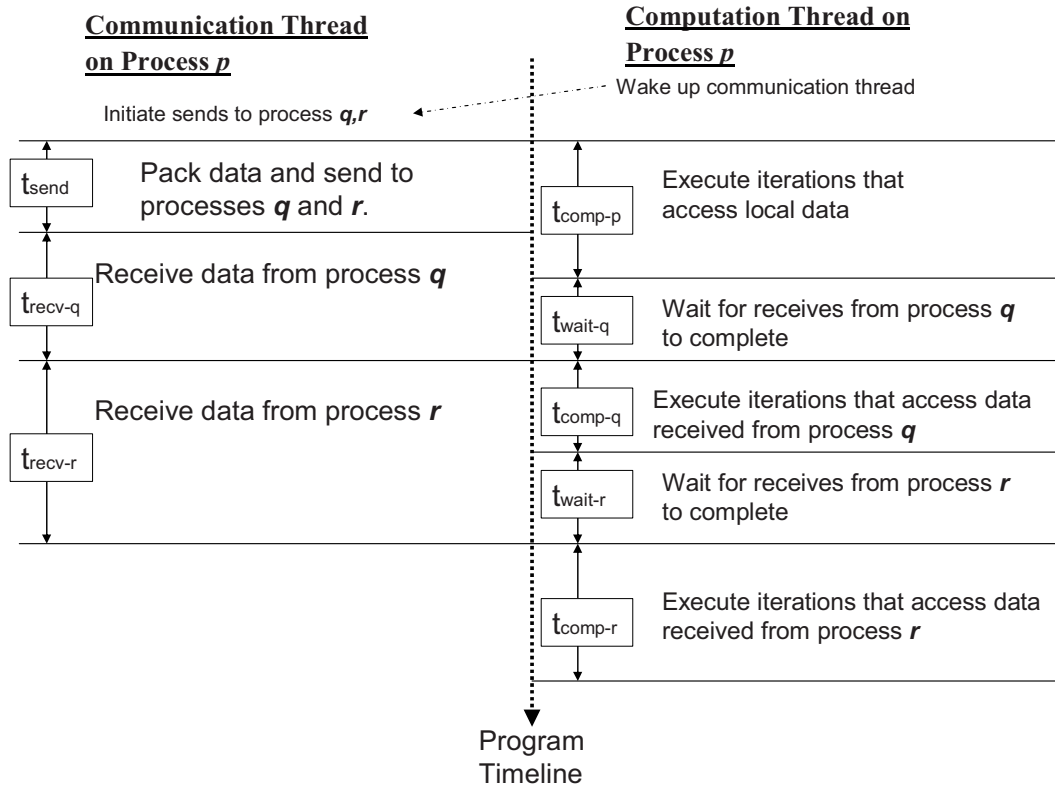


Fig. 5.7. Measurement of Computation-Communication Overlap : A scenario with three processes -  $p, q$  and  $r$ . All processes send data computed locally to all other processes that need these data in the next step. Measured on process  $p$ , the time spent in sends/recvs is  $(t_{send} + t_{recv-q} + t_{recv-r})$ . The computation available for overlap is  $(t_{comp-p} + t_{comp-q})$ . The actual wait time (time spent by the computation thread on  $p$  waiting for sends/receives to complete) is  $(t_{wait-q} + t_{wait-r})$ .



is  $t_{comp} = t_{comp-p} + t_{comp-q}$ . The actual time spent by the computation thread waiting for communication to complete (which we refer to as the “actual wait time” is  $t_{wait} = t_{wait-q} + t_{wait-r}$ . We would thus achieve a reduction in execution time if we have  $t_{wait} < t_{send-recv}$ . Ideally, we should also see  $t_{send-recv}$  being equal to  $(t_{comp} + t_{wait})$ . In practice, we observe  $(t_{comp} + t_{wait}) > t_{send-recv}$  and the difference is because of overheads in threading (waking up waiting computation or communication threads) and in copying data between computation and communication threads.

The platform we have used for our experiments is sixteen IBM SP2 WinterHawk-II dual-processor nodes, connected by a high-performance switch. To ensure background communication progress, as discussed in Section 5.5, we use one processor per node for the communication thread. Our setup is thus more general and can be extended to other commodity platforms such as multiprocessor workstations connected by regular ethernet. For comparison, wherever relevant, we shall refer to the performance achieved by the inspector-only versions where all thirty two processors were used as compute nodes. The MPI library used is the IBM MPI library with xlc version 6 (at optimization level O3) as the back-end compiler. We now examine in detail the performance of Equake, CG and MOLDYN using the two metrics discussed above.

### 5.6.1 Performance of *Equake*

Equake is an OpenMP application that is part of the SPEC OMPM2001 benchmark suite. Equake performs a simulation of an earthquake. For our experiment, we have used the *ref* dataset. However, instead of the 3333 timesteps for the dataset, we have performed measurements for 330 timesteps. The most time-consuming part of this application is the *smvp* subroutine, which computes the product of a sparse matrix (which contains the grid coordinates) and a vector (which contains displacements in time). This subroutine is called in every timestep. The sparse matrix-vector multiplication loop is parallelized in OpenMP to accumulate the product in a private copy in each thread, which is then merged by a global reduction.

Figure 5.8 shows the performance of the hand-coded, baseline, the inspector-only and the inspector-with-reordering versions for Equake. The sparse matrix-vector product in the *smvp* subroutine accesses the displacement vector irregularly. However, the access pattern remains the same through all timesteps. The vector read in one timestep is produced blockwise by all threads in the previous timestep. The compiler analysis for the OpenMP-to-MPI baseline translation assumes that each thread reads the entire vector and thus inserts an all-to-all communication between processes to communicate the displacement vector. The resultant large volume of communication limits the scalability of this baseline translation. Using an inspector in the “inspector-only” version cuts down the communication requirement by recording the actual accesses pattern for the displacement vector and shows a speedup of close to three times (for 16 processes) over a serial version. The hand-coded version uses inspection of the indirection vector as well to cut down communication, but does not overlap computation with communication. Its performance is better than the baseline and inspector-only versions. But re-ordering iterations and using computation-communication overlapping further reduces execution time by about 32% on 8 nodes and 45% on sixteen nodes compared to the inspector-only version and is faster than the hand-coded version on eight and sixteen processes as well. The increased single-processor execution time for the inspector-only and the inspector-with-reordering versions is the inspection and reordering overhead. This overhead also reduces in going from one to sixteen processors because the iteration space (that needs to be inspected and reordered) on each process reduces with increasing number of processes. Even if all 32 processors in the 16 nodes are used, the 16 node inspector-with-reordering version is faster than the 32-process hand coded version.

Figure 5.9 shows the computation-communication overlap achieved for Equake. Specifically, we have measured this metric for the matrix-vector part in the *smvp* subroutine. Even after inspection, the communication volume is very high. However, overlap allows us to tolerate the communication latency considerably, cutting down the effective communication latency ( $t_{wait}$  as discussed previously) by about 50 %

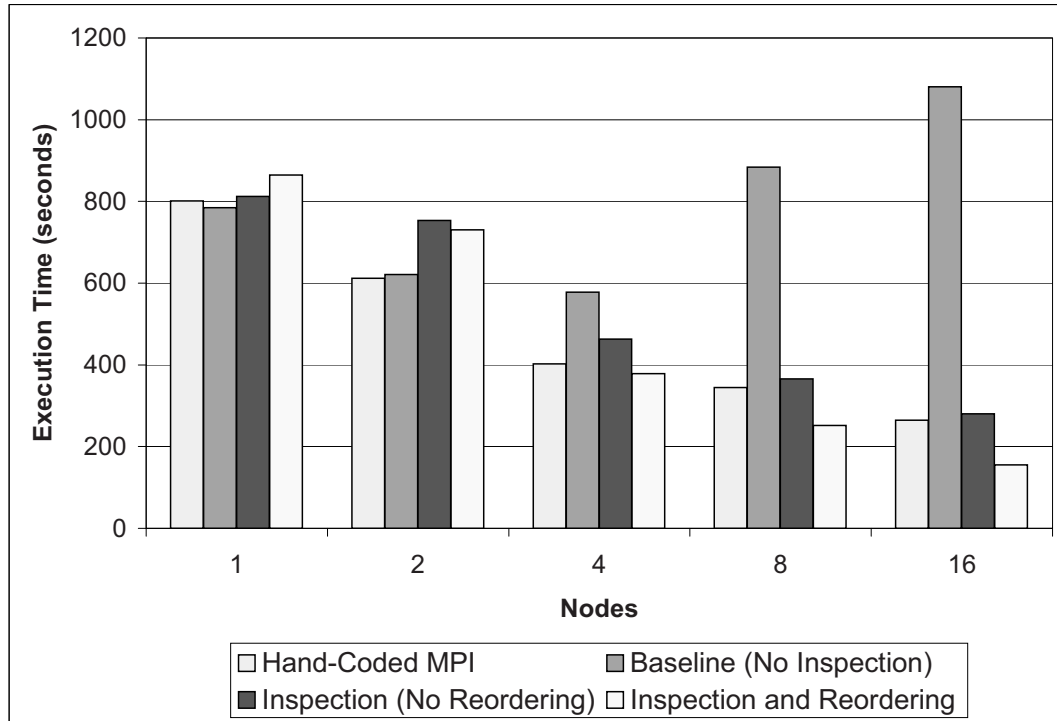


Fig. 5.8. Performance of *Equake*.

on two processes to 68 % on sixteen processes. This reduction of communication latency of the matrix-vector product, as well as the communication latency reduction for the subsequent accumulation of the result vectors, makes the re-ordered version almost twice as fast as the inspector-only version on sixteen nodes. An observation from Figure 5.9 is that the actual-time spent in sends/receives reduces in going from two to sixteen nodes. This trend depends on the actual pattern of communication between processes and the volume of data that needs to be communicated. For CG, the time spent in sends/receives increases in going from two to sixteen nodes.

### 5.6.2 Performance of *CG*

CG implements the conjugate-gradient method and is part of the NAS benchmark suite. The most time-consuming part of this application is the *conj\_grad* subroutine and a large chunk of this time is spent in computing the product of a sparse matrix

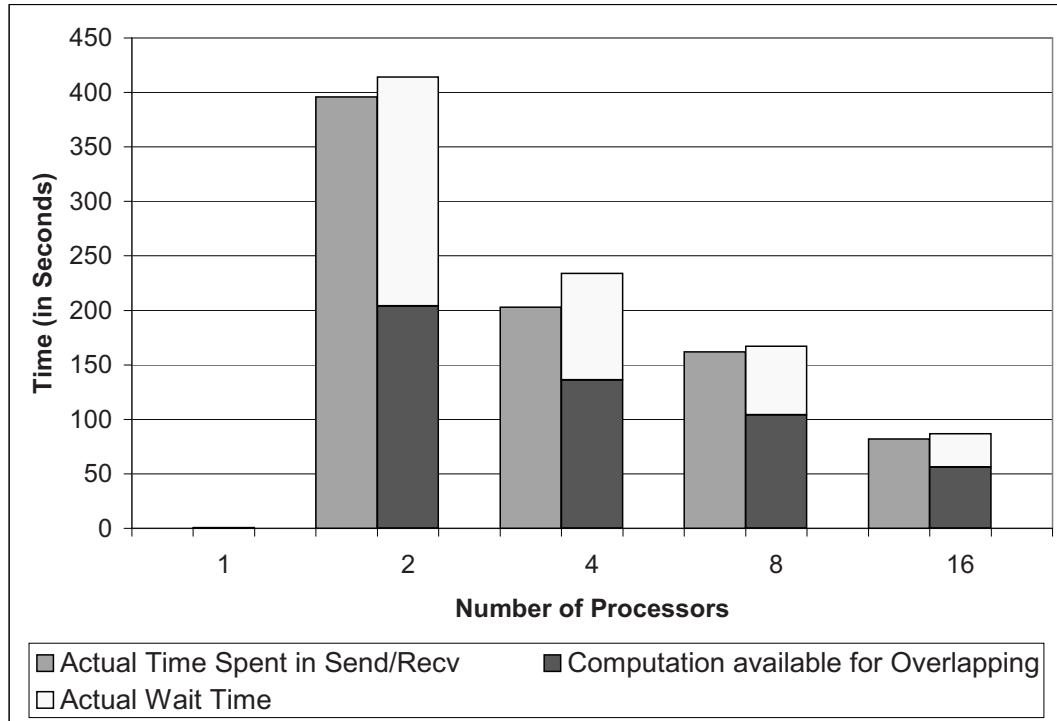


Fig. 5.9. Computation-Communication Overlap in *Equake*.

$A$  and a vector  $p$ . In the OpenMP matrix-vector multiplication loop,  $A$  is accessed blockwise and  $p$  is irregularly read. Figure 5.10 shows the performance of the baseline, inspector-only and inspector-with-reordering versions.

The baseline translation assumes that the entire vector  $p$  is read by all processes. As it turns out, this assumption is actually valid. Therefore, the inspector-only version does not reduce any communication time and is not any faster than the baseline version. The inspector-with-reordering version succeeds in tolerating the communication latency and reducing the execution time by almost 22% on 16 nodes compared to the baseline version. Figure 5.10 also shows the execution of the hand-coded NPB2.3-MPI version of CG. This version employs a sophisticated strategy of logically dividing processes into a 2-D grid, distributing  $p$  along columns and replicating it along rows in order to cut down communication overhead. Still, we find that the MPI version created by translation from the comparatively simple OpenMP version is just about 10% slower than the hand-coded MPI version. The CG benchmark performs one

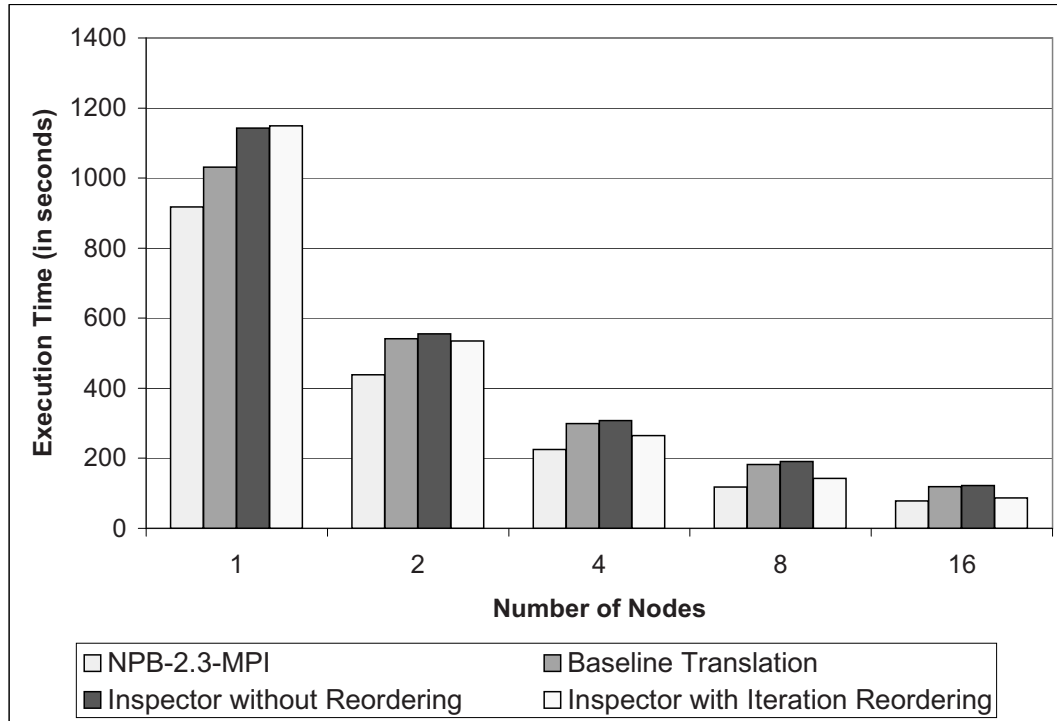


Fig. 5.10. Performance of *CG*.

iteration for "free" before it starts the timed iterations. The inspector and reordering code executes during this free iteration and the overheads are not directly visible in this performance comparison. However, we separately measured this overhead and found it to be about 3.8% of the complete application's execution time.

Figure 5.11 shows the computation-communication overlap in *CG*. The actual time spent in sends/recvs ( $t_{send-recv}$ ) for *CG* is (for up to 16 nodes) lower than the computation time available for overlapping. Therefore, the communication latency can be almost entirely tolerated except for overheads involved in threading and in moving data between the computation and communication threads. Thus, the actual time for the entire application spent by the computation thread waiting for communication to complete ( $t_{wait}$ ) is very low (about 2 seconds).

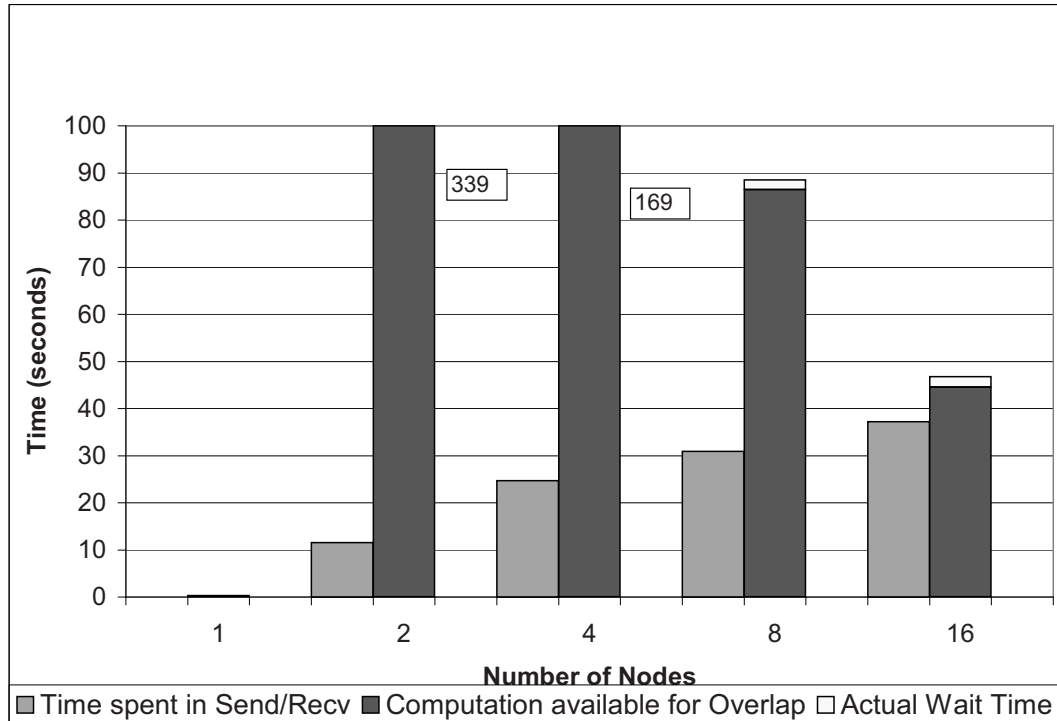


Fig. 5.11. Computation-Communication overlap in *CG*.

### 5.6.3 Performance of *MOLDYN*

*MOLDYN* is a kernel within the CHARMM molecular mechanics application. *MOLDYN* simulates the interaction between particles by considering only those pairs of particles that are within a cutoff distance from each other. It builds an interaction list of such "neighbors" right at the beginning and computes changes in each particle's position, velocity and energy over certain number of timesteps. After a given number of timesteps, it recomputes the list of neighbors.

The two most time-consuming parts of *MOLDYN* are (a) computing the list of interacting particles (neighbors) and (b) computing the forces between interacting pairs for all pairs in the interaction list. In creating the OpenMP version of *MOLDYN*, we followed an SPMD style parallelization approach [87] and were able to effectively parallelize both the force computation and neighbor list creation. Our serial *MOLDYN* version uses a Recursive Coordinate Bisection partitioner [88] to partition particles

prior to building the interaction list. For the OpenMP versions, we have continued to use this partitioner. Irregular accesses occur to the coordinate vectors of the particles in the force computation loop - where each iteration calculates the forces between a pair in the interaction list. The forces are accumulated in a local copy and copies are merged in a subsequent global reduction.

In our experiments, the dataset used for MOLDYN had 23328 particles, the interaction list was computed once at the beginning and then after every 20 timesteps. The simulation was done for a total of 40 timesteps. Figure 5.12 shows the performance of the hand-coded, baseline, inspector-only and inspector-with-reordering versions of MOLDYN in terms of execution time. The baseline version assumes, for the force computation loop, that each process accesses the entire coordinates vector. It therefore suffers from redundant computation. The hand-coded version inspects the indirection vectors implicitly while the interaction list is being built and is thus more efficient than the inspector-only version. Inspection reduces the volume of communication significantly by eliminating redundant communication and the inspector-only is almost 40% faster than the baseline version. However, iteration reordering and computation-communication overlap tolerates the communication latency even further and that version is faster than both the inspector-only and hand coded versions on four, eight and sixteen nodes.

Figure 5.13 shows the computation-communication overlap for the inspector-with-reordering version. On two and four nodes, the available computation is more than the send-recv latency and the communication latency is almost entirely tolerated. For the two node case, there is a slight communication imbalance (for the force calculation, process 1 must receive some coordinates from process 0, but process 0 does not need any coordinates from process 1. Therefore, process 0 completes its force calculation earlier and has to wait a bit in the subsequent global reduction phase, while process 1 completes its force calculation).

Compared to CG and Equake, MOLDYN has a higher overhead associated with inspection of irregular accesses and iteration reordering. For MOLDYN, every time

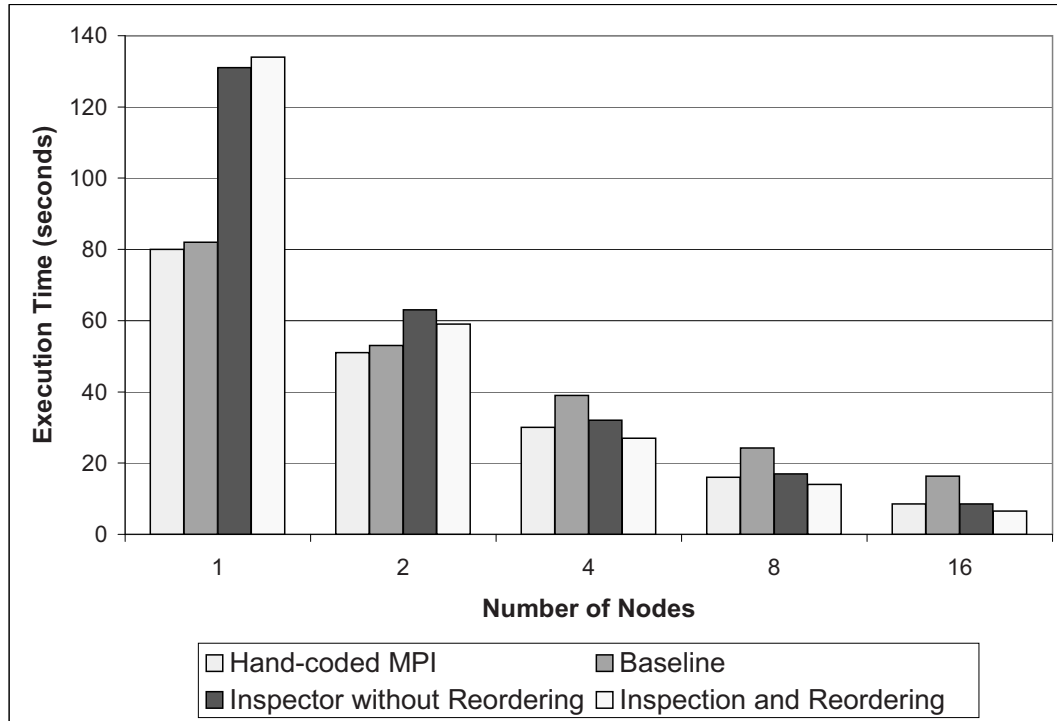


Fig. 5.12. Performance of *MOLDYN*.

the interaction list is recomputed, inspection of accesses and iteration reordering has to be repeated. However, as has been explained in Section 5.4, this overhead decreases with increasing number of processes. Despite this overhead, the inspector-with-reordering version on 16 nodes shows a speedup of 12.6 relative to the baseline serial version, compared to 9.6 for the inspector-only version and 5.03 for the baseline version on 16 nodes.

## 5.7 Related Work

Irregular applications have been studied extensively. There is a large body of work on both compile-time and dynamic schemes [89–93] which have focussed on improving data locality in irregular applications. While these were important starting points for our work, our focus is on transformations that expose opportunities for computation-communication overlap, rather than data locality.



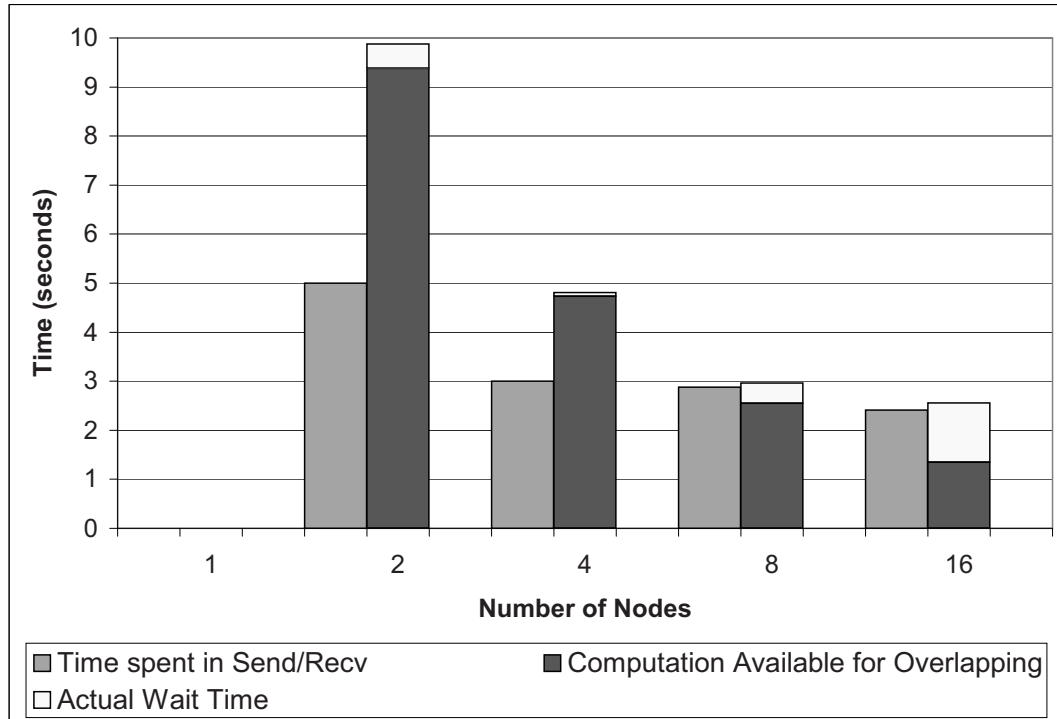


Fig. 5.13. Computation-Communication overlap in *MOLDYN*.

In the context of irregular parallel applications using HPF, the inspector-executor scheme [28] was proposed to resolve remote accesses. While related approaches proposed communication optimizations [94], none of these approaches examined the restructuring of irregular loops to overlap computation with communication. An instance of loop restructuring within the inspector-executor model was proposed to improve data locality [95].

The notion of tolerating communication latency using computation-communication overlap has been used before, mainly in the context of regular applications. HPF compilers [25] proposed the posting of sends as early as possible and receiving as late as possible in order to overlap with available intervening computation. A key distinction of these with our work is that we overlap communication with *related* computation. Considering the loops L1 and L2 in Figure 5.1, the simple approach of sending early and receiving late would not achieve any overlap. The vector  $p$  produced in loop L1 is consumed in loop L2 and there is no intervening computation between L1 and L2

with which the communication of  $p$  can be overlapped. This pattern is found often. In contrast to HPF compilers, we overlap communication of data used in loop L2 with computation in the same loop.

Some later approaches, in the context of HPF, have suggested the overlapping of computation and communication for a single-loop but these have been compile-time approaches, applicable only for regular loops. Liu and Abdelrahman [96] suggested loop peeling in regular HPF applications to differentiate iterations with local and non-local accesses. They did not consider loop restructuring techniques to maximize overlap opportunities. Ishizaki et al [97] proposed a tiling-based approach for computation-communication overlap in HPF applications. Both these efforts are limited to regular applications. Su and Yelick [98] examined irregular applications in the context of the Titanium [35] language. Their work evaluated message optimizations based on runtime inspection of accesses. They did not propose any loop restructuring to enable computation-communication overlap.

Lu et al [99] proposed compiler support for irregular applications on software distributed shared memory. Their approach was to use the compiler to identify induction arrays for irregular accesses and to prefetch them, aggregating the irregular accesses. Their approach amortizes the communication latency that would occur in a page-based DSM, but they do not achieve any computation-communication overlap, which could reduce latencies further. Lain and Banerjee [100] have examined a graph-coloring based partitioning approach for iterative irregular applications. Their work is focussed on computation partitioning, rather than any restructuring to expose overlap opportunities in partitions already assigned to a process.

To the best of our knowledge, this dissertation presents the first approach to optimization shared-memory irregular applications on distributed-memory systems using loop restructuring coupled with run-time inspection to achieve computation-communication overlap.

## 5.8 Conclusion

In Chapter 4, we have presented techniques that can transform shared-memory programs, written in standard OpenMP, into MPI message-passing programs. In all but a number of irregular applications, these techniques achieved good performance. In this chapter we have now added techniques that fill this void, leading to good performance in all of our test programs. These include codes from the SPEC OMP applications, NAS benchmarks, and several others.

Our new optimizations achieve overlap of computation and communication, even in the case where data is produced in one computation loop and consumed in the immediately following loop. It does so by reordering computation in a way that locally available data is used immediately, while remote data is communicated for the next computation step. A combined compile-time/run-time scheme is used to accomplish this. Our runtime techniques also include a method to guarantee that overlap of computation and communication actually happens, which is not the case in many current MPI implementations.

## 6. EPILOGUE

### 6.1 Conclusions

This dissertation has investigated compiler and runtime techniques for deploying shared-memory programs, written in OpenMP, on distributed-memory systems. Writing shared-memory programs is arguably much easier than writing message-passing code. However, commodity platforms such as networks of workstations continue to be the dominant platforms for High-Performance Computing. This dissertation aims to extend the ease and elegance of shared-memory parallel programming to these distributed-memory platforms.

To this end, we have investigated two approaches in this dissertation. The first approach utilizes a Software Distributed Shared Memory (SDSM) System to provide a shared-memory abstraction. We have presented techniques to translate OpenMP programs to SDSM programs. We have evaluated the performance of this baseline translation in terms of overheads introduced in the translation of different OpenMP constructs and overall execution time using micro-benchmarks, kernel programs and representative OpenMP applications from the SPEC OMPM2001 benchmark suite. We have also discussed optimizations such as *computation repartitioning*, *page-aware optimizations* and *access privatization* that improve the performance of these translated applications by an average of 70% compared to the baseline translation.

The second approach explored by this dissertation is to translate OpenMP applications directly to message-passing applications. To accomplish this, we have incorporated Regular Section Analysis to summarize array accesses and proposed novel techniques for analysis of shared-data, incorporation of OpenMP's memory consistency semantics, message-generation and an SPMD execution model based on redundant execution of serial regions and partial replication of shared data. We have

also proposed optimizations such as *computation alignment*, *recognition of reduction idioms* and *casting of point-to-point communication to collective communication* that improve the performance of the baseline scheme significantly. We have evaluated the performance of our translation scheme by measuring the performance of representative OpenMP applications from the NAS Parallel Benchmarks and SPEC OMPM2001 suite and comparing them with the performance of corresponding hand-coded MPI applications. Overall, the average scalability in terms of execution time on one processor for the translated OpenMP applications is within 12% of their hand-coded MPI counterparts. The OpenMP applications translated to MPI also significantly outperform their HPF and SDSM counterparts.

In case of OpenMP applications that have irregular accesses, a compile-time analysis leads to a conservative summarization of array accesses which in turn limits the performance of these applications translated to MPI. For such cases, this dissertation proposes a combined compile-time/runtime scheme. A key insight here is that runtime inspection can not only distinguish between local and remote accesses but also map iterations to specific accesses. We propose a scheme that then overlaps computation on data from one source with communication of data from another source. To maximize this overlap, we proposed a novel computation restructuring technique. To guarantee overlap, we implemented a threaded MPI library. We evaluate these techniques by measuring the performance of three OpenMP applications with irregular accesses. On one application, the translated OpenMP version achieves performance within 10% of the official MPI version while in the other two, the translated OpenMP versions outperform hand-coded MPI versions that we created with considerable programming efforts.

The approaches to deploy OpenMP applications on distributed-memory platforms proposed in this dissertation thus achieve reasonable performance for both regular and irregular applications and thus a significant step towards increasing the productivity of parallel programming for commodity platforms.

## 6.2 Future Work

While this dissertation in itself represents a significant step towards shared-memory programming support for distributed-memory platforms, it also opens up some possibilities for future research in this area. To conclude this dissertation, we discuss two promising future directions.

### 6.2.1 Shared-Memory Programming for Hybrid Platforms

Current processors have *multi-core* architectures. Thus, future commodity platforms may be distributed-memory systems but will have multiple shared-memory cores on each node. Researchers have proposed a *Hybrid Programming Model* for such systems with programs being written with a mixture of OpenMP and MPI to make use of these intra and inter-node parallelism. MPI programming itself is considered effort-intensive. Therefore, a hybrid programming model may be unreasonably demanding for a programmer.

The translation scheme proposed in this dissertation in Chapter 4 conserves the structure of the original OpenMP program and OpenMP directives. With minimal extensions, this scheme could be used to translate OpenMP programs to hybrid OpenMP/MPI programs. Thus, it would extend the ease and productivity of OpenMP programming to hybrid platforms that have intra-node shared memory.

### 6.2.2 Runtime and Interactive Optimizations

Certain optimizations explored briefly in these dissertation, such as computation repartitioning, would be more efficient as runtime or interactive optimizations. For example, consider a program with nested loops where each nested loop is parallel. Depending on runtime parameters, specifying one level as parallel may result in extra communication. In such cases, different versions of the application could be created

with each version selecting one level of the loop nest to be parallel and at runtime, the version with the best performance could be selected.

Or, the compiler could be part of an integrated development environment whereby it could warn the user where certain parallel loops could result in increased communication because of the direction along which it accesses a multi-dimensional array. In this way, it could provide important performance feedback to the user even at compile-time.

## LIST OF REFERENCES



## LIST OF REFERENCES

- [1] OpenMP Forum, “OpenMP: A Proposed Industry Standard API for Shared Memory Programming,” tech. rep., October 1997.
- [2] M. P. I. Forum, “MPI: Message passing interface standard.” [www.mpi-forum.org](http://www.mpi-forum.org), 1999.
- [3] J.M.Bull, “Measuring Synchronization And Scheduling Overheads in OpenMP,” in *Proc. of the European Workshop on OpenMP (EWOMP99)*, October 1999.
- [4] H. Jin, M. Frumkin, and J. Yan, “The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance,” Tech. Rep. NAS-99-011.
- [5] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, “SPECComp: A New Benchmark Suite for Measuring Parallel Computer Performance,” in *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001)*, *Lecture Notes in Computer Science*, 2104, pp. 1–10, July 2001.
- [6] M. Booth and K. Misegades, “Microtasking: A New Way to Harness Multiprocessors,” *Cray Channels*, pp. 24–27, 1986.
- [7] M. Martonosi and A. Gupta, “Tradeoffs in message passing and shared memory implementations of a standard cell router,” in *Proceedings of the 1989 International Conference on Parallel Processing.(IEEE) Volume III*, pp. 88–96, August 1989.
- [8] S. Chandra, J. R. Larus, and A. Rogers, “Where is time spent in message-passing and shared-memory programs,” in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, California), pp. 61–73, 1994.
- [9] A. C. Klaiber and H. M. Levy, “A comparison of message passing and shared memory architectures for data parallel programs,” in *Proceedings of the 21th International Symposium on Computer Architecture*, 1994.
- [10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, “Active messages: a mechanism for integrated communication and computation,” in *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, (New York, NY, USA), pp. 256–266, ACM Press, 1992.
- [11] V. S. Sunderam, “PVM: a framework for parallel distributed computing,” *Concurrency, Practice and Experience*, vol. 2, no. 4, pp. 315–340, 1990.
- [12] M. P. I. Forum, “MPI: A Message-Passing Interface Standard,” Tech. Rep. UT-CS-94-230, 1994.

- [13] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.
- [14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *SIGPLAN Not.*, vol. 30, no. 8, pp. 207–216, 1995.
- [15] 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*, vol. 29, no. 2, pp. 18–28, 1996.
- [16] B. N. Bershad and M. J. Zekauskas, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," Tech. Rep. CMU-CS-91-170, Pittsburgh, PA (USA), 1991.
- [17] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," in *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pp. 168–177, Mar. 1990.
- [18] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel, "OpenMP for Networks of SMPs," *Journal of Parallel and Distributed Computing*, vol. 60, pp. 1512–1530, December 2000.
- [19] J. Zhu and J. Hoefflinger, "Compiling for a Hybrid Programming Model Using the LMAD Representation," in *Proc. of the 14th annual workshop on Languages and Compilers for Parallel Computing (LCPC2001)*, August 2001.
- [20] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers," in *The First International Workshop on Parallel Processing*, (Bangalore, India), pp. 322–330, Dec. 1994.
- [21] J. Li and M. Chen, "Generating explicit communication from shared-memory program references," in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pp. 865–876, IEEE Computer Society, 1990.
- [22] J. Li and M. Chen, "Compiling communication-efficient programs for massively parallel machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 3, pp. 361–376, 1991.
- [23] High Performance Fortran Forum, "High Performance Fortran language specification, version 1.0," Tech. Rep. CRPC-TR92225, Houston, Tex., 1993.
- [24] C. Koelbel, D. Loveman, R. Schreiber, G. S. Jr., and M. Zosel, *The High Performance Fortran Handbook*. MIT Press, 1994.
- [25] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo, "An HPF compiler for the IBM SP2," in *Proceedings of Supercomputing '95*, (San Diego, CA), 1995.
- [26] V.J.Schuster, "Pghpf from the portland group," in *IEEE Parallel and Distributed Technologies*, p. 72, 1994.

- [27] U. Kremer, “Automatic data layout for distributed memory machines,” Tech. Rep. TR96-261, 14, 1996.
- [28] R. Das, M. Uysal, J. Saltz, and Y.-S. S. Hwang, “Communication optimizations for irregular scientific computations on distributed memory architectures,” *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 462–478, 1994.
- [29] M. Frumkin, H. Jin, and J. Yan, “Implementation of NAS Parallel Benchmarks in High Performance Fortran,” Tech. Rep. NAS-98-009.
- [30] R. V. D. Wijngaart, “NAS Parallel Benchmarks, Version 2.4,” Tech. Rep. NAS-02-007.
- [31] B. L. Chamberlain, S. J. Deitz, and L. Snyder, “A comparative study of the NAS MG benchmark across parallel languages and architectures,” in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, p. 46, IEEE Computer Society, 2000.
- [32] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. A. Yelick, “Parallel Programming in Split-C,” in *Supercomputing*, pp. 262–273, 1993.
- [33] T. El-Ghazawi, W. Carlson, and J. Draper, “UPC Language Specifications V1.0,” Feb. 2001.
- [34] R. W. Numrich and J. Reid, “Co-array fortran for parallel programming,” *SIG-PLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.
- [35] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, “Titanium language reference manual,” tech. rep., Berkeley, CA, USA, 2001.
- [36] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, “Global arrays: a portable “shared-memory” programming model for distributed memory computers,” in *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, (New York, NY, USA), pp. 340–349, ACM Press, 1994.
- [37] T. El-Ghazawi and F. Cantonnet, “Upc performance and potential: a npb experimental study,” in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–26, IEEE Computer Society Press, 2002.
- [38] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi, “OmniRPC: A Grid RPC facility for cluster and global computing in OpenMP,” *Lecture Notes in Computer Science*, vol. 2104, pp. 130–136, 2001.
- [39] L. Smith and M. Bull, “Development of Mixed Mode MPI / OpenMP Applications,” in *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2000)*, July 2000.
- [40] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. Nelson, and C. Offner, “Extending OpenMP for NUMA Machines,” in *Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference (SC2000)*, November 2000.

- [41] V. Schuster and D. Miles, "Distributed OpenMP, Extensions to OpenMP for SMP Clusters," in *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2000)*, July 2000.
- [42] T. Abdelrahman and T. Wong, "Compiler support for data distribution on NUMA multiprocessors," *Journal of Supercomputing*, vol. 12, pp. 349–371, October 1998.
- [43] B. Chapman, P. Mehrotra, and H. Zima, "Enhancing openmp with features for locality control," 1999.
- [44] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, vol. 29, pp. 18–28, February 1996.
- [45] S. V. Adve and M. D. Hill, "A Unified Formalization of Four Shared-Memory Models," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 613–624, June 1993.
- [46] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy Release Consistency for Software Distributed Shared Memory," in *Proc. of the 18th Annual Int'l Symp. on Computer Architecture (ISCA '92)*, pp. 13–21, May 1992.
- [47] S.-J. Min, A. Basumallik, and R. Eigenmann, "Optimizing OpenMP programs on Software Distributed Shared Memory Systems," *International Journal of Parallel Programming*, vol. 31, pp. 225–249, June 2003.
- [48] A. Basumallik, S.-J. Min, and R. Eigenmann, "Towards OpenMP execution on software distributed shared memory systems," in *Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI'02)*, Lecture Notes in Computer Science, 2327, Springer Verlag, May 2002.
- [49] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, "SPECComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in *Proc. of the Workshop on OpenMP Applications and Tools (WOMPAT2001)*, Lecture Notes in Computer Science, 2104, pp. 1–10, July 2001.
- [50] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel programming with Polaris," *IEEE Computer*, pp. 78–82, December 1996.
- [51] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "Quantifying the performance differences between PVM and TreadMarks," *Journal of Parallel and Distributed Computing*, vol. 43, no. 2, pp. 65–78, 1997.
- [52] A. K. W. L. Todd C. Mowry, Charles Q. C. Chan, "Comparative evaluation of latency tolerance techniques for software distributed shared memory," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA '98)*, Feb. 1998.
- [53] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "An integrated compile-time/run-time software distributed shared memory system," in *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pp. 186–197, 1996.

- [54] R. N. Taylor, "Complexity of analyzing the synchronization structure of concurrent programs," *Acta Informatica*, vol. 19, no. 1, pp. 57–84, 1983.
- [55] S.-I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation," in *Proc. of the Workshop on Languages and Compilers for Parallel Computing (LCPC'03)*, pp. 539–553, (Springer-Verlag Lecture Notes in Computer Science), Oct. 2003.
- [56] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for expec/fortran," *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.
- [57] M. P. I. Forum, "MPI-2: Extensions to the Message-Passing Interface." Technical Report, University of Tennessee, Knoxville, 1996.
- [58] Y. Lin, "Static Nonconcurrency Analysis of OpenMP Programs," in *Proceedings of the first International Workshop on OpenMP (IWOMP 2005)*, 2005.
- [59] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 2, pp. 282–312, 1988.
- [60] A. Krishnamurthy and K. Yelick, "Analyses and optimizations for shared address space programs," *Journal of Parallel and Distributed Computing*, vol. 38, no. 2, pp. 130–144, 1996.
- [61] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, (New York, NY, USA), pp. 285–294, ACM Press, 2003.
- [62] J. Lee and D. A. Padua, "Hiding relaxed memory consistency with a compiler," *IEEE Trans. Comput.*, vol. 50, no. 8, pp. 824–833, 2001.
- [63] S. P. Midkiff, J. Lee, and D. A. Padua, "A compiler for multiple memory models," *Concurrency and Computation : Practice and Experience*, vol. 16, pp. 197–220, 2004.
- [64] Y. Paek, J. Hoefflinger, and D. Padua, "Efficient and precise array access analysis," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 1, pp. 65–109, 2002.
- [65] V. Balasundaram and K. Kennedy, "A technique for summarizing data access and its use in parallelism enhancing transformations," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pp. 41–53, ACM Press, 1989.
- [66] R. von Hanxleden and K. Kennedy, "Give-n-takea balanced code placement framework," in *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pp. 107–120, ACM Press, 1994.
- [67] M. Gupta, E. Schonberg, and H. Srinivasan, "A unified framework for optimizing communication in data-parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 7, pp. 689–704, 1996.



- [68] W. M. Pottenger and R. Eigenmann, "Idiom recognition in the Polaris Parallelizing Compiler," in *International Conference on Supercomputing*, pp. 444–448, 1995.
- [69] 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*, vol. 5, pp. 63–73, Fall 1991.
- [70] G. Krawezik and F. Capallo, "Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors," in *Proceedings of the fifteenth annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 118–127, ACM Press, 2003.
- [71] C.-H. Huang and P. Sadayappan, "Communication-free hyperplane partitioning of nested loops," *J. Parallel Distrib. Comput.*, vol. 19, no. 2, pp. 90–102, 1993.
- [72] L. Lamport, "The parallel execution of do loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, 1974.
- [73] S.-J. Min, A. Basumallik, and R. Eigenmann, "Supporting realistic OpenMP applications on a commodity cluster of workstations," in *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools, WOMPAT 2003, Toronto, Canada, June 26-27, 2003. Proceedings Editors: M.J. Voss (Ed.)*, pp. 170 – 179, 2003.
- [74] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 350–360, 1991.
- [75] A. Yoshida, K. Koshizuka, and H. Kasahara, "Data-localization for fortran macro-dataflow computation using partial static task assignment," in *ICS '96: Proceedings of the 10th international conference on Supercomputing*, (Philadelphia, Pennsylvania, USA), pp. 61–68, ACM Press, 1996.
- [76] W. Blume and R. Eigenmann, "The range test: a dependence test for symbolic, non-linear expressions," in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pp. 528–537, ACM Press, 1994.
- [77] Y. Lin and D. Padua, "Compiler analysis of irregular memory accesses," in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pp. 157–168, ACM Press, 2000.
- [78] R. Sass and M. Mutka, "Enabling unimodular transformations," in *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 753–762, IEEE Computer Society Press, 1994.
- [79] P. Tu and D. Padua, "Array privatization for shared and distributed memory machines (extended abstract)," *SIGPLAN Not.*, vol. 28, no. 1, pp. 64–67, 1993.
- [80] K. Kennedy and K. S. McKinley, "Loop distribution with arbitrary control flow," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, (Washington, DC, USA), pp. 407–416, IEEE Computer Society, 1990.

- [81] A. Basumallik and R. Eigenmann, "Towards automatic translation of openmp to mpi," in *ICS '05: Proceedings of the 19th annual International Conference on Supercomputing*, (Cambridge, Massachusetts, USA), pp. 189–198, ACM Press, 2005.
- [82] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, Sept. 1996.
- [83] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," in *Proceedings of Supercomputing Symposium*, pp. 379–386, 1994.
- [84] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, and J., "An overview of the BlueGene/L supercomputer," in *SC2002 – High Performance Networking and Computing*, (Baltimore, MD), November 2002.
- [85] G. Almási, R. Bellofatto, J. Brunheroto, C. Caşcaval, J. G. C. nos, L. Ceze, P. Crumley, C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, "An overview of the BlueGene/L system software organization," in *Proceedings of Euro-Par 2003 Conference*, Lecture Notes in Computer Science, (Klagenfurt, Austria), Springer-Verlag, August 2003.
- [86] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus, "Charmm: A program for macromolecular energy, minimization, and dynamics calculations," *J. Comp. Chem.*, vol. 4, pp. 187–217, 1983.
- [87] D. Hisley, G. Agrawal, P. Satya-narayana, and L. Pollock, "Porting and performance evaluation of irregular codes using OpenMP," *Concurrency: Practice and Experience*, vol. 12, no. 12, pp. 1241–1259, 2000.
- [88] H. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering*, vol. 2, no. 2-3, pp. 135–148, 1991.
- [89] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, (New York, NY, USA), pp. 229–241, ACM Press, 1999.
- [90] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, (New York, NY, USA), pp. 252–262, ACM Press, 1994.
- [91] H. Han and C.-W. Tseng, "A comparison of locality transformations for irregular codes," in *LCR '00: Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, (London, UK), pp. 70–84, Springer-Verlag, 2000.
- [92] E.-J. Im and K. A. Yelick, "Optimizing sparse matrix computations for register reuse in sparsity," in *ICCS '01: Proceedings of the International Conference on Computational Sciences-Part I*, (London, UK), pp. 127–136, Springer-Verlag, 2001.

- [93] N. Mitchell, L. Carter, and J. Ferrante, "Localizing non-affine array references," in *1999 International Conference on Parallel Architectures and Compilation Techniques*, pp. 192–202, 1999.
- [94] R. Das, M. Uysal, J. Saltz, and Y.-S. S. Hwang, "Communication optimizations for irregular scientific computations on distributed memory architectures," *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 462–478, 1994.
- [95] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, "The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562," in *Proceedings of the 30th Aerospace Sciences Meeting*, 1992.
- [96] T. S. Abdelrahman and G. Liu, "Overlap of computation and communication on shared-memory networks-of-workstations," *Cluster computing*, pp. 35–45, 2001.
- [97] K. Ishizaki, H. Komatsu, and T. Nakatani, "'a loop transformation algorithm for communication overlapping'," *International Journal of Parallel Programming*, vol. 28, no. 2, pp. 135–154, 2000.
- [98] J. Su and K. Yelick, "Automatic support for irregular computations in a high-level language," in *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, (Denver, Colorado), April 2005.
- [99] H. Lu, A. L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel, "Compiler and software distributed shared memory support for irregular applications," in *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pp. 48–56, 1997.
- [100] A. Lain and P. Banerjee, "Techniques to overlap computation and communication in irregular iterative applications," in *ICS '94: Proceedings of the 8th international conference on Supercomputing*, (New York, NY, USA), pp. 236–245, ACM Press, 1994.



VITA

## VITA

Ayon Basumallik was born in 1976 in Calcutta, India. He received his B.Tech (Honors) degree in Electronics and E.C.E from the Indian Institute of Technology at Kharagpur, India, in May 1999. From August 2000, he pursued a Ph.D. in Computer Engineering at the School of Electrical and Computer Engineering, Purdue University, West Lafayette in Indiana, USA. He successfully defended his Ph.D. thesis in August 2007 and received his Ph.D. degree in December, 2007.

From May 1999 to June 2000, Ayon Basumallik worked at Oracle India on elucidative techniques for the latest Java technologies embedded in Oracle databases. From May 2002 to August 2002, he worked with the Standard Performance Evaluation Corporation to design a new Mail Server benchmark. From May 2006 to August 2006, he worked at Intel to incorporate novel performance enhancements into Intel's Cluster OpenMP toolkit. From August 2005 to September 2007, he worked with the Office of Information Technology at Purdue University on Grid Technologies. A significant outcome of this work was the creation of a Climatology Portal for the TeraGrid.

Ayon Basumallik has worked extensively on Parallel Programming. For his B.Tech dissertation, Ayon Basumallik proposed and implemented a Parallel Image Registration algorithm that greatly speeds up tasks such as fingerprint matching on distributed databases. For his Ph.D. dissertation, Ayon Basumallik conducted research in compiler techniques to extend the advantages of Shared-Memory Parallel Programming to Distributed-Memory Systems, which continue to be the commodity platforms for High Performance Computing. As part of this research, he proposed and implemented techniques to translate Shared-Memory Parallel Programs written in OpenMP to Message-Passing MPI programs which then be deployed on Distributed-Memory systems. This work provides a novel approach to improving programmer productivity on current and emerging High Performance Computing platforms.