

## **Shared Memory and OpenMP on Clusters**



**Shared Memory and OpenMP on Clusters**  
**Performance Aspects, Compilers, Methods, and**  
**Run-time Systems**

**Sven Karlsson**

A dissertation submitted to the Royal Institute of Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Department of Microelectronics and Information Technology, KTH,  
Stockholm

TRITA-IMIT-LECS AVH 04:11  
ISSN 1651-4076  
ISRN KTH/IMIT/LECS/AVH-04/11--SE

KTH, Royal Institute of Technology  
Department of Microelectronics and Information Technology  
P.O. Box Electrum 229  
SE-164 40 Kista  
Sweden

© Copyright 2004 by Sven Karlsson, All rights reserved

Some of the papers which are part of the thesis are reprinted with kind permissions from Kluwer Academic Publishers, IEEE Computer Society Press, and Springer Verlag. Copyright notices are put with each paper where applicable.

## Table of Contents

	List of papers included in thesis	vii
	Abstract	ix
	Preface	xi
1	Introduction	1
2	Terminology	3
3	Clusters in high performance computing	11
3.1	The architecture of processing elements	11
3.2	Network technologies	14
3.3	System software	16
4	Shared memory programming models	19
4.1	The POSIX thread API	20
4.2	ANL macros	21
4.3	OpenMP	22
5	Software distributed shared memory systems	27
5.1	The pioneer	27
5.2	The breakthrough	28
5.3	Non page-based systems	30
5.4	Hardware hybrids	30
6	Summary of contributions	33
6.1	A study of message size distributions	33
6.2	A proposal for an OpenMP capable software DSM system	34
6.3	Latency hiding and network traffic pattern restructuring	35
6.4	Using priorities to reduce latencies	38
6.5	Implementing OpenMP on Software DSM	39
6.6	An OpenMP source to source compiler for C and C++	40
6.7	An OpenMP run-time library for clusters	42
7	Concluding remarks	45



## List of papers included in thesis

This thesis is a summary of the following seven papers. The papers are appended after the summary text. References to the papers will be made using the roman numbers associated with the papers.

- I S. Karlsson and M. Brorsson, A Comparative Characterization of Message Communication in Applications using MPI and Shared Memory on an IBM SP2, in *Proceedings of 1998 Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Las Vegas, January 31 - February 1, 1998, pp. 189-201
- II S. Karlsson and M. Brorsson, An Infrastructure for Portable and Efficient Software DSM, In *Proceedings of 1st Workshop on Software Distributed Shared Memory (WSDSM '99)*, Rhodes, Greece, June 25, 1999, also available from Department of Information Technology, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden
- III S. Karlsson and M. Brorsson, Producer-Push - a Protocol Enhancement to Page-based Software Distributed Shared Memory Systems, in *Proceedings of the 1999 International Conference on Parallel Processing (ICPP'99)*, Aizu-Wakamatsu, Japan, September 1999, pp. 291-300
- IV S. Karlsson and M. Brorsson, Priority Based Messaging for Software Distributed Shared Memory, in *Journal on Cluster Computing*, 6 (2): 161-169, April 2003  
also published as:  
S. Karlsson and M. Brorsson, Priority Based Messaging for Software Distributed Shared Memory, in *Proceedings of Workshop on Communication Architecture for Clusters (CAC '01)*, April 2001
- V S. Karlsson, S-W. Lee, and M. Brorsson, A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory, in *Proceedings of 9th International Conference on High Performance Computing (HiPC 2002)*, December 2002, pp. 195-206

- VI S. Karlsson, and M. Brorsson, A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing, accepted for publication in *Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, November 2004
- VII S. Karlsson, *Balder – An OpenMP run-time library for clusters of SMPs*, Technical report, TRITA-IMIT/LECS R 04:01, ISSN 1651-4661, ISRN KTH/IMIT/LECS/R-04/01--SE, Department of Microelectronics and Information Technology, Royal Institute of Technology, KTH, August 2004

## **Abstract**

Clusters, i.e., several computers interconnected with a communication network, provide a cost efficient way to achieve high performance. Messages are the natural way of communication in this kind of systems. However, it is widely argued that using a shared memory programming model reduces the programming effort. Hence it is interesting to investigate systems that provide shared memory on clusters.

This thesis describes some performance aspects of providing such a shared memory using software. The systems that provide a shared memory in software are commonly called software distributed shared memory systems, software DSM systems. The thesis consists of seven papers that each describe different aspects of software DSM systems.

One of the main performance bottlenecks is the communication network and three papers in the thesis investigate this bottleneck. One paper analyzes the traffic sent on the network when applications are executed. In another paper a latency hiding technique is described and evaluated that substantially increases the performance of iterative applications, i.e., applications with loops. The last paper investigates the use of priorities to reduce the latency of certain messages used by the software DSM system.

One paper is devoted to discussing how a portable software DSM system should be designed and this paper forms the basis for the remaining three papers. These papers also present a compiler and run-time library for OpenMP which is a recent industry shared memory programming model. The compiler and the run-time library were developed as part of the thesis. One of the three papers describes a prototype system for running OpenMP applications on clusters. The second presents the compiler and compares the performance of applications compiled with the compiler with that of the applications compiled with a commercial compiler. The last paper describes the OpenMP run-time library.



## Preface

First of all I would like to thank my advisor Professor Mats Brorsson for his guidance, patience and support. Without him this work would surely never have been conceived. Professor Mats Brorsson together with Dr. Fredrik Dahlgren and Professor Per Stenström inspired me to pursue graduate studies.

A special thanks goes to my former and present colleagues at Lund University and the Royal Institute of Technology, and all the people in the Intone project. Perhaps most especially Jonas Alowersson, Eduard Ayguade, Anders Berkeman, Mikael Collin, Fredrik Dahlstrand, Karl-Filip Faxén, Marc Gonzalez, Seif Haridi, Rita Johnsson, Elisabeth Larsson, Sven Lämmermann, Xavier Martorell, Nguyen-Thai Nguyen-Phan, Mladen Nikitovic, Andreas Rodman, Thomas Sjöland, Lars Selander, Patrik Sundström, Vlad Vlassov, Håkan Zeffer, and Bengt Öhman. There are of course even more people that have been important to this thesis. Although excluded from this list, I gratefully acknowledge their support.

I also would like to thank my parents and brothers for always encouraging me to study hard. Tack!

During the end of the 1980's and the first few years of the 1990's I spent a tremendous amount of time in the Atari ST demo scene. The time spent competing in contests really made me grow as a programmer and I made some, I hope, life long friends. Some of these are Anders Gustavsson, Johan Klockars, Conny Pettersson, and Rasmus and Mandus Söderberg. They have all influenced this thesis.

I would, furthermore, like to thank, Daniel Elvin, Ilja Hallberg, and all other friends and colleagues that I have worked with over the years. Thank you for expressing interest in my research.

A lot of people have helped me over the years. Unfortunately, the list is too long to put here but you should not feel overlooked or left out. Thank you all!

As always, I couldn't have done this without Madelen. Thank you.

The research in this thesis has been in part financially supported by the Swedish Research Council for Engineering Sciences under contract number, TFR 1999-376, and by the Swedish National Board for Industrial and Technical Development (NUTEK) under project number P855. It was also partially financed by the European Commission

under contract number IST-1999-20252. I also gratefully acknowledge the use of computing resources at Centre for Parallel Computing at Royal Institute of Technology, Stockholm.

They say that you change to the better when you pursue a Ph.D. I do not know if I'm a better person after these years but I'm certainly not the same.

*Sven Karlsson*  
Stockholm, August 2004

# 1 Introduction

During the last years there has been an increasing interest in using clusters as a way to build multicomputers. The motivation for this is that commodity, off-the-shelf, hardware can be used without modifications. As with quite a few trends in computer architecture, this trend is fueled by the remarkable price and performance developments of personal computers. This has led clusters to be very popular for high performance computing and 291 of the 500 most powerful computers in the world are clusters including the second most powerful [54].

Clusters have traditionally been programmed using message passing. It is, however, widely argued that a machine with a shared memory is much easier to program than a machine without and this thesis deals with performance aspects of systems that provide a shared memory abstraction on a cluster.

Applications that make use of a shared memory can be written in several ways. OpenMP is an industry standard for programming machines with a shared memory [35, 36]. This thesis is also concerned with executing OpenMP applications on clusters.

Very briefly, the contributions in this thesis include an investigation of the communication patterns in clusters, the formulation, implementation, and evaluation of an performance enhancing technique for clusters and the design, implementation, and evaluation of compilers and run-time library systems enabling the use of OpenMP on clusters.

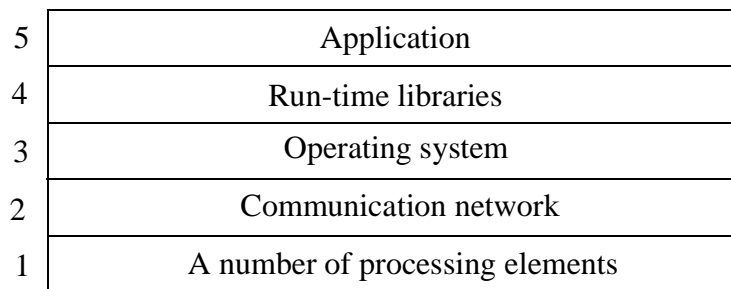
This thesis consists of seven papers and a summary. The summary is laid out as follows. Chapter 2 holds a list of definitions used in the summary. Chapter 3 is an introduction to how high performance clusters are designed. In chapter 4 the most common programming models for shared memory machines, including OpenMP, are introduced while chapter 5 outlines the history of systems that provide shared memory on clusters. Chapter 6 consists of summaries of the papers including problem statements, contributions, and results. The summary itself is concluded in chapter 7.



## 2 Terminology

In this chapter I will define some key terms and concepts that will be used throughout the rest of this summary. The definitions and terms might not be the same as those used in other works.

A parallel computer built from several general purpose microprocessors can be seen as a layered system, see figure 1.



**Figure 1: A computer as a layered system.**

The five layers are denoted 1 to 5. The view of lower layers as found on a boundary between layers is referred to as a *programming interface*. The layers up to and including layer 4 are collectively referred to as the *parallel machine*.

The functions of the different software and hardware parts in a layer are referred to as the mechanisms in that layer. The mechanisms that reside in each layer are listed below.

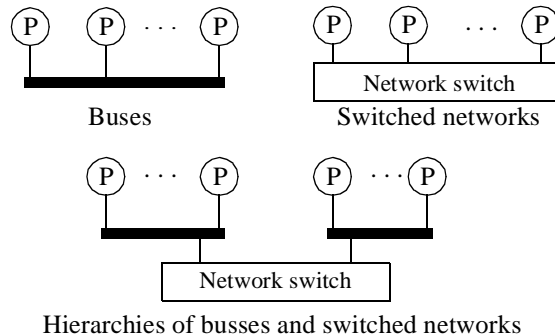
### Layer 1

Layer 1 consists of a number of *processing elements* with one or several *microprocessors*, possibly *memory* and possibly one or several levels of *caches*.

The microprocessors access memory via a *virtual address space* provided by the help of a *translation look-aside buffer* which provides a cache for address translations. In addition, there is support for *exceptions* and hardware for restarting an instruction after a miss in the translation look-aside buffer. The *context* of a microprocessor is the set of registers needed to execute instructions.

## Layer 2

The *communication network* in layer 2 interconnects the processing elements. Some important topologies of communication networks are sketched in figure 2.



**Figure 2: Communication network topologies.**

A common and simple topology is a bus as can be seen in the top left of figure 2. Here all processing elements or *devices* are connected with a shared media called the *bus*. Only one device at a time can transmit on the bus but several devices can, however, receive, at the same time. The process of selecting the device that is allowed to transmit is called *arbitration*.

A more advanced structure is to use a *network switch*. The switch is a hardware device which connects a sub-set of the devices to each other making it possible for them to communicate to each other. This structure is called a *switched network* and it makes it possible for several devices to transmit and receive at the same time. The connections on the switch where devices connect are called *ports*. The switch can be rapidly re-configured to change the subsets of communicating devices. The re-configuration is triggered by the transmitted data or the connected devices. Hierarchies of network switches and busses can be built as illustrated in the bottom of figure 2.

A network where two devices are connected directly to each other is called a *point-to-point* network.

A common address space may be available, accessible using ordinary load and store instructions, and may be *coherent*, i.e., all micro-processors see the same memory contents whether caches are used or

not. The coherency is maintained by hardware, located at the memory and the caches, and traffic on the communication network. This traffic and the actions taken by the hardware is referred to as the *coherency protocol*. The coherency protocol is governed by the *memory consistency model*.

The memory consistency model describes when the result of memory write operations as performed by one microprocessor becomes visible to other microprocessors. Several memory consistency models have been defined. The most simple one is *sequential consistency* where, when issuing a memory write operation, the microprocessor waits until the write operation has completed before continuing with any other memory operation. When performing a memory read operation the microprocessor have to wait for the read to complete and also for the write operation whose value is returned by the read operation to complete. In essence, this means that memory operations are performed in the same order as in the program code.

The performance of the microprocessor and also the coherency protocol is often increased if memory operations are allowed to deviate from the program code order. A number of memory consistency models have been proposed which allow that. These are collectively called *loose* or *relaxed memory models*. These memory models allow memory operations to be reordered and some even let the microprocessor execute multiple memory operations simultaneously.

*Message communication* may be possible, i.e., it may be possible to transfer data between memory buffers associated with the processing elements using privileged operations.

The *data rate* for  $n$ -byte messages is the maximal sustainable amount of bytes per second that can be transferred between a pair of processing elements using messages of size  $n$  bytes. The *aggregate data rate* of a communication network is the total amount of bytes per second that can be simultaneously transferred between all pairs of processing elements. The bandwidth seen by a pair of processing elements may be limited by the traffic generated by other processing elements. This phenomenon is called *congestion*.

The *latency* for  $n$ -byte messages is the minimal time from the transmission of the first byte of the message from one processing element until the reception of the last byte in the message to another processing element. Congestion may cause the latency to rise.

The process of determining how a message is sent to processing element is called *routing*. This involves selecting the right networks or network switch ports to traverse. Routing can be performed by the network switches themselves.

### Layer 3

In a *shared memory parallel machine*, a single virtual address space can span several processing elements. Such a virtual address space and additional context information is referred to as a *process*.

In the address space of a process one or several individually scheduled *threads* are executing. The threads communicate using *shared variables* located in the common address space and are synchronized using explicit synchronization primitives. For the purpose of this thesis I assume that threads executing in different processes can only communicate using messages. The primitives used to send and receive messages also acts as synchronization primitives.

In short, the operations used for communication and synchronization between threads are:

**Table 1: Summary of methods for synchronization and communication.**

	Threads in the same process	Threads in different processes
Data communication	Shared variables	Messages
Synchronization	Explicit primitives	Implicit using message transfer primitives

The use of explicit primitives for synchronization allows very loose memory models such as the *release consistency* memory model. Release consistency divides the synchronization primitives into *acquire* operations, which are used to gain access to a shared resource, and *release* operations which are used to grant other threads access to a resource. The microprocessor waits at a release operation until all proceeding memory operations have completed. No memory operations succeeding an acquire operation is allowed to proceed unless the acquire operation has completed. This allows for very aggressive reordering of operations and also aggressive coherency protocols as the

results of write operations do not have to be visible to other microprocessors until a release operation have completed.

In a *message passing parallel machine* it is not possible to extend, in hardware, a virtual address space beyond a single processing element. This means that all threads belonging to a process are allocated to the same processing element and that communication between threads executing on different processing elements has to be done via messages.

A message passing parallel machine built from several different computers interconnected with a network is called a *cluster*.

#### **Layer 4**

This layer consists of the *compiler run-time systems, libraries* and high level language interfaces to operating system calls. The interfaces to libraries, run-time systems, and operating system calls are called *application programming interfaces, or APIs*. One example of a library is a *software distributed shared memory system*, software DSM system, which is a software library that makes it possible for two threads in different processes to communicate and synchronize as though they were executing in a single process. A parallel machine that contains a software DSM system is called a software DSM machine.

#### **Layer 5**

Layer 5 represents the *application*. An application is several related processes or threads cooperating to solve a single problem.

Shared memory parallel machines are typically programmed using multiple threads in a single process, see figure 3, while message passing parallel machines are typically programmed using multiple processes each having a single thread, see figure 4. The figures show an example using pseudo code where two threads update a variable, *x*. In figure 3, a mutex is used to ensure that only one thread at a time access the variable. After the variable is updated, each thread uses a barrier. The barrier is a synchronization primitive that suspends a thread until all other threads in the application have reached the barrier. At the barrier the threads are synchronized and after the barrier both threads can assume that the shared variable has been updated by both threads.

In figure 4, the same example is implemented using a message passing parallel machine. Here the processes do not have direct access to

variable  $x$  and so it is not shared. Instead the variable resides in the memory of the processing element where process B executes and process A must access it using messages. In this example, process A sends an update of variable  $x$  using a message to process B that applies the update. Both processes must know each other's identity and process B must collect updates from each and every process that may update the

Thread A	Thread B
shared mutex m; <i>/* a global mutex seen by both thread A and B*/</i>	shared int x; <i>/* shared by thread A and B */</i>
int i,j; <i>/* a piece of code */</i>	i int k; <i>/* some code */</i>
<b>acquire(m);</b>	<b>acquire(m);</b>
if (i<56)	x=x+k;
x=x+j;	<b>release(m);</b>
<b>release(m);</b>	<b>barrier();</b> <i>/* synchronize the threads */</i>
<b>barrier();</b> <i>/* synchronize the threads */</i>	<i>/* some other code */</i>
<i>/* some other piece of code */</i>	

**Figure 3: A pseudo-code example using a shared memory parallel machine.**

Process A	Process B
int i,j; <i>/* some piece of code */</i>	int x,k; <i>/* some code */</i>
{	x=x+k; <i>/* perform own update */</i>
int tmp_j=0;	{
if (i<56)	int tmp_j;
tmp_j = j;	<b>receive(A, &amp;tmp_j, sizeof(int));</b>
<b>send(B, &amp;tmp_j, sizeof(int));</b>	<i>/* wait for A to send the update of     the x variable */</i>
<i>/* send a message to B     with the update */</i>	x=x+tmp_j; <i>/* perform the update */</i>
}	}
<i>/* yet another piece of code */</i>	<i>/* yet another code */</i>

**Figure 4: A pseudo-code example using a message passing parallel machine. Messages both transfer data and act as synchronization primitives.**

variable. Also, as the primitives are blocking, process B must collect the updates in the correct order or else all processes will be blocked in message passing primitives.

Compared to sequential non-parallel programming this is a radically different way of programming. The example in figure 3 is much more similar to sequential programming and it is widely argued that shared memory parallel machines are much easier to program than message passing parallel machines [48].



### **3 Clusters in high performance computing**

Clusters have been used for some time to achieve high performance or fault tolerance. I will not discuss the fault tolerance aspects of clusters in this thesis but I will instead focus on the high performance aspects. There is a plethora of clusters being used today and in this chapter I will explain some of the reasons for using a cluster, the anatomy of a cluster, and discuss some applications for clusters.

The idea of clusters is not new and various forms of clusters have been used for decades. During the last decade, however, there has been a trend to build high performance machines from commodity off-the-shelf components. This trend has been made possible by the tremendous performance increase and cost decrease of personal computers, providing cheap computing resources, and the introduction of capable computer networks. Equally important, however, has been the development of supporting software and operating systems making it possible to run software, on personal computers, that was previously only feasible to run on mainframes or supercomputers.

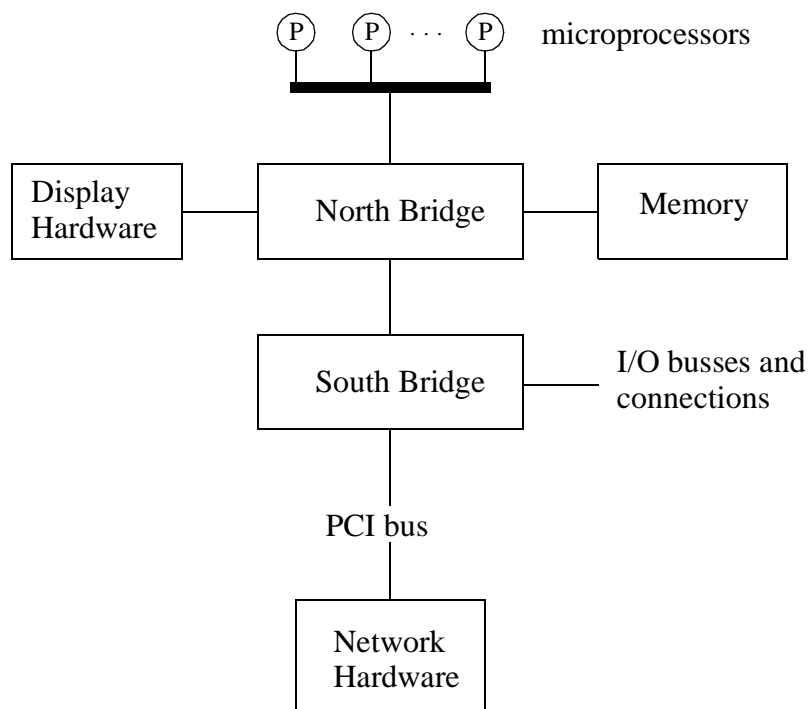
The kind of cluster I am going to discuss in this thesis consists of a set of processing elements, which normally are ordinary personal computers but could also be larger shared memory machines, interconnected with a network which normally is a dedicated and highly specialized network. This type of cluster is at the time of this writing very popular and the number of such clusters among the fastest computers in the world is staggering. 291 of the 500 most powerful computers, listed on the TOP500 list as published in June 2004, are clusters including the second most powerful computer in the world [54].

I am now going to go through the different parts of a cluster starting with the processing elements.

#### **3.1 The architecture of processing elements**

As mentioned, processing elements are normally ordinary personal computers or even larger shared memory machines. The idea is to use the type of machines which provides the most computing power at the lowest cost. The number of microprocessors per processing element is dictated by the price of available components. At the time of this writing the “sweet-spot”, i.e., the best price per performance, is at processing elements with two to four microprocessors. Personal computers are

sold at high volumes and so the cost is very low compared to machines made for server or supercomputing usage and this is one of the major reasons for building clusters. A typical processing element is designed as sketched in figure 5. The overview is based on the architecture of personal computers as of mid 2004. It is, however, applicable to recent workstations and even servers as these do make use of parts designed for personal computers and so the architecture of servers and workstations do not deviate much from personal computers.



**Figure 5: Schematical overview of a cluster processing element.**

As can be seen a number of microprocessors are connected through a bus. There is also the possibility of using a network such as machines using the recent Opteron or Athlon 64 processors from AMD [1]. These processors are interconnected by a point-to-point network, called HyperTransport, which is based on the PCI bus [21, 38]. Each

processor can include a number of levels of cache and there can be cache levels external to the microprocessors.

The microprocessors used are at the time of this writing capable of performing SPECint\_2000 and SPECfp\_2000 results of more than 1600, and are running at clock frequencies close to 4 GHz [50].

On the processor bus there is a device frequently called the *north bridge*. The north bridge's purpose is to connect the microprocessors to memory and to various I/O busses. It is also possible for processors to connect directly to memory as Opteron and Athlon 64 processors do. In this case, part of the functionality of the north bridge is integrated into the processor. The main advantage of this design is the possibility of scaling memory data rate with the number of processors. The memory data rate, i.e., the rate in which memory can be read or written, is at the time of this writing is close to 7 gigabyte/s.

For technical reasons the number of physical connections to the north bridge is limited making it not feasible to connect all I/O busses to the north bridge. Instead, a so called *south bridge* is used. The south bridge is connected to the north bridge with a dedicated point-to-point network. The south bridge then hosts the various I/O busses and connections while the north bridge manages the memory, processor bus, and a connection for graphics and display hardware which commonly is connected to the north bridge via an AGP bus or PCI express [23, 39].

The south bridge connects to hard drives, I/O busses and connections, and most importantly a PCI bus or a number of PCI Express connections. The PCI bus is important since the network interconnect hardware generally connects to the PCI bus.

PCI is an industry standard and is a synchronous bus with central arbitration [38]. The arbitration and data transfer phases on the PCI bus takes quite a few bus cycles so to attain reasonable data rates, devices on the PCI bus hold on to the ownership of the bus after getting access to it and performs several data transfers after each other. While this behavior increases the usable data rate of the bus it also increases the time for arbitration as any data transfer in progress has to be finished or aborted before control of the bus is handed over. This is further complicated by the fact that any of the devices participating in the access can add wait states, i.e., idle bus cycles, to the transfer procedure. In all a single bus transfer on the PCI bus can take several microseconds to

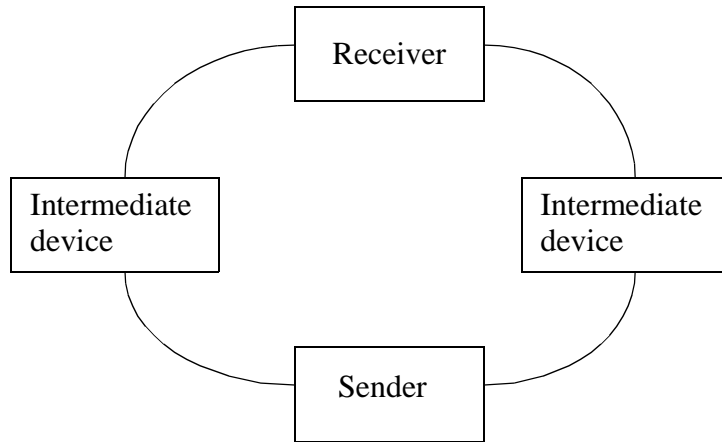
perform. The actual time, depend on the revision of PCI that is implemented. Low-end clusters normally use a 32-bit PCI bus running at 33 MHz where a total data rate of roughly 120 megabyte/s is attainable while high end systems use PCI-X which allows 64bit data transfers and bus frequencies up to 133MHz yielding data rates up to 1 gigabyte/s. A recent revision of PCI-X allows bus frequencies up to 533 MHz allowing data rates of over 4 gigabytes/s [40]. PCI Express is a new derivative of the PCI bus which, at the time of this writing, is starting to appear in off-the-shelf components. PCI Express does not make use of a bus structure. Instead, it uses point-to-point connections which removes some of the overhead of arbitration [39]. Several variants of PCI Express is available and data rates can exceed 3 gigabyte/s.

The PCI bus and its derivatives are all designed for high data rates and good bus utilization. Latency is not a primary focus and, although PCI Express does look promising in this respect, latencies are fairly high. To further complicate things, data has to traverse multiple busses until it reaches the PCI bus and the network interconnect which increases the latency.

All this means that the data rate available for transporting data to and from the processing element is, depending on the actual PCI bus version used, fairly high as well as the latencies involved. Traditionally clusters has been programmed using message passing where typically a high data rate is more important than low latencies. However, as we will see later in section 5.1, software DSM systems would benefit from low latencies and the design of such layers are complicated by the high latencies.

### **3.2 Network technologies**

There are three major network technologies used in clusters. The cheapest one, which also has the lowest performance, is ordinary Ethernet which is commonly used to connect personal computers to corporate networks or the Internet. Today Ethernet should be considered a low-end solution even if gigabit Ethernet or even 10 gigabit Ethernet is used. Ethernet is designed for general data communication and is not well suited for clusters. One of the reasons is the way data is transported on the network. The data is put into packets in a way that is suitable for transporting to any node in a very large network, normally the Internet, while in a cluster the number of processing elements is



**Figure 6: An SCI ring. The sender removes data from the ring while the receiver modifies data on the fly and intermediate devices forward data.**

fairly small and the way Ethernet lays out data will only increase the latency of routing.

There are however network technologies which are tailor-made for clusters but which also are more expensive. The two most important ones are SCI and Myrinet [11, 34]. Both of these put data into packets which are efficiently routed. SCI is a so-called *remote operation* network which means that it is possible to access memory in a remote processing element through the network and perform operations such as load, store, and even atomic fetch-and-add operations. The latency of such operations is typically in the range of 1 microsecond for write operations and up to 3-4 microseconds for read operations [16]. Several write operations can be simultaneously in transit which increases the data rate. With recent versions of SCI it is possible to achieve data rates exceeding 300 megabyte/s, for 66 MHz PCI busses, at latencies, as seen by the application, as small as 1.4 micro seconds with favorable PCI bus conditions. The SCI network is based on ring structures. The data is forwarded in each device until it reaches the transmitter where it will be removed. The receiver changes the data on the fly to indicate that it has received properly, see figure 6. In most cases no switches are needed although SCI switches are available. This because current SCI cards as those from Dolphin can participate in several rings at once and

very efficiently route packets on-the-fly from ring to ring [16]. It is thus possible to build elaborate structures such as a 2-D torus or even a 3-D torus without switches.

Myrinet is a more traditional network in that the normal programming API provides primitives for transmitting and receiving packets. What makes Myrinet stand out is, however, the ability to program the network hardware. A large part of the network handling is performed in software which is executed by a custom CPU which is on the Myrinet network card and it is possible to replace the software so as to add functionality. Doing that, it is possible to implement remote operations and even higher level operations such as lock operations, barrier operations, or even memory transfer operations. The drawback of the Myrinet hardware is its fairly high latency of roughly 3.5 microseconds for current network cards and small messages. The data rate goes up to 250 megabyte/s for 133 MHz PCI-X. The performance is thus slightly lower than current SCI cards and this is likely due to the software network handling as comparable Myrinet cards which use slower custom processors have significantly higher latencies. It is also likely that Myrinet has designed their cards using a more pessimistic model of the PCI bus latency where the overhead of the software can be overlapped with the PCI bus latency. Myrinet is a switched network with a central network switch.

Most high-end clusters use either SCI or Myrinet networks. Roughly 37% of the 500 most powerful computers in the world use Myrinet.

The attainable network latency and data rate is to large extent limited by the way the network card is connected to the processing element. The technology trend during the last decade has been that the attainable network latency has not changed very much while the data rate has risen but has not scaled in the same rapid pace as processor or even memory performance. Therefore, in quite a few clusters, the network is today the bottleneck and it is likely it will continue to be that for the foreseeable future.

### **3.3 System software**

As mentioned earlier the system software is a vital part of clusters as means for high performance computing. A major mile-stone for cluster system software was when the first MPI standard arrived in 1994 [32].

It was now possible to write reasonably portable applications that could run on several different supercomputers. Personal computers became, at about the same time, powerful enough to run Unix operating systems such as BSD or Linux and this spurred the development of several different cluster projects. One of the most well known projects is the Beowolf project which in 1994 produced a cluster for NASA [51].

Many clusters are now in use. Examples are the clusters for web page indexing run by Google and the, in June 2004, second fastest computer which is a cluster, located at Lawrence Livermore National Laboratory, with 1024 processing elements each having four Itanium2 processors.

Application areas for high performance clusters include automotive design, geophysics, weather and climate research, databases, and semiconductor development.

Clusters are message passing parallel machines by design and are typically programmed using message passing using PVM, MPI, or any other message passing system [8, 32]. This thesis is, however, concerned with making it possible to program a cluster as though it was a shared memory parallel machine.



## 4 Shared memory programming models

There are several proposed ways of programming a machine with shared memory. The different proposals are commonly called *programming models*. There are a few abstract definitions of what a programming model is. One definition is that a programming model is an abstract machine providing certain operations to the programming level. In practice, for the programming models I'm discussing in this chapter, this means that a shared memory is available and accessible using ordinary memory operations such as loads and stores, and that there exists primitives, as run-time library functions, macros or compiler constructs that allows for creation and synchronization of threads. The semantics of these primitives are different for the various programming models and in the next sections these differences will be discussed.

The programming models I'm going to discuss are the *POSIX thread API*, *ANL-macros*, and *OpenMP*. All of these use a thread-level fork-join programming model. I will use a simple vector addition example while describing the different models. A sequential version of the example in pseudo-code similar to C is in figure 7.

```
int A[10];
int B[10];          /* A and B are the two vectors that are to be added. */
int Result[10];    /* This vector holds the result. */

void VectorAdd(void)
{
    int i; /* i is the index variable. */
    for(i=0;i<10;i++)
    {
        Result[i]=A[i]+B[i]; /*Add each individual element. */
    }
}
```

**Figure 7: A simple vector addition example.**

I will use this example and present variants consistent with the different programming models.

## 4.1 The POSIX thread API

The POSIX thread API, or *pthread*s, as it is often called is a standard describing an operating system interface for multithreaded applications [22]. It is very general since it is supposed to be use by any kind of multithreaded application. It is usually implemented as a run-time library and the API itself consists of a number of C functions. There are functions to create threads, manipulate and destroy threads, synchronize threads using mutual exclusion, and also signal threads using condition variables. The standard defines variants of the synchronization primitives that avoids priority inversion and other anomalies. The POSIX thread API is often used for coarse grain multithreading where for instance threads are spawned to handle I/O, graphical user interfaces, and similar tasks. In such applications, threads are created or synchronized rather infrequently. The vector addition example in a parallel pthreads version is presented in figure 8. The example is simplified for brevity. Error handling is for instance omitted.

The POSIX thread API forces you to implement the parallel portions of your code as functions. These functions are then executed by the spawned threads. The functions take one pointer as argument and can return an integer. Threads are created with the `pthread_create` primitive which takes four parameters. The first is a pointer to a handle which is used to identify the spawned thread. The second parameter is a pointer to a structure which is used to set attributes such as stack size on the created thread. The third is a pointer to the function to be executed and the fourth is the argument to pass on to the function. I'm using the argument, in the example, to differentiate between the threads. Two threads are used in the example. It is relatively cumbersome to adapt a program based on pthreads to different levels of parallelism, i.e., different number of threads.

You can wait for spawned threads to finish using the `pthread_join` primitive which takes a pointer to the thread handle as parameter.

In the example, one thread is spawned and then the `ParallelFunction` function is called explicitly yielding two threads executing in parallel. The vector addition is performed in the function and the iteration space as executed by the two threads is adjusted so as to no iterations are executed twice. This has to be done by hand as pthreads do not provide any primitives to help work sharing.

```

int A[10];
int B[10];          /* A and B are the two vectors that are to be added. */
int Result[10];    /* This vector holds the result. */

void ParallelFunction(void * argument)
{
    /* The argument is used to differentiate between the threads. */
    int thread_id=(argument!=0);

    for(i=0+5*thread_id;i<5+5*thread_id;i++)
    {
        Result[i]=A[i]+B[i]; /* Add each individual element. */
    }
}

void VectorAdd(void)
{
    int i;          /* i is the index variable. */
    pthread_t thread1; /* thread1 is a handle for keeping track
                        of the spawned thread. */
    pthread_create(&thread1,(void *)0,&ParallelFunction,(void *) 0);
                        /* pthread_create spawns a thread. It takes several
                        arguments. Many of them are normally left unused. */
                        /* Now execute some work myself. */
    ParallelFunction((void *)1); /* 1 is passed as argument so that the function
                                can keep track of thread identities. */
    pthread_join(&thread1); /* pthread_join waits until the thread identified
                            by the handle has finished executing. */
}

```

**Figure 8: A simple parallel vector addition example using pthreads.**

## 4.2 ANL macros

ANL macros were developed at the Argonne National Laboratory to aid in developing parallel shared memory applications. Several revisions of the macros are currently available ranging from the original m4 package for HEP Fortran, to the variant used in the SPLASH benchmarks, and to p4 [31, 49]. I refer to these different packages collectively as the ANL macros.

The ANL macros are, in contrast to pthreads, designed to aid in high performance shared memory applications and provide a number of primitives which are useful to such applications. Such primitives include primitives for allocating shared memory, lock operations, barriers, and primitives for distributing for-loop iterations among threads.

The vector addition example in an ANL macro version is in figure 9. The example is again simplified for brevity. Error handling is omitted and the syntax of the macros are simplified.

A number of macros have been inserted into the example. MAIN\_ENV is used to set up variables that the ANL macro environment use internally. MAIN\_INITENV and MAIN\_END is used to initialize and shut down the ANL macro environment respectively. GSDEC is used to declare parallel for-loops while GSINIT initializes a declared loop. Iterations from a parallel for-loop is requested with GETSUB. Threads are created with CREATE and are joined with WAIT\_FOR\_END. Shared variables are explicitly allocated with G\_MALLOC and declarations of shared variables must thus be rewritten.

The parallel portions need to be rewritten into functions just like when using pthreads. The threads are explicitly spawned and joined, again just like with pthreads. There is however no need for micro management of threads and there is support for distributing the iterations in a for-loop among threads. This makes it easier to adapt the application to varying degrees of parallelism. The program will however still have to be in large parts rewritten when parallelized.

### **4.3 OpenMP**

One recent and very promising proposal is OpenMP which uses directives inserted into the source code [35, 36]. These directives can be used by a compiler to generate a parallel executable, see figure 10 for a OpenMP version of the vector addition example. The example is, in contrast to the pthreads and ANL macro examples, not simplified.

There is one directive in the example. The directive instructs the compiler to transform the following for loop into a parallel for-loop. It also instructs the compiler to spawn threads before the for-loop and join the spawned threads afterwards. The number of threads spawned is determined at runtime and the for-loop adapts to the number of threads. The use of OpenMP will thus, in comparison to pthreads and

```

MAIN_ENV      /* This is a macro that sets up global variables used by the
               rest of the macros and the environment. */

               /* You have to explicitly allocate shared memory when
               using the macros. The vectors are therefore replaced
               with pointers. */

int *A;
int *B;
int *Result;

GSDEC(loop);  /* This macro declares the data structures needed to
               implement a parallel loop. */

void ParallelFunction(void ) /* Note that the ParallelFunction function does
                             not take any parameters. */
{
    int i;          /* i is the index variable. */
    while(GETSUB(loop,i,9,2),i>=0) /* The GETSUB macro requests a new
                                    iteration to perform. */
    {
        Result[i]=A[i]+B[i]; /* Add each individual element. */
    }
}

void VectorAdd(void)
{
    MAIN_INITENV(,80000000); /* This macro sets up the environment. */
    GSINIT(loop); /* This macro initializes the parallel loop. */
    A=(int *) G_MALLOC(10*sizeof(int)); /* Allocate memory for the shared
                                         variables. */
    B=(int *) G_MALLOC(10*sizeof(int));
    Result=(int *) G_MALLOC(10*sizeof(int));
    CREATE(ParallelFunction); /* Spawn one thread */
    ParallelFunction(); /* Do some work myself. */
    WAIT_FOR_END(1) /* Wait for one spawned thread to finish. */
    MAIN_END; /* This macro shuts down the environment. */
}

```

**Figure 9: A simple parallel vector addition example using ANL macros.**

```

int A[10];
int B[10];          /* A and B are the two vectors that are to be added. */
int Result[10];    /* This vector holds the result. */

void VectorAdd(void)
{
    int i;          /* i is the index variable. */

    /* The following line is an OpenMP directive. It instructs the compiler to rewrite
       the following for statement into a parallel for statement which is executed by
       several threads. It also instructs the compiler to spawn threads before the for
       statement and join them afterwards. A, B, and Result is treated as shared
       variables. */
    #pragma omp parallel for shared(A,B,Result)
                          /* The index variable i is private by default */
    for(i=0;i<10;i++)
    {
        Result[i]=A[i]+B[i]; /* Add each individual element. */
    }
}

```

**Figure 10: A simple parallel vector addition example using OpenMP.**

ANL macros, make it possible for the programmer to avoid restructuring of the code.

OpenMP directives exist for creation, coordination, and synchronization of threads. The directives allow for both loop-level and functional parallelism. Data clauses that describe whether variables are shared or private can be added to the directives. In addition to the directives, the OpenMP specification describes a number of run-time library functions that for instance can be used to inquire the number of currently running threads.

One interesting feature of OpenMP is that if all directives are removed from a well-written program the resulting program would still be a valid, although sequential, program. The directives are in fact designed so that properly written OpenMP programs can be compiled with non-OpenMP compilers and the resulting executable is a sequential program.

OpenMP differs from previous shared memory programming standards in the use of compiler directives which makes it possible to do

advanced compiler optimizations where the compiler restructures the program to better suit the target platform. This opens up for interesting research topics and in fact some of the studies proposed in this thesis deal with compiler optimizations. The programmer do not have to restructure the code when parallelizing which leads to a more rapid application development.

The first OpenMP specification was released in 1997 by an industry consortia and was largely based on a previous standardization effort called X3H5 [35]. There has since been several revisions to the specifications and the current version is 2.0 which is available both for the Fortran, C, and C++ languages [36].

Several compiler vendors were part of the specification process and rather soon after the release of the specification, commercial OpenMP compilation systems were available from several vendors. An OpenMP compilation system typically consists of a compiler and a run-time library. Research compilation systems, although rather few, were soon to follow the commercial systems one of them is the compilation system developed as part of this thesis and supported by the Intone project [6, 25].

The research compilers are normally source-to-source compilers, i.e., they take source code as input and generates source code as output. This makes them fairly architecture independent and thus portable [5, 12, 25, 43].

Vendor compilers on the other hand generate object code directly and because of that optimize the code more efficiently.



## 5 Software distributed shared memory systems

The history of software DSM systems is now well over a decade long. This chapter highlights the most important research advances during that time.

### 5.1 The pioneer

IVY, Integrated shared Virtual memory at Yale, designed by Kai Li, was the first software DSM system described in literature [30]. It maintained coherency on a virtual memory page grain size using only software and the virtual memory system. Systems like IVY are called *page-based software DSM systems*.

With IVY, Li opened up the research field and identified several important research topics that still are being investigated. Most importantly it was found that:

- The average memory latency in a software DSM machine — which is mainly determined by the communication network latency — is the main performance bottleneck. When a shared variable is found to be located in another processing element's memory, it must be requested. This is done by a handler similar to a virtual memory page fault handler that sends messages and receives messages over the communication network. The application itself is not aware of this but sees an increased latency for certain memory accesses.
- The experiments on IVY showed that virtual memory page sizes larger than 1 kilobyte caused congestion in the communication network. In fact, the performance of the coherency protocol is limited by the size of the virtual memory page.

The majority of the research conducted since IVY have consisted of proposals for new coherency protocols and memory consistency models which reduce the risk of communication network congestion and which try to hide the memory latency. The main reasons for this are that modern microprocessors can only support virtual memory page sizes equal to or larger than 4 kilobyte and that the performance of microprocessors have increased much more rapidly than the performance of communication networks.

## 5.2 The breakthrough

IVY spurred several research projects. One of them was Munin [14] which used several different coherency protocols and let the programmer decide which protocol to use with each shared variable. This was done by inserting variable type modifiers into the source code. For a listing of the available variable types, see table 2.

**Table 2: Types of shared variables in Munin.**

Variable type	Description
Read-only	The variable is only read after initialization.
Migratory	The variable is read and written by one thread and later read and written by another thread. Only one thread at a time access the variable.
Write-shared	Several threads are concurrently writing to different parts of the variable.
Producer-consumer	The variable is written by one thread and read by another.
Reduction	The variable is updated using atomic fetch-and-update primitives.
Result	The variable is updated in two phases. In the first phase several threads update the variable followed by a phase where one single thread is exclusively accessing the variable.
Conventional	The variable is of no specific type. IVY's coherency protocol is used.

Munin also supported a release consistency memory model and the use of release consistency enabled much more aggressive hiding of memory latency and nearly all software DSM systems have since then supported release consistency [18], or even weaker memory consistency models. IVY, in contrast, supported only sequential consistency [28] which means that applications assume that modifications to a shared variable immediately become visible to the other processing elements. The multiple coherency protocols helped Munin to be very efficient. However, its main disadvantage is that the programmer must annotate each shared variable in the source code to get optimum appli-

cation performance. This can be very cumbersome in a large application.

The research on Munin led to the most well known system, TreadMarks, that is arguably the first really usable software DSM system [2, 27]. TreadMarks introduced a version of the release consistency memory model called *lazy release consistency* [27]. Lazy release consistency means that modifications to shared variables are not conveyed to other processing elements unless those processing elements, subsequent to the modification, actually access the variable. The main advantage of supporting lazy release consistency instead of release consistency is that the communication network traffic is drastically reduced. Thus, the risk of congestion in the communication network is reduced.

By using lazy release consistency TreadMarks did not need to use multiple coherency protocols to yield good application performance. This greatly simplified the task of writing applications. TreadMarks is designed as a software library that is linked with the application. The TreadMarks library provides synchronization primitives and means to allocate memory blocks that are shared by all threads. The application is augmented with calls to synchronization primitives and calls to initialization routines that initialize the library. Because of the performance and ease of use, TreadMarks gained widespread acceptance.

Since TreadMarks, the research on page-based software DSM systems has, with some exceptions, focused on enhancing lazy release consistency with aggressive latency hiding techniques such as prefetching, and extending the idea of multiple protocols that Munin introduced. Several so called *adaptive protocols* have been developed that automatically, guided by a set of heuristics, choose a coherency protocol at runtime for each shared variable or virtual memory page [3, 4, 33].

Research has also been conducted on *home-based lazy release consistency* where each virtual memory page is assigned to a processing element [56]. Home-based lazy release consistency simplifies the design of the coherency protocols and also reduces the memory footprint needed to implement the protocols in software. The performance difference between systems based on home-based lazy release consistency and ordinary lazy release consistency is small [15].

All the systems up to TreadMarks assumed that there is only one microprocessor per processing element. As computers equipped with several microprocessors have become increasingly more common the research on software DSM systems has shifted to systems for machines whose processing elements consists of multiple microprocessors [42, 52].

### **5.3 Non page-based systems**

It is not necessary to base a software DSM system on the virtual memory system. Some systems, such as Blizzard-S [46], DSZOOM [41], and Shasta [45], instead use instrumentation of load and store instructions to check state and to maintain coherency on a smaller grain size than a virtual memory page. This instrumentation is done automatically in the compiled executable code of the application. In essence, each access to a shared variable is wrapped with code that checks whether a copy of the variable corresponding to the access is present in the memory of the processing element or must be requested from another processing element.

The advantage of this approach is that the coherency protocols are not limited to using virtual memory pages but can use much smaller memory objects. This yields a more efficient use of the communication network and better application performance. The disadvantage is that it must be possible to rewrite the application object code to use this technique. This is only practical on certain microprocessor architectures and so this technique has not become widespread. In addition, the instrumentation add additional microprocessor instructions which means that the instrumented version of an application have a longer execution time on a single processor than a not instrumented version. The difference in execution time can be quite significant.

### **5.4 Hardware hybrids**

During the last few years several interesting proposals have been made where techniques originating from software DSM machines are used to either make the hardware for maintaining coherency in shared memory parallel machines more simple or to increase the performance of such machines. An example of this is Simple COMA [44] where basic support for the memory coherency is maintained in hardware while high level functions in the coherency protocol is implemented in software.

Simple COMA is a practical approach for implementing a COMA [19] protocol in a shared memory parallel machine and the Wildfire machines are one commercial example of simple COMA [20].

Another commercial example is the Origin 2000 in which techniques from software DSM systems are used to automatically move data at runtime between processing elements as to increase the performance of the running application [29].



## 6 Summary of contributions

This thesis consists of several studies. These studies and their contributions are briefly summarized in the following sections.

### 6.1 A study of message size distributions

It is known that the performance of an application running on a software DSM machine is lower than when it runs on a message passing machine, and that the bandwidth of most communication networks is higher for larger messages than for smaller. In the study of paper **I**, it was found that the relative communication time in the software DSM machine is higher than in the message passing machine, and that this difference increases with the number of processing elements. A decreasing fraction of small messages was found with increasing system size in the message passing machine but not in the software DSM case.

Although a lot of research on coherency protocols for software DSM system have been conducted, very little has been known about the achievable performance of a software DSM machine given an ideal coherency protocol. The performance depends very much on the usage of the communication network and so studying the behavior of a software DSM machine and a corresponding message passing machine would give valuable insight into the performance aspects of software DSM machines.

In the study the most important parameters measured and analyzed were the message sizes and the time when the transmissions of messages take place. These parameters were measured for the execution of message passing and shared memory versions of scientific benchmarks with different communication behavior. In addition, the execution time and its breakdown into, i.e., time spent communicating and busy time, was measured.

The message passing parallel machine used MPI [32] and the software DSM machine used TreadMarks. Both machines are based on the same hardware, an IBM SP2, which is optimized for message communication. Three scientific computing benchmarks from the SPLASH [49] and NAS [7] benchmark suites were used: Barnes-Hut, Water, and FFT.

It was found that the message passing versions outperform the software DSM versions when the number of processing element increases. The performance of the shared memory applications running on top of the software DSM system is nevertheless comparable with the corresponding message passing versions when the number of used processing elements is small, i.e., up to eight processing elements. One reason for this is a difference in the network traffic patterns.

The study showed that the processing elements running message passing based applications communicates using fewer and larger messages than a corresponding version running on a software DSM machine causing a shorter communication time. The difference in network traffic patterns becomes even more pronounced when the number of processing elements increases leading to the difference in measured performance. For instance, the fraction of small messages, caused by applications running on the software DSM machine, remains approximately constant when doubling the number of processing elements while in the message passing versions the fraction of small messages is approximately reduced by 50%.

Two approaches for increasing the performance of shared memory applications on software DSM systems were suggested in paper **I**. Most importantly, work should be focused on altering the coherency protocols so that the fraction of small messages is reduced. Also, the performance would increase if the latency of messages sent on the communication network is reduced.

## **6.2 A proposal for an OpenMP capable software DSM system**

Even though the research on software DSM systems has matured there are issues remaining. Paper **II** discussed the shortcomings of software DSM systems as of 1999 and proposed solutions.

One of the most important issues with the software DSM systems was the use of non standard APIs and programming models. The state-of-the art of that time was TreadMarks which used its own API [2]. The use of OpenMP as a unifying programming model was suggested [35, 36].

It was observed that SMP nodes, i.e, processing elements with multiple microprocessors, had become common and cost efficient. It was

stated that future software DSM systems should handle processing elements with multiple microprocessors efficiently due to this.

Stability, network performance, and network management were pinpointed as areas where effort should also be spent. A system where monitoring daemons observe the activity on the processing elements and also intervene and communicate with the processes was suggested in the paper. This, thus, making it possible for the user to monitor and control the execution of the applications. The use of, and integration of support for, batch systems into the software DSM systems was also suggested as means for gaining higher acceptance among end users as monitoring systems and batch systems had been part of cluster system software for some time.

It was discussed, in the paper, that the support for high performance communication networks in software DSM systems was lacking. The implementation of the coherency protocols was often intermixed with the code for handling the network which led to systems that could not easily be ported to new network technologies. A special network library, called *Balder Messages*, was proposed and introduced as a solution. Balder Messages provides a network technology independent interface suitable to software DSM systems. The reasons for it being suitable and an overview of the library were discussed.

It was also observed, in the paper, that the use of the network resources were not optimal making it not possible to use the full data rate of the networks. It was suggested in the paper that effort is spent on changing the network traffic patterns of the software DSM systems.

### **6.3 Latency hiding and network traffic pattern restructuring**

In a software DSM system when a variable in another process is accessed a time consuming message procedure will occur. The objective of the study in paper **III** was to measure the effect on the communication time of a pushing strategy based on the assumption that shared variables in a loop are accessed equally each time and using the associated synchronization primitives as runtime markers. For a set of test applications the communication time was reduced in average by 70%.

In a software DSM system, shared variables are fetched from the processing element that produced the most recent value when it is not present in the local memory upon a memory access. This high-latency

operation can be hidden if the data is moved ahead of time, before it is needed. Prefetching techniques, both manual and automatic, have been proposed but they do not alter the basic request-reply scheme and generally requires a larger bandwidth on the network.

An alternative approach, investigated in paper **III**, is to let the producer send shared data to presumed users before it is requested. It is thus possible to avoid requesting shared data, which takes valuable execution time, without sending more data on the communication network.

The technique, called producer-push, described in this paper is automatic and based on the repetitive behavior of the communication in iterative scientific and engineering applications. It predicts the communication patterns of repetitive applications and sends data from the processing element that update shared data, the *producer* to the presumed user of the data, the *consumer*.

Producer-push is applicable to page-based software DSM system where consumers requests data corresponding to an entire virtual memory page. Slightly altered, however, it would be possible to apply producer-push also to shared memory parallel machines.

Producers in page-base software DSM systems do not respond to requests of shared data with the contents of an entire virtual memory page. Instead, only information regarding the most recent updates is sent. These updates are called *diffs*. The system has a global notion of time and each processing node knows when it last received a diff for each of the virtual memory pages. When data is requested, this time-stamp information is sent with the request so that the producer can provide the correct diff.

The prediction of communication patterns is performed using a set of heuristic algorithms:

- For each incoming data request, the memory address of the corresponding virtual memory page is stored and is appended to a linked list structure associated with the most recently executed synchronization primitive. Each consumer has its own linked list.

- When a synchronization primitive is executed, the associated linked lists are checked. If a list is not empty, diffs corresponding to the stored requests are sent, i.e., *pushed*, to the consumer corresponding to the list. Several diffs are pushed in one single message if possible.
- The consumer stores each of the pushed diffs in a cache. Whenever the consumer needs an update it checks the cache to see if there is a corresponding diff in the cache. If a cached diff is present it is immediately applied and a request need not be issued.

When using producer-push, the consumer do not need to request shared data if a corresponding diff is already pushed to it. The time spent waiting for the producer to respond is thus avoided. In contrast to prefetching [9, 24], several small request messages are eliminated when using producer-push and the corresponding replies can be aggregated into a single message and a restructuring of the network traffic pattern is achieved. The result is an increased fraction of large messages which means that the communication network can be used more efficiently which is not the case when prefetching is used.

Previous suggestions for techniques that proactively send data, e.g. adaptive protocols, either require source code modifications to the application, which producer-push do not require, or they use too indiscriminate heuristics that yield excessive usage of the available bandwidth [27, 47].

In order to quantitatively measure the performance of producer-push, an implementation using TreadMarks was made. Several commonly used benchmarks were run on an IBM SP2 using both the original TreadMarks and the producer-push enhanced version. The benchmarks used were IS, FFT, CG, and MG from the NAS benchmark suite [7] and Water and Barnes-Hut from the SPASH suite [49].

It was found that the fraction of the execution time spent waiting for diffs is effectively reduced by using producer-push. The average reduction was 74%. This resulted in a average increase of speedup, when using 8 processing elements each containing one processor, of 23%. In no case did producer-push cause any performance degradation.

The accuracy of the prediction of communication patterns was also determined. It was found that producer push can predict on the average 79% of the data requests and on the average over 85% of the pushed diffs were used by consumers.

The amount of data sent on the communication network were also observed. It was found that producer-push indeed reduces the number of messages sent and that the average message size is increased. Even though not all diffs can be predicted the total communication network data rate used by the applications was only increased with on average 4% when producer-push was used.

## 6.4 Using priorities to reduce latencies

Some software DSM systems make no difference between synchronization and data messages. The purpose of the study in paper **IV** was to evaluate the performance consequences of reducing synchronization latency by introducing priorities in the message buffers maintained by the software DSM system. It was shown that the latency of synchronization messages is effectively reduced regardless of network design parameters when priorities are used and if the messages are delayed by data messages.

In a software DSM machine, there are two types of coherency messages: *Data messages* that contain memory data and *control messages* that are used for synchronization and exchange of coherency information. These messages are sent and received in FIFO order. For some latency hiding techniques, such as producer-push, and certain coherency protocols, the control messages get delayed behind data messages in the FIFO buffers [10]. This leads, in turn, to a higher latency of synchronization primitives which has a negative effect the execution time of applications executing on top of the software DSM system.

In paper **IV**, it was proposed to let the control messages bypass the FIFO. Each message is assigned a priority and high priority messages are transferred ahead of lower priority messages. This functionality has been implemented into a portable software communication library called Balder Messages which is designed to support a software DSM system. Balder Messages have a small set of primitives capable of the most common types of message based communication. All primitives operate on a linked-list based structure called *chunk lists* where each link corresponds to a memory block. The use of chunk lists provides a way to efficiently use DMA engines in the communication network.

Balder Messages implements flow control to not overload the communication network and the mechanisms that enforce priorities among messages are implemented into the flow control algorithms. It is found

that the added complexity to the flow control algorithms is negligible. The main differences between Balder Messages and other communication libraries, such as Fast Messages [37], VMMC [17], and PM [53], are the use of priorities and the portability.

The communication library was evaluated using a cluster of Pentium II nodes interconnected with a 100 Mbit ethernet. The cluster was running a synthetic benchmark on top of a Unix operating system. The evaluation shows that by using priorities the impact on the latency on synchronization messages caused by other messages is very effectively reduced.

A quantitative model was developed as a tool to investigate the impact of priorities on a wide range of communication network technologies. It was shown that priorities has a large impact regardless of network technology. The latency of control messages is reduced with up to 25% per each delaying message if priorities are used. The impact is greater when there is a mismatch between the latency and data rate of the communication network. Generally speaking, the impact of priorities depends more on the bandwidth than on the latency of the communication network and also depend on the size of the delaying messages.

## **6.5 Implementing OpenMP on Software DSM**

An OpenMP compliant software DSM system has many benefits which would attract users of high performance clusters. The objective of paper V was to describe a fully compliant OpenMP 1.0 prototype implementation on software DSM and to evaluate its performance. The performance, as evaluated using a set of benchmarks from the SPLASH-2 benchmark suite [55], was found to be competitive and comparable to a commercial OpenMP system running on a shared memory machine.

The software DSM system was built on-top of an existing software DSM system called CVM [26]. The software DSM system was using home-based lazy release consistency due to its robustness and simplicity. The OpenMP implementation was the first complete OpenMP 1.0 cluster implementation without any limitations.

The software DSM system was targeted by an OpenMP compiler, OdinMP, which was enhanced to target clusters as well as SMPs. The enhancements were in two areas both related to shared variables:

- The compiler was altered so that it explicitly allocated all global shared variables and replaced all accesses to such variables to accesses through pointers.
- Variables located on the stack of the thread executing the serial portions of the code, the master thread, can be made shared. The compiler was augmented to move all shared variables located on the master thread's stack into a single shared stack located in the shared memory.

Significant effort was put into reducing the amount of network traffic in the cluster as caused by coherency operations. In OpenMP the flush directive is the basis for consistency. OpenMP flushes can be performed on a set of variables or the entire memory space. The software DSM system in paper V differentiates between these two types. It implements flushes using a scheme similar to distributed locks and piggy-backed coherency information onto an imaginary lock which is acquired and then released during a flush operation. Doing this reduces the amount of network traffic, especially in the case when the flush is performed on a set of variables. It was shown in the paper, using the EPCC micro-benchmarks [13], that by carefully using the flush implementation a number of OpenMP constructs could be implemented efficiently.

The entire compilation system was evaluated on an IBM SP2 machine using FFT, LU and Water-spatial from the SPLASH-2 benchmark suite [55]. It was compared to MIPSpro, a commercial OpenMP compiler whose executable output were run on an SGI Origin 3800. The relative speedups, as compared to sequential versions of the benchmarks, on 8 processors ranged from 4.1 to 6.5 for the software DSM system. The speedups for the executables generated by the MIPSpro compiler was 6.8 to 7.8 for 8 processors.

## **6.6 An OpenMP source to source compiler for C and C++**

OpenMP relies on an OpenMP aware compiler to do code transformations so as to produce a parallel program. There are however few efficient open source OpenMP compiler systems suitable for research. Paper VI presented such a compiler system for C and C++, showed the code transformations that are performed, and evaluated the perfor-

mance of the generated code. The efficiency of code generated by the compiler was shown in the paper to be good and the performance of the generated code to be par with commercial vendor compilation systems for a wide range of benchmark applications.

The compilation system presented in paper **VI** consists of a compiler and a run-time library. The functionality of the run-time library was outlined and an overview of the compiler was discussed. The compiler is a source-to-source compiler written in C++. The source code of the compiler is highly portable and available under a very liberal licence.

Some limitations to the compiler were listed in the paper. These limitations have been removed since the paper was written and, as of the time of this being written, a new version of OdinMP, the compiler, is under preparation which supports all of OpenMP 2.0 including so called threadprivate variables, and volatile variables [36]. The language support is also improved to fully support ANSI C99.

It was shown, in the paper, how a few different OpenMP directives such as the parallel directive, the for directive, and some synchronization directives are transformed. It was also described how storage attributes are handled.

The performance of the compilation system was evaluated using the EPCC micro-benchmarks and OpenMP versions of Barnes, Cholesky, FFT, LU, and Ocean from the SPLASH-2 benchmark suite [13, 55]. The system was evaluated on an Intel based machine running Linux and an SGI Origin 3800 running IRIX.

The data gathered with the micro-benchmarks showed that the compilation system produces code with about the same, or better, overheads than code generated by the SGI MIPSpro 7.3 and the Intel C/C++ compiler version 8.0 compilers. The only exception was the overhead of parallel for-loops where the vendor compilers can do better optimized code since they generate object code directly.

The applications from the SLASH-2 benchmark suite were run on the SGI machine and compared to versions compiled with the SGI MIPSpro compiler. It was shown that the performance of the code generated by the presented compilation system was very close to code generated by the vendor compiler and even better than the vendor compiler in a few cases.

The performance of the generated code when running on one processor was very close to the performance of a purely sequential version. The performance difference was in one case 3% but generally less than 1%.

## 6.7 An OpenMP run-time library for clusters

A compilation system for OpenMP typically consists of a compiler and a run-time library. Paper **VII** described a portable run-time for OpenMP called Balder. The run-time can handle both clusters and individual processing elements. The design of the library and the functionality provided by the primitives in the library were discussed in the paper. The performance of the library was found to be competitive when compared to a commercial compilation system.

The presented run-time library complements the compiler presented in paper **VI** and forms a compilation system with the compiler. The design of the run-time library is modular to help porting efforts and the library has already been ported to several different types of operating systems and microprocessors. The design of the library was discussed in the paper. Internally, the library has several sub-libraries which are used to hide operating system and architecture specific features from the rest of the library.

OpenMP 2.0 is supported and the library provides the compiler with a number of primitives that the compiler can use to implement OpenMP constructs [36]. The paper presented the functionality of some of these primitives and their implementations on clusters were discussed. Most of the functionality outlined in paper **II** is provided.

The run-time library has a built-in software DSM system based on home-based lazy release consistency to provide a shared address space on a cluster. The software DSM system was briefly described in the paper and the transformations performed by the compiler to handle shared variables were presented.

Two planned advanced features under implementation in Balder were also described in the paper. One of the features is an extension to the coherency protocol which makes it possible to achieve coherency on a smaller grain size through the use of compiler inserted coherency checks.

The other advanced feature is support for a set of compiler directives that describes the data sharing patterns. The information in the

directives is passed on to the run-time library through code generated by the compiler. The information is then used by the run-time to proactively send data between processing elements and perform message passing so as to increase the performance of the coherency protocols.

The run-time library has been evaluated using the EPCC micro-benchmarks on a dual-Pentium III machine [13]. The overheads of common OpenMP constructs as measured by the benchmarks were found to be very competitive when compared to the Intel C/C++ compiler version 8.0.



## 7 Concluding remarks

The work I have performed is rather well summarized in the title of the thesis. I have investigated the performance aspects of software DSM systems and have tried to improve on the performance with a few different technologies. A large part of the thesis is devoted to OpenMP compilers, OpenMP run-time libraries and methods for executing OpenMP applications on clusters.

To recap, the main contributions in the thesis are:

- An in-depth study of the message communication in software DSM systems and message passing systems indicating that one major contributing factor to the performance difference between applications running on software DSM systems and message passing applications is the use of the communication network. The software DSM systems tend to use many and very small messages which will not utilize the network efficiently. Message passing applications tend to use fewer but larger messages which suits high performance networks better.
- A pushing technique which, by predicting the communication patterns of iterative applications, can proactively send data before it is used and can therefore significantly increase the performance of software DSM systems.
- A critique of software DSM systems as of 1999 and a proposal for a software DSM system infrastructure based on OpenMP.
- A messaging library for software DSM systems and an investigation of the use of priorities to improve the latency of time-critical messages.
- An OpenMP compiler for C and C++ which is very competitive when compared to commercial vendor compilers and can generate code that, given a suitable run-time library, can be executed on a cluster.
- A prototype OpenMP run-time library for clusters of uni-processor processing elements.
- A more mature OpenMP run-time library that can handle clusters of multi-processors and is competitive when compared to commercial systems.

The efficient use of the communication network is the key to achieve good performance on clusters and so a major part of this thesis is devoted to studies and techniques aimed at optimizing the network utilization.

The compiler has also become very important with the acceptance of OpenMP as a standard programming model. Compiler technology is part of several of the papers in this thesis.

It is my belief that the work I have done in the areas of software DSM system, OpenMP compilers, and OpenMP run-time libraries has advanced the current state-of-the-art in those areas and with that I am happy.

I see several directions for future work. Even though OpenMP can be used on clusters, there is still work to do in defining programming models to be used to program clusters efficiently. More and more shared memory parallel machines are organized hierarchically yielding radically different memory access latency for different microprocessors. This is very much the same as when software DSM systems are used on clusters of multi-processors. This non-uniformity in memory access latencies is something that I believe needs to be addressed in the programming models. This effort can be focused on entirely new programming models or, as I think is very much possible, extensions to OpenMP. Part of this effort is compilers, compiler transformations, and compiler optimizations which provide interesting problems.

There is still work to be done in the areas of fault tolerance and system management for clusters. I have not touched on fault tolerance at all in this thesis but, as the size of new clusters tend to increase, the probability of one or even several processing elements failing during execution of larger applications is increasing. I believe fault tolerance and availability issues will be increasingly important in high performance computing clusters.

The increasing interest in grid computing has introduced some interesting problems. Grids are very large systems where large supercomputers or clusters are interconnected and the problems outlined above become even more accentuated. The area of computational grids have several interesting problems and is likely to be in the research focus for quite some time.

## References

- [1] A. Ahmed et. al., Hammer Shared Memory Multi Processor Systems, in *Proceedings of Hot Chips 14*, August 2002
- [2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, Vol. 29, no. 2, pp. 18-28, February 1996
- [3] C. Amza, A.L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel, Adaptive Protocols for Software Distributed Shared Memory, in *Proceedings of IEEE, Special Issue on Distributed Shared Memory*, Vol. 87, No. 3, March 1999, pp. 467-475
- [4] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel, Software DSM Protocols that Adapt between Single Writer and Multiple Writer, in *Proceedings of the third High Performance Computer Architecture Conference*, February 1997, pp. 261-271
- [5] E. Ayguadé et. al., NanosCompiler: A Research platform for OpenMP Extensions, in *Proceedings of First European Workshop on OpenMP*, Sept. 1999
- [6] E. Ayguadé, M. Brorsson, H. Brunst, H.-C. Hoppe, S. Karlsson, X. Martorell, W. E. Nagel, F. Schlimbach, G. Utrera and M. Winkler. OpenMP Performance Analysis Approach in the INTONE Project, In *Proceedings of EWOMP'01*, September 2001
- [7] D. Bailey, T. Harris, W. Saphir, R vd Wijngaart, A. Woo, and M. Yarrow, *The NAS Parallel Benchmarks 2.0*, Report NAS-95-020, Nasa Ames Research Center, Moffett Field, Ca, 94035, USA. December 1995
- [8] A. Beguelin, J. Dongarra, A. Geist, S. Otto, J. Walpole, PVM: Experiences, Current Status and Future Direction, in *the Supercomputing '93 Proceedings*, pp. 765-766, November 1993
- [9] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs, In *Proceedings of the 7th ACM/IEEE International Conference on Architectural Sup-*

*port for Programming Languages and Operating Systems (ASP-LOS 7)*, October 1996, pp. 198-209

- [10] A. Bilas, L. Iftode, J. P. Singh, Evaluation of Hardware Write Propagation Support for Next-Generation Shared Virtual Memory Clusters, in *Proceedings of 1998 International Conference on Supercomputing*, Orlando, November 1998, pp. 274-281
- [11] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29--36, Feb. 1995
- [12] C. Brunschen and M. Brorsson, OdinMP/CCp - a portable implementation of OpenMP for C, *Concurrency: Practice and Experience*, 2000; 12:1193-1203
- [13] J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, in *Proceedings of the First European Workshop on OpenMP*, Sept. 1999, pp. 99-105
- [14] J. Carter, J. Bennett, and W. Zwaenepoel, Implementation and Performance of Munin, in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991, pp. 152-164
- [15] A. Cox, E. de Lara, Y. Hu, and W. Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory. in *Proceedings of the Fifth International Symposium on High Performance Computer Architecture (HPCA '99)*, Feb 1999
- [16] Dolphin Interconnect Solutions Inc., *PCI-SCI Adapter Card D320/D321 Functional Overview*, version 1.01, November 1999
- [17] Dubnicki et al., Design and Implementation of Virtual Memory-Mapped Communication on Myrinet, in *Proceedings of 1997 International Parallel Processing Symposium*, Los Alamitos, CA., 1997, pp. 388-396
- [18] K. Gharachorloo, et. al., Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, in *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, Washington, May 1990, pp. 15-26

- [19] E. Hagersten. *Toward Scalable Cache-Only Memory Architectures*, PhD Thesis, Royal Institute of Technology, Stockholm/Swedish Institute of Computer Science, 1992
- [20] E. Hagersten, M. Koster, WildFire: A Scalable Path for SMPs, in *Proceedings of the fifth International Symposium on High Performance Computer Architecture*, Orlando 1999, pp. 172-181
- [21] HyperTransport Consortium, *HyperTransport technology I/O Link Specification*, 2004
- [22] IEEE, *IEEE std 1003.1-1996 POSIX part 1: System Application Programming Interface*, 1996
- [23] Intel Corporation, *AGP v3.0 Interface Specification*, 2003
- [24] M. Karlsson and P. Stenström, Evaluation of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems, in *Journal of Parallel and Distributed Computing*, vol. 43, no. 7, July 1997, pp.79-93
- [25] S. Karlsson, *OdinMP homepage*, <http://www.odinmp.com>, retrieved on August 8th 2004
- [26] P. Keleher, *The CVM Manual*, Technical report, Computer Science Department, University of Maryland, May 1995
- [27] P. Keleher, *Lazy Release Consistency for Distributed Shared Memory*, PhD Thesis, Rice University, 1994
- [28] L. Lamport, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers* 28(9), 1979, pp. 690-691
- [29] J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server. in *Proceedings of the International Symposium on Computer Architecture*, Denver, June 1997, pp. 241-251
- [30] K. Li and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM trans. on Computer Systems* 7(4), November 1989, pp. 321-359
- [31] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, Inc., 1987
- [32] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1, June 12, 1995

- [33] L. R. Monnerat and R. Bianchini, Efficiently Adapting to Sharing Patterns in Software DSMs, in *Proceedings of the 4th IEEE International Symposium on High-Performance Computer Architecture (HPCA98)*, Feb 1998, pp. 289-299
- [34] K. Omang and B. Parady, Scalability of SCI Workstation Clusters, a Preliminary Study. in *Proceedings of the 11th IEEE International Parallel Processing Symposium (IPPS'97)*, Geneve, April 1997, pp. 750-755
- [35] OpenMP Architecture Review Board, *OpenMP specification, C/C++ version 1.0*, October 1998
- [36] OpenMP Architecture Review Board, *OpenMP specification, C/C++ version 2.0*, March 2002
- [37] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. in *Proceedings of Supercomputing '95*, San Diego, California, December 1995
- [38] PCI-SIG, *Conventional PCI V2.3 specification*, 2002
- [39] PCI-SIG, *PCI Express specifications*, 2004
- [40] PCI-SIG, *PCI-X V2.0 specification*, 2002
- [41] Z. Radovic and E. Hagersten, Removing the overhead from software-based shared memory, in *Proceedings of Supercomputing 2001*, November 2001
- [42] R. Samanta, A. Bilas, L. Iftode, and J.P. Singh, Home-based SVM protocols for SMP clusters: Design and performance, in *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, Nevada, January 1998, pp. 113-124
- [43] D. M. Sato, Design of OpenMP Compiler for an SMP Cluster, in *Proceedings of First European Workshop on OpenMP*, Sept. 1999
- [44] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin, An Argument For Simple COMA, in *Proceeding of the first IEEE Symposium on High Performance Computer Architecture*, Rayleigh, North Carolina, January 1995, pp. 276-285
- [45] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-

Grain Shared Memory, in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996, pp. pages 174-185

- [46] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, Fine-grain Access Control for Distributed Shared Memory, in *Proceedings of the sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, November 1994, pp. 297-307
- [47] B Seidel, R. Bianchini, and C. L Amorim, The Affinity Entry Consistency Protocol, in *Proceedings of the International Conference on Parallel Processing*, August 1997, pp. 208-217
- [48] J. P. Singh, A. Gupta, and M. Levoy, Parallel Visualization Algorithms and their Implications for Multiprocessor Architecture, *IEEE Computer*, special issue on Visualization, vol. 27, no. 6, June 1994
- [49] J. P. Singh, W.-D. Weber, and A. Gupta, SPLASH: Stanford parallel applications for shared-memory, *Computer Architecture News* 20(1), March 1992, pp. 5-44
- [50] Standard Performance Evaluation Corporation, SPEC homepage, <http://www.spec.org>, retrieved on August 6th 2004
- [51] T. Sterling, D. Becker, D. Savarese, et al. BEOWULF: A Parallel Workstation for Scientific Computation, in *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, August 1995, Vol. 1, pp. 11-14
- [52] R. Stets et al., CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network, in *Proceedings of SOSP '97*, Saint Malo, France, October 1997, pp 170-182
- [53] H. Tezuka, A. Hori, and Y. Ishikawa, *PM: a High-Performance Communication Library for Multi-user Parallel Environments*, Technical Report TR-96015, Real World Computing Partnership, 1996
- [54] *Top 500 Supercomputer sites*, June 2004, <http://www.top500.org/lists/2004/06/>, retrieved on August 6th 2004

- [55] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations, in *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 24-36
- [56] Y. Zhou, L. Iftode, and K. Li., Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. in *Proceedings of the 2nd Operating Systems Design and Implementation Symposium*, Oct. 1996

S. Karlsson and M. Brorsson, A Comparative Characterization of Message Communication in Applications using MPI and Shared Memory on an IBM SP2, in *Proceedings of 1998 Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Las Vegas, January 31 - February 1, 1998, pp. 189-201

© Copyright 1998 Springer Verlag, All rights reserved. Reprinted with permission.



# A Comparative Characterization of Communication Patterns in Applications using MPI and Shared Memory on an IBM SP2

Sven Karlsson and Mats Brorsson

Department of Information Technology,  
Lund University, P.O. Box 118, SE-221 00 LUND, Sweden  
email: Sven.Karlsson@it.lth.se

**Abstract.** In this paper we analyze the characteristics of communication in three different applications, FFT, Barnes and Water, on an IBM SP2. We contrast the communication using two different programming models: message-passing, MPI, and shared memory, represented by a state-of-the-art distributed virtual shared memory package, TreadMarks. We show that while communication time and busy times are comparable for small systems, the communication patterns are fundamentally different leading to poor performance for TreadMarks-based applications when the number of processors increase. This is due to the request/reply technique used in TreadMarks that results in a large fraction of very small messages. However, if the application can be tuned to reduce the impact of small message communication it is possible to achieve acceptable performance at least up to 32 nodes. Our measurements also show that TreadMarks programs tend to cause a more even network load compared to MPI programs.

## 1 Introduction

It has recently been quite popular to investigate the use of networks of workstations (NOWs) as parallel computing platforms with the motivation that they should be more cost-effective than multiprocessors with highly customized hardware and software. Since there is no hardware support in a NOW for shared memory, the programming model of choice has been message-passing. While message-passing is natural for some problems, it can be very cumbersome to use for applications with irregular communications patterns and many programmers feel that the shared memory programming model is a more natural approach to parallel programming.

The choice of programming model is, however, not merely a choice of convenience. It can also affect performance as well. Message-passing programs might suffer from extra calculations and memory overhead to partition the data in a regular manner in order to facilitate the programming of communication. Shared memory programs might suffer from the separation of synchronization and data

communication and most importantly, in the case of a NOW, the lack of hardware support for shared memory. It is therefore important to understand the performance trade-offs in choosing one programming model over the other.

This paper characterizes the communication patterns for three different applications on the IBM SP2 coded in two different programming models: message-passing using MPI and shared memory using the TreadMarks DVSM package [1]. For this purpose, the IBM SP2 multiprocessor can be seen as a network of rs6000 workstations, albeit with a specialized and high-performance network switch. Compared to related work that compare these two programming models on platforms that do not support shared memory in hardware [4], we have used more processors and concentrate more directly on the characterization of communication patterns in the two programming paradigms.

To summarize, we have found that by using TreadMarks, there is a more even load on the network compared to the MPI versions of the applications which tend to have peaks of communication demanding very high bandwidth. On the other hand, TreadMarks inherent request/reply nature results in at least 50% very small messages leading to poor end-to-end bandwidth utilization. Moreover, when the number of processors increase, the MPI programs tend to result in a smaller fraction small messages while the situation is the opposite for TreadMarks further reducing the performance of TreadMarks.

The rest of the paper is organized as follows: Section 2 presents the two programming models used in the study. This is followed by a description of the three applications and the differences in communication. In Sect. 4 we present results from measurements on an IBM SP2 and discusses the results before we conclude in Sect. 5.

## 2 MPI and Distributed Virtual Shared Memory

Different programming models will lead to different communication patterns even if the basic algorithm of the application is the same. We have used MPI (message passing) and TreadMarks (shared memory) and compare the resulting communication patterns.

### 2.1 MPI

The Message Passing Interface (MPI) standard was defined in a concerted effort by both high-performance computing vendors and research institutes [5]. The standard defines a variety of point-to-point send and receive primitives as well some collective communication primitives that involves more than two processing nodes. Table 1 lists the MPI routines used in the applications studied in this paper. In addition to the primitives listed in Table 1, there are primitives to find out the number of participating processors, and the rank, i.e. identity, of the local processor.

MPI primitive	Application	Meaning
MPI_SEND	Barnes	Standard point-to-point send of messages. Does not synchronize with the receiver.
MPI_RECV	Barnes	Standard blocking receive of message from a named sending node. When the program returns from this call, a message has been received.
MPI_BARRIER	FFT	Blocks execution until all processors have called MPI_BARRIER. Does not communicate data but is used to synchronize the program.
MPI_BCAST	Barnes, Water	A message is distributed from one processor to all other processors.
MPI_SCATTER	Water	A root node distributes its data across all the other nodes.
MPI_ALLGATHER	Water, FFT	All nodes fetches data from each node and stores it in a local array.
MPI_REDUCE	FFT, Barnes, Water	All nodes send a piece of data to the root node that performs a reduce operation, e.g. min., max or sum, on the collected data.
MPI_ALLREDUCE	Water	The same as MPI_REDUCE except that the result is distributed to all participating processors.

**Table 1.** MPI communication primitives used in the applications in this study.

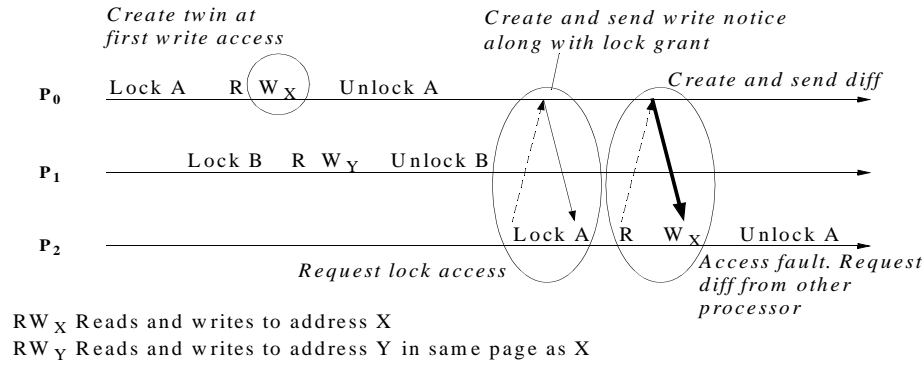
## 2.2 Lazy Release Consistency shared memory

The other programming model we use for the SP2 is shared memory with the Lazy Release Consistency (LRC) memory model as found in TreadMarks [1]. In shared memory multiprocessors, communication occur implicitly based on what data is modified and when. Synchronization primitives such as locks for mutual exclusion and barriers are often used to ensure that only one processor at a time is modifying a particular shared data object.

The IBM SP2 does not implement shared memory in hardware and we therefore use TreadMarks, a Distributed Shared Virtual Memory (DVSM) system, to provide a shared memory programming model in software [1], using the existing virtual memory management system. Compared to a multiprocessor that implement shared memory in hardware, a DVSM system has a very high overhead to maintain a coherent shared memory. In order to overlap some of the shared memory communication with computation, TreadMarks uses a relaxed memory consistency model with a multiple-writer, invalidate protocol.

The main idea behind relaxed memory models is that write operations need not be conveyed to other processors immediately and that subsequent memory operations may bypass outstanding writes in order to increase the performance [3]. Since most shared memory programs are data-race free, i.e., updates to shared data are protected by critical regions, e.g. locks-unlocks, it is sufficient

if information on updates are propagated to the other processors at the next synchronization operation. In TreadMarks this memory model has been further improved by delaying the time at which other processors are notified of memory updates until a processor acquires the lock that was used to protect the shared data. At the time of lock acquisition, all pages modified by the previous lock holder are invalidated. However, the update of the actual page contents is deferred to the time when it is actually needed.



**Fig. 1.** The Lazy Release Consistency protocol in TreadMarks defers the propagation of consistency information to the latest possible moment.

Figure 1 shows a scenario that exemplifies how TreadMarks maintains consistency. Processors 0 and 2 take turn in accessing a variable  $X$  protected by lock A. When processor 0 first modifies  $X$ , a copy of the page, in which this variable is located, is created. This copy is called a *twin*. The twin is later used to make an encoding, called a *diff*, of the changes made. The release of lock A will not result in any communication as long as no other processor has requested the lock. When processor 2 acquires the lock, it sends a message to the last holder of the lock (via a lock manager) and requests information on which pages the previous lock holder has modified. Processor 0 responds with a write notice for each modified page along with the lock grant message. Processor 2 invalidates these pages by removing the access privileges. Processor 2 will take an access fault at the first access to the page in which  $X$  is located. It then requests the changes, the *diffs*, from processor 0, the previous holder of lock A.

Note that during these operations, other processors can very well be accessing other locations within the same page as long as these memory locations do not overlap with  $X$  and they are protected by some other lock, e.g. lock B. This multiple-writer protocol effectively removes the issue of false-sharing from TreadMarks unless the same synchronization variable is used to protect an entire page.

The barrier operation, which is an important shared memory synchronization operation, can with respect to memory consistency be modeled as all processors performing a release followed by an acquire at the exit of the barrier. The effect is that all processors become updated on all changes made by all other processors and it is therefore a rather communication intensive operation.

Let us now describe the applications that we have used to characterize the differences in communication between MPI and Shared memory versions of the same algorithms.

### 3 Applications

Three different benchmark applications, that all are part of the TreadMarks application distribution, were used in the study. Barnes-Hut and Water originally from the SPLASH benchmark suite [7] and 3D FFT from the NAS benchmarks [2]. The MPI versions were written based on the corresponding TreadMarks versions.

#### 3.1 Barnes-Hut

This application is a simulation of the motion of bodies interacting with gravitational forces. The bodies are hierarchically ordered in a tree where the internal nodes are called cells and represents collections of bodies in relatively close physical proximity. The algorithm loops over a number of time steps, in which the acceleration, velocity and position of each body are updated and the tree is reconstructed. Some system-wide parameters, e.g., total energy and system boundaries, are also calculated in each time step. All processors must have access to all of the database since the gravitational forces from all bodies on each body must be evaluated.

The TreadMarks version is directly taken from the TreadMarks distribution and uses shared memory for the bodies and the cells. At the beginning of each time step the root processor builds the tree and each processor processes its part of the bodies. The databases and system parameters are distributed using shared memory and the processors are synchronized using barriers.

The MPI version was developed from the shared memory version. It uses exactly the same algorithm but the root processor broadcasts the bodies and tree at the start of the time step and the slave processors send the updated bodies back to the root processor. The system wide parameters are collected by the root processor with reduction operations.

#### 3.2 Water

Water simulates the dynamics of water molecules. Both intra- and inter molecule forces are evaluated. Similarly to Barnes-Hut this means that all processors must be able to read all of the database. However, unlike Barnes-Hut each processor

not only update its own molecules but also those of others. In the original TreadMarks version the race hazards are solved by using one lock per molecule. This approach proved to yield too much communication when directly ported to MPI so instead a different method was used. Each processor has a private copy of its modifications of all of the molecules. At the end of the time step all processor's "diffs" are merged and applied. This optimization proved to be effective in reducing the communication needed in the MPI version and it was therefore also used in the TreadMarks version used in this study. The TreadMarks version became more than twice as fast as a unmodified version when running on two nodes. When running on more nodes the difference was even higher due to the reduced communication.

The algorithm uses a couple of small global reduction operations in each time step. It also uses one large global sum for the "diffs" and a broadcast for the distribution of the database. The MPI version uses the specialized MPI operations for reduction etc. while the TreadMarks version uses shared memory protected by barriers and locks.

### 3.3 3D FFT

This application is a PDE solver using a three dimensional FFT. It solves the PDE by first doing a forward FFT and then stepping in time by updating the frequency space and doing inverse FFT back into the time space. The FFT and inverse FFT are done by allocating planes of the cube to the processors and then performing two one-dimensional transforms along the two axis in the plane. The cube is then transposed and a third one-dimensional transform is performed. The transpose is the main communication problem. It is solved in the TreadMarks version by simply using the shared memory and copying while in the MPI version each processor needs to build a partially transposed cube which is then broadcast to the other processors.

## 4 Results

### 4.1 Experimental test-bed

All measurements have been done on an IBM SP2 with 110 nodes using IBM's native MPI library and TreadMarks v0.10.1. TreadMarks was modified so that profiling data could be collected during program execution. The network switch of the SP2 has an end-to-end latency of about 40  $\mu$ s using MPI [6] and measurements have shown that the corresponding UDP/IP latency is about 200  $\mu$ s.

The programs were compiled using AIX's own C compiler using identical compile switches. They were executed on 1,2,4,8,16 and 32 node configurations. Two runs were done for each program, with and without profiling. The run without profiling measured the execution time while the profiling run gathered execution statistics, see below. Two runs had to be performed since the profiling code induces some overhead.

Sequential execution time, together with workload parameters, are listed in Table 2. The sequential execution time is the execution time of the parallel version running on a single processor and is the same for both the MPI and the TreadMarks version of the applications. The measured time does not include the startup phase since the timing is started after the first iteration. This to ensure a true speedup is calculated, i.e. the speedup of the parallel section of the application.

The MPI versions were profiled using IBM’s own trace format and the TreadMarks versions by profiling code inserted into TreadMarks itself. Unless otherwise stated, all profiling data were collected from all of the execution phases, i.e. even the startup phase. During the profiling runs communication and processor usage statistics were obtained. Profiling was done at the MPI API layer for the MPI versions and at the TreadMarks API layer and at the UDP/IP socket API layer for the TreadMarks versions. The execution times of all MPI and TreadMarks functions respective were measured as well as all socket read and write calls in the TreadMarks library. Using these data the execution time of the applications could be broken down into busy time, communication time, i.e. time spent sending or waiting for messages, and time spent in the TreadMarks library, i.e. time spent ensuring consistency.

Application	Workload	Time (s)
Barnes-Hut	65536 bodies, 6 time steps	279
Water	1728 molecules, 5 time steps.	400
3D FFT	64 * 64 * 64 Cube, 64 iterations	54

**Table 2.** Application workloads and sequential execution times

## 4.2 Overall performance characterization

Figure 2 shows the speedup of the MPI and TreadMarks versions of the applications. As seen, there is a great variation in speedup among the applications and implementations. In general, the MPI versions do relatively well in terms of speedup, while the TreadMarks versions of FFT and Barnes perform very poorly for large configurations. Let us examine the communication characteristics in more detail.

Figure 3 displays the execution times of the applications subdivided into *busy time*, *communication time* and *TreadMarks time*. The busy time for both MPI and TreadMarks programs is the fraction spent executing useful code in the algorithm. The Communication time in the TreadMarks versions is the time spent blocking in socket calls while for the MPI programs it is the time spent in MPI calls. The TreadMarks time is the time spent inside the TreadMarks

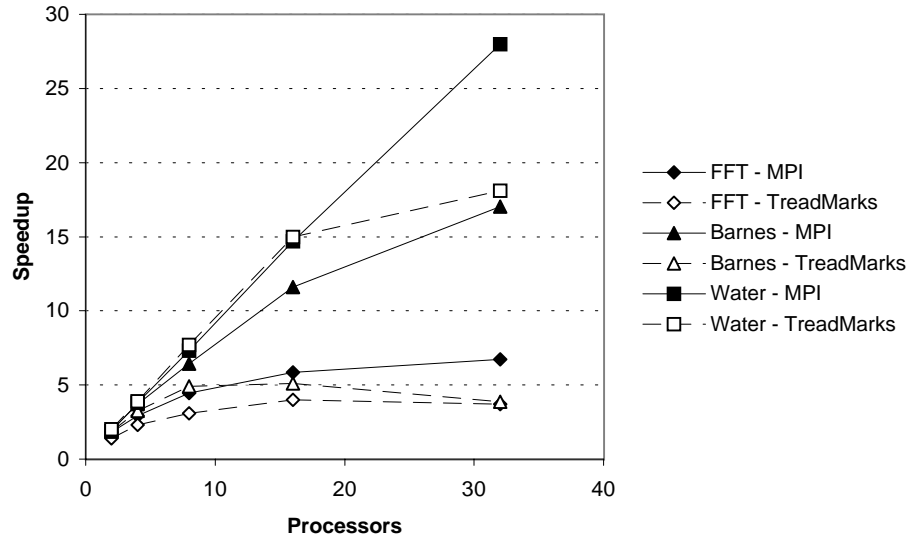


Fig. 2. Application speedups.

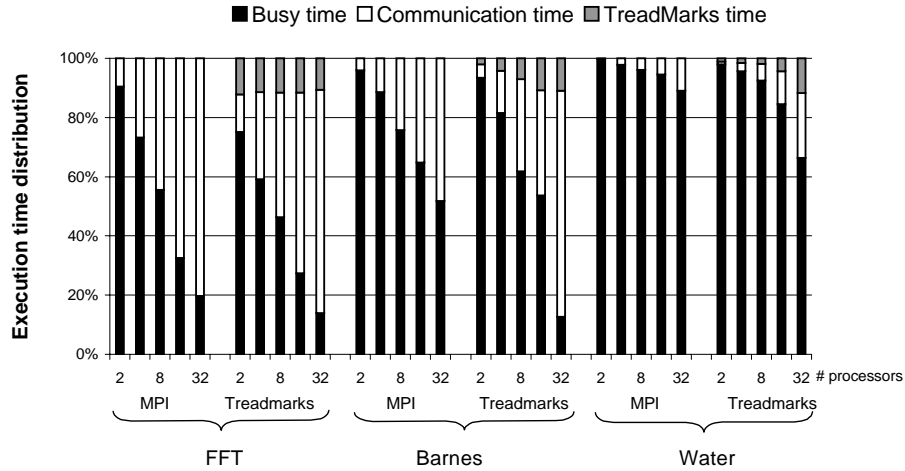
library itself, i.e. everything that is not in the algorithm and not in the socket communication protocol.

Figure 3 shows that the relative importance of the communication time for FFT and Barnes increases with the number of processors. In fact, it also increases in absolute time. Water, on the other hand, which has been tuned to aggregate changes, displays a behavior for the TreadMarks version which is much more similar to the MPI version.

Tables 3 and 4 show the aggregate number of messages and the total amount of data communicated in the MPI and TreadMarks versions of the applications respectively. The MPI versions generally communicate more data than the corresponding TreadMarks versions. However, since this is done in fewer but larger messages, the overall effect is a better speedup.

no. procs.	FFT			Barnes			Water		
	Mbytes sent	Msgs sent	kbytes/message	Mbytes sent	Msgs sent	kbytes/message	Mbytes sent	Msgs sent	kbytes/message
2	260.0	194	1372.4	68.76	108	652.0	14.0	108	174.9
4	780.0	972	821.7	178.0	324	562.5	38.8	286	139.0
8	1820.0	4088	455.9	381.2	756	516.3	88.0	934	96.5
16	3900.0	16560	241.2	781.7	1620	494.1	186.2	3190	59.8
32	8060.0	66464	124.2	1579.7	3348	483.1	382.6	11542	34.0

Table 3. MPI program communication characteristics



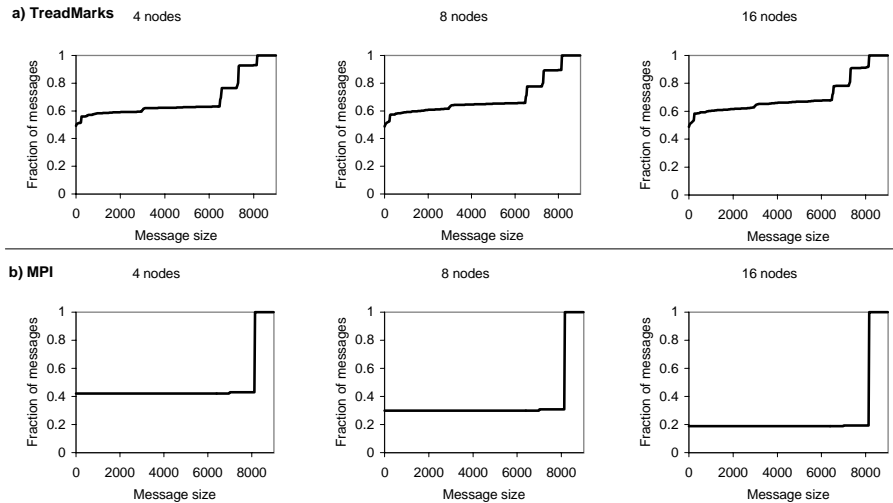
**Fig. 3.** Execution time distribution for the different applications.

no. procs.	FFT			Barnes			Water		
	Mbytes sent	Msgs sent	kbytes/message	Mbytes sent	Msgs sent	kbytes/message	Mbytes sent	Msgs sent	kbytes/message
2	136.5	35478	3.9	47.5	20508	2.4	10.7	3572	3.1
4	208.3	55536	3.8	141.4	91570	1.6	22.8	8381	2.8
8	246.5	71582	3.5	282.5	275965	1.0	47.8	18256	2.7
16	523.2	165960	3.2	533.9	856078	0.6	100.5	39936	2.6
32	1088.8	402504	2.8	976.4	2502602	0.4	222.4	92283	2.5

**Table 4.** TreadMarks program communication characteristics

### 4.3 Results and discussion

We will now further investigate the reasons to poor speedup for applications using TreadMarks. From the communication statistics collected we have built histograms on the size of the messages sent during the course of execution. These show that TreadMarks distribution of message sizes differ radically from MPI's. MPI's distribution is polarized, i.e. the messages sent are either very small or very large, while TreadMarks' are much more evenly distributed. A typical distribution is shown in Fig. 4 for Water. The figure shows the cumulative distribution of the fraction of sent messages. The last value, corresponding to a message size of 8192 bytes, in the distribution is the fraction of messages larger than or equal to 8 k-bytes.



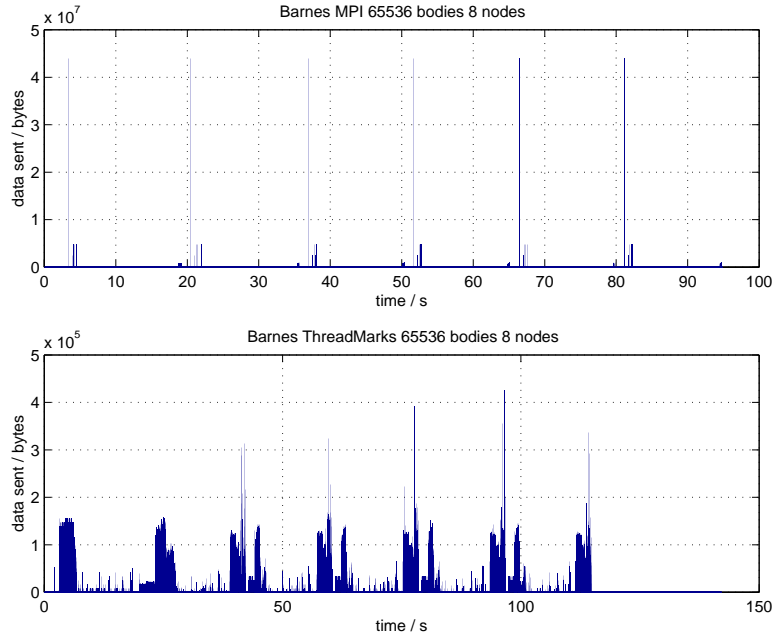
**Fig. 4.** The distribution of message sizes for Water, a) TreadMarks, b) MPI.

A general trend that we have observed for all three applications is that the fraction of small messages tends to decrease for the MPI versions when the number of processors increases, see Fig. 4. Although difficult to discern from Fig. 4, the opposite situation, i.e. the fraction of smaller messages increases, tends to be the case for the TreadMarks versions. This will of course further reduce the performance of TreadMarks as the number of processors increases. There are a number of reasons to this difference in message size distribution. First there is a difference in how the message size is measured. The communication data for MPI is collected at the MPI API layer which means that no acknowledgment messages that MPI itself might need are shown. Since TreadMarks' communication data is collected at the socket layer all TreadMarks' messages are shown. Thus the fraction of small messages in the MPI's distributions could in fact be higher! By running the FFT application on the free MPI implementation MPICH we have seen that each MPI message produces roughly 3 to 4 network messages. These additional messages are acknowledge and control messages. However, since IBM's native MPI implementation is supported by the hardware in the network interfaces of the SP2 nodes, it should produce fewer explicit control and acknowledgment messages than MPICH.

Secondly, while an MPI version of a program can use arbitrarily large messages and thus optimize the program to a given network, a TreadMarks program relies on the TreadMarks system for communication. Since TreadMarks uses the virtual memory system, the maximum message size is limited by the virtual memory page size. This can partly be alleviated by using larger page sizes in the TreadMarks code [4], but the optimal page size is application dependent and not always easy to estimate.

Furthermore, due to the request/reply technique used in TreadMarks at least 50% of the total number of messages are small, i.e., requests.

Interestingly enough, the overhead of TreadMarks LRC protocol itself is relatively small for Water and Barnes (up to eight nodes, see Fig. 3). For FFT it is much more significant. A closer examination reveals that the diffs in FFT's case are three times as large as those in Barnes and Water. The diff generation and application is the single largest component in TreadMarks overhead.



**Fig. 5.** Message traffic vs time for Barnes. Note the difference in scale on the y-axis.

We have also measured the network load induced by the different applications. A typical example of how many bytes sent per time interval is shown for Barnes in Fig 5. The other applications displayed similar characteristics. Note that MPI sends up to 100 times as much data as TreadMarks per time interval. Also note the more even distribution of messages caused by TreadMarks as opposed to MPI's discrete distribution. The difference in the distribution of communication over time indicates that TreadMarks might do better than MPI on a network with limited bandwidth but low latency.

## 5 Conclusions

We have studied and compared the communication patterns of three applications under two different programming models executing on the IBM SP2, message-passing using MPI and shared memory using TreadMarks. The results show that, for the applications studied, both programming models can be used with acceptable performance using up to eight processors. However, for large configurations, the fraction of small messages inherent in TreadMarks leads to poor performance.

TreadMarks causes a more even distribution of communication over time which suggests that it is not as prone to congestion in the network as are MPI applications. However, for the type of network used in the SP2, where the bandwidth is not a problem, and if the message latency is still high the message overhead will outweigh the potential benefits from less bandwidth demand.

We can therefore conclude that in order to improve the performance of a DVSM system work should be done to reduce the latency of the network system, decrease the number of sent packets and increase the size of sent network packets. This can be done by using network drivers that implement more functionality in hardware and by incorporating prefetch and producer push techniques into the DVSM protocols.

## 6 Acknowledgments

The research in this paper was in part supported by the Swedish National Board for Industrial and Technical Development (NUTEK) under project number P855. It was also supported with computing resources by the Swedish council for High Performance Computing (HPDR) and Center for High Performance Computing (PDC), Royal Institute of Technology.

## References

1. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel.: *Tread Marks: Shared Memory Computing on Networks of Workstations*, IEEE Computer, Vol. 29, no. 2, pp. 18- 28, February 1996.
2. D. Bailey, T. Harris, W. Saphir, R vd Wijngaart, A. Woo, and M. Yarrow, *The NAS Parallel Benchmarks 2.0*, Report NAS-95-020, Nasa Ames Research Center, Moffett Field, Ca, 94035, USA. December, 1995.
3. K. Gharachorloo, D. E. Lenoski, J. P. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 15-26, May 1990.
4. 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, June 1997.
5. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1, June 12, 1995.

6. J. Miguel, A. Arruabarrena, R. Beivide and J. A. Gregorio, *Assessing the Performance of the New IBM SP2 Communication Subsystem*, IEEE Parallel & Distributed Technology, Winter 1996, pp. 12-22.
7. J. P. Singh, W.-D. Weber, and A. Gupta. *SPLASH: Stanford parallel applications for shared-memory*. Computer Architecture News, 20(1):5-44, March 1992.



S. Karlsson and M. Brorsson, An Infrastructure for Portable and Efficient Software DSM, in *Proceedings of 1st Workshop on Software Distributed Shared Memory (WSDSM '99)*, Rhodes, Greece, June 25, 1999, also available from Department of Information Technology, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden





# An Infrastructure for Portable and Efficient Software DSM

Sven Karlsson and Mats Brorsson  
Department of Information Technology,  
Lund University, P.O. Box 118,  
SE-221 00 LUND, Sweden  
phone: +46-46-222 7579  
Sven.Karlsson@it.lth.se

## 1. ABSTRACT

**In this paper, some of the weaknesses of the current software DSM systems are identified. Among these weaknesses are the lack of user friendly network management functions, limited performance on high performance networks, limited portability, and no standard APIs. OpenMP is proposed as a standard API for future software DSM systems and the network management subsystem and network layer of the Balder system is described as an example of an infrastructure of a future DSM system.**

### 1.1 Keywords

Software DSM systems, Network Protocols, Network Management

## 2. INTRODUCTION

Recent research on software DSM systems [4] has shown that these systems can provide decent performance even when compared to hardware based cc-NUMA machines. Furthermore, software DSM systems yield good price/performance as systems can be built from commodity parts such as network interfaces and workstations or PCs.

In this paper we describe the management and network infrastructure we have chosen for a new Software DSM system, called Balder, which is in development at Lund University, Sweden. The ideas and views expressed are based on years of experience conducting performance studies and development work on existing software DSM systems such as TreadMarks [1] and CVM [6].

There are many evidences that software DSM has the potential to provide a simple programming model for parallel computing on distributed memory systems without hardware support for shared memory. One of the main weaknesses of current software DSM efforts lies in that the

programming interface does not conform to any standard and it is our firm belief that there is a future for software DSM systems only if we can base them on standards such as OpenMP [11]. While this is probably a requirement for acceptance, an OpenMP compiler can also make more extensive analyses of the dataflow in the parallel regions than an ordinary compiler due to the OpenMP directives. The information obtained can be used to further tune the software DSM system.

Furthermore, in order to gain broad acceptance, a great deal of work must be done in the areas of stability, network performance, and network management. Current software DSM systems have not paid attention to these areas enough. As cheap Intel-based SMPs recently have become common, we also feel that future software DSM system should be targeted to SMP nodes.

In Section 3 we describe some of the problems of the current software DSM systems. We describe how the Balder system solves some of these problems in Section 4 and in Section 5 we end the report with some conclusions.

## 3. SOFTWARE DSM SYSTEMS

For several years the dominant state-of-the-art software DSM system for research has been TreadMarks. While decent performance can be achieved on a wide range of applications, TreadMarks has some limitations as described below. These limitations are shared by many other software DSM systems.

TreadMarks does not have any network management support. It is hard to retrieve status regarding a specific node in a running TreadMarks systems. Message passing systems such as PVM [3], LAM [9] etc. are in this respect much more advanced. Since TreadMarks run on loosely coupled systems where hardware or software failures are not uncommon, it is vital to be able to monitor the network. Secondly, in TreadMarks there are no distinct interfaces between the network handling code and the consistency protocol handling code. This means in practice that it is very cumbersome to implement support for new networks or even to optimize the consistency protocols since these two are moulded together. In the end this means that in order to optimize performance on a specific network one will have to rewrite a great deal of the consistency handling routines. This is not only a waste of time, but it is also error prone and it leads to code that is difficult to maintain. It should,

however, be noted that there has been some recent work on network independent protocols, for instance the GeNIMA system [4].

There are no public software DSM systems, known to the authors, that have network management functions. There is some work going on in the field of fault tolerance though, for instance by the Brazos [15] team. Some software DSM systems like Cashmere-2L [16] and Shasta [14] utilize high performance networks like Memory Channel. However, most systems only support UDP or TCP sockets. Shasta and Cashmere-2L are very hard to port as they rely on the remote write feature of Memory Channel.

Finally, we can also note that the performance of TreadMarks on high performance networks is limited. The typical message size in TreadMarks is 2-3 kilobytes which leads to only 63% of the asymptotical bandwidth on a high-performance network such as SCI, see Figure 1. In order to obtain good bandwidth on high-performance networks, it is often necessary to increase the message size substantially, [5].

#### 4. AN INFRASTRUCTURE FOR S-DSM

We will now describe the network layer and the network management system used in Balder, the software DSM we are designing.

##### 4.1 Server processes

The network is in Balder managed using dedicated daemons in a way similar as to what LAM do [9]. Two types of daemons are used: *server* daemons and *slave* daemons.

###### 4.1.1 Server daemons

The server daemon provides basic network management functions such as job status listings, node and network load statistics and status to the user. The server collects this information from several slave daemons that are run on the actual computation nodes. The slaves communicate with the server daemon using standard TCP/IP sockets. The server does thus not need to run on any of the computation nodes

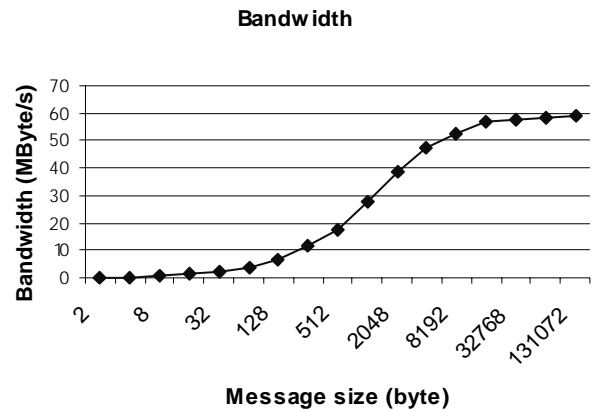


Figure 1. Bandwidth as a function of MPI message size for a network of 400 MHz Pentium II-based PCs interconnected with an SCI [10] network and PCI-card network interfaces.

and it is typically run on a login node. It is the server's responsibility to spawn the slave daemons at system start time.

The management system and the network layer are closely coupled as can be seen in Figure 2. Note that only the management system use TCP/IP. The applications running on the nodes use the high performance network and not TCP/IP.

The user communicates with the server daemon using a command line interface that can be reached by using a telnet session. It is possible for the user to submit jobs to the system, monitor and kill running jobs, and view various statistics regarding the system.

###### 4.1.2 Slave daemons

The slave daemons periodically check the status of the running processes and report any anomalies to the server daemon. They also send so called heart beat messages to the

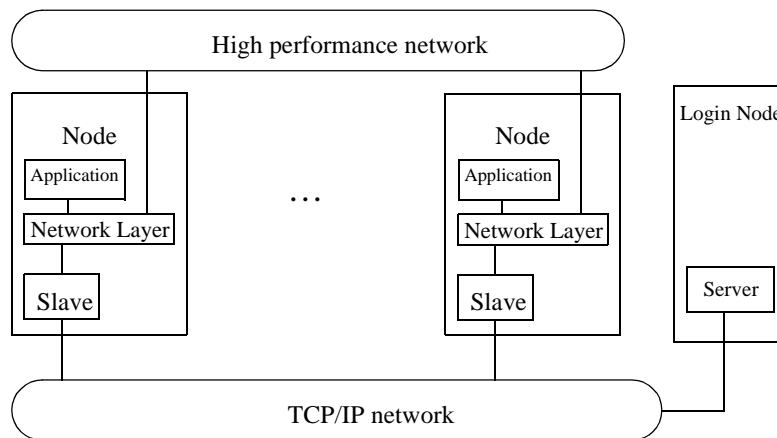


Figure 2. Overview of the management system and network layer.

server. From this information the server can decide whether a process has aborted, a deadlock has occurred, a node has failed and so on. The slaves also start the application processes and provide additional services to the network layer like providing identities of the computation nodes in the network. The slaves are, however, not involved in the applications' communication. Finally the slaves can kill stray processes.

When the user submits a job to the system, the server first checks whether there are enough computation nodes for it. If so, a virtual network consisting of the allocated nodes is formed and the job is associated with it. The server daemon orders the slave daemons running on the nodes in the virtual network to run the application. After this, the server has mainly two tasks. It monitors the execution of the job and provides services to the slave daemons. When the application has started on one of the computation nodes and wants to utilize the high performance network, it needs to know the identities of the other nodes. The network layer sends a request for the identities to the slave daemon using a TCP/IP socket and the slave relays the request to the server.

Whenever the network layer needs to know a system specific parameter it asks the slave which might need to consult the server. Each job has its own virtual network and consequently the network layer can only obtain information regarding its job's virtual network. In practice the network layer will not send any requests to the slave during the actual execution of the job so this scheme will not affect performance. The slaves collect the stdout and stderr outputs from the application and relay them to the server which in turn merges them with the outputs of all nodes in the virtual network. Periodically the network layer reports its current state to the slave daemon. This information is used by the system to detect deadlocks in the communication on the high performance network. Deadlocks can occur, for instance, if one node in some way loses connection to the network. The network layer sends these reports relatively seldom and always tries to utilize the time when the node is idle in order not to degrade performance.

Finally, when the job is terminated, all nodes in the virtual network are released and can be used for subsequent jobs. During the job's execution it is possible for the user to monitor each of the computational nodes in the virtual network. It is also possible for the user to kill a running job and the server will also kill the job if it detects or suspects that one of the nodes in the associated virtual network has lost the connection to the network.

## 4.2 Networking support

### 4.2.1 Interface

One of the most important issues in software DSM system design is to provide an efficient interface to the network. We have designed a simple message-passing layer to support our system. The network layer relies on gather/scatter operations to ensure that copying of data is kept at a

minimum. Messages can be delivered in-order or out-of-order. The in-order mode is intended to be used for coherency messages and it is possible to queue messages from several nodes using the same receive queue when ordering among all nodes is needed. Each message can be assigned an integer tag which simplifies the protocol handling code when several messages from the same node are to be received. It is also possible to register callback functions which are run when a message with a specific tag is received. This makes it possible to implement primitives similar to active messages [2] when needed. A simple window based flow control algorithm is used to ensure reliable transfer of all sent data. Checksums are only used if the network hardware does not detect bit errors.

All send and receive operations operate on linked lists of descriptors identifying pieces of memory where the actual message data is located or is to be received, see Figure 3. The head of the lists contains information such as destination and tag regarding the message. The links consist of descriptors that hold information, i.e. addresses and sizes of memory blocks. The same type of links are used for transmission and reception of messages, i.e., the descriptors points to memory where either message data exist or will be put when the message is received. This scatter/gather approach eliminates, in most cases, the need for dedicated receive buffers and reduces the number of copy operations needed to transfer a message. It also makes it possible for the network layer to efficiently build large network messages from smaller messages by simply linking several lists together. It is possible through the API to instruct the layer when to actually send messages over the network. This makes it possible to transfer large amounts of data without increasing the latency of high priority messages like consistency and synchronization messages.

The message layer has been designed to be easily ported to different networks. The underlying network drivers can operate in user or kernel space. We are currently working on UDP, LAPI [8] and SCI [10] ports. Our experience so far is that the scatter/gather approach makes it simple to utilize the special capabilities of the various networks like memory put and get operations.

It should be noted that a standard message passing API such as MPI [7] can be built on top of the network layer or the network layer can be built on top of MPI. This makes it

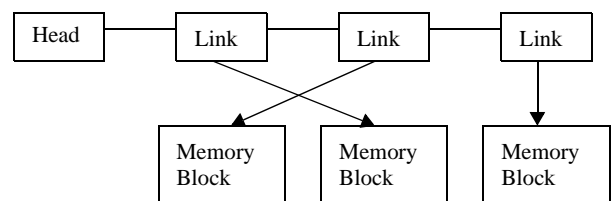


Figure 3. Linked list of descriptors.

possible to have both MPI and for instance OpenMP [11] coexisting in the same system providing a very interesting programming paradigm [12].

The API of the network layer largely consists of functions that manipulate, create and destroy the linked lists of descriptors. There are also functions that are used to register callbacks into the reception system, that sends or receive descriptor lists either synchronously or asynchronously, and that polls or wait for outstanding asynchronous receive operations. Finally there are functions that perform broadcasts among all nodes in a set of nodes.

## 5. CONCLUSIONS

We have in this paper described some of the problems that exist in current software DSM systems and how two of the sub-systems in the Balder software DSM system solves some of these problems. The network management system, which consists of daemons running on the nodes of the system, monitors the network and provides the user with status information and the possibility of controlling the network from a central point. The network layer provides a clean interface between the consistency protocol handling code and the network drivers. The layer has been designed to be portable and is currently being ported to three different networks. The network layer is also suitable for integration of message passing primitives into the software DSM system.

## 6. REFERENCES

- [1] 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, February 1996.
- [2] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauer, Active Messages: a Mechanism for Integrated Communication and Computation, in *Proceedings of the 19th International Symposium on Computer Architecture*. Gold Coast, Qld., Australia. May 1992. pp. 256-66.
- [3] G.A. Geist, V.S. Sunderam, Network-Based Concurrent Computing on the PVM System, *Concurrency: Practice and Experience*, 4 (4):293-311, June 1992.
- [4] D. Jiang et al., Application Scaling under Shared Virtual Memory on a Cluster of SMPs, in *Proceedings of the ISC'99*, June 1999. (to appear)
- [5] S. Karlsson and M. Brorsson, A Comparative Characterization of Communication Patterns in Applications using MPI and Shared Memory on an IBM SP2, in *Proceedings of 1998 Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Las Vegas, January 31 - February 1, 1998, pp. 189-201.
- [6] Pete Keleher, The Relative Importance of Concurrent Writers and Weak Consistency Models, in *Proceedings of the 16th International Conference on Distributed Computing Systems*, May 28, 1996.
- [7] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, version 1.1, June 12, 1995.
- [8] G. Shah, et al., Performance and Experience with LAPI – a New High-Performance Communication Library for the IBM RS/6000 SP, in *Proceedings of IPPS/SPDP'98*, 30 March-3 April 1998, Orlando, Florida.
- [9] Ohio Supercomputer Center, MPI Primer/Developing with LAM, Technical report, The Ohio State University, 1996
- [10] Knut Omang and Bodo Parady, Scalability of SCI Workstation Clusters, a Preliminary Study, in *Proceedings of the 11th IEEE International Parallel Processing Symposium (IPPS'97)*, Geneva, April 1997, pp. 750-755.
- [11] OpenMP Architecture Review Board, OpenMP C and C++ Application Program Interface, Version 1.0, October 1998.
- [12] A. Rodman and M. Brorsson, Programming Effort vs. Performance with a Hybrid Programming Model for Distributed Memory Parallel Architectures, in *Proceedings of EuroPar'99*, September 1999. (to appear)
- [13] R. Samanta, A. Bilas, L. Iftode, and J.P. Singh, Home-based SVM protocols for SMP clusters: Design and performance, in *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, Nevada, January 1998, pp. 113-124.
- [14] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath, Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory, in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October, 1996.
- [15] E. Speight and J.K. Bennett, Brazos: A third generation DSM system, in *Proceedings of the 1997 USENIX Windows/NT Workshop*, August, 1997.
- [16] Robert Stets et al., CASHMERE-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network, in *Proceedings of SOSP '97*, Saint Malo, France, October 1997

S. Karlsson and M. Brorsson, Producer-Push - a Protocol Enhancement to Page-based Software Distributed Shared Memory Systems, in *Proceedings of the 1999 International Conference on Parallel Processing (ICPP'99)*, Aizu-Wakamatsu, Japan, September 1999, pp. 291-300

© Copyright 1999 Computer Society Press, All rights reserved.  
Reprinted with permission.





# Producer-Push – a Protocol Enhancement to Page-based Software Distributed Shared Memory Systems

Sven Karlsson and Mats Brorsson

Computer Systems Group, Department of Information Technology, Lund University

P.O. Box 118, SE-221 00 LUND, Sweden

email: [Sven.Karlsson@it.lth.se](mailto:Sven.Karlsson@it.lth.se), URL: <http://www.it.lth.se>

## Abstract

*This paper describes a technique called producer-push that enhances the performance of a page-based software distributed shared memory system. Shared data, in software DSM systems, must normally be requested from the node that produced the latest value. Producer-push utilizes the execution history to predict this communication so that the data is pushed to the consumer before it is requested. In contrast to previously proposed mechanisms to proactively send data to where it is needed, producer-push uses information about the source code location of communication to more accurately predict the needed communication.*

*Producer-push requires no source code modifications of the application and it effectively reduces the latency of shared memory accesses. This is confirmed by our performance evaluation which shows that the average time to wait for memory updates is reduced by 74%. Producer-push also changes the communication pattern of an application making it more suitable for modern networks. The latter is a result of a 44% reduction of the average number of messages and an enlargement of the average message size by 65%.*

## 1. Introduction

Systems that implement a shared memory programming model on a distributed memory architecture provide a cost-effective means to create parallel computing platforms that are relatively easy to program. Such systems are generally called *software distributed shared memory*, software DSM, systems and can easily be put together from commodity

components such as PCs and standard networks. Current LAN technologies are possible to use but generally deliver poor performance. However, high-bandwidth, low-latency networks such as SCI [15], have been available as off-the-shelf components for some time and it is likely that more high-performance network technologies will emerge.

Even with a standard ATM T1 network it is possible to achieve reasonable efficiency (50%-90%) for a wide range of applications on a software DSM system [16, 8]. However, given the current performance growth of microprocessor technology, it has been shown that even if the network technology keeps up with the processor performance trend, aggressive latency hiding techniques will be needed to sustain the parallel performance efficiency in the future [5, 16].

The existing standard technique to tolerate shared memory latency is to use a relaxed memory model [6]. In this paper we present a technique that complements this idea to mask and overlap some of the communication latency in shared memory applications on a software DSM system. In a shared memory parallel computing model with relaxed memory the actual transfer of data does not occur when the data is produced but is deferred to when it is needed, i.e. when a processor issues a load- or store-instruction to a data item that has previously been written by some other processor. This leads to a request-reply scheme where data is requested and the processor that has an up-to-date copy replies. As a consequence, the communication latency will directly contribute to the execution time.

We have implemented and evaluated a technique called *producer-push* that uses simple heuristics to determine when data that has been modified by one processor is needed by some other processor and to push that data to its anticipated destination before it has been requested. This communication can thus be almost completely overlapped with computation on the receiving processor. An additional advantage is that the processor that needs the data will not

---

The research in this paper was supported with computing resources by the Swedish council for High Performance Computing (HPDR) and Center for High Performance Computing (PDC), Royal Institute of Technology.

be required to send a request message since the data is already present in the local memory.

We have integrated the producer-push technique in TreadMarks, a state-of-the-art software DSM system [2]. The heuristics trigger producer-initiated communication automatically without modification of the application and, based on repetitive access patterns, is found to predict communication in certain parallel applications. We have performed an evaluation on seven applications from the TreadMarks distribution that we executed with and without the producer-push technique on an IBM SP2 using up to eight processing nodes. The evaluation shows that the producer-push technique indeed effectively reduces the time to request shared data. The resulting performance improvement depends on the original efficiency of the application. Our experiments show a performance improvement of 55% in one case. Producer-push also does not hurt performance for applications that do not fit into the behaviour assumed by the simple heuristics.

The performance improvement is mainly due to a reduction in the time an application spends in the communication routines. There are two reasons for this. First, the requests for memory updates can in many cases be eliminated since the updated memory is already in place; second, the interconnection network is better utilized since producer-push results in fewer and larger messages.

## 1.1 Contributions

The idea to transfer data before it has been requested is not new in the context of software DSM systems. The success, or lack of success, depends on the quality of the mechanism that decides what to send and when. If the mechanism decides to send more data than is actually needed, or to the wrong destinations, the performance gain is usually smaller than the performance loss because of the increased network traffic.

Keleher proposed a lazy-hybrid protocol in his Ph.D. thesis that appears similar to the protocol enhancement proposed here [11]. He later re-evaluated this protocol in the context of home-based software DSM systems [12]. The lazy-hybrid protocol is, like our technique, based on repetitive access patterns but uses more indiscriminate heuristics that lead it to send substantially more data than needed and to processors that do not use it. Seidel et al. have proposed an almost identical technique in their Affinity Entry Consistency Protocol [17].

In contrast to the techniques proposed by Keleher and Seidel, the heuristics in our producer-push technique go one step further to identify the correct time to send data proactively by means of an identification of where in the program new data is generated. This is described in detail in section 2.

Producer-push is related to producer initiated communication commonly found in message-passing applications [1]. However, it is also fundamentally different in that it does not require that explicit message passing code is inserted into the applications. Instead the software DSM system takes care of all message passing. No modification of the application's source code is needed at all.

Prefetching [4, 9] is also closely related to producer-push in that it tries to move data to its destination before it has been requested. However, while producer-push reduces the number of messages since the data is moved proactively, prefetching uses the same request-reply scheme as the normal protocol does and if the efficiency of the prefetching scheme is not very high it will send much more data over the network than needed. Producer-push eliminates many of the small request messages and thus utilizes the network better.

In short, the contributions in this paper are:

- the specification and implementation of producer-push in TreadMarks
- a thorough performance evaluation that provides valuable insight into parallel application behaviour

The performance evaluation was done on an upgraded IBM SP2 system with substantially higher performance than the experimental systems used by Keleher and Seidel.

In the rest of the paper we next describe the lazy release consistency protocol and how producer-push has a potential to increase performance. In section 3 we then describe our experimental methodology that we have used to evaluate the producer-push technique. The results are presented in section 4 and after a short discussion on potential improvements of producer-push in section 5, the paper is concluded in section 6.

## 2. Lazy release consistency and producer-push

### 2.1 The TreadMarks software DSM system

TreadMarks is a software package that provides programmers with a shared memory programming model on top of a network of workstations, NOW [2]. Although TreadMarks defines its own programming model, it has been shown that it is possible to implement a standard programming model such as OpenMP on top of TreadMarks [7, 14]. As the nodes in a NOW do not physically share memory, information about changes to the shared memory is captured through the use of the address translation mechanism in the processor which then invokes the consistency protocol. The coherence granularity is thus a virtual page. The performance penalty to maintain memory coherency in software and with such a coarse granularity is very high and therefore TreadMarks uses a relaxed memory consistency model with a multiple-writer, invalidate protocol.

In a relaxed memory model information about changes in the shared memory may be delayed to a synchronization point [6]. If all accesses to shared data in a shared memory program are protected by synchronization mechanisms, e.g. barriers or locks, we say that the program is data race free. It is thus not necessary that all processors see the same shared memory image at a given time. If a variable is protected by a lock and modified by a processor, it is only necessary to inform another processor about this update when the lock has been released and acquired by the other processor.

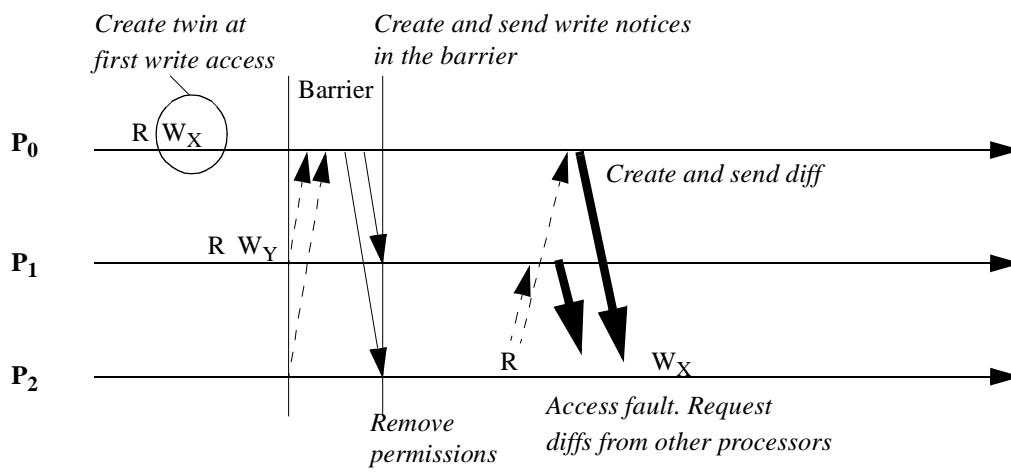
This memory model has been further improved in TreadMarks to reduce communication. A processor is notified about the changes to the shared memory made by other processors at the time of a lock acquire (instead of at the lock release as is normal in relaxed memory models). The actual data is of course not transferred until it is needed which is signalled by a memory access fault. This mechanism is called *lazy release consistency*, LRC. Since the implementation of producer-push is intimately related to LRC we shortly explain how LRC works with a simple example in figure 1.

In this example three processors access two variables within the same virtual page. Processor 0 first reads and writes to variable X and processor 1 reads and writes to variable Y. The processors are then synchronized with a barrier and after this, processor 2 reads and writes variable X. It should then see the value of X that was updated by processor 0 before the barrier. To achieve this, a copy of the page, called a *twin*, is created when processor 0 modifies the variable for the first time. This is done by the page fault handler that is invoked at the write access since the page is write-protected from the start of the execution.

In the code for the barrier, each processor sends a message to processor 0 to notify that it has reached the barrier. Together with this message it transmits information about which pages it has modified during the previous *interval*. This information is called a *write notice*. Processor 0 collects all write notices, including the ones that have been created by itself, and redistributes them to all processors with the same message that signals to all processors that they can proceed past the barrier.

When a processor receives the barrier release message, it continues execution after having removed all permissions to access pages for which it has received write notices. Thus, when processor 2 later accesses variable X it will experience a page fault. Processor 2 has received two write notices for this particular page since both processor 0 and 1 have modified data in the page. It will then send a message to both processor 0 and 1 and request the updated memory. In order to save time and to easier combine the changes by processors 0 and 1, these processors creates a *diff* that is an encoding of the changes made by only this processor. For this reason it uses the twin created earlier. When the diff has been created, it is sent to processor 2 that applies the changes and when all diffs have arrived it continues with its memory reference.

Lazy release protocols reduce network bandwidth requirements at the expense of increased latency caused when faulting processors have to wait for diffs to arrive. This latency is analogous to coherency miss latencies in hardware based systems. Another approach would be to let the processor broadcast all of its diffs directly when it enters a barrier. Protocols that behave like this are called eager protocols [13]. It has been shown that lazy protocols



RW<sub>X</sub> Reads and writes to address X

RW<sub>Y</sub> Reads and writes to address Y in same page as X

**Figure 1. The Lazy Release Consistency protocol in TreadMarks defers the propagation of consistency information to the latest possible moment.**

outperform eager protocols since eager protocols send too much data so that even very high performance networks become saturated.

## 2.2 Producer-push overview

The vast majority of parallel scientific applications use some sort of iterative algorithm. It could be a system simulation by small time steps or an iterative algorithm of a numerical problem. Such an application typically performs the same calculations repetitively in each iteration and also often uses the same data. We could thus use information gathered during one iteration to predict the communication in the following iterations. Really simple applications only have one phase which is repeated over and over again. For such applications the lazy-hybrid protocol as proposed by Keleher would perform very well [11]. However, many applications consist of several phases and for these applications the simple heuristics used to send data proactively in the lazy-hybrid protocol will send too much data.

A processor can collect information on which other processors that have requested diffs during an interval, i.e., a TreadMarks interval as described earlier. The processors that have issued diff requests are called *consumers*, since they want to use data. The processor that responds to the diff requests are in this context called *producer*. From the diff requests, the producer can, when it enters a barrier or releases a lock, collect information regarding which diffs the *consumers* need in a phase of the computation. Knowing this, the producer can send these diffs directly after it has released a lock or entered a barrier the next time it comes to this phase. We call this *pushing*. The consumer stores the pushed diffs in a diff cache and when it later needs a diff, it first looks in the diff cache to see if the diff has been pushed. If so, the consumer does not need to request the diff from the producer and the latency due to the diff request is greatly reduced. This is the essence of the *producer-push* technique.

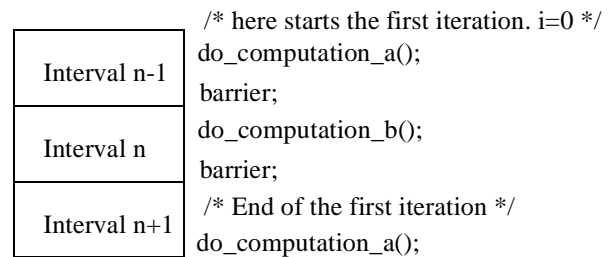
Besides reducing the latency for diff requests, producer-push also has the advantage that it cause the average network message size to increase. This can further boost performance since modern high bandwidth networks cannot reach their peak bandwidth unless fairly large messages are used [10]. Another advantage is that the technique does not require any source code modifications at all.

## 2.3 Heuristics and implementation

In order to make producer-push work well we need some kind of heuristic that can guide us in the selection of diffs to push, when the diffs should be pushed and which processor they should be pushed to. We describe here the heuristic that we have used and some details about the actual implementation. We first describe how we find *logical intervals* in the execution related to phases in the execution.

```
int i;
for (i=0; i<N; i++)
{
    do_computation_a();
    barrier;
    do_computation_b();
    barrier;
}
```

**Figure 2. A simple example of an iterative algorithm.**



**Figure 3. An illustration of the intervals in the simple example.**

We use this information to determine when diffs are to be pushed.

Internally TreadMarks uses so called intervals to keep accurate time stamps on all diffs. The interval is a description of a small part of the application's execution time. New intervals are created at barriers and when a lock is released. A diff is associated with the specific interval when it was created. It is also the interval mechanism that drives the invalidate consistency protocol. We have implemented producer-push on top of the interval mechanism and perform pushing of selected diffs directly after a new interval is created.

Since the creation of intervals is forced by synchronization primitives like locks and barriers it is possible to bind intervals to specific regions in the application code. We will show this by using a simple example, see figure 2. Figure 3 shows the intervals created in one of the processor nodes. The example is based on barriers but the described heuristic is analogous for locks.

A new interval is created for each execution of a barrier. In this example all iterations are identical in the sense that the same instructions are executed repetitively in each interval. Therefore, interval n+1 is considered to be identical to interval n-1. We say that they belong to the same *logical interval*. In general, applications may in one loop synchronize several times and it is therefore common that a two consecutive TreadMarks intervals refer to different parts in the program. In order to bind intervals to logical intervals we assign a unique identifier to each barrier. We

Logical interval ?	Interval n-1	do_computation_a(); barrier(0);
Logical interval 0	Interval n	do_computation_b(); barrier(1);
Logical interval 1	Interval n+1	do_computation_a(); barrier(0);
Logical interval 0	Interval n+2	do_computation_b(); barrier(1);
Logical interval 1	Interval n+3	

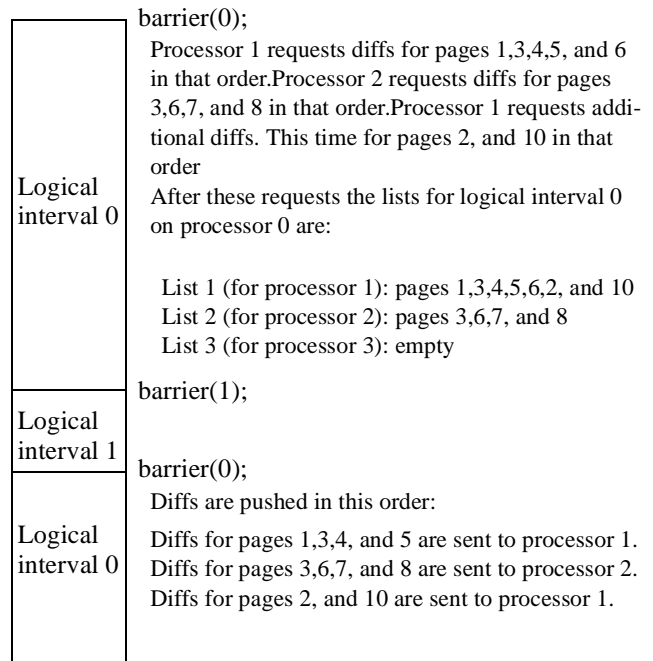
**Figure 4. The simple example augmented with logical intervals.**

have in our experiments done this with a label for each barrier in the source code but it could just as well be done automatically, either statically by the compiler or in run-time by TreadMarks by observing the program counter value. We have in figure 4 augmented our example with logical intervals. Note that we cannot know which logical interval is bound to interval n-1 from the example. By assigning an identifier to each barrier in an application we can during the second iteration always link a TreadMarks interval to a logical interval.

We next need to predict which diffs to send to which processor in each logical interval. The producing processor gets diff requests from a consuming processor whenever the consumer faults and it cannot find a needed diff in its diff cache. From these requests it is possible for the producer to build up sets of recently requested pages for each processor. Since the diffs are requested in a particular order it is important that this order is maintained so that diffs are sent out in the correct order.

In our implementation we maintain a linked list of page entries corresponding to pages for each logical interval and consuming processor. All lists are initially empty when the application starts. Whenever a consumer sends a request for a page diff, a page entry is inserted at the end of the list associated with the consuming processor and the current logical interval on the producer, i.e. the processor that serves the diff request.

When a processor enters a logical interval it pushes all diffs corresponding to the pages in the lists in the order that these pages occur in the list. This ensures that data that is needed early by the consumer, is pushed as early as possible. If several consuming processors have requested diffs previously in this logical interval, the producer will push diffs to all of these processors. In order not to favour any particular consumer, the producer interleaves the pushing of diffs among the consumers. In practice this means that approximately equally sized chunks of data are sent in turn



**Figure 5. The simple example with producer-push actions.**

to the consumers. The maximal size of these chunks depend on the network layer and is typically on the order of a few tens of kilobytes. Figure 5 shows how the lists of pages are maintained and how the pushing of data is done in the case of 4 processors. Once again our simple example is used and we show the lists maintained and the diffs pushed by processor 0. For clarity only the actions in logical interval 0 and the barrier primitives are shown.

## 2.4 What kind of application benefits from Producer-Push?

Some algorithms benefit more than others from producer-push. The algorithm used must first of all be iterative, i.e., several iterations in which the communication pattern is repeated. Otherwise the current heuristic will not do any predictions. Furthermore, the algorithm should be fairly regular in its data usage. It should basically use the same data-set in each iteration. Also, since producer-push reduces the time to request diffs, the number of coherency misses caused by the algorithms should be fairly high. Even though the mechanism works for lock-based algorithms, we have tuned the heuristics for barrier-based applications since iterative algorithms often use barriers as the main synchronization mechanism.

A typical application that will benefit from producer-push is eigen-value calculations where the same vector manipulations are performed on the same set of data for several iterations. An example of application that will not benefit much from producer-push is an algorithm using

searches where the search pattern is irregular. Algorithms that perform different calculations in different iterations, like some heuristic-based algorithms do, will also not gain much performance from producer-push. The rule of thumb is that if an algorithm will benefit from normal prefetching, it will also benefit from producer-push. It should be noted that even with the current simple heuristic we have not noticed any negative effects on any of the applications we have run. Let us now go over to the evaluation of producer-push.

### 3. Experimental methodology

#### 3.1 Experimental platform

We have integrated the producer-push algorithm in version 1.0.1 of TreadMarks. In order to evaluate the performance of producer-push we have used seven standard scientific benchmarks on an IBM SP2 with POWER2 processors running at 160MHz and an upgraded 110 Mbyte/s switch. The benchmarks were compiled with IBM's C-compiler xlc using optimization level -O. The TreadMarks library was augmented to be able to collect profiling information. The profiling adds approximately 15% overhead and thus two separate runs have been done for each application; one run without profiling to measure execution times and one with profiling to collect information on where the execution time is spent.

#### 3.2 Applications

Seven different benchmarks from the TreadMarks 1.0.1 benchmark distribution were used in the evaluation of the relative performance benefits of producer-push. SOR is a standard stencil iterative algorithm. IS, FFT, CG and MG are originally from the NAS benchmark suite [3], and Water and Barnes-Hut are originally from the SPLASH benchmark suite [18] but rewritten to suit TreadMarks better. Like SOR, all other benchmarks also use iterative algorithms and according to the discussion at the end of the previous section, it is thus probable that their performance can be boosted by the producer-push technique. A summary of the application workloads is shown in table 1 and they are briefly described below.

Execution times are measured according to the existing timing instrumentation in the applications as they are distributed in TreadMarks. This means that only the parallel section of the program is taken into account. Our profiling measurements have been done on the same part of the execution as the timing. This means that profiling has been done on all iterations in all applications but for Barnes-Hut where profiling is done from the fifth iteration.

**3.2.1 SOR and IS.** SOR iteratively applies an averaging kernel to a matrix. The algorithm is extremely regular and very little data is communicated in each time step. IS sorts a

**Table 1. Applications and workloads used in the study.**

Application	Workload
SOR	10 iterations, 2000*1000 matrix
IS	10 rankings, 32768 keys
FFT	6 time steps, 64*64*64 cube
CG	Kernel C
MG	Kernel B
Water	5 time steps, 512 molecules
Barnes-Hut	10 time steps, 16384 bodies

vector of integers and this algorithm is also regular in its access pattern. We can expect that producer-push will predict the communication well for both applications. However, since SOR already has a good relative speedup even without producer-push, we cannot expect a large absolute performance improvement since there is very little latency to hide.

**3.2.2 FFT, CG and MG.** FFT is a PDE solver that uses a three-dimensional FFT. The solution is iterated in frequency space where each iteration consists of an inverse FFT. The CG benchmark finds an estimate of the largest eigen-value to a sparse matrix using the inverse power method. MG solves the Poisson problem in three dimensions using a finite element method. These programs all consist of simple matrix operators operating on the same data in every iteration. We thus expect that producer-push will work well on them.

**3.2.3 Water.** Water is a simulation of the dynamics of several water molecules. The effect of both intra- and inter molecular forces are simulated. The version used is the so called n-squared Water where each processor is assigned a range of molecules and calculates the effect these molecules exert on all the other molecules. This means that processors will manipulate the data structures that represents the molecules assigned to other processors. Race hazards are avoided by using one lock per molecule. This means that a sizeable fraction of the execution time is spent waiting for lock acquisitions. The lock-mechanism in TreadMarks uses intervals in the same way as barriers do. A logical interval is in this case identified by the molecule for which there is a unique lock. The communication in Water is mostly regular and when a processor acquires a lock, it is often released by the same processor as the last time but as the processors are not tightly synchronized it is possible that the communication patterns change slightly from iteration to

iteration. Water also uses locks to implement global reductions which yields highly unpredictable communication. We can, however, conclude that producer-push should be able to predict at least a major part the resulting communication which in turn should reduce the diff request time.

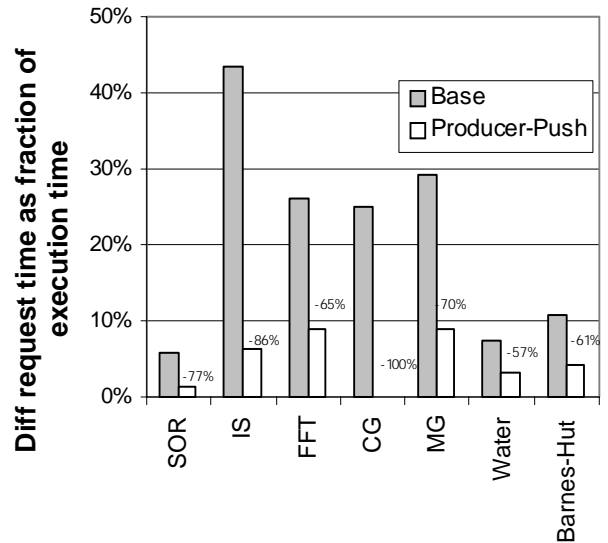
**3.2.4 Barnes-Hut.** This is an N-body simulation of the motion of bodies that interact using gravitational forces. The bodies are hierarchically ordered in a tree that is rebuilt by the root node for each time step. As in Water the bodies are distributed among the processors and the effect of the forces from all bodies on each body must be evaluated. Unlike Water, however, no processor updates the body structures of another processor. The main communication is thus to and from the root node when the tree is rebuilt. Since the tree is rebuilt for each time step the set of bodies assigned to a processor changes from one time step to the next. This communication cannot be predicted but we can expect producer-push to effectively push all data corresponding to bodies that do not migrate between processors.

#### 4. Experimental results

The main idea behind producer-push is to reduce the time to request diffs in software DSM systems. Figure 6 shows the difference in the fraction of execution time spent doing diff requests in the seven applications, without and with producer-push. As expected, the diff request is greatly reduced for all applications. The average reduction of diff request time is 74%. The reduction is smaller for Barnes-Hut. This is due to the migration of bodies between processors which the heuristics cannot predict. Thus a smaller fraction of the diffs needed by the consumer is pushed from the producer.

The reduction for Water is larger than 50%. This should normally directly correspond to a decrease in execution time of a corresponding amount but the execution time is unchanged which we can see from figure 7. This is due to the fact that the time spent waiting for a lock to be released, which in this case is the dominating latency, is increased by the same amount of time. The reason for this is that at the same time as ownership of the lock is acquired, the releasing node is also pushing data to the acquiring node which must handle the incoming data. This causes a small overhead which in this case is equal to the time gained from producer-push. It should be stated that Water is an extreme case and even so, producer-push will not decrease the performance of the application.

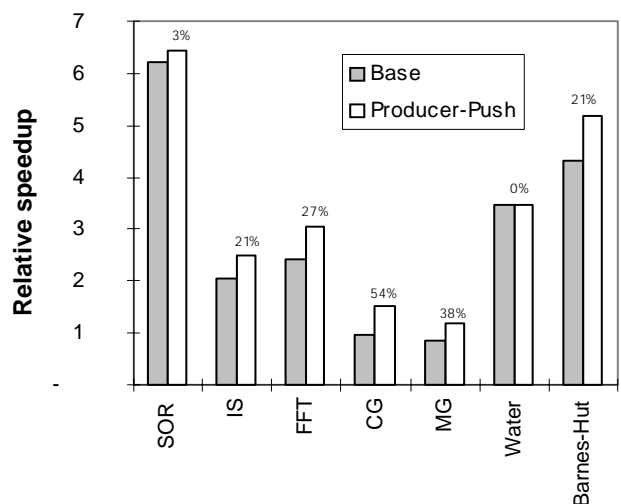
In figure 7 we show relative speedup with eight processors for all applications. The grey bar corresponds to executions of TreadMarks and for the white bar we have also used producer-push. The relative speedup is calculated as the execution time of an application using one processor



**Figure 6. The fraction of execution time spent doing diff requests with and without producer-push.**

divided by the execution time using eight processors. The data in figure 7 was of course obtained from executions without profiling overhead.

The numbers above the bars in figure 7 show the performance improvement we get when producer-push is used. We can see that the applications can be divided into three different groups depending on the performance improvement obtained. SOR and Water do not benefit much



**Figure 7. A comparison of the relative speedup with 8 processors, without and with producer-push. The number above the bars is the performance improvement achieved by producer-push.**

from producer-push. For FFT, IS and Barnes-Hut there is a moderate performance improvement, and for CG and MG there is a relatively large performance improvement.

Although the diff request overhead is reduced for Water this has no impact on the execution time. This is, as stated earlier, due to the fact that the lock release time is increased. SOR also has a low relative speedup. In this case producer-push actually works well but the latency that can be hidden is very small. The diff request overhead is reduced by 5% to only 1% and this results in a performance improvement of about 3% on 8 nodes.

The second group of applications have all moderate relative speedups of 20-27% on 8 nodes. The applications in this group, consisting of FFT, IS and Barnes-Hut, are all affected by producer-push as one would expect. The diff request time is reduced and the overall execution time is decreased.

The last group, i.e., MG and CG, have large performance gains from using producer-push. Figure 8. shows the normalized execution time broken down into busy time, time spent in TreadMarks code and time spent in the network layers including communication time in the actual network and protocol handling time. We see in figure 8 that the execution time of these applications are totally dominated by the communication time. Note that the execution times have been normalized so that 100% corresponds to the execution time without producer-push. Roughly half of the communication time is due to diff requests and the other half is due to waiting for other processors in barriers. Producer-push effectively reduces the diff request time and thus the communication time. Since the busy time of CG and MG is so much shorter compared to the other applications, the reduced communication time has a much greater impact. Experiments show that the performance improvement of producer-push for MG is even higher for four processors. This is due to the fact that when eight processors are used, MG uses so much network bandwidth that it saturates the network.

Let us now investigate the performance of the heuristics that decides when and where to push data. Table 2 lists the efficiency, fraction late diffs and the coverage for the different applications. The *efficiency* is defined as the fraction of the pushed diffs that are found in the diff cache at the time of a page fault and thus used by the consumer. An efficiency near 100% means that almost all pushed diffs are used and that network bandwidth is not being wasted. The *late diffs* are the fraction of the pushed diffs that arrive too late to the consumer in order to be useful. This means that they arrive after the page fault has occurred. These pushed diffs waste bandwidth just as the diffs that never are requested. However, if they could have been sent earlier,

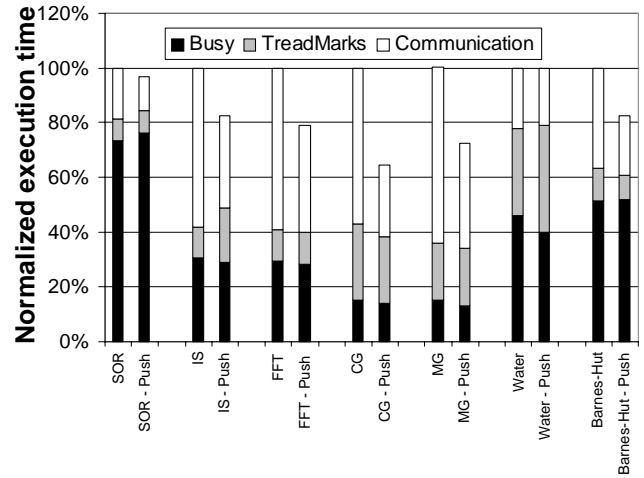


Figure 8. Normalized execution time breakdown.

they would be useful. Finally, the *coverage* is defined as the fraction of diffs requested by the consumer that were successfully pushed and used. The coverage is a metric of the ability to predict communication.

Table 2. Efficiency, late diffs and coverage of producer-push on the seven applications.

Application	Efficiency	Late diffs	Coverage
SOR	92%	2%	85%
IS	97%	0%	90%
FFT	97%	0%	72%
CG	100%	0%	100%
MG	68%	25%	75%
Water	45%	7%	64%
Barnes-Hut	98%	4%	67%

The efficiency is very high for all applications except for MG and Water. For MG the reason for the relatively low efficiency is that 25% of all pushed diffs arrive too late and not that the heuristic mispredicts the communication. However, the low efficiency for Water is a result of the inability to correctly predict the communication. As described in section 3.2, some parts of the communication in Water is hard to predict due to the fact that locks are used to implement reduction operations, and that the processors are loosely synchronized.

The coverage of producer-push varies among the applications. This is mostly because the timed regions are

different in the applications. When the timing and profiling start, the only application in which the heuristic is warmed up is Barnes-Hut. Besides Water, the lowest coverage is for Barnes-Hut. This is due to the migration of bodies as described in section 3.2. The coverage of producer-push for FT, IS and SOR is a bit higher but not perfect as one would expect. The reason here is the presence of cold misses that are impossible to predict. Producer-push has an ideal coverage for CG, but at first glance not for MG. However, for MG all diffs are predicted but, as mentioned before, 25% of the pushed diffs arrive late due to network contention. In this case the consumer cannot find the late diffs in its diff cache and therefore performs a normal diff request.

One final observation is that producer-push results in a better utilization of the available network bandwidth. It has been shown that the main reason for poor performance in software DSM systems such as TreadMarks is due to communication protocol processing [16]. Therefore, each protocol improvement that reduces the number of messages will also lead to a performance improvement unless it also results in an increase of some other execution time component. Table 3 shows the number of messages and the average message sizes for the base system compared to when producer-push is used. For all applications except for Water, the actual number of bytes communicated is about the same with and without pushing. However, for the applications that perform better using producer-push, IS, FFT, CG, MG, and Barnes-Hut, the number of messages sent is only 43-64% of the number of messages without pushing. Consequently the average message size is increased for these applications.

## 5. Potential improvements

Even if the current implementation works well on barrier-based applications, as we saw in the previous section, there are some possible improvements to the heuristics that can be done. We will describe a few in the following passages. Although we have not implemented any of these, we feel that they can boost performance even higher.

The order in which diffs are sent to the producer to the consumer is important. MG observed a sizeable fraction late diffs and part of the reason to why these are late is because they are sent to late from the producer. The mechanism could be thus modified to send diffs earlier than is determined by the current heuristics if they are detected as late by an *adaptive-order* mechanism. It is also possible in some cases to move a page from a list in a logical interval to a list in an earlier logical interval. This is a variation of adaptive order.

If the current heuristics has determined that a diff is to be pushed in a logical interval, it will always be pushed in this interval even if it no longer will be needed by the consumer.

**Table 3. Number of messages and average message sizes without and with producer-push. The numbers in parentheses show the reduction in number of messages and increase in message size respectively.**

	Number of messages		Average message size (kbyte)	
	Base	Push	Base	Push
SOR	2709	2485 (92%)	2.68	2.92 (1.1x)
IS	6023	3670 (61%)	12.06	20.26 (1.7x)
FFT	11207	7072 (63%)	3.70	5.86 (1.6x)
CG	160349	83279 (52%)	0.43	0.83 (1.9x)
MG	40981	22180 (54%)	2.41	4.69 (1.9x)
Water	43812	42412 (97%)	0.41	0.50 (1.2x)
Barnes-Hut	105456	45012 (43%)	0.97	2.29 (2.4x)

In order to keep the network traffic as low as possible improvements can be done to remove pages from the producer's lists if they have not been used by the consumer. We call this extension *adaptive push-set*. We intend to investigate the effects of these improvements in our future research.

## 6. Conclusions

Networks of workstations provide a cost-effective way to build parallel computing environments. Software DSM systems can be utilized to make NOWs more easy to program. However, the high communication overhead in a NOW makes it necessary to reduce the implicit communication in a shared memory system as much as possible. Lazy release consistency is one such technique that reduces and hides latency in software DSM systems.

We have in this paper presented a technique called producer-push which is an extension to a lazy release consistency protocol. Producer-push boosts the performance of regular iterative applications, an important class of applications for parallel systems. Producer-push uses repetitive communication patterns to predict communication and can, for the applications used in the study, reduce stall time to wait for shared memory updates from remote processing nodes by 74% on average. The resulting performance improvement is on average 23%.

Given that the performance improvement of single-processor systems tend to improve at a faster pace than

network technology that tend to have longer life-times, the relative performance of applications on software DSM systems will decrease since they will be more and more dependent on network performance. It is therefore important to be able to overlap communication and computation as much as possible and also to utilize the network as good as possible.

Although there are improvements to be made for producer-push, it is already usable to reduce remote memory access latency by overlapping communication and computation. One important side-effect is that the number of messages needed is greatly reduced and, since the amount of communication is the same, the message sizes are increased by roughly the same amount. This favours even off-the-shelf networks that have relatively long latencies but high available bandwidth.

## 7. References

- [1] H. Abdel-Shafi et al. An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors. In *Proceedings of the Third International Symposium on High-Performance Computer Architecture*, pages 204-215, February 1997.
- [2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, *IEEE Computer*, Vol. 29, no. 2, pp. 18-28, February 1996.
- [3] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0, Report NAS-95-020, Nasa Ames Research Center, Moffett Field, Ca, 94035, USA. December, 1995.
- [4] R. Bianchini, L. Kontothanassis, R. Pinto, M. De Maria, M. Abud, C. L. Amorim. Hiding Communication Latency and Coherence Overhead in Software DSMs, In *Proceedings of the 7th ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 7)*, pp. 198-209, October 1996.
- [5] A. L. Cox, S. Dwarkadas, and P. Keleher. Software Versus Hardware Shared-Memory Implementations: A Case Study, In *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 106-117, April 1994.
- [6] K. Gharachorloo, D. E. Lenoski, J. P. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15-26, May 1990.
- [7] J. Hård, *OdinMP – A Proposal for the Implementation of OpenMP for a Network of Workstations*, M.Sc. thesis. Department of Information Technology, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden, January 1999.
- [8] D. Jiang et al., Application Scaling under Shared Virtual Memory on a Cluster of SMPs, in *Proceedings of the ISC'99*, June 1999, pp. 165-174.
- [9] M. Karlsson and P. Stenström, Evaluation of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems, In *Journal of Parallel and Distributed Computing*, vol. 43, no. 7 (July 1997), pp.79-93.
- [10] S. Karlsson and M. Brorsson, A Comparative Characterization of Communication Patterns in Applications using MPI and Shared Memory on an IBM SP2, In *Proceedings of 1998 Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, pp. 189-201, Las Vegas, January 31 - February 1, 1998
- [11] P. Keleher, Lazy Release Consistency for Distributed Shared Memory, PhD Thesis, Rice University, 1994
- [12] P. Keleher, Update Protocols and Iterative Scientific Applications, In *Proceedings of the International Parallel Processing Symposium*, pp. 675-681, 1998.
- [13] P. Keheler, A. L. Cox and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory, In *Proc. of the 19th Annual Int'l Symp on Computer Architecture (ISCA '92)*, pp. 13-21, May 1992
- [14] H. Lu, Y.C. Hu, and W. Zwaenepoel, OpenMP on Network of Workstations, In *Proceedings of Supercomputing '98*, October 1998.
- [15] K. Omang and B. Parady, Scalability of SCI Workstation Clusters, a Preliminary Study, in *Proceedings of the 11th IEEE International Parallel Processing Symposium (IPPS '97)*, Geneve, April 1997, pp. 750-755.
- [16] E. W. Parsons, M. Brorsson and K. C. Sevcik, Predicting the Performance of Distributed Virtual Shared Memory Applications, *IBM Systems Journal*, Volume 36, No. 4, 1997, pp. 527-549.
- [17] C. B Seidel, R. Bianchini, and C. L Amorim, The Affinity Entry Consistency Protocol, In *Proceedings of the International Conference on Parallel Processing*, pp. 208-217, August 1997.
- [18] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):5-44, March 1992.

S. Karlsson and M. Brorsson, Priority Based Messaging for Software Distributed Shared Memory, in *Journal on Cluster Computing*, 6 (2): 161-169, April 2003

also published as:

S. Karlsson and M. Brorsson, Priority Based Messaging for Software Distributed Shared Memory, in *Proceedings of Workshop on Communication Architecture for Clusters (CAC '01)*, April 2001

© Copyright 2003 Kluwer Academic Publishers, All rights reserved.  
Reprinted with permission.



IV





# Priority Based Messaging for Software Distributed Shared Memory

SVEN KARLSSON\* and MATS BRORSSON

*Department of Microelectronics and Information Technology, Royal Institute of Technology, Stockholm, Sweden*

**Abstract.** Software Distributed Shared Memory (DSM) systems can be used to provide a coherent shared address space on multicomputers and other parallel systems without support for shared memory in hardware. The coherency software automatically translates shared memory accesses to explicit messages exchanged among the nodes in the system. Many applications exhibit a good performance on such systems but it has been shown that, for some applications, performance critical messages can be delayed behind less important messages because of the enqueueing behavior in the communication libraries used in current systems. We present in this paper a new portable communication library that supports priorities to remedy this situation. We describe an implementation of the communication library and a quantitative model that is used to estimate the performance impact of priorities for a typical situation. Using the model, we show that the use of high-priority communication reduces the latency of performance critical messages substantially over a wide range of network design parameters. The latency is reduced with up to 10–25% for each delaying low priority message in the queue ahead.

**Keywords:** cluster computing, software distributed shared memory, software DSM, communication libraries, parallel computing

## 1. Introduction

The performance of modern computer interconnects is rapidly increasing and it is no longer uncommon with latencies lower than a few microseconds and data rates exceeding hundreds of megabytes per second. There is, however, an increasing difference in performance between processors and network interconnects since the performance of processors have increased even more rapidly than the performance of networks. The network interconnect is already a performance bottleneck in many systems and will continue to be that for the foreseeable future.

Systems for software distributed shared memory, software DSM, are examples where the network subsystem is a major performance bottleneck. A software DSM system provides a coherent shared memory programming model on a non-cache-coherent multicomputer [1]. Most such systems use the virtual memory protection mechanism to detect memory accesses to addresses that currently do not reside locally and it is the task of the software DSM system to request data from other nodes through explicit messages over the interconnection network. In general, smaller and more numerous messages are sent between the nodes if an application is written using shared memory with a software DSM system than if the same algorithm is coded with message-passing [10].

Most software DSM systems make use of a communication library as the interface to the network drivers and hardware. There are several advantages of using a communication library instead of the low-level network drivers directly. The most important advantage is that a communication library can

be made portable which means that the software DSM system can be used unchanged on many different platforms.

Software DSM systems send messages of two categories through the communication library: *data messages* that carry shared memory contents, and *control messages* that are sent to update the states in the coherency protocol or to synchronize nodes. Even though it is important that data messages are not delayed unduly, it is the latencies of the control messages that are the most critical for the overall performance of the system. It has, however, been shown that control messages in certain applications get delayed by data messages ahead in the message queues [2]. This comes from the fact that in a software DSM system using a traditional communication library, there is only one available communication channel between any two nodes and all messages are delivered in the same order as they have been sent.

We argue here for the use of message priorities as a way of improving the latency of performance-critical control messages. This means that messages with a high priority, e.g., control messages, are transferred before messages with lower priority.

We have implemented prioritized communication in a portable communication library for software DSM with a significant reduction of latency as result. We also present a simple model for the expected latency with and without using priorities. We have found that the decrease in latency for control messages when priorities are used depends more on the bandwidth of the network than on the zero-size message latency. A network with high bandwidth thus benefits more from introducing priorities than a network with low bandwidth given the same message latency.

The concept of priorities is certainly not new. There are, however, no existing communication libraries for either soft-

\* Corresponding author.  
E-mail: svenka@it.kth.se

ware DSM or message-passing that make use of priorities. Below, we briefly describe some systems presently used.

Practically all current software DSM systems either only include support for standard UNIX sockets or are targeted at a specific network technology and are thus very hard to port to new hardware technologies [2,18,22]. One exception is GeNIMA which takes a more modular approach that in theory could be retargeted at other interconnects [3]. GeNIMA is currently implemented on Myrinet and implements some of the coherency protocol directly in Myrinet's programmable hardware which will cause problems when porting to other network technologies [4]. GeNIMA is the only system that attacks the problem with delayed control messages. However, it does so by moving the handling of the control messages into the network hardware which again, is not portable.

There are several communication libraries for user space communication, e.g., VMMC [7], PM [23], Fast Messages [16], Active Messages [24], and Basic Interface for Parallelism [17]. However, none of them support prioritized communication. The VI architecture support several communication channels between nodes but there are no semantics for priorities [5]. There are no quantitative models known to us that describe the performance impact of prioritized communication.

In short, the main contributions in this paper are:

- The design and implementation of a portable communication library, called *Balder Messages*, utilizing prioritized message communication.
- A simple performance model for the latency of control messages with and without priorities.

We next describe Balder Messages in section 2. In section 3 the quantitative performance model is introduced and some important results are obtained in section 4. The paper is concluded in section 5.

## 2. Communication library with priorities

### 2.1. Balder Messages overview

In this section we describe the structure and implementation of Balder Messages, a portable communication library supporting priorities and which is targeted at software DSM systems. Balder Messages is a part of a software DSM system being developed called Balder [11]. The main distinguishing feature of Balder Messages compared to previous high-performance, low latency communication libraries such as Protected Messages (PM) and Fast Messages [16,23] is the use of priorities. The implementation of priorities is also portable in contrast to GeNIMA which uses the hardware features of a particular network interface to handle control messages [3,4].

The communication needs of a software DSM system puts several constraints on a communication library:

- It is crucial that the latency is kept at an absolute minimum since the coherency in a software DSM system is

kept through a complex scheme of messages sent between the nodes. Any excessive delay of messages would reduce the performance of the system.

- The order of coherency messages should be maintained. This will make the coherency protocols more simple and more efficiently implemented.
- It is important, for performance reasons, that the system can utilize the raw bandwidth of the underlying interconnect hardware. In addition, it must support a wide range of interconnect hardware since the network technology is constantly improving.

In order to fulfil all these requirements, the API of Balder Messages has been designed to have only very simple primitives instead of specialized primitives. Furthermore, user space communication is used to keep latency low. The API consists of functions for both traditional message passing and active messages [24].

It should be noted that while the previous communication libraries used in software DSM systems either were targeted at a particular interconnect topology, hardware, or were tightly integrated with the rest of the system, Balder Messages takes another approach utilizing a modular design and a retargetable communication library that supports a wide range of interconnects. For instance when operating on an IP-based network, Balder Messages implements flow control, ensures the correct ordering of messages and also handles the case when messages are lost in the network. Some or all of these functions might not be needed when using more capable networks where the network hardware takes care of, for instance, message ordering. Balder Message use polling based message reception on most high performance network technologies as interrupts are usually inefficient. Balder Messages can, however, support interrupt based message reception on technologies that have efficient interrupt handling.

It should be noted that although Balder Messages is a general communication library, it is targeted at software DSM systems. We will in the rest of the paper therefore only discuss Balder Messages in the context of software DSM systems even though it can be used in other environments as well.

We will now start a more detailed description of Balder Messages and its API.

### 2.2. Message passing primitives

Balder Messages supports, as indicated above, asynchronous, synchronous, and active message communication. All of these types of communication are useful in a software DSM system. Asynchronous communication makes it possible to overlap computation with communication. Synchronous communication, on the other hand, can potentially reduce the latency of communication when the system or application is otherwise idle. Finally, active messages make server functions efficient. It should also be noted that Balder Messages

Table 1  
Basic message functions in Balder Messages.

Function	Description
ASync send	Asynchronous send
Sync_recv_list	Synchronous receive
ASync recv_list	Asynchronous receive
Poll_reception	Reception status poll
Wait for reception	Blocking reception status poll
Poll transmission	Transmission status poll
Wait for transmission	Blocking transmission status poll
Register_msg_callback	Register active messages callback

supports multithreading and is fully reentrant which makes it suitable for SMP-nodes.

Each message sent and received has a tag attached to it. This tag works just like the message tag used in MPI [13]. This means that in a receive operation you will have to specify a tag and only a message with identical tag will be received. Balder Messages has no notion of individual threads so if multithreading is used, the message tag can be used to direct a message to a specific thread. The message tag can of course also be used to specify the type of message. This is crucial in software DSM systems which implement the coherence protocol using request-reply schemes. The tags are used to ensure that the respective nodes receive and process the correct requests and replies in the right order.

The upper bits of the message tag are used to specify the priority of a message. The communication in each priority level is independent of the other levels, although messages with a higher priority are always sent ahead of those with lower priority. If possible, several messages are aggregated and sent together in a single network packet. Another way of looking at this is that each priority level has its own communication channel which is independent from the other priority levels.

Table 1 summarizes the functions in Balder Messages for sending and receiving messages. In addition to these functions, there are also functions to manipulate chunk lists which are important data structures in the design of Balder Messages, see section 2.3.

Balder Messages provides a small number of simple primitives which can be used to build more complex messaging schemes if needed. This is in contrast to the large number of complex functions high-level messaging libraries, such as MPI, provides.

While this makes porting more easy due to fewer functions to implement, it also makes it potentially harder to optimize Balder Messages for a specific network hardware. Fortunately, most network interconnect cards works in more or less the same way in that they only provide basic packet reception and transmission. They do, however, often differ in the way that DMA engines are setup. We therefore use an internal data structure, chunk lists, which makes it possible to use several different DMA engine architectures without changing the actual code in the software DSM system. The use of this data structure in Balder Messages is described next.

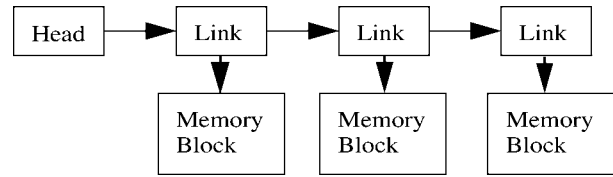


Figure 1. Chunk links and their application. The head link marks the beginning of a chunk list.

### 2.3. Chunk lists

A *chunk list* is a linked list where each link in the list corresponds to a block, or *chunk*, of memory, see figure 1. Apart from being easy to manipulate, the chunk lists closely resembles the DMA descriptors commonly used in chained DMA engines. The software DSM system never manipulates these lists directly. They are maintained using functions and macros provided by the API of Balder Messages. It is thus possible to alter the actual implementation of the chunk list in order to better suit a DMA engine in case the underlying hardware incorporates one. Each link in the list contains a pointer to the corresponding memory block and the size of the block. Since the software DSM system never touches these fields itself, several other fields might be present if needed by a DMA engine.

The chunk list is a convenient data structure in a software DSM system as all messages in such systems consists of a header followed by several data records. The header and each of the records are typically located in different memory regions upon transmission and are to be scattered into different memory regions upon reception. The use of chunk lists thus reduces the number of times data needs to be copied in memory when a message is processed. In fact, in networks that support remote load and store – e.g., SCI where a portion of a remote nodes memory can be mapped into the local nodes address space [6] – all excessive memory copy operations can be eliminated with the use of chunk lists. The sending node can then copy the message directly into the receiving node’s memory. Similarly, the receiving node can copy the message from the sending node’s memory into its own memory. This both reduces the latency of the message passing operations and also reduces the load on the CPU resources which in turn makes it possible to utilize the high bandwidth of modern high performance networks.

We will now continue and describe the flow control mechanism which is the foundation for implementing priorities in Balder Messages.

### 2.4. Flow control and priorities

One important issue in the design of any communication layer or library is the implementation of flow control. Flow control is essential as to avoid overloading the network interconnect fabric or to avoid creating hot spots that in turn may result in high communication latencies or even packet loss.

In Balder Messages we use a version of the rather well known sliding window flow control approach used in TCP [9].

With sequence numbers added to each network packet it is possible to enforce deterministic message ordering as well as making all communicating nodes aware of the reception status, i.e., the number of messages each node have received and how many more it can receive. From this information it is possible for the sending part to safely send several consecutive messages without overloading the receiving part's reception buffer. Acknowledgements and retransmissions is used to ensure that all data is correctly transferred even if packets are lost in the network

The possibility to use priorities for messages is implemented on top of the flow control algorithm. A priority level is assigned to each message based on the most significant bits in the message tag. Each bit combination corresponds to a priority level and so if 4 bits are used to describe the priority, you get 16 priority levels. The actual number of priority levels can be set when Balder Messages is compiled.

The priority levels can be seen as virtual channels and messages belonging to different priority levels are transferred independently of each other. We call these virtual channels *priority channels*. Messages sent in a priority channel are delivered in FIFO order. Priorities are enforced by only transferring messages in a priority channel with a low priority if there are no messages of higher priority waiting to be sent in other priority channels. We will now explain the algorithm in more detail.

A message being transferred is not considered to be sent until it has been acknowledged by the receiver through the flow control mechanism. An acknowledgement message may be used for several messages and acknowledgements can also be piggy-backed on outgoing data traffic, if any. In addition, the acknowledgement messages convey information regarding the status of the reception buffer of the receiving node.

Each byte of application data sent in a priority channel has a *sequence* number and the sequence number of the first byte in each data message is sent with the message. The sequence number is incremented with each byte sent so a data message of 63 bytes would cause the sequence number to increase with 63. The receiving node acknowledges received data by sending back the sequence number of the next byte it expects in the priority channel as well as how many more data bytes it can accept of any priority. These two numbers are called the *acknowledge sequence number* and the *window size*, respectively. Each acknowledgement message consists of several acknowledge sequence numbers, one for each priority channel, and one single window size. The acknowledgement messages are sent independently of the priority channels but are considered to have higher priority than any message sent by the application.

Through this information it is possible for each node to decide, at any time, the amount of data that can be transferred to each of the other nodes without overflowing the reception buffers. Note that this limit applies to an entire node and not to a specific priority level. In other words, all messages that are to be sent to a specific node share the same capacity. A strict priority order is enforced by sending available messages with the highest priority until the capacity limit is

reached, i.e., no low priority messages are sent if messages with higher priority are present. It is important to observe here that Balder Messages tries to send messages as soon as possible. The reason for this is to reduce latency even though eagerly sending messages instead of aggregating messages to the same destination wastes some bandwidth.

It is possible to have several pending receives on the receiving side as Balder Messages supports asynchronous receives. Naturally, these might receive from different priority channels. Also, it is possible that there are several messages pending to be received, e.g., lying in buffers in the network hardware. Even in this case the priorities are enforced by receiving the message of highest priority first.

In short, the flow control and priorities are implemented by:

- All application messages sent are acknowledged. With each acknowledgement message additional information is sent so that all nodes can calculate the amount of data they can send to any of the other nodes.
- Each message has a priority and the messages with highest available priority are sent first. The number of messages sent are limited by the capacity limit conveyed through the acknowledgement information. In other words, messages with low priority are not sent if messages of higher priority are available. Similarly on the receiving side, the message with the highest priority is processed first if there are several messages pending.

This way of implementing priorities has proven to be efficient as the added complexity in the flow control algorithm and the extra bandwidth needed is negligible. We will now go on and describe a model for control message latency which we developed in order to be able to quantify the performance impact of priorities on a variety of network technologies.

### 3. A quantitative model

#### 3.1. Model background

All messages exchanged in a software DSM system can be seen as belonging to one of two classes, *data messages* that carry shared memory content, and *control messages* that are used by the system to update the states in the coherency protocol or to synchronize nodes. The control messages are sent in a request-reply scheme so there is a reply message sent in response to each control message received. We will, in the discussion below, call the node that sends a specific control message the *client*, and the node responding to that control message the *server*.

We have developed a simple synthetic benchmark to evaluate the impact of priorities. The synthetic benchmark mimics the messaging behavior observed in the application Water from SPLASH on a software DSM system, see figure 2 [21]. This particular application was chosen as this is one of the applications where enqueueing of control messages occurs and the inner loop of Water is simple enough to be modeled in a

```

Client Process:
for(i = 0; i < P; i++)
  send(data_message);
start_timer();
send(control_message);
receive(reply); /* wait for reply */
stop_timer();

Server Process:
while(1) {
  if (data_message_available()) receive(data_message);
  if (control_message_available) {
    receive(control_message);
    send(reply);
  }
}

```

Figure 2. Pseudo-code for the synthetic benchmark used to develop the model.

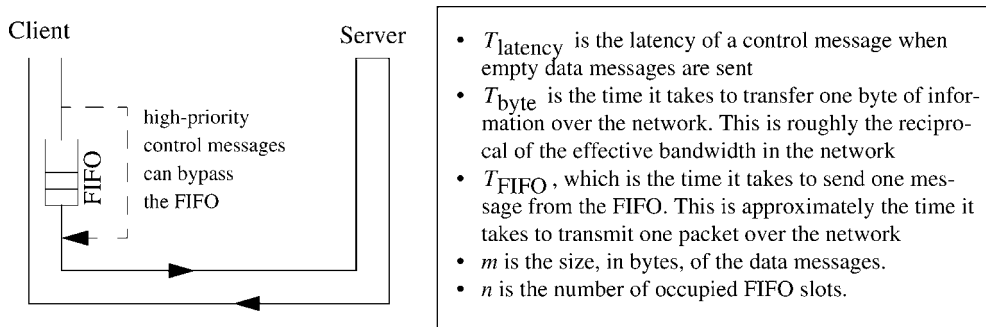


Figure 3. Overview of the communication model and its parameters.

few source code lines. It should be noted that the benchmark only models the communication behavior and that no consideration is taken for the computation phases in Water.

The synthetic benchmark consists of two processes running on different nodes. The client process sends  $P$  data messages of a given size, followed by a control message of a fixed small size, 32 bytes. It then waits for a reply to the control message and measures the time from the start of the transmission of the control message until the reception of the reply. The server process receives the messages and replies to control messages. The replies are of the same size as the control messages, i.e., 32 bytes and have the same priority. All message data, both from control, reply and data messages, is acknowledged.

A control message will be delayed, if it is sent directly after a number of data messages have been sent to the server, and if priorities are not used. This is due to the fact that messages in a traditional communication library are sent in FIFO order. Any handling of data messages on the server side will further delay the reply of the control message. From now on, when we discuss the latency of control messages we mean the time, as can be measured on the client side, from the start of a send operation of a control message until the completion of the receive operation of a corresponding reply.

### 3.2. Model overview

In order to fully understand the performance consequences of using priorities and to quantitatively describe the enqueueing behavior of the system we have developed a simple but useful model of the control message latencies in the synthetic benchmark application. We assume that only one control message is being sent at a time and thus there is no enqueueing of control messages. From the description of how Balder Messages

are implemented in section 2 it should be clear that the latency of a control message is a complex function of software and hardware latencies. However, for the purpose of this discussion we can model it according to figure 3.

In Balder Messages, certain buffers are found both at the receiving and transmitting side. However, those buffers actually work as one single FIFO buffer and so we have only one buffer in the model. In addition, some acknowledgement messages are sent between the client and the server. These messages are not explicitly modelled. Instead, they contribute to the values of all three parameters and in particular. It is assumed that all data messages have the same size and that the control messages and their replies have another much smaller size.

Using parameters from figure 3 the latency of control messages without priorities can be written as:

$$T_{\text{no priorities used}} = T_{\text{latency}} + T_{\text{byte}} \cdot \frac{m}{2} + T_{\text{FIFO}} \cdot n. \quad (1)$$

In equation (1),  $m$  is the data message size in bytes and  $n$  is the number of occupied FIFO slots. For the purpose of the model, we assume that the FIFO is infinite and that for each FIFO slot that is filled,  $n$ , the control message is delayed by time  $T_{\text{FIFO}}$ . This comes from the fact that even though messages are sent as soon as possible, all messages must also be acknowledged, see section 2.4. The time needed to send an acknowledgement is approximately independent of the size of the data messages.

The total latency also depends on the size of the data messages. This is due to the fact that during the time a data message is sent, a succeeding control message cannot be sent since the network can only transport one message at a time. The  $m/2$  factor in the equation comes from the fact that when a send operation of a control message is performed it is very

likely that a data message is being transmitted. However, we cannot cancel the transmission of the data message and we do not know how much of it that have been sent. We therefore assume that there is a data message being transmitted. The delay caused by this message depends on how much of it is left to be transmitted. We assume this fraction to be random from a rectangular distribution and so we approximate the delay caused by the data message with on the average half the time it takes to transmit a data message.

If control messages are sent with a higher priority, the FIFO is effectively bypassed and so we can in this case assume that the number of FIFO slots becomes zero. Measurements on Balder Messages indicates that this is correct. The term can  $T_{\text{FIFO}}$  therefore be dropped from the equation which leads to the following form when priorities are used:

$$T_{\text{priorities used}} = T_{\text{latency}} + T_{\text{byte}} \cdot \frac{m}{2}. \quad (2)$$

Note that we do not really care about where the actual delay of control messages occur when priorities are not used. It could happen at the client side, the server side or at both. In figure 3 we have shown the FIFO as being at the client side but there could, in fact be a FIFO also at the server side. This distinction is not important for the model which is why we have chosen to ignore it. In addition, the model only takes two priority levels into account and applies to the scenario in figure 2.

### 3.3. Validation of model

The model above has been validated by running the synthetic benchmark on top of Balder Messages using Fast Ethernet as interconnecting hardware. Table 2 summarizes the properties of the experimental platform.

We have augmented Balder Messages with instrumentation code so that the number of filled FIFO slots, i.e.,  $n$  in the model, can be measured. The control message latency was measured, with and without using priorities, for various data message sizes and various  $n$ . We applied linear regression to the measurement data in order to extract the model parameters. The values obtained using these parameters and the model closely resemble the measured data both with and without using priorities. Hence, we conclude that for this scenario, the model is accurate and valid.

Given the model presented in this section, we will now go over and discuss the effect of priority based messaging for other network technologies.

Table 2  
Experimental platform properties.

Item	Description
CPU	Dual Intel Pentium II, 350 MHz
Memory	256 MB
Network adapter	3Com 3c905
Operating system	Linux version 2.2.10

## 4. Extrapolation to other technologies

The model described above provides us with a tool to investigate the impact of priorities on time-critical messages with evolving software and hardware technology. The most important aspect of using priorities is the ability to lower the latency for performance critical control messages or in other words: the latency will increase if priorities are not used. Equation (3) combines equations (1) and (2) to show the relative increase of the latency of control messages,  $L$ , as a fraction of the absolute latency observed when priorities are used,

$$\begin{aligned} L &= \frac{T_{\text{no priorities used}} - T_{\text{priorities used}}}{T_{\text{priorities used}}} \\ &= \frac{T_{\text{latency}} + T_{\text{byte}} \frac{m}{2} + T_{\text{FIFO}}n - (T_{\text{latency}} + T_{\text{byte}} \frac{m}{2})}{T_{\text{latency}} + T_{\text{byte}} \frac{m}{2}} \\ &= \frac{T_{\text{FIFO}}n}{T_{\text{latency}} + T_{\text{byte}} \frac{m}{2}}. \end{aligned} \quad (3)$$

Obviously, the relative latency increase is a function of both the message size,  $m$ , and the number of previous messages in the FIFO,  $n$ . We can further re-write the formula using two parameters,  $A = T_{\text{FIFO}}/T_{\text{latency}}$  and  $B = T_{\text{byte}}/(2T_{\text{latency}})$ . We then get equation

$$L = A \frac{1}{1 + Bm} n. \quad (4)$$

Parameter  $A$  above is relatively independent on the technology by which a particular computer is designed since both  $T_{\text{FIFO}}$  and  $T_{\text{latency}}$  scale approximately in the same way.  $T_{\text{FIFO}}$  is roughly equivalent to the time it takes to send one message into the network.  $T_{\text{latency}}$  on the other hand is close to the roundtrip latency of the network for an efficient implementation. However,  $T_{\text{latency}}$  and  $T_{\text{FIFO}}$  relate to each other. Modern high performance networks are commonly connected to the nodes using I/O-cards connected a I/O-bus on the nodes. All data transferred to and from the network must be transferred over the I/O-bus and therefore in the ideal case,  $T_{\text{FIFO}}$  corresponds to one bus transfer while  $T_{\text{latency}}$  corresponds to four transfers, i.e., one send and one receive operation on each of the two participating nodes. The usage of packet switches in the network fabric also adds to  $T_{\text{latency}}$ . However, the latency of current, and probably also future, packet switches is relatively small [25]. These basic design trade-offs will likely remain unchanged and it is therefore probable that the value of  $A$  in equation (4) will be a little lower than 1/4 for the foreseeable future. Table 3 shows that this is indeed so for a number of current systems.

Parameter  $B$ , on the other hand, is a function of two relatively independent parameters: the basic overall latency of small messages and the data rate. With a high data rate, we get a small  $B$  and the queuing of messages in the FIFO will be more pronounced. The data rate of the current high performance computer networks is often limited by the bus with which the network interface is connected to the processing node and the properties of the bus therefore strongly influence the value of  $B$ . The use of long burst lengths and/or

Table 3

Network technologies, model parameters, and relative latency increase used in the discussion.

Network technology	$T_{\text{FIFO}}$	$T_{\text{latency}}$	$T_{\text{byte}}$	$A$	$B$
Fast Ethernet	68 $\mu\text{s}$	360 $\mu\text{s}$	110 ns	0.189	$0.16 \cdot 10^{-3}$
SCI	2 $\mu\text{s}$	9.4 $\mu\text{s}$	40 ns	0.213	$2.13 \cdot 10^{-3}$
SCI-scaled	875 ns	4.11 $\mu\text{s}$	13.2 ns	0.213	$1.63 \cdot 10^{-3}$
LAPI	28 $\mu\text{s}$	109 $\mu\text{s}$	40 ns	0.257	$0.18 \cdot 10^{-3}$
RapidIO	36 ns	166 ns	3 ns	0.218	$9 \cdot 10^{-3}$
RapidIO-scaled	36 ns	166 ns	0.3 ns	0.218	$0.9 \cdot 10^{-3}$

wide buses, for instance, effectively lowers  $B$ . However, it should be noted that a high data rate by itself does not mean a low  $B$  as lowering  $T_{\text{latency}}$  will make it possible to increase the data rate and still maintain an unchanged  $B$ .

Six different network technologies, with their  $A$  and  $B$  values, are summarized in table 3. Some of these technologies are current state-of-the-art while others are estimated by extrapolation. Figure 4 shows the relative increase of the control message latency for various data message sizes and one FIFO slot filled when priorities are not used. We see that the performance impact of priorities is significant for all technologies, even those with very low latency.

The values for Fast Ethernet were obtained from the model validation as described in section 3.3. The values for SCI were estimated from measurements on Scali AS's implementation of MPI [19] on Dolphin Interconnect Solutions' 32-bit PCI adapter [6]. The scaled SCI values were extrapolated from the non-scaled SCI values to match that of a 66 MHz 64-bit PCI SCI adapter. The LAPI values were estimated from direct measurements on LAPI [20] on an IBM SP-2 while the values for the upcoming RapidIO technology were estimated from performance values made public by Motorola [14]. The scaled RapidIO values were obtained by increasing the data rate ten times while keeping the latencies unchanged.

It should be noted here that the upcoming InfiniBand architecture would be a better representative than RapidIO for high performance networks [8]. However, at the time of this writing, no performance data, known to the authors, for InfiniBand has been released. Please also note that the values obtained by estimation and extrapolation should only be seen as an indication of the behavior of a corresponding system. Both parameter  $A$  and  $B$  depend on the actual hardware and software implementation of the communication library.

As we can see in figure 4, LAPI has a much larger relative increase and thus enqueueing effect than the other technologies and the main reason for this is that the latencies in the network are not matched to the high data rate which results in a small value of  $B$ . It is interesting to see that 100Base-T perform better in this respect, i.e., it has a lower relative latency increase than LAPI, although its overall latencies are much larger. Again this is due to the fact that the data rate is lower and so the data rate and the latencies are much better matched.

Both LAPI and 100Base-T are designed primarily for message passing. SCI and RapidIO on the other hand are designed to provide at least a crude shared memory programming model which means that a great deal effort has been spent on keeping the latencies low. It should be noted that even though the latencies are low, both SCI and RapidIO provide a very high data rate. However,  $B$  is kept at a high level because of the low latencies. Both the scaled SCI and RapidIO values are higher than the non-scaled. Here, the data rate is higher so that the latencies become mismatched and parameter  $B$  therefore becomes smaller.

From the discussion above we can see that there are two possible ways to avoid delays as a consequence of message queues. Either we introduce priorities in the communication library or we aggressively reduce the latency in the driver software and network hardware. The latter might in many cases be difficult to achieve and we therefore suggest that user-

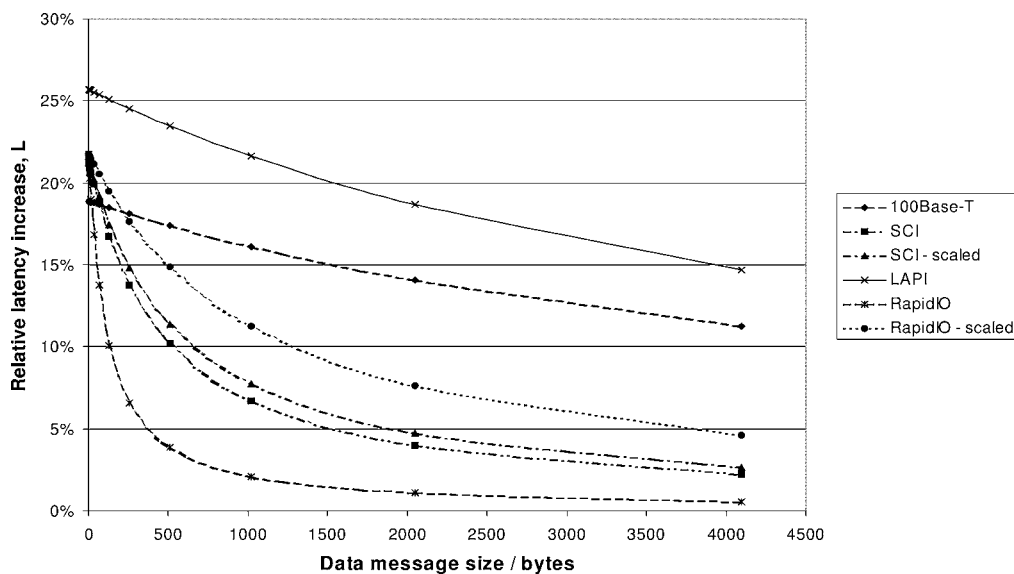


Figure 4. Relative increase of the latency of control messages,  $L$ , when priorities are not used, as a function of data message size. The number of filled FIFO slots,  $n$ , is one.

space communication libraries support message priorities as they can be efficiently implemented in the higher levels of an existing flow control algorithm. This approach is portable which is particularly beneficial for software DSM systems.

## 5. Conclusions

Modern commodity network interconnects and user space communication libraries provide low latency and high bandwidth, and the performance is increasing. The performance of processors, however, is improving much more rapidly than the performance of networks and so the networks have become performance bottlenecks. It has previously been shown that, in software DSM systems, latency critical messages are delayed by less critical messages, thus limiting the performance.

In this paper we have argued that priorities among the messages should be introduced to solve this problem. An implementation of a communication library with priorities has been done and a quantitative performance model has been developed. The model shows that, over a wide range of network design parameters, priorities are beneficial. The latency of critical messages is reduced with up to 25% for each delaying message if priorities are used. The performance impact depends more on the network bandwidth and the size of the delaying messages than on the overhead latency. However when the delaying messages are small, the reduction in latency is around 20% per delaying message regardless of bandwidth.

We also found that networks with high bandwidth benefit more from priorities than those with lower bandwidth. In general, the impact of priorities is smaller if the latency of the network is low compared to the bandwidth. Since lowering latency is in practice very difficult, and the bandwidth demands are increasing, we propose that priorities should be introduced into the communication libraries as priorities can efficiently be implemented in flow control algorithms.

Future studies involve studying how priorities should be assigned to different types of messages in a software DSM system and evaluating the impact on end application performance.

## Acknowledgements

The research in this paper has been financially supported by the Swedish Research Council for Engineering Sciences under contract number TFR 1999-376. We also gratefully acknowledge the use of computing resources at Centre for Parallel Computing at Royal Institute of Technology, Stockholm.

## References

- [1] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, TreadMarks: Shared memory computing on networks of workstations, *IEEE Computer* 29(2) (1996) 18–28.
- [2] A. Bilas, L. Iftode and J.P. Singh, Evaluation of hardware write propagation support for next-generation shared virtual memory clusters, in: *Proceedings of 1998 International Conference on Supercomputing* (1998) pp. 274–281.
- [3] A. Bilas, C. Liao and J.P. Singh, Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems, in: *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, Georgia, May 1999, pp. 282–293.
- [4] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic and W.-K. Su, Myrinet: A gigabit-per-second local area network, *IEEE Micro* 15(1) (February 1995) 29–36.
- [5] Compaq Computer Corp. et al., *VI Architecture Specification V1.0* (December 1997).
- [6] Dolphin Interconnect Solutions Inc., *PCI-SCI Adapter Card D320/D321 Functional Overview*, version 1.01 (November 1999).
- [7] Dubnicki et al., Design and implementation of virtual memory-mapped communication on Myrinet, in: *Proceedings of 1997 International Parallel Processing Symposium* (IEEE Computer Society Press, Los Alamitos, CA, 1997) pp. 388–396.
- [8] InfiniBand Trade Association, *InfiniBand specification 1.0.a* (June 2001) available at <http://www.infinibandta.org>.
- [9] Information Sciences Institute, University of Southern California, *RFC 793: Transmission Control Protocol* (September 1980).
- [10] S. Karlsson and M. Brorsson, A comparative characterization of message communication in applications using MPI and shared memory on an IBM SP2, in: *Proceedings of 1998 Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, Las Vegas, 31 January–1 February 1998, pp. 189–201.
- [11] S. Karlsson and M. Brorsson, An infrastructure for portable and efficient software DSM, in: *Proceedings of 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, Rhodes, Greece, 25 June 1999; also available from Department of Information Technology, Lund University, PO Box 118, SE-221 00 Lund, Sweden.
- [12] P. Keleher, *Lazy release consistency for distributed shared memory*, Ph.D. thesis, Rice University (1994).
- [13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1 (12 June 1995).
- [14] Motorola Semiconductor, *RapidIO: An Embedded System Component Network Architecture* (February 2000).
- [15] K. Omang and B. Parady, Scalability of SCI workstation clusters, a preliminary study, in: *Proceedings of the 11th IEEE International Parallel Processing Symposium (IPPS'97)*, Geneva, April 1997, pp. 750–755.
- [16] S. Pakin, M. Lauria and A. Chien, High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet, in: *Proceedings of Supercomputing '95* (1995).
- [17] L. Prylli and B. Tourancheau, BIP: A new protocol designed for high performance, in: *Proceedings of PC-NOW Workshop, Held in Parallel with IPPS/SPDP'98*, Orlando, USA, 30 March–3 April 1998, pp. 272–485.
- [18] R. Samanta, A. Bilas, L. Iftode and J.P. Singh, Home-based SVM protocols for SMP clusters: Design and performance, in: *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, Nevada, January 1998, pp. 113–124.
- [19] Scali AS, *ScaMPI Installation and User's Guide*, version 1.6 (1999).
- [20] G. Shah et al., Performance and experience with LAPI – a new high-performance communication library for the IBM RS/6000 SP, in: *Proceedings of IPPS/SPDP'98*, Orlando, FL, 30 March–3 April 1998, pp. 260–266.
- [21] J.P. Singh, W.-D. Weber and A. Gupta, SPLASH: Stanford parallel applications for shared-memory, *Computer Architecture News* 20(1) (March 1992) 5–44.
- [22] R. Stets et al., CASHMERE-2L: Software coherent shared memory on a clustered remote-write network, in: *Proceedings of SOSP '97*, Saint Malo, France, October 1997, pp. 170–182.
- [23] H. Tezuka, A. Hori and Y. Ishikawa, PM: A high-performance communication library for multi-user parallel environments, Technical report TR-96015, Real World Computing Partnership (1996).
- [24] T. von Eicken, D.E. Culler, S.C. Goldstein and K.E. Schauer, Active messages: A mechanism for integrated communication and computa-

tion, in: *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Qld., Australia, May 1992, pp. 256–266.

- [25] W. Weber et al., Mercury interconnect architecture: A cost-effective infrastructure for high-performance servers, in: *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, Denver, Colorado, May 1997, pp. 98–107.



**Sven Karlsson** is a Ph.D. student at KTH, Royal Institute of Technology in Stockholm, Sweden. He holds a M.Sc. from Lund University. Mr. Karlsson's research interests are in system software for parallel computers, most notably software distributed shared memory systems, and in compilers for parallel computers. His home page is: <http://www.imit.kth.se/~svenka>.  
E-mail: [svenka@it.kth.se](mailto:svenka@it.kth.se)

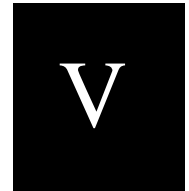


**Mats Brorsson** is a Professor in computer architecture at KTH, Royal Institute of Technology in Stockholm, Sweden. He holds a Ph.D. and M.Sc. from Lund University. Dr. Brorsson's research interests are in energy-efficient computer architectures, chip-multiprocessors and other parallel architectures and in system software for parallel computers. He is also interested in computer organization and architecture education, particularly in training material supporting the learning process of students. His home page is: <http://www.imit.kth.se/~matsbror>.



S. Karlsson, S-W. Lee, and M. Brorsson, A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory, in *Proceedings of 9th International Conference on High Performance Computing (HiPC 2002)*, December 2002, pp. 195-206

© Copyright 2002 Springer Verlag, All rights reserved. Reprinted with permission.





# A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory

Sven Karlsson<sup>1</sup>, Sung-Woo Lee<sup>2</sup>, and Mats Brorsson<sup>1</sup>

<sup>1</sup> Royal Institute of Technology, KTH, Stockholm, Sweden  
{Sven.Karlsson,Mats.Brorsson}@imit.kth.se

<sup>2</sup> Ditto Information Technology Inc., South Korea  
swlee@dittotec.com

**Abstract.** OpenMP is a relatively new industry standard for programming parallel computers with a shared memory programming model. Given that clusters of workstations are a cost-effective solution for building parallel platforms, it would of course be highly interesting if the OpenMP model could be extended to these systems as well as to the standard shared memory architectures for which it was originally intended.

We present in this paper a fully compliant implementation of the OpenMP specification 1.0 for C targeting networks of workstations. We have used an experimental software distributed shared memory system called Coherent Virtual Machine to implement a run-time library which is the target of a source-to-source OpenMP translator also developed in this project.

The system has been evaluated using an OpenMP micro-benchmark suite as to evaluate the effect of some memory coherence protocol improvements. We have also used OpenMP versions of three Splash-2 applications concluding in reasonable speedups on an IBM SP2 machine. This also is the first study to investigate the subtle mechanisms of consistency in OpenMP on software distributed shared memory systems.

## 1 Introduction

Workstation clusters are cost-effective when it comes to implementing platforms for parallel computation. Initiatives such as the Beowulf project have gained attention to these platforms and there are now several such systems among the top 500 most powerful computers in the world [12, 20]. They are a viable alternative to SMP servers, even for small systems, just because they are cost-effective. There is one major drawback, though. The topology and hardware of a cluster of workstations only support a message-passing programming model. However, it is widely acknowledged that a shared memory model, such as OpenMP [13, 14], is preferred by most programmers when it comes to ease of programming and maintenance of the software base [19]. Software distributed shared memory, software DSM, systems have been developed for quite some time now to provide a shared memory programming model on clusters of workstations [2, 6, 8, 9, 18, 16]. It

has been shown that it is indeed possible to achieve reasonable performance for a range of applications using modern software DSM systems even though there is still much work to do as to improve performance [7, 15].

In this paper we present a fully compliant OpenMP implementation of the 1.0 specification for C for a cluster of workstations. It is based on an existing software DSM system called Coherent Virtual Machine [8] and is embodied in the form of a run-time library and an OpenMP compilation system originally developed for an SMP target [5].

The performance of the SMP version of the OpenMP implementation is on par with commercial OpenMP compilers. For the Software DSM version, we have changed the allocation of shared variables since Software DSM systems assume explicit allocation of shared memory and OpenMP programs assumes shared storage by default. The speedup of the same, unmodified, applications on the Software DSM system with eight processors ranges from 4.1 to 6.5 which should be compared to 6.8 to 7.8 achieved on an SGI Origin 3800 which is a distributed shared memory architecture machine. Given the relatively small data set used in these programs, and the difference in inter processor communication time, this difference is not very big. It is important to note here that we are referring to a full implementation of the OpenMP 1.0 specification for C, not a modified version to suit the software DSM system better.

In the next section we briefly introduce the OpenMP specification for C, in section 3 we present the compilation system and the run-time library we developed for shared memory target platforms. This is followed in section 4 with a discussion on software DSM systems. Section 5 outlines the implementation aspects of the run-time library for software DSM followed by a performance evaluation in section 6 before the paper is concluded.

## 2 OpenMP

OpenMP is a recent effort to provide an informal standard for how shared memory parallel computers should be programmed. It is a thread-level fork-join programming model based on compiler directives. Several types of directives are provided. Parallel directives are used to spawn parallel activity, and there are work sharing directives to make the different threads do different things. Both loop-level parallelism and functional parallelism are supported by the directives. Directives for critical sections and other synchronization primitives are also available as well as a number of directives to control which variables are shared and private to threads. However, in total there are relatively few directives which makes it quite easy to start using OpenMP.

In addition to the directives, OpenMP specifies a number of intrinsic functions that can be used to divide work based on the number of parallel threads and thread identities similar to how the rank is used in MPI programs [11].

The following example shows an excerpt of a C program with a parallel region and one of the more common work-sharing constructs. All examples are in this

paper given in C since the software DSM OpenMP compilation system currently only targets C programs. Equivalent structures exist for Fortran programs.

```

1   #pragma omp parallel
2   {
3       foo();
4
5   #pragma omp for
6       for (i = 0; i < 200; i++)
7           a[i] = bar(i);
8   }
```

Line 1 in the example above starts a new parallel region using the parallel directive. OpenMP directives are inserted as pragma compiler directives and typically operate on the lexically next C statement. In this case, a team of threads is created and the compound statement beginning on line 2 is executed in parallel by each of the threads. The number of threads that is created is determined at runtime based on the value of environment variables. The function `foo` is called by each of the created threads in parallel. The programmer has to make sure that `foo` does not have any side-effects that can cause race conditions.

Line 5 contains a work-sharing construct, the `for`-directive. This construct means that the iterations of the `for`-loop that follows are divided among the threads in the parallel region. Each thread executes its share of the, in this case, 200 iterations. It is the responsibility of the programmer to make sure that the loop iterations are independent.

Notice that without the directives, the program is still a valid program making it easy to maintain a sequential and a parallel version of the code with the same source code files.

We will now have a look at how the SMP-version of the OpenMP translator that we have developed works before we look at what changes we needed to do as to support software distributed shared memory.

### 3 The OpenMP Compilation System

We have implemented a C compilation system for OpenMP. The basis for this system is a source-to-source code translator which given a source code with OpenMP directives generates a new transformed source code with calls to a runtime library. The library has primitives for spawning and synchronizing threads. The actual translation process is described in detail elsewhere and we only provide a rough sketch here [5]. In essence, each parallel region is transformed into a function called a parallel function. A pointer to one of these parallel functions is passed as an argument to a library function when threads are to be spawned and the spawned threads all execute the parallel function. The translator also inserts various library calls when needed as to implement work sharing and thread synchronization.

There are a number of problems when it comes to implementing OpenMP on a cluster of workstations. First of all we need a shared address space, and secondly there are some particular issues of OpenMP that forces us to modify the OpenMP translator when targeting a software DSM system. We will first discuss the issue of supporting a shared address space at all.

## 4 Software DSM Systems

A software distributed shared memory, software DSM, system implements a shared memory programming model on top of a machine architecture that does not support shared memory in hardware. The software DSM system is a piece of user-level software that intercepts memory accesses that cannot be satisfied locally, sends messages to the node(s) that has the requested memory contents and resumes execution after that the local node has received what it needs.

We have based our implementation of the run-time library on top of an existing software DSM system, Coherent Virtual Machine, CVM [8]. CVM implements a number of consistency models, but we have focused on using the Home-based Lazy Release Consistency model, HLRC [22]. Like many other software DSM systems [2], HLRC is page based. This means that the system uses the virtual memory page protection mechanism to detect accesses to shared memory and memory contents are replicated on a virtual memory page basis.

In order to simplify the protocol, each page in an HLRC system has a home node which always holds an up-to-date copy of the page. Thus, whenever a processor accesses a page which is not present locally, a copy is retrieved from the home. Several nodes can have write permissions to the same page in which case several data structures are used to locally keep track of changes to the page. These changes are transferred at synchronization points to the home node where they are merged. The synchronization points also forces updated pages to be invalidated. The synchronization points normally occur when synchronization primitives are performed.

We chose to base our OpenMP run-time library on HLRC because of its relative simplicity and because it performs quite robustly. There is, however, no implicit assumption in our system on HLRC so it should be relatively straightforward to port the system also to other types of software DSM protocols, e.g. homeless protocols.

In the next section we describe why we cannot use CVM and HLRC just as they are as target for our OpenMP translator and why we had to augment both the translator and CVM. We also briefly describe some implementation aspects of the translator and the run-time library.

## 5 OpenMP on a Software DSM System

For obvious reasons we would like to implement the OpenMP run-time library, without changing the interface of the existing SMP implementation. This enables us to keep the exact same OpenMP translator for both the SMP version of the

OpenMP implementation and the software DSM version. Unfortunately, this is not entirely possible since variables declared outside a parallel region, and visible from within that parallel region, are shared among the threads by default. It is possible to explicitly declare variables to be shared, but the OpenMP specification does not require it. This is in conflict to all previous software DSM systems that require shared variables to be explicitly allocated. Therefore, we have augmented the translator to automatically identify all variables that could possibly be accessed from within a parallel region, and thus should be shared, and to allocate them explicitly in the shared address space in the translated program. Automatically allocated variables which usually are stored on a stack can also be shared and to handle this we allocate these variables on a single shared stack. However, automatically allocated variables that are not shared are still put on the respective thread's stack. Naturally, the support for the shared stack and the allocation primitives have been added to the run-time library.

Mapping OpenMP's memory model onto the HLRC model of our software DSM system also caused a few potential performance problems. The only construct that enforces consistency in OpenMP is the flush construct. The flush operation requires that all shared memory variables are written back to the shared memory and it is thus a synchronization point. Most OpenMP constructs that contain some sort of synchronization implies that a flush is performed and there is also the possibility to insert explicit flush statement in an OpenMP program via directives.

A flush with no argument implicitly forces the system to perform consistency operations for the entire shared memory area. The OpenMP flush construct can also contain arguments with named variables in which case the consistency is performed only on those variables. We call the former global-flush and the latter selective-flush. Since OpenMP is intended to be used on a variety of consistency models, including sequential consistency, total store ordering and release consistency [1], the specification requires that implicit global-flush operations are made in connection to many OpenMP constructs, such as entry and exit of parallel regions, barriers, critical sections and work-sharing constructs. On an SMP that implements a sequential consistent memory model, a global-flush is not much more time-consuming than writing all register allocated variables to memory, but for a more relaxed memory consistency model, and in particular on a software DSM system, global-flushes can be very expensive operations. Therefore, it is important that we try to use selective-flush, where only the named variables are affected, as much as possible and that we should reduce the number of global-flushes if possible.

We have implemented the functionality of the consistency operation of global-flush by using a scheme very similar to how locks are handled in a software DSM system. An imaginary lock called the flush-lock is used for this purpose. A flush-operation consists of an acquire of the flush-lock, immediately followed by a release. The acquire and release operations are implemented using message passing making it possible to piggy-back information on the messages sent between the previous flush-lock holder and the acquirer. The information transferred makes it

possible to invalidate any updated pages and thus obtaining a coherent memory state on all cluster nodes.

Selective-flush only need to force a coherent memory state for the specified variables and it should thus cause a much lower overhead. This is not considered in current page-based software DSM systems and so we have to augment the coherency protocols. To do this we use the flush-lock again but only the information needed to update the specified variables are sent between the nodes. This greatly reduces the overhead.

It should be noted here that OpenMP also has intrinsic lock functions, however, these locks only perform mutual exclusion and does not enforce consistency.

Work-sharing constructs in OpenMP can have a reduction clause which causes a reduction operation on a variable. The simplest implementation of the reduction operation is to use a critical section where the global reduction variable is updated from the local copies of each process. However, the OpenMP specification states that the result of a reduction is not guaranteed to be visible until a barrier and this made possible for us to merge the reduction operation with the barrier thus eliminating quite a bit of coherency traffic.

The atomic construct in OpenMP has almost the same semantic as a critical section but allows for a more efficient implementation. The code excerpt below is an example of the atomic operation.

```
1 #pragma omp atomic
2 v = v + foo();
```

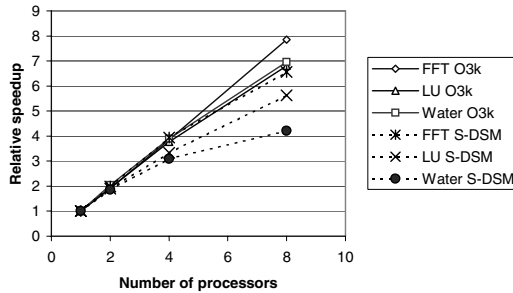
The update of variable *v* must be done atomically. The invocation of the function *foo*, however, is not guaranteed to be atomic and it should therefore not contain any side-effects. The simplest implementation of atomic is to use a critical section with *global-flush*. However, we have a chance to optimize it using a refined version of our selective-flush approach. We here first execute function *foo* to get its return value, then acquire the flush-lock asking for updates for variable *v* only but we will not release it until we have updated *v*'s value exploiting the mutual exclusion properties of the flush-lock.

Before presenting some performance measurements from our prototype we would like to point out that we have through-out the run-time tried to reduce the number of messages sent by merging messages whenever possible.

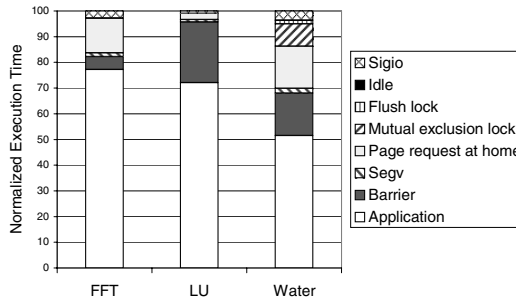
## 6 Performance

We have used an IBM SP2 machine with 160 MHz POWER2 uniprocessor nodes as our experimental platform. All the nodes have 256 megabytes of memory and are interconnected with a 110 megabyte/s network. The nodes were running AIX 4.2. A virtual memory page has a size of 8 kilobytes in this version. We have also used a SGI Origin 3800 with 400MHz MIPS R12000 processors for making comparisons.

We evaluated our prototype with three applications from SPLASH-2 [21]. The applications and data sets used were 1D-FFT ( $2^{20}$  points), LU ( $1024 \times 1024$



**Fig. 1.** The relative speedup of the three applications on an SGI Origin 3800 (O3k) and our prototype running on an IBM SP2 (S-DSM)

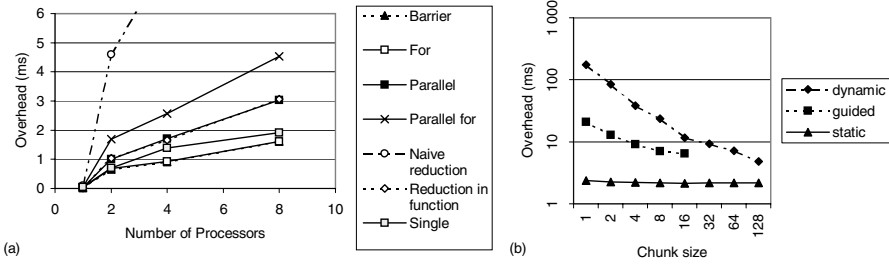


**Fig. 2.** Execution time breakdown for the three applications

matrix), and Water-spatial (512 molecules, 6 time steps). All applications were run with up to 8 nodes. The EPCC micro-benchmarks were used to measure the overhead of OpenMP constructs and scheduling schemes [3].

Figure 1 presents the relative speedups for the three SPLASH-2 applications. All applications were also run the SGI machine using SGI’s own OpenMP compiler. As expected, the speedups are better on the SGI, which is a CC-NUMA and implements shared memory in hardware, but the differences are not extremely large. We believe the good performance of our prototype comes from the coherency protocol enhancements we have made. The OpenMP specification allowed enough freedom to do these optimizations and this indicates that OpenMP is a viable approach for a broad range of parallel computing platforms.

Figure 2 shows the normalized execution time breakdown observed in the software DSM system for the three applications. The fraction labeled “Application” is the application busy time fraction. For 1D-FFT and Water-spatial, the most dominant overhead is page requests which is labeled “Page request at home”. Page requests are very common as a page has to be requested from the home node each time a page is accessed after being invalidated. The second largest overhead is the barrier fraction, labeled “Barrier”. This time includes



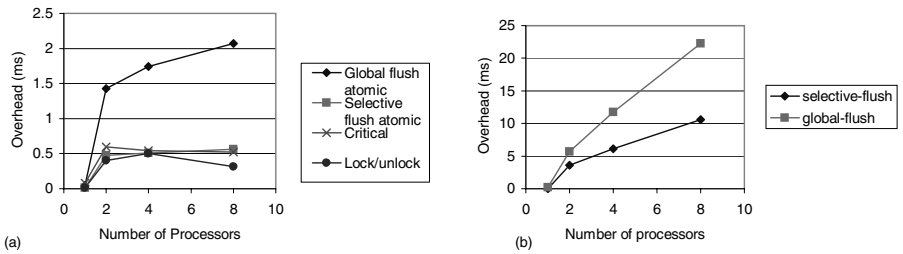
**Fig. 3.** OpenMP constructs (a) and work sharing scheduling overheads (b) in milliseconds

the time it takes to perform the barrier operation, the time to fully update the home nodes and possible idle time because of load imbalance. Finally, in Water, mutual exclusion locks, labeled “Mutual exclusion lock”, constitute a large overhead as well. Whenever possible, we combine the flush-lock operations, labeled “Flush lock”, with ordinary mutual exclusion lock operations which is here seen as a relatively large overhead for the mutual exclusion locks. The fraction of the execution time it takes to handle incoming messages is labeled “Sigio” while the time fraction it takes to handle page faults is labeled “Segv”. The idle time in the system is simply labeled “Idle”. These values suggest that our approach to optimize the flush operations is valid.

Figure 3 (a) shows the measured overheads for a few different OpenMP constructs with implied barriers as reported by the EPCC OpenMP micro-benchmarks.

We here clearly see the difference in overhead between using a naïve reduction implementation, using global-flush and locks, and our optimized implementation that delays the reduction operation until the next barrier. The overhead of a parallel construct with a reduction using the naïve implementation, labeled “Naive reduction”, is extremely high due to the high cost of flush operations and the serialization caused by the locks. The optimized implementation, labeled “Reduction in function”, on the other hand has about the same overhead as a plain parallel construct and we can thus argue that with our implementation the reduction adds no extra overhead.

The parallel for construct is the combination of the parallel directive and a work-sharing construct with static scheduling. It has a much larger overhead than the parallel construct and this is due to an excess barrier introduced by our OpenMP translator. This excess overhead will be removed in the next version of OpenMP translator and was due to the translator treating the parallel for construct as two separated parallel and for constructs and inserted a not needed barrier at the end of the for construct. With these changes we expect the parallel for construct to have essentially the same overhead as a plain parallel construct.



**Fig. 4.** Mutual exclusion (a) and flush (b) overhead in milliseconds

The barrier construct has approximately the same overhead as a plain work-sharing construct, labeled “For”, and this is expected as the static for work-sharing construct is very efficient. We here basically only see the overhead of the barrier at the end of the for construct.

The single construct has a little bit higher overhead than a plain for. Our run-time causes a slight overhead when starting a new work-sharing construct and this leads to the difference in overheads. When measuring the overhead of the for construct the micro-benchmark runs several iterations and so the start-up overhead is not as pronounced in the values of the for construct.

Figure 3 (b) shows the measured overheads of different schedules in work-sharing constructs. The x-axis in the figure shows the chunk size used when distributing iterations of a for-loop on eight processors with 1024 iterations. The static schedule distributes the iterations statically and equally to the different processes and therefore has very little run-time overhead. The dynamic schedule distributes iterations dynamically in chunks and the guided schedule is a dynamic schedule where the chunk size from start is the number of iterations divided by the number of processes. As expected, both the dynamic schedules have a substantial overhead compared to the static scheduling policy. Also, our prototype has the same behavior as a SMP implementation would have although the overheads are higher.

Figure 4 (a) shows the overheads for a few mutual exclusion constructs as measured by the EPCC micro-benchmarks. We see the striking difference in overhead between using global-flush, labeled “Global flush atomic” in the figure, to implement atomic and our optimization using a variant of our selective-flush approach, labeled “Selective atomic flush” in the figure. The overhead of a critical construct is almost as low as the one of just using the intrinsic functions, i.e. lock and unlock. This comes from the fact that the loop that measures the overhead of critical constructs do not touch any shared data and so very little coherency information is sent. The optimized version of atomic has approximately the same overhead as the intrinsic functions. We believe this comes from the fact that the rather large lock latencies in the system can hide the coherency work done in the atomic construct.

Finally, figure 4 (b) shows the performance difference of only the global-flush and selective flush for a micro-benchmark based on an example from the OpenMP specification [14]. There are two shared variables in this micro-benchmark, a data array and a synchronization variable array. In one iteration each processor updates an element in the data array and then signals this update to a neighboring processor with selective-flush. Again, we see the growing difference between the two flush-implementations as the number of processors increases.

## 7 Related Work

There are three related studies on OpenMP for software DSM systems known to us.

One study was made by H. Lu et al. [10]. They deviate from the OpenMP specification in two ways. Variables in parallel region are treated as private by default and all shared variables must be explicitly declared. Furthermore to alleviate the expensive run-time cost for flush, they introduced condition variables and semaphores, which would replace flushes in pipelined or task-queue-based parallelism.

Y. C. Hu et al. reports on an OpenMP implementation on networks of SMPs [4]. Here no apparent deviations from the OpenMP specification has been made. However, it is unclear how flushes are handled.

M. Sato implemented a OpenMP translation tool which instruments the application's code with communication primitives as to uphold coherency [17]. However, as to work their system required a specialized network interconnect and runtime environment.

In short, our work differs from the related work above in that we fully implement the OpenMP specification in a highly portable way. We have put considerable effort into a correct and efficient implementation of the flush operation. We also evaluate the performance of our system using several benchmarks.

## 8 Conclusions

Clusters of workstations are becoming more and more sought-after as cost-effective parallel computing platforms. We have in this paper presented a system consisting of an OpenMP translator, a run-time library, and a software distributed shared memory system that together form a fully compliant implementation of the C OpenMP specification version 1.0 for a cluster of workstations. It is one of the first in this kind and we expect to release an open source version of both the compiler and the software DSM system shortly. In contrast to previous work on OpenMP for clusters of workstations/SMPs, we have focused on the compliance aspect of OpenMP and to provide the first published data on the overheads of OpenMP constructs in a software DSM system.

We find the performance of the resulting system quite satisfactory. The speedups are similar to previously reported results and, although the OpenMP

construct overheads are high, we find that they exhibit the same general behavior as on SMP systems when the number of processors scale. During the work on the system we have found numerous ways to improve performance. Most notably through a cooperation between the OpenMP translator and the software DSM run-time library. This will be the focus of future research. We will also work on adding support for SMP nodes.

It was surprisingly complicated to get the semantic meaning of some of the more subtle issues of OpenMP correct. Flushes of individual variables turned out to be particularly troublesome to implement correctly.

## Acknowledgements

The work in this paper has been partly financed by the European Commission in the Intone project under contract number IST-1999-20252. The OpenMP translator was in part implemented also by Håkan Zeffler, Samer Al-Kassimi and Örjan Friberg. Their contribution is hereby greatly acknowledged. We also gratefully acknowledge the use of computing resources at the Centre for Parallel Computing at the Royal Institute of Technology, Stockholm. Anna Thelin wrote the OpenMP versions of the SPLASH-2 benchmarks.

## References

- [1] S. V. Adve, K. Gharachorloo, *Shared memory consistency models: a tutorial*, IEEE Computer, Volume: 29 Issue: 12, Dec. 1996 pp. 66–76. 199
- [2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel. *TreadMarks: Shared Memory Computing on Networks of Workstations*, IEEE Computer, Vol. 29, no. 2, pp. 18-28, February 1996. 195, 198
- [3] J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, in *Proceedings of the First European Workshop on OpenMP*, Sept. 1999, pp. 99-105. <http://www.it.lth.se/ewomp99>. 201
- [4] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel, OpenMP for Networks of SMPs, in *Proceedings of IPPS/SPDP'99*, April 1999, pp. 302-310. 204
- [5] S. Karlsson, et al., *A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing*, Technical Report, Department of Microelectronics and Information Technology, KTH, Royal Institute of Technology, 2002. 196, 197
- [6] S. Karlsson, and M. Brorsson, Producer-push-a protocol enhancement to page-based software distributed shared memory systems *Proceedings of 1999 International Conference on Parallel Processing*, September 1999, pp. 291-300. 195
- [7] S. Karlsson, and M. Brorsson, A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SP2, in *Network-Based Parallel Computing. Communication, Architecture, and Applications. Second International Workshop, CANPC '98*, January 1998 pp. 189-201. 196
- [8] P. Keleher, *The CVM Manual*, Technical report, Computer Science Department, University of Maryland, May 1995. 195, 196, 198

- [9] K. Li, IVY: A shared Virtual Memory System for Parallel Computing. In *Proceedings of 1988 International Conference on Parallel Processing*, 1988, pp. 94-101. 195
- [10] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on Networks of Workstations, in *Proceedings of Supercomputing'98*, Nov. 1998. 204
- [11] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1, June 12, 1995. 196
- [12] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon, *TOP500 Supercomputer Sites, 18th ed.*, Technical report, Lawrence Berkely National Laboratory LBNL-49122, Nov. 2001. 195
- [13] OpenMP consortium, *OpenMP: A Proposed Standard API for Shared Memory Programming*, White paper, <http://www.openmp.org>. 195
- [14] OpenMP consortium, *OpenMP C and C++ Application Program Interface*, Version 1.0, October 1998. 195, 204
- [15] E. W. Parsons, M. Brorsson and K. C. Sevcik, *Predicting the Performance of Distributed Virtual Shared Memory Applications*, IBM Systems Journal, Volume 36, No. 4, 1997, pp. 527-549. 196
- [16] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh, Home-based SVM protocols for SMP clusters: Design and performance, in *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture (HPCA-4)*, Las Vegas, Nevada, January 1998, pp. 113-124. 195
- [17] D. M. Sato, Design of OpenMP Compiler for an SMP Cluster, in *Proceedings of First European Workshop on OpenMP*, Sept. 1999. <http://www.it.lth.se/ewomp99>. 204
- [18] D. J. Scales, K. Gharachorloo, and C. A. Thekkath, Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grained Shared Memory, in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, October 1996, pp. 174-185. 195
- [19] J. P. Singh, A. Gupta and M. Levoy, *Parallel Visualization Algorithms: Performance and Architectural Implications*, IEEE Computer Magazine, July 1994, pp. 45-55. 195
- [20] T. Sterling, D. Becker, D. Savarese, et al., BEOWULF: A Parallel Workstation for Scientific Computation, in *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, Vol. 1, August 1995, pp. 11-14. 195
- [21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations, in *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 24-36. 200
- [22] Y. Zhou, L. Iftode, and K. Li, Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems, in *Proceedings of the 2nd Symposium on Operating System Design and Implementation*, October 1996, pp. 75-88. 198

S. Karlsson, and M. Brorsson, A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing, accepted for publication in *Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, November 2004



# A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing

Sven Karlsson and Mats Brorsson

Department of Microelectronics and Information Technology

KTH, Royal Institute of Technology, Sweden

email: {Sven.Karlsson,Mats.Brorsson}@imit.kth.se

## **ABSTRACT**

OpenMP is an informal industry standard for programming parallel computers with a shared memory and has during the last few years achieved considerable acceptance in both the academic world and the industry. OpenMP is a thread-level fork-join programming model and relies on a set of compiler directives. An OpenMP aware compiler uses these directives to generate a multi-threaded application. In practice, an OpenMP run-time library is also needed as OpenMP specifies a set of run-time library calls.

In this paper we report on a free OpenMP compiler and run-time library infrastructure. We present an OpenMP compiler for C called OdinMP and briefly discuss the run-time library that the compiler targets. The source code to both the compiler and the run-time libraries are available and can be freely used for OpenMP research.

The compilation system is evaluated using the EPCC micro-benchmark suite for OpenMP and a set of applications from the SPLASH-2 benchmarks suite ported to OpenMP. Comparisons are made to OpenMP aware compiler systems from SGI and Intel.

The performance of code generated with the presented compilation

system is shown to be very close to or exceeding that of commercial compilers for a wide range of benchmark applications.

## 1 Introduction

Parallel computing with a shared-address-space, SAS, programming model was for a long time hampered by the fact that there was no standard way of writing programs that were portable across different shared-memory machines. This made independent software vendors reluctant to develop parallel shared address space applications. The scientific computing community developed the Parallel Virtual Machine, PVM, and, with support from a machine vendor consortium, the Message Passing Interface, MPI, in order to program parallel machines with a message passing programming model [10, 15].

It has long been argued that a shared address space programming model is to prefer from a programming productivity standpoint and that it is easier to come to a working parallel code from a serial code base than if a message-passing programming model is used. But then again, the lack of a standard high-level programming model has constituted a hindrance for wide-spread use of this model.

Since the introduction of OpenMP in 1997, for Fortran, and in 1998 for C/C++, this has changed [12, 13]. OpenMP is an informal standard maintained by a non-profit board constituted by industrial and academic partners. OpenMP allows software developers to maintain portable serial and parallel versions of their code with the same code-base. It leaves a lot of the low-level technical details in maintaining parallelism to the compiler so that the programmer can concentrate on the actual parallel algorithm. It is also a small standard making it possible to master it in its entirety.

OpenMP is constantly evolving and the OpenMP architecture review board, ARB, is open for new suggestions to add to the programming model. However, there are two important requirements that need to be fulfilled: (i) the proposal should be demonstrably effective, i.e., it should solve some problem that the standard currently cannot handle, and (ii) it should be possible to implement effectively. Both of these requirements basically require that the proposer has implemented the

feature in an experimental infrastructure.

In this paper we report on a free OpenMP compiler, called OdinMP, and a run-time library infrastructure for C/C++ which has been made available to the research community for exactly this purpose. The system consists of an OpenMP source-to-source translator, a compiler driver and a run-time library. We show some of the transformations done in the translator and provides a performance analysis of the compilation system for a micro-benchmark and some scientific/engineering codes in comparison with two commercial OpenMP compilers. We find the performance of OdinMP to be on par with both the Intel C/C++ compiler version 8.0 and the SGI MIPSpro 7.3 OpenMP compilers.

While almost all server and compiler vendors provides OpenMP compilers, there are few compilers to which you can obtain the source code and implement your own modifications. To the best of the authors knowledge there are only one such compiler, the Omni compiler [14]. The compiler presented in this paper can in contrast to the Omni compiler handle more of the C99 and ANSI C++ specifications [4, 5]. The previous OdinMP/CCp compiler was not open source, was very limited and only the Java class files were provided [2]. OdinMP is written entirely in C++, is a full rewrite, is more efficient and is provided with full source.

The rest of this paper is organized as follows. Section 2 gives a brief introduction to OpenMP while the OdinMP compiler and associated run-time libraries are introduced in section 3. The performance of code generated with the compilation system is evaluated in section 4. Finally, the paper is summarized in section 5.

## 2 OpenMP

OpenMP is an informal industry standard for programming shared memory parallel computers [11–13]. It is a thread-level fork-join programming model based on compiler directives and a few library functions. It is hence necessary to use a compiler which understands the directives to write a parallel program with OpenMP.

A range of compiler directives are provided in OpenMP. A *parallel directive* is the basic structure for starting parallel activity. *Work-*

*sharing directives* provide the functionality for distributing work among threads. The OpenMP directives are devised so that both loop-level parallelism and functional parallelism are supported. Apart from these two types of directives there are directives for implementing critical regions and barrier synchronization. It is, furthermore, possible to tag the parallel directives with storage attributes which makes it possible to instruct the compiler as to which variables should be treated as shared between threads or private to each thread.

Although so much functionality is provided, the number of directives is quite small which makes it rather straightforward to learn and use OpenMP.

The OpenMP specification also describes intrinsic run-time functions which, for example, can be used to deduce the number of currently running threads. There are also functions implementing lock primitives and functions for obtaining the identity of threads.

OpenMP directives are in C inserted as compiler pragma directives. All examples in this paper are written in C since the compilation system described targets C and C++ [4, 5]. Equivalent structures for Fortran programs exist, however.

We will now continue with a description of the OdinMP compilation system.

### **3 The compilation system**

There are two primary services that the compilation system must provide. The first is the actual processing of OpenMP directives and the second is the intrinsic run-time library functions specified by the OpenMP specification.

We have developed a source-to-source compiler for C and C++ called OdinMP, which parses and handles the directives. OdinMP calls on other tools such as compilers, linkers and pre-processors so that it looks and feels to the programmer as an ordinary C compiler.

An OpenMP program generally needs some support for creating and synchronizing threads. While an OpenMP-aware compiler could emit all supporting functions into the generated code, it is often more convenient to use a run-time library. We have chosen the latter.

The run-time library holds the intrinsic OpenMP run-time functions as well as basic support for thread creation and synchronization. The OdinMP compiler supports several different run-time libraries. One which is developed at UPC in Barcelona and another one, called Balder, which is developed by the authors. The Balder library also has full support for nested parallelism and support for running OpenMP applications on top of clusters. We are, however, not going to discuss those features. Some preliminary cluster work has already been published [8]. The run-time libraries were developed within the context of a EU financed research project called Intone and are sometimes called the *Intone run-time libraries* [1,6].

In short, the libraries provides:

- implementations for all run-time functions as stipulated by the OpenMP specification.
- support for spawning of threads. This is described in more detail in section 3.2.
- support for work-sharing directives. All work-sharing constructs in OpenMP can be mapped to parallel for-loops and so the run-time provides support for such for-loops as described in section 3.4.
- synchronization of threads. The basic synchronization methods provided in OpenMP are barrier synchronization, locks, and mutual exclusion. The programmer do barrier synchronization by using a directive while mutual exclusion can be either achieved by using lock operations directly or by using OpenMP constructs for critical regions or atomic operations.

We will now continue with an overview of the entire compilation system.

### 3.1 Compilation system overview

Figure 1 shows the general flow of information in the compilation system. The compiler implements the two lightly shaded boxes: *The OpenMP*

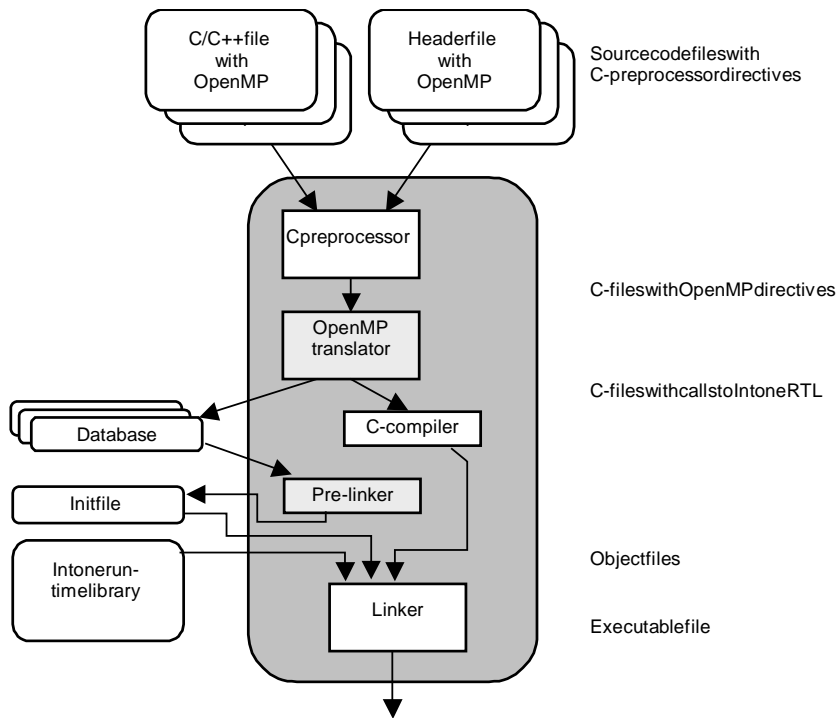


Figure 1: General overview of the work-flow in the OpenMP C/C++ compiler.

*translator*, *Pre-linker* and the bigger darker shaded box which consists of a wrapper calling the other program components as needed. The wrapper implements an interface similar to a traditional Unix type of C compiler and works as follows.

First, all C/C++-files are run through the ordinary C/C++ pre-processor. This has the effect that all include-files are included in the source files and that OdinMP does not have to parse pre-processor directives. Then OdinMP is run on each source file separately to parse the OpenMP directives and generate parallel code using the run-time library as a target. In addition, it creates a database file with information about initialization routines and possible named locks for critical regions. The pre-linker takes the database files as input and generates a C-file which contains an initialization routine for the program. This routine initializes global locks and calls the initialization routines that might exist for

different source code files. These routines are emitted by the compiler to handle initialization of variables used by the transformed OpenMP constructs.

After transforming the OpenMP source code, the wrapper will compile the generated source code with a compiler to generate object code. The actual compiler, called the *back-end compiler*, used for this step is controlled by the programmer using environment variables. In fact, all phases of the compilation as orchestrated by the wrapper can be controlled by environment variables. This makes it possible to customize the OdinMP compiler to the back-end compilation system.

The wrapper also implements an interface which resembles that of standard Unix linkers. This interface performs the pre-link step as previously described and will also compile the initialization routine previously described and run a linker to generate a complete executable application.

The OdinMP compiler and the run-time library developed by UPC are available in source code and can freely be used for research [6, 9]. The Balder run-time library has not been made public at the time of this paper being written. It is, however, the intention of the authors to make the source code of the library public.

Being a source-to-source compiler, OdinMP is very easy to port to new architectures as it does not produce object code directly but relies on the back-end compiler to generate all object code. OdinMP is written in C++ and can be ported to virtually any platform which provides a C++ compiler.

We will now continue with looking at some OpenMP constructs and how OdinMP will generate code for these constructs.

## 3.2 The parallel construct

The examples have been made somewhat simplified for brevity and so the actual output from the compiler might differ from these examples. We will start with the parallel construct, which is the basic form of achieving parallelism in OpenMP programs.

The parallel construct describes a region of the OpenMP application in which several threads may be executing in parallel. The OdinMP compiler will turn each such parallel region into a function as in the

following example.

```
#pragma omp parallel
{
  int my_id = omp_get_thread_num();
  /* returns the identity of the
     current thread. */
  foo(my_id);
  /* Some function which may be executed
     by multiple threads in parallel. */
}
```

The parallel region described by the parallel construct is translated into a function which looks as follows. The `omp_get_thread_num()` is an intrinsic function in the OpenMP specification.

```
static void in__tone_c_pf0()
{
  int my_id = omp_get_thread_num();
  foo(my_id);
}
```

We refer to these parallel regions turned into functions as *parallel functions*. The original parallel section is replaced with the following code:

```
in__tone_spawnparallel(in__tone_c_pf0, 1,
                      in__tone_cpus_current());
```

The `in__tone_spawnparallel()` function is a function in the run-time library and is used to spawn threads. It does not return until all the spawned threads have ended. For efficiency the run-time library does not actually create or destroy threads when `in__tone_spawnparallel()` is executed. Instead, a system is implemented where a pool of previously created threads is used, if possible, so as to avoid the overhead of thread creation. The function `in__tone_spawnparallel()` takes at least three arguments. The first argument is the pointer to a parallel function. The second argument is

an integer holding the number of arguments starting with the third argument. Hence, the constant 1 in this case. The third argument is an integer holding the number of threads to spawn. The function can furthermore pass multiple arguments to the function corresponding to the parallel region. This functionality is used to implement certain types of shared variables as described below. In this example, however, no arguments are passed to the parallel function.

The call to function `in__tone_cpus_current()` returns the number of threads to spawn. This value is based on the environment variables and run-time options that control the run-time library.

### **3.3 Storage attributes**

Variables that are accessible in a parallel region are by default shared among all threads that execute in parallel. There are exceptions for loop-index variables used in loops controlled by a work-sharing construct and automatically allocated variables, e.g. variables declared within dynamic context of a parallel region. These are private to each thread. The default behavior of variable allocation can be changed with storage attributes to the parallel construct.

As mentioned earlier each parallel region is put in its own function. Private variables are then allocated on the stack of each thread, i.e., within the function that represent a parallel section. Globally declared shared variables can remain global as they are globally accessible from all functions. Automatically allocated shared variables cannot be reached directly. Instead, a pointer to the shared variable is passed as an argument to the parallel function and all accesses to this variable are replaced with pointer de-references.

We will now continue looking at how OpenMP work-sharing constructs are handled by OdinMP.

### 3.4 Work-sharing constructs

The following example illustrates the use of the for-loop work sharing construct. It must be dynamically enclosed in a parallel region.

```
1 #pragma omp for
2 for (i = 0; i < 99; i += 3) {
3     /* parallel loop */
4 }
```

In this example 99 iterations will be split among the threads. The run-time library exposes an interface to support-functions for parallel loops. Three functions are used to implement all work-sharing constructs in OpenMP.

The function `in__tone_begin_for()` is called before the start of a new parallel for-loop. OpenMP allows the programmer to more closely control how parallel loops are executed through the directives. The programmer can for example control the smallest number of iterations handed out to each thread. It is also possible to control how iterations are scheduled among threads. The compiler extracts all this information from the directives and passes it on as arguments to `in__tone_begin_for()`.

The function `in__tone_next_iters()` is used to fetch new iterations to execute. It is called by threads as they need more iterations to execute and returns a range of iterations which is then performed by the calling thread.

Finally, `in__tone_end_for()` marks the end of a work sharing construct. Using these three functions it is possible to implement all OpenMP work-sharing constructs as they are strictly speaking special cases of parallel for-loops. The exact mapping of the various constructs can be found in the OdinMP user's manual [7].

We will now continue looking at how synchronization is handled.

### 3.5 Synchronization

As mentioned earlier, OpenMP supports atomic operations, barriers and locks in various forms as synchronization primitives. The run-time library has direct support for all these primitives and the code generated

by the compiler is thus straight forward. Each construct is directly replaced with calls to the run-time library.

The limitations of the compilation system is discussed in the next section.

### **3.6 Limitations**

At the time of this writing, the OdinMP compiler version 0.281.1 partially supports ANSI C++ and supports the C99 specification with one exception [4, 5]. OdinMP does not fully support initialization of array and struct variables using designators, which is a new feature of the C99 standard. As of now designators within initializations of variables are not fully supported. So called *threadprivate* variables are not supported as well as the semantics of volatile variables. OpenMP version 1.0 is otherwise supported and OpenMP version 2.0 is partially supported [12, 13].

This concludes the discussion of the compilation system and we will now proceed with a performance evaluation of the system. Further information on the compilation system can be found in OdinMP's documentation [7]. The development of OdinMP is still on-going and it is likely that the limitations above are removed in later versions.

## **4 Experiments**

Several experiments have been performed so as to judge the performance and efficiency of the code generated by the compilation system and we will now continue describing these experiments.

### **4.1 Experimental setup**

We will start the discussion with the benchmarks.

#### **4.1.1 Benchmarks**

The EPCC micro-benchmarks have been used to measure the basic overhead latencies in the run-time libraries and generated code [3]. These micro-benchmarks are widely adopted by both the industry and academia.

The performance of applications is evaluated using applications from the SPLASH-2 benchmark suite [16]. The SPLASH-2 applications were originally written using ANL macros but have been ported to OpenMP. Six applications were chosen from the suite based on their behavior. The applications were Barnes, Cholesky, FFT, two versions of the LU kernel, and one version of Ocean. The data sets were chosen for each application so that the execution times of the serial versions of the applications ranged from fairly short up to several minutes. We will now briefly describe each application and the chosen data sets.

**Barnes:** The Barnes application implements a N-body simulation using the Barnes-Hut method. The application simulates how a number of bodies, i.e., particles, interacts with each other. The data set for Barnes was chosen to be 262144 bodies which corresponds to a serial execution time of 459 seconds on a SGI Origin 3800. Barnes is fairly regular and we expect it have good parallel performance.

**Cholesky:** This application performs a Cholesky factorization on a sparse matrix. The usage of a sparse matrix causes the communication to computation ratio to be fairly high and we thus expect Cholesky to have much worse parallel performance than Barnes. The data set used was the tk29\_O data set which has a serial execution time of 1.5 seconds on a SGI Origin 3800. The execution time is very short and this means that any underlying overheads in the OpenMP constructs or run-times will be more pronounced.

**FFT:** The FFT application performs a one dimensional complex fast Fourier transform. Such a transform is highly parallelizable and we expect good parallel performance. The data set was chosen to be 16777216 complex values which yields a serial execution time on a SGI Origin 3800 of 40 seconds.

**LU kernel:** The LU kernel performs a triangulation of a dense matrix. The kernel comes in two versions. One which uses contiguous partitions and one which uses non contiguous partitions. We have chosen to use both versions. The data set chosen was a 5000x5000 matrix which yields a serial execution time on a SGI Origin 3800 of 473 and 541 seconds for the contiguous partitions and the non contiguous partitions versions respectively.

**Ocean:** The ocean application simulates ocean water movements. We have here chosen to use the contiguous partitions version with an ocean size of 2050x2050. The serial execution time is 293 seconds on an SGI Origin 3800.

#### 4.1.2 Experimental platforms

Two different experimental platforms have been used. The first is a dual Pentium III workstation, with 1 GHz processors, running Linux version 2.4.25. This machine was used to do simple comparisons to Intel's C/C++ compiler version 8.0 using the EPCC micro-benchmarks. The back-end compiler used by the OdinMP compilation system was gcc 3.3.3 and the run-time library developed by UPC, hereafter called the *UPC run-time*, was used.

The other experimental platform used is a 128 CPU SGI Origin 3800 running IRIX 6.5. This machine was used to evaluate the performance of compiled applications. SGI MIPSpro 7.3 was used for comparison. Here the MIPSpro compiler was also used as back-end compiler together with the UPC run-time.

The UPC run-time released in July 2002 was used on both platforms and the version of OdinMP used was 0.281.1.

All benchmarks have been run at least ten times. All values presented are average values formed by taking data from all runs. All executables were compiled at optimization level -O.

## 4.2 Results and discussion

We will start with a discussion of the results from the runs of the EPCC micro-benchmarks. These benchmarks consist of two applications which in turn measure the overhead of several different OpenMP constructs. We have for brevity chosen to present the overheads of the most commonly used constructs and will start with the results from the runs on the Pentium III machine using 2 CPUs, see table 1. Values for code generated with the Intel compiler and OdinMP with the UPC run-time are presented. The *Parallel construct*, *For construct*, and *Barrier construct* rows show the overhead of one single parallel, for and barrier construct respectively. The *Lock and unlock primitives* row shows the overhead of

Table 1: Overheads in microseconds of common OpenMP constructs as measured by the EPCC micro-benchmarks on a dual Pentium III workstation.

<b>Construct</b>	<b>Intel compiler</b>	<b>OdinMP with the UPC run-time</b>
Parallel construct	1.43 +/- 0.11	1.43 +/- 0.53
For construct	0.79 +/- 0.17	2.48 +/- 0.24
Barrier construct	0.48 +/- 0.19	0.37 +/- 0.17
Lock and unlock primitives	0.48 +/- 0.33	0.34 +/- 0.12

using first locking and then unlocking a single lock once. All overheads are presented with their 95% confidence interval.

OdinMP paired with the UPC run-time, yields roughly the same overheads, if not better, as the Intel compiler except for the for construct. This is not very surprising as the Intel compiler generates object code directly and can do better optimizations than OdinMP which emits source code. OdinMP is, for example, limited to using generic run-time functions while the Intel compiler can specialize the for code. We have disassembled the code generated by the Intel compiler and it does seem to specialize the code.

The EPCC benchmarks were also run on the SGI Origin machine and the results are the similar to the results gathered on the Pentium III system. Code generated with OdinMP and run with the UPC run-time has approximately the same overheads as code generated with the MIPSpro compiler. The overheads are larger for the for construct where the MIPSpro compiler can do better code than OdinMP and for the parallel construct in that the MIPSpro compilation system is more efficient when spawning threads than the UPC run-time. The overhead for code generated with the MIPSpro compiler grows less than linear while the overhead with the UPC run-time grows linearly with the number of threads. However, as we will see later on in this section, this has little impact on the performance of larger applications as most applications use few and fairly large parallel regions and parallel for-loops with relatively many iterations.

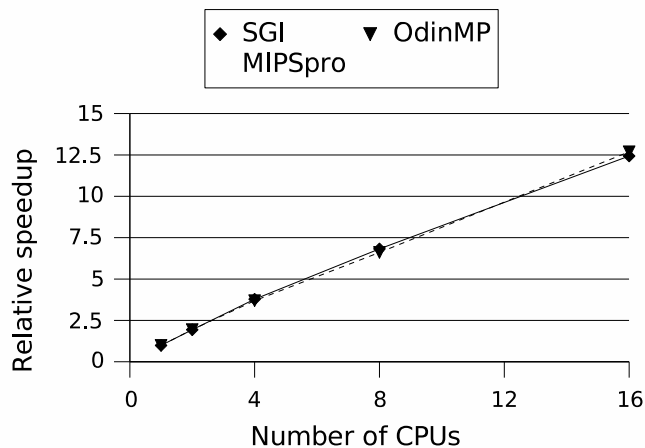


Figure 2: Relative speedups of the Barnes application.

We will now have a closer look at the performance of the SPLASH-2 applications on the SGI Origin machine when compiled with the two compilation systems. Generally, the two compilation systems yield approximately the same performance. The differences are very small and can be explained by the differences in overheads as previously described. We will now discuss each application.

The performance of Barnes for both compilation systems can be seen in figure 2. The figure shows the relative speedup which in this paper is the execution time for a purely serial version of the application divided by the execution time of a parallel version of the same application. This measure thus takes into account all extra code generated to handle parallelism. The serial versions of the applications were generated by compiling the applications with the back-end compiler forcing it to not honor OpenMP directives. The result is an executable which is without any OpenMP related code and will not execute in parallel.

The SPLASH-2 benchmarks outputs two measurements of the execution time. One of these measurements cover all parts of the benchmarks including the initialization phase which can include file IO operations while the other one measure the time spent in the parallel portion of the benchmarks. We are primarily interested in the parallel performance of the applications and so we have chosen to only use the latter

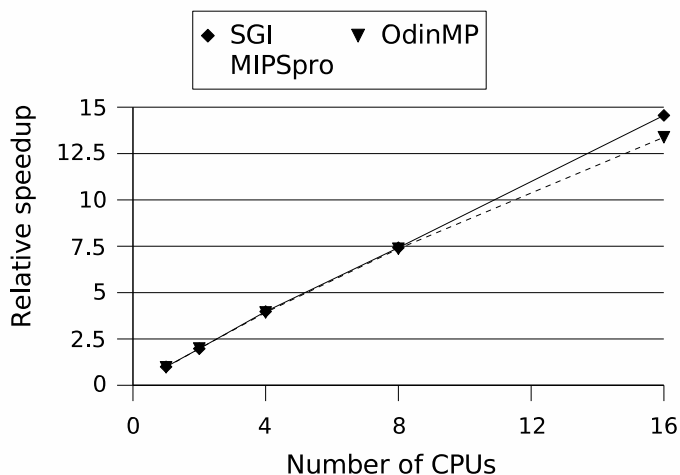


Figure 3: Relative speedups of the FFT application.

measurement when forming the relative speedup. We have for all applications used the same number of threads as CPUs making it possible to run one thread on each CPU.

Both compilation systems yield more or less the same performance for Barnes. The differences in performance are slightly larger for the FFT and Cholesky benchmarks, see figure 3 and figure 4. The differences get more exaggerated with increasing number of CPUs and the reason for this is that the parallel execution time is fairly short for these benchmarks causing the differences in overheads outlined above to have more impact.

The two variants of the LU kernel behave more like Barnes in that the performance differences are small, see figure 5 and figure 6. The graphs show no surprises.

The contiguous partitions version of Ocean performs, however, much more interestingly, see figure 7. Ocean has super-linear speedup and this is because the working data set is so small that it will fit into the combined second level caches of eight CPUs effectively reducing the number of cache misses. The difference in speedups when using 16 CPUs is fairly large but the absolute difference in execution time is fairly small and can be explained with differences in the overheads of synchronization primitives.

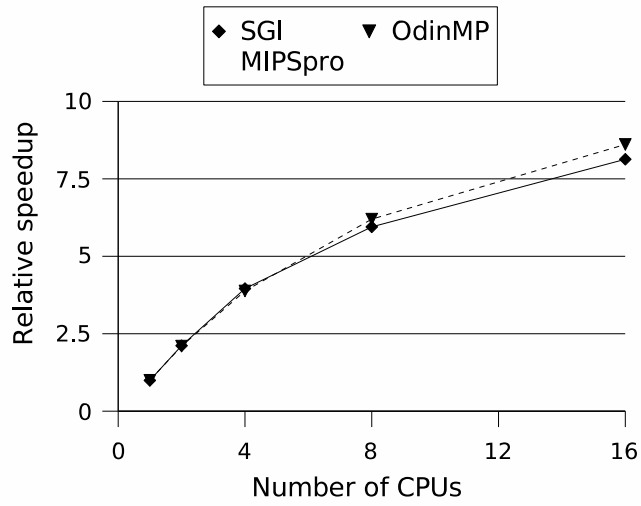


Figure 4: Relative speedups of the Cholesky application.

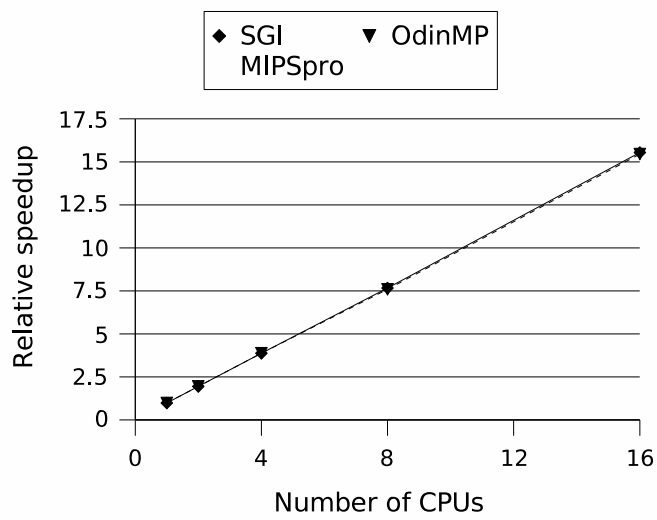


Figure 5: Relative speedup of the contiguous partitions version of the LU kernel.

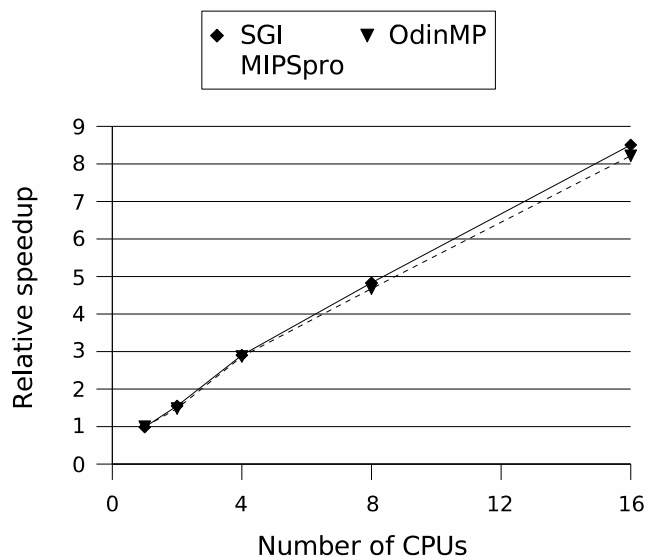


Figure 6: Relative speedup of the non contiguous partitions version of the LU kernel.

Experiments with more CPUs, i.e., up to 64 CPUs, show that the performance trends outlined above continue as the number of CPUs increases. The differences in underlying overheads of OpenMP constructs in the two compilation systems become more accentuated as the execution times decreases.

The relative speedup on one CPU is very close to 1 for all of the benchmarks and both compilation systems. The difference is at most 3% for code generated with OdinMP and less than 1% for code generated with MIPSpro, see table 2. This indicates that the fact that OdinMP is a source-to-source compiler has very little impact on performance.

## 5 Conclusions and summary

In this paper we have presented a free C and C++ compiler for OpenMP, called OdinMP, which paired with a suitable run-time library yields application performance which is very close to or exceeding the performance obtained by using commercial OpenMP compilers. This even

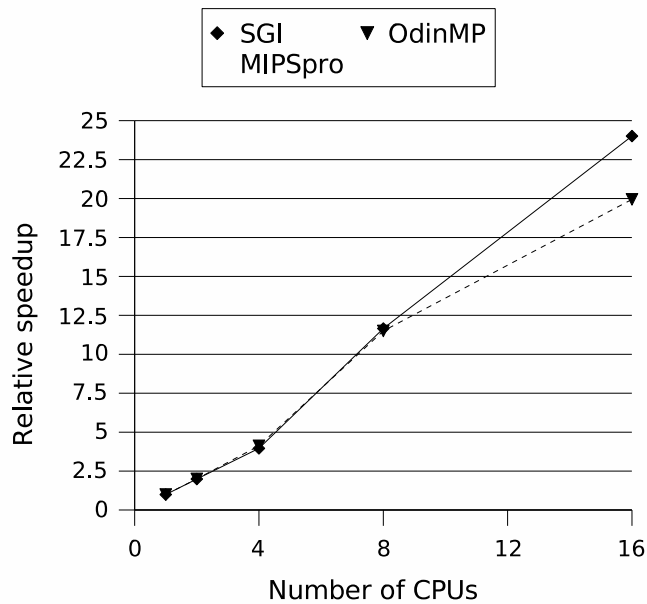


Figure 7: Relative speedup of the contiguous partitions version of the ocean application.

Table 2: Relative speedups for the benchmarks when running on one CPU.

Benchmark	MIPSPro compiler	OdinMP with the UPC run-time
Barnes	0.99	0.99
FFT	1.00	0.97
Cholesky	1.00	1.00
LU contiguous	0.99	0.99
LU non contiguous	0.98	1.00
Ocean contiguous	0.99	0.99

though OdinMP is a source-to-source code compiler and can not perform the more elaborate optimizations made possible to compilers that emits object code.

Several benchmarks have been run and the performance difference between code compiled with OdinMP and code compiled with a com-

mercial compiler is very small. One can therefore conclude that OdinMP is a viable alternative to commercial compilers and a strong foundation for research on OpenMP and shared memory programming models in general.

## Acknowledgments

The work in this paper has been partly financed by the European Commission in the Intone project under contract number IST-1999-20252. The authors gratefully acknowledge Xavier Martorell et. al. at UPC, Spain for their efforts working on their run-time library. The OpenMP translator was in part implemented also by Håkan Zeffer, Samer Al-Kassimi and Örjan Friberg. Their contribution is hereby greatly acknowledged. Anna Thelin wrote the initial OpenMP versions of the SPLASH-2 benchmarks. Håkan Zeffer also did a tremendous job helping out with running some of the benchmarks on the SGI Origin machine.

## References

- [1] E. Ayguad et. al., OpenMP Performance Analysis Approach in the INTONE Project, in *Proceedings of 3rd European Workshop on OpenMP*, Barcelona, Spain, September 2001
- [2] C. Brunschen and M. Brorsson, *OdinMP/CCp - a portable implementation of OpenMP for C*, *Concurrency: Practice and Experience*, 2000; 12:1193-1203.
- [3] J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, in *Proceedings of the First European Workshop on OpenMP*, Sept. 1999, pp. 99-105. <http://www.it.lth.se/ewomp99>
- [4] Information Technology Industry Council, *ISO/IEC 9899:1999 Programming languages - C second edition*, American National Standards Institute, 1999

- [5] Information Technology Industry Council, *ISO/IEC 14882:1998 Programming languages - C++*, American National Standards Institute, 1998
- [6] Intone project, *Intone project homepage*, accessed July 9th 2004, <http://www.cepba.upc.es/intone/>
- [7] KTH, *OdinMP documentation*, accessed July 9th 2004, <http://renaldo.imit.kth.se/odinmp/docs/>
- [8] S. Karlsson, S-W. Lee, and M. Brorsson. A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory, in *Proceedings of 2002 International Conference on High Performance Computing*, Bangalore, India, December 2002, pp 195-208.
- [9] KTH, *OdinMP homepage*, accessed July 9th 2004, <http://www.odinmp.com>
- [10] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1, June 12, 1995.
- [11] OpenMP consortium, *OpenMP: A Proposed Standard API for Shared Memory Programming*, White paper, <http://www.openmp.org>.
- [12] OpenMP consortium, *OpenMP C and C++ Application Program Interface*, Version 1.0, October 1998
- [13] OpenMP consortium, *OpenMP C and C++ Application Program Interface*, Version 2.0, March 2002
- [14] D. M. Sato, Design of OpenMP Compiler for an SMP Cluster, in *Proceedings of First European Workshop on OpenMP*, Sept. 1999. <http://www.it.lth.se/ewomp99>
- [15] V. S. Sunderam, *PVM: A framework for parallel distributed computing*, *Concurrency: Practice and Experience*, vol. 2(4), Dec. 1990, pp. 315-339

- [16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations, in *Proceedings of the 22nd International Symposium on Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 24-36.

S. Karlsson, *Balder – An OpenMP run-time library for clusters of SMPs*, Technical report, TRITA-IMIT/LECS R 04:01, ISSN 1651-4661, ISRN KTH/IMIT/LECS/R-04/01--SE, Department of Microelectronics and Information Technology, Royal Institute of Technology, KTH, August 2004



# **Balder – An OpenMP run-time library for clusters of SMPs**

**Sven Karlsson**

**Department of Microelectronics and Information Technology**

**Royal Institute of Technology, KTH, Sweden**

**Sven.Karlsson@imit.kth.se**

## **Abstract**

In this paper an run-time library, called Balder, for OpenMP 2.0 is presented. OpenMP 2.0 is an industry standard for programming shared memory machines. The run-time library presented can be used on SMPs and clusters of SMPs and it will provide a shared address space on a cluster.

The functionality and design of the library is discussed as well as some features that are being worked on.

The performance of the library is evaluated and is shown to be competitive when compared to a commercial compiler from Intel.

## **1 Introduction**

OpenMP has during the last few years gained considerable acceptance as the shared memory programming model of choice. OpenMP is an industry standard and utilizes a fork-join programming model based on compiler directives [14, 15].

The directives are used by the programmer to instruct an OpenMP aware compiler to transform the program into a parallel program. In addition to the directives, OpenMP also specifies a number of run-time library functions.

To use OpenMP, an OpenMP aware compilation system is thus needed. The compilation system generally consists of a compiler and

an OpenMP run-time library. The library is not only used to handle the run-time library functions as defined by the OpenMP specification but also to aid the compiler with a number of functions that efficiently spawns threads, synchronize threads, and help share work between threads.

In this paper, an open source OpenMP run-time library, called Balder, is presented which is capable of fully handling OpenMP 2.0 including nested parallelism [15]. The library is capable of handling single SMPs efficiently as well as clusters of SMPs making it possible to do research on extensions to the OpenMP specification both in the areas of SMP centric and cluster centric extensions. A compiler, called OdinMP, targeting the library is already readily available [7, 10]. A more detailed description of the transformations OdinMP transforms is available at OdinMP's homepage [10]. I will not discuss the API of Balder in detail in this paper and instead I refer the interested reader to the aforementioned website.

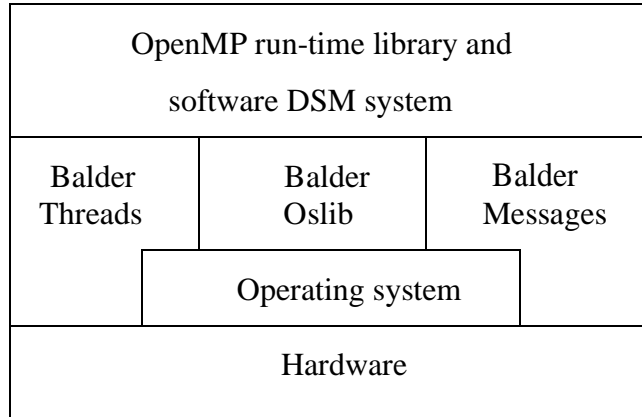
The source code of the library is not currently openly available although it will be released soon under a very liberal license. The library is highly portable and there are currently ports available to several processor architectures such as x86 and ARM and to several different operating systems such as several different Unix variants and Windows versions. Balder is currently at version 0.105.1.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the library while section 3 provides information of the sub-libraries that Balder builds on. Section 4 describes the OpenMP run-time itself and presents some details on the implementation of important primitives while section 5 discussed work-in-progress and future features for the Balder run-time. Experimental results are presented in section 6. The paper is summarized in section 7.

## **2 Overview of the run-time library**

The run-time library is implemented in ANSI C [5] and provides the following functionality:

- Full implementation of all OpenMP 2.0 intrinsic functions, i.e., the run-time library functions described by the specification.



**Figure 1: Overview of the design of the Balder library.**

- Efficient handling of threads such as thread creation, and thread synchronization.
- Parallel for-loop primitives to aid the compiler when transforming work sharing constructs.
- Support for OpenMP's threadprivate variables.
- Support for OpenMP's copyprivate clause.
- A built-in software DSM system to achieve a shared address space on a cluster.
- Memory management for the shared address space.
- Support for shared stack storage.

The library builds on previous experience [7, 8]. It is designed to be highly portable and to achieve portability it is designed as a layered system, see figure 1. Most of the functionality outlined in a previous paper is implemented [9].

The library utilizes three sub libraries: *Balder Threads*, *Balder Oslib*, and *Balder Messages*. These libraries provides thread services, operating system services, and cluster messaging services respectively.

The software DSM system and OpenMP run-time library is then implemented on top of the sub-libraries making it possible to implement the run-time in ordinary ANSI C without any dependencies on processor architecture or operating system features. In essence, this means that porting Balder is a matter of porting the well-defined prim-

itives in the three sub-libraries and I will now continue with a description of the run-time sub-libraries.

### **3 The Balder sub-libraries**

The three sub-libraries are instrumental in achieving a high degree of functionality and portability. A layered design approach has, whenever possible, been used even in the sub-libraries to aid porting efforts.

#### **3.1 Balder Messages**

Balder Messages is a packet-based, network technology independent messaging library with support for prioritized communication [9, 11]. The transmission and reception primitives use a data format based on linked lists for the packets. This facilitates scatter-gather I/O. It is possible to allocate memory for packets in network hardware buffers so as to achieve zero-copy communication.

There exists primitives for both synchronous and asynchronous communication as well as active message communication [3].

The library is divided into several parts. One part handles the construction of packets as linked lists. Another one handles flow control and queuing of packets and the last one is the network back-end. There exist network back-ends for UDP and MPI [13]. Back-ends with partial support for Myrinet, LAPI, and Sys V's shared memory primitives exist [1].

#### **3.2 Balder Oslib**

Balder relies heavily on the virtual memory management system of the operating system. The virtual memory systems of different operating systems are, however, quite different from each other. The Balder Oslib sub-library acts as a virtualization layer and provides an operating system independent API. There is support for:

- Creation of a virtual memory region which is guaranteed to be at the same address range on all cluster nodes.
- Setting of access permissions on individual virtual memory pages.
- Catching of page-faults or similar exceptions.
- Timers and timing management.

In addition, Balder Oslib implements a registry under which arbitrary data structures can be filed using strings as keys. This registry is used to store global system options and data structures and is instrumental in making it possible to modularize Balder properly without being hampered by artificial cross-module dependencies.

The Oslib is the sub-library which is the least layered. This is largely due to the virtual memory systems differing so much between operating systems. The Windows port, for instance, requires a completely different implementation than Unixes and cannot share any code with other ports.

### **3.3 Balder Threads**

Balder Threads provides an efficient processor architecture independent multithreading API. Balder Threads has primitives for:

- Thread creation and destruction.
- Thread synchronization using monitors, monitor signals, and barriers.
- Work queues.
- Stack frame creation so that arbitrary function can be called.

Balder Threads uses a system of assembly macros that describes the processor architecture. When porting to a new processor architecture is most cases only needed to change the assembly macros to complete a port.

The assembly macros describe how a function stack frame is organized, how functions are called, i.e., which parameters are passed on the stack and which are passed in registers, and provide implementations for important low-level primitives. Balder Thread uses pthreads as underlying thread library but implements efficient synchronization primitives using the mentioned low-level primitives [4]. The primitives used by Balder Threads are test-and-set, fetch-and-add, and a memory fence operation if required by the memory consistency model. In the absence of a required primitive, in the form of macros, Balder Threads will first revert to an implementation which only uses test-and-set and then, if no test-and-set primitive is provided, to pthreads.

Using this scheme of portable synchronization primitives based on low-level assembly macros, Balder Threads is able to achieve a synchronization overhead close to an order of magnitude less than

pthread. The synchronization primitives are generally based on test-and-set with a time-out so as to avoid excessive busy wait.

Balder Threads only provides primitives which can be used by threads running on the same cluster node. The OpenMP run-time layer provides primitives which can be used across cluster nodes.

## **4 Software DSM system and OpenMP run-time library**

The OpenMP run-time library is built on-top of the sub-libraries and can thus be written completely architecture independent. It provides support for a shared address space on-top of a cluster via a software DSM system, and support for high-level OpenMP primitives as outlined in section 2.

The software DSM system uses the virtual memory management system to provide a shared address space, on a cluster, which acts as a shared memory. The system does not rely on any particular hardware except a decent virtual memory system and a network interconnect system.

In Balder, the software DSM system is built on-top of Balder Oslib and Balder Messages. The software DSM system uses home-based lazy release consistency, HLRC [18]. The reason for using HLRC is HLRC robustness and relative simplicity. Special features of the HLRC variant used in Balder will be discussed when discussing the support functions for OpenMP.

As mentioned in section 2, the OpenMP run-time library provides a number of primitives in different areas of functionality. I will now go through these areas and provide an overview of the functionality provided and how the implementation is done.

### **4.1 Parallel regions**

OpenMP is a fork-join programming model and the basis of parallelism are parallel regions as defined by compiler directives. The compiler will transform each parallel region into a function which is then executed in parallel by a number of threads. The compiler then inserts a call to a run-time library function, which handles creation of threads and execution of the function, into the program. The run-time library

function is the basis for parallelism and is called `in__tone_spawnparallel`.

Balder uses a pool of threads to avoid excessive and time consuming thread creation and destruction operations. Threads are never destroyed and are only created when no threads are left in the pool. All threads in the thread pool wait on a work queue as provided by Balder Threads.

Internally the implementation of `in__tone_spawnparallel` is straightforward. It merely performs book-keeping, adds work to the thread pool's work queue, and then waits for the threads to finish executing the function. After finishing, the spawned threads returns to the thread pool and can be reused in other parallel regions.

Each cluster node has its own thread pool and so message passing is used to hand out work when running on a cluster. The cluster code also make use of an limitation to simplify the message passing. Balder does not allow nested parallelism when running on a cluster. All nested parallel regions are serialized yielding one level of parallelism. This is allowed by the OpenMP specification and does not break OpenMP compliance. Balder does, however, fully support nested parallelism when running on one single SMP node.

The limitation is used to simplify the message passing and the memory coherence protocol.

## 4.2 Lock functions

The OpenMP specification describes a number of lock functions as run-time library functions. To handle these function, a set of distributed lock primitives have been implemented on top of Balder Threads and Balder Messages. The lock primitives are un-fair in the sense that if a lock is held on a cluster node, the threads on that node get precedence over other threads when setting the lock. This reduces the network activity and the latency in the case of lock contention [16].

Intra-node synchronization is performed with Balder Thread primitives while inter-node synchronization is performed with message passing.

The lock primitives revert back to the efficient primitives in the Balder Thread sub-library, if the system is running on one single node, thus avoiding the overhead of the distributed lock algorithm.

### 4.3 Barriers

The threads executing a specific parallel region forms a thread team. The threads in a thread team can synchronize with a barrier. The OpenMP run-time provides a function for barrier synchronization.

Inter-node barriers are performed in two phases. First there is an intra-node barrier as provided by Balder Threads. After all threads internal to a node are synchronized, message passing takes place which synchronizes all the nodes to each other using a centralized barrier algorithm.

The barrier operation is an operation with particularly high overhead on clusters. The overhead, is however, overlapped with memory coherence operations so as to not waste CPU resources.

As in the previous section, no message passing or memory coherence activities are performed when only one cluster node is used.

### 4.4 Worksharing primitives

The worksharing constructs in OpenMP can all be mapped onto parallel for-loops and so Balder only provides support for such loops. The primitives essentially split a range of iterations into smaller pieces which then are executed in parallel by different threads.

It is, when running on one single cluster node, straight-forward to implement such primitives and in Balder the fetch-and-add primitive as provided by the Balder Thread assembly macros is used, if available, to minimize thread synchronization.

A cluster implementation is, however, much more complicated and easily leads to excessive message passing. A trick is used to reduce the message passing. The entire iteration space is first divided statically among the cluster nodes so that each cluster node receives a piece of the iteration space proportional to the number of threads, taking part of the parallel for-loop, that are executing on the node. These smaller pieces are then divided and handed out to the threads. This way no message passing is needed to implement parallel for-loops at the expense of potentially worse load imbalance. Load balancing algorithms, to reduce any load imbalance, are planned but not implemented

I believe this trick to be OpenMP compliant as the statical assignment of iterations performed follows all rules for the scheduling of for-loops stated in the OpenMP specification.

## 4.5 Advanced data clauses

The OpenMP specification defines several data clauses which describe how different variables are handled. One such data clause is the thread-private data clause which defines a variable to be private to a thread and keep its value between parallel regions. Such variables are called *threadprivate variables*. This essentially means that the threadprivate variables cannot be stored on the stack but must be associated with the threads themselves.

The run-time provides, for this, a thread local storage space and two primitives to access the storage space. One primitive is used to allocate and initialize space and another is used to retrieve a pointer to the space.

The compiler generates code which at startup, and using the first primitive, defines the thread local storage. The storage space is, however, not allocated or initialized until it is used.

All accesses to threadprivate variables are changed by the compiler so that the pointer to the storage space is retrieved using the second primitive and all accesses are performed relative to that pointer. Allocation and initialization of the storage space will be performed before the pointer is returned if no space has been allocated for the thread.

Another data clause is the copyprivate data clause. It makes it possible to broadcast the value of a variable private to a thread to the other threads in the same thread team. The copyprivate clause has been added in the second revision of the OpenMP specification to aid programming of applications utilizing nested parallelism.

A set of primitives has been added to the run-time library to implement the copyprivate data clause. These primitives makes it possible for the broadcasting thread to send a variable via the run-time library to receiving threads and then wait for the receiving threads to properly receive the variable. For efficiency, the primitives are devised so that several variables can be sent and received after each other.

Naturally, message passing is used for inter cluster node broadcasts.

## 4.6 Handling of shared memory

The software DSM system is providing a shared address space across the cluster which can be accessed as a shared memory. To manage this address space, the run-time library provides memory allocation and de-

allocation functions similar to ANSI C's malloc and free [5]. These are used instead of malloc and free when running on a cluster. The software DSM system is inactive when running on a single node, i.e., a single SMP, and so the normal heap as provided by the operation system is used.

A few advanced code transformations are required to run OpenMP applications on clusters. The transformations involve the handling of shared global variables and shared stack variables. The OdinMP compiler performs these transformations if instructed to do so through a command line option. The default is to not perform the transformations as they are not needed when generating code for SMPs.

Shared global variables must be allocated in the shared address space when running on a cluster. The OdinMP compiler can emit code to do the allocation and it also transforms declarations of, and accesses to, shared variables so that shared variables are accessed through pointers pointing to allocated memory regions in the shared address space.

In OpenMP, variables located on a thread's stack can also be shared. This can occur if a parallel region is started. The thread that starts the parallel region, i.e., executes the `in__tone_spawnparallel` primitive, is called the master thread. Variables on the master thread's stack can be shared among the threads in the spawned thread team.

Balder handles this by implementing one single shared stack located in the shared address space. One stack is enough as, on clusters, only one level of parallelism is allowed which means that only the thread that executes the serial portions of the OpenMP application can spawn parallelism.

The OdinMP compiler can transform the application code to make use of the stack. It changes how variables are allocated so that the potentially shared variables are allocated on the shared stack and it inserts code that builds stack frames on, and removes them from, the shared stack upon entry and exit from functions respectively.

One shared variable is in the example in figure 2. The example is simplified for brevity.

The transformed code is, albeit simplified, presented in figure 3.

The code inserted by the OdinMP is very much similar to what a compiler emits for function entry and exit to build and remove stack frames. One thing differing is, however, the checks to see if the function is being called from a serial or parallel region of the code. This is

```

void f(void)
{
    int shared_variable; /* The shared variable */

    shared_variable=5; /* The shared variable is accessed. */
}

```

**Figure 2: An example with a shared variable.**

necessary as the shared stack must only be used from a serial portion of the code. A frame pointer is inserted as a local variable and used for accessing the shared variable.

In OpenMP, the flush directive is used to enforce consistency. The flush directive is a memory fence operation that controls when memory updates are conveyed and when memory operations is performed.

You must, essentially, insert a flush directive before any shared variable that could have been updated by another thread and you must insert a flush directive after an update if you want that update to be conveyed to other threads. Quite a few directives such as barrier directives have implied flush directives and this reduced the need to insert extra flush directives. The OpenMP directives are defined so that extra flush directives are only needed some very special cases such when implementing thread synchronization not using the OpenMP synchronization primitives.

The semantics of the flush directive is implemented in Balder just like in an earlier prototype [8]. A special distributed lock is used. The lock is not accessible from the OpenMP application but is handled just like any other OpenMP lock. A flush directive consists of a set of the lock followed by a release. Information on memory updates are piggybacked onto the lock. When setting the lock, the received information is used to invalidate memory contents so as to drive coherency. The received update information is merged with information on locally performed memory updates and then sent with the lock when the lock is released to a remote node.

The implied flush directives can and are in most cases optimized. The barrier directive, as an example, has an implied flush directive but

```

void f(void )
{
    struct in__tone_c_dt0
    {
        int shared_variable;
    }; /* The declaration of the stack frame which
        is used on the shared stack. */

        /* The shared stack must only be used in the serial portions of
        the code. The declaration below makes space on the thread's
        stack to be used in parallel regions. */
    struct in__tone_c_dt0 in__tone_c_dt0;

        /* The variable below is true when executing in a parallel region.
        in__tone_in_parallel() is a run-time library call that returns 1
        iff the calling thread is executing in a parallel region. */
    const int in__tone_sdsm_in_parallel=in__tone_in_parallel();

        /* The declaration below is for the frame pointer. The frame pointer
        is set to either the shared stack or the stack frame on the thread's
        stack. */
    struct in__tone_c_dt0 * const
    in__tone_sdsm_i_framep=in__tone_sdsm_in_parallel ?
    &in__tone_c_dt0 :
        /* The shared stack pointer is called in__tone_sdsm_stackptr.
        It is below subtracted to make space for a new frame. This
        piece of code will only be executed if executing in a serial
        portion of the code.*/
        (struct in__tone_c_dt0 * const ) (in__tone_sdsm_stackptr=
        in__tone_sdsm_stackptr-sizeof(struct in__tone_c_dt0 ));

        /* Below is the access to the shared_variable. It is now done through
        the frame pointer. */
    in__tone_sdsm_i_framep->in__tone_c_dt0.shared_variable=5;

        /* When leaving the function, either at the end of the function or
        with return, the shared stack pointer must be updated so as to
        remove the frame if executing in a serial portion of the code. The
        code below does that. */
    if (!in__tone_sdsm_in_parallel)
        in__tone_sdsm_stackptr=
        in__tone_sdsm_stackptr+sizeof(struct in__tone_c_dt0 );
}

```

**Figure 3: Transformed example with shared variable.**

the coherency information is piggy-backed on the barrier message passing thus removing the use of the special lock mentioned above.

## **4.7 OpenMP intrinsic functions**

All the OpenMP 2.0 intrinsic functions are implemented in Balder. The implementation is rather straight-forward and mainly involves inquiring or updating the internal state of the run-time library.

# **5 Advanced features**

Balder has a few planned features currently under implementation. Apart from prefetch and producer-push [6, 12], two more advanced features are under implementation as outlined in the next few sections.

## **5.1 A compiler supported hybrid fine-grain/coarse-grain Software DSM system**

The OdinMP compiler can gather information from the source code of OpenMP applications which can be used to further optimize the performance of said applications on Balder. This can be used to insert coherency checks into the compiler output so as to reduce the granularity of the coherency while still being able to fall back on a page-based system.

In short this means that whenever a shared variable can potentially be accessed the compiler needs to make sure that there is code inserted, prior to the access, which makes sure the shared variable is cached locally. The key point here is that only the shared variable itself, and not necessarily the virtual memory pages on which the variable is located, has to be locally cached. This greatly reduces the latency of remotely requesting shared data and also reduce the average memory access latency of shared variables thus increasing the performance, i.e., reducing the execution time, of applications running on the Software DSM system provided the overhead of the inserted coherency checks is small enough.

OdinMP is being augmented to insert the mentioned coherency checks and two primitives is being added to Balder. The coherency checks are based on a check-in/check-out scheme where a piece of

shared address space is requested and then later returned. Prototypes for the two primitives are:

```
void *in__tone_sdsm_checkout_memory(void *shared_address,  
                                     unsigned int length);  
void in__tone_sdsm_commit_memory(void *shared_address,  
                                  void *local_address,  
                                  unsigned int length);
```

The `in__tone_sdsm_checkout_memory` function is used to request an up-to-date version of a memory region. The memory region is defined by the parameters `shared_address` which is the shared address to the region and `length` which is the length in bytes of the region. The primitive returns a pointer to a copy of the memory region.

The `in__tone_sdsm_commit_memory` primitive is used to hand a previously requested copy back to the system and commit changes. It takes as parameters the shared address in `shared_address`, the pointer to the copy of the region in `local_address`, and the length of the region in `length`.

## **5.2 Sharing pattern hints to improve communication performance**

It has been shown that the performance of applications running on software DSM systems can be greatly enhanced by using a small number of explicit message passing operations [17].

An extension to OpenMP is being worked on which adds message passing operations to the C variant of OpenMP in the form of added directives. These directives do not directly correspond to message passing operations but rather reflects the communication of shared data in the application as estimated by the programmer. The programmer describes via the directives how shared data is shared between threads. The proposed directives is then used by OdinMP to insert code, into the generated code for an application, that perform message passing or to efficiently distribute data structures among nodes. The directives aid the coherency protocols in Balder so that data can be moved before data is accessed and thus reducing the time it would take to remotely request data yielding to reduced application execution time due to reduced overhead of shared memory accesses.

The most common worksharing construct in OpenMP is a parallel for loop and the proposed directives builds on that construct. This means that the programmer can use a notion of iteration variables when describing the shared data.

Planned directives have support for:

- Broadcast operations which sends the same data from the master thread to all the other threads. See example in figure 4.

```
int i;
/      * i is a shared variable. */
#pragma ompd broadcast(i)
      /* This is a directive specifying a broadcast of i to all other threads */
#pragma omp parallel for
      /* This OpenMP directive indicates the start of a parallel for loop and
      a parallel region and threads will thus be spawn. Here the shared
      variable i can be broadcast so that all threads can access a cached
      copy */
for (j=0; j<MAX; j++)
  foo(i);
```

**Figure 4: Example with the broadcast directive.**

- Scatted and gather operations which splits up and sends data from and to the master thread. This models the case where a single thread acts as a master to several slaves and hands out work and to the slaves and later collects the data. See figure 5 for an example.
- All-to-all operations where a data structure which has been updated by several threads in parallel is broadcast to all threads. See example in figure 6.

The directives are planned to be processed by OdinMP which will insert run-time library calls so as to add attributes to the parallel for loops. These attributes are then used by the run-time library when creating parallel regions and distributing parallel loops to give hints to the coherency protocol about data sharing patterns and also possibly perform message passing. The directives are a complement to producer-push techniques [12].

```

int i[MAX];
    /* i is a shared variable. */
#pragma ompd scatter(i,1,0)
    /* This is a directive suggesting i to be scattered among the threads.
       The first parameter is the array to be scattered. The second
       parameter is the number of records being used in each iteration, i.e.,
       the number 1 means here that each iteration in the for loop uses one
       record. The third parameter describes which record is being used by
       iteration 0, i.e., the number 0 means that iteration 0 will use array
       record 0. This means leads to i[0] being sent to the thread executing
       iteration 0 i [1] to the thread executing iteration 1 and so on.*/
#pragma ompd post gather(i,1,0)
    /* This directive is instructing i to be gathered from the threads after
       the for loop. The numbers are the same as for scatter. After the for
       loop is executed i[1] is sent from the thread executing iteration 1
       to the master thread, i[2] is sent from the thread executing iteration
       2 and so on. */
#pragma omp for
    /* This OpenMP directive indicates the start of a for loop inside a
       parallel region, i.e., threads have already been spawn. Here the
       shared variable i is split among the threads so that again each
       thread can access a cached copy. */
for (j=0; j<MAX; j++)
    i[j]=i[j]+foo();

```

**Figure 5: Example with the scatter and gather directives.**

## 6 Experimental results

Some experiments have been conducted so as to evaluate the performance of Balder. These experiments are not exhaustive but they do give an indication of Balder's performance. The cluster parts of Balder are not evaluated as they are being tested and are not ready to be evaluated yet.

A dual Pentium-III workstation running Linux version 2.4.25 was used as experimental platform. The processors were running at a clock rate of 1 GHz.

The EPCC micro-benchmark suite was used in the experiments [2]. The EPPC micro-benchmark suite is a set of benchmarks which mea-

```

int i[MAX], k[MAX];
    /* i and k are shared variables */
#pragma ompd post alltoall(i,1,0)
    /* The alltoall directive uses the same "parameters" as scatter and
       gather. Here the parameters describes which part of the array that
       has been updated in which iterations. Here i[0] is updated in iteration
       0, i [1] in iteration 1 and so on. */
#pragma omp for
    /* OpenMP directive which indicates the start of a for loop inside a
       parallel region, i.e., threads have already been spawn. Here the
       shared variable i is updated by all threads in parallel. */
for (j=0; j<MAX; j++)
    i[j]=foo();
    /* After the loop, parts of i are broadcast by the thread that updated the
       respective part. In this case, i[0] is broadcast by the thread executing
       iteration 0, i[1] is broadcast by the thread executing iteration 1 and so
       on. */
for (j=0; j<MAX; j++)
    k[j]=foo(i);
    /* here all threads have already received i and can act on a cached copy */

```

**Figure 6: Example with alltoall directive.**

sure the overhead of individual OpenMP constructs and thus also the run-time library. The benchmarks were compiled with OdinMP version 0.284.1 and GCC 3.3.4. The Balder library version 0.105.1 was used and was also compiled with GCC 3.3.4.

For comparison, the same set of benchmarks were compiled with the Intel C/C++ compiler version 8.0 and run on the experimental platform. The Intel compiler supports C/C++. The Balder library cannot currently be compiled with the Intel compiler. The highest available optimization level was used in both compilation systems.

The overheads in microseconds for the most common OpenMP constructs are summarized in table 1. The overheads are presented with their 95% confidence interval.

The overheads in the first three rows are for one single parallel region, parallel for-loop, and barrier respectively. The lock and unlock primitives row is the overhead of setting and then releasing a single lock once.

**Table 1: Overheads in microseconds of common OpenMP constructs.**

<b>OpenMP Construct</b>	<b>Intel compiler</b>	<b>Balder with OdinMP</b>
Parallel construct	1.43 +/- 0.11	2.91 +/- 0.15
For construct	0.79 +/- 0.17	2.93 +/- 0.30
Barrier construct	0.48 +/- 0.19	0.49 +/- 0.12
Lock and unlock primitives	0.48 +/- 0.33	0.47 +/- 0.12

The overheads of the primitives in the Balder library are very competitive for barrier and lock synchronization. The overhead of parallel for loops are much higher for the Balder run-time. The reason for this is the fact that OdinMP is a source-to-source compiler and cannot do aggressive optimizations of the parallel for-loops as the Intel compiler can as it is compiling to object code.

The overhead of parallel regions are also higher for the Balder run-time. One contributing factor is the transformation of parallel regions as outlined in section 4.1. The functions created out of the parallel regions by the OdinMP compiler can take arguments. These arguments are managed by the `in__tone_spawnparallel` run-time primitive. The arguments are copied once more than actually needed on an SMP during the handling of the arguments and the creation of stack frames performed in the run-time library. The extra copying performed is, however, necessary when running on clusters. It is, though, being investigated to see if it can be removed in future versions of Balder.

The differences in overheads between code generated by the Intel compiler and the OdinMP/Balder combination is very small. The overheads are in the same range as found in a previous study [7]. It was found in the same study that differences in overheads these small is unlikely to influence end performance of OpenMP applications. The overheads measured thus suggests that Balder paired with OdinMP should be very competitive to commercial compilation systems.

## 7 Summary

This paper provides an overview of the current status of the Balder, OpenMP run-time library. The organization of the library is presented and the functionality and design of the different modules are described. Some selected parts of the implementation are discussed. Planned future and work in progress for the Balder library and the OdinMP compiler is presented. Some experimental data is presented which shows the Balder library to be competitive when compared to a commercial OpenMP compiler.

## Acknowledgements

The research in this thesis has been in part financially supported by the Swedish Research Council for Engineering Sciences under contract number, TFR 1999-376, and by the Swedish National Board for Industrial and Technical Development (NUTEK) under project number P855. It was also partially financed by the European Commission under contract number IST-1999-20252.

Nguyen-Thai Nguyen-Phan has implemented parts of the work-sharing, memory allocation and distributed lock primitives. He has been instrumental in testing the library. The API used by Balder is a super-set of the API for OpenMP run-time libraries developed during the Intone project by the Intone project partners. Eduard Ayguadé, Marc González, and Xavier Martorell at UPC in Spain were particularly instrumental in that effort.

## References

- [1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. *Myrinet: A gigabit-per-second local area network*. IEEE Micro, 15(1):29--36, Feb. 1995
- [2] J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, in *Proceedings of the First European Workshop on OpenMP*, Sept. 1999, pp. 99-105.
- [3] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauer, Active Messages: a Mechanism for Integrated Communication and Computation, In *Proceedings of the 19th International Sym-*

*posium on Computer Architecture*. Gold Coast, Qld., Australia. May 1992. pp. 256-66

- [4] IEEE, *IEEE std 1003.1-1996 POSIX part 1: System Application Programming Interface*, 1996
- [5] Information Technology Industry Council, *ISO/IEC 9899:1999 Programming languages - C second edition*, American National Standards Institute, 1999
- [6] M. Karlsson and P. Stenström, Evaluation of Dynamic Prefetching in Multiple-Writer Distributed Virtual Shared Memory Systems, In *Journal of Parallel and Distributed Computing*, vol. 43, no. 7, July 1997, pp.79-93
- [7] S. Karlsson, and M. Brorsson, A Free OpenMP Compiler and Run-Time Library Infrastructure for Research on Shared Memory Parallel Computing, *to appear in proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004), November 2004*
- [8] S. Karlsson, S-W. Lee, and M. Brorsson, A Fully Compliant OpenMP Implementation on Software Distributed Shared Memory, In *Proceedings of 9th International Conference on High Performance Computing (HiPC 2002)*, December 2002, pp. 195-206
- [9] S. Karlsson and M. Brorsson, An Infrastructure for Portable and Efficient Software DSM, In *Proceedings of 1st Workshop on Software Distributed Shared Memory (WSDSM '99)*, Rhodes, Greece, June 25, 1999, also available from Department of Information Technology, Lund University, P.O. Box 118, SE-221 00 Lund, Sweden
- [10] S. Karlsson, *OdinMP homepage*, <http://www.odinmp.com>, retrieved on August 8th 2004
- [11] S. Karlsson and M. Brorsson, Priority Based Messaging for Software Distributed Shared Memory, In *Journal on Cluster Computing*, 6 (2): 161-169, April 2003
- [12] S. Karlsson and M. Brorsson, Producer-Push - a Protocol Enhancement to Page-based Software Distributed Shared Memory Systems, in *Proceedings of the 1999 International Conference on Parallel Processing (ICPP'99)*, September 1999, pp. 291-300

- [13] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, version 1.1, June 12, 1995
- [14] OpenMP Architecture Review Board, *OpenMP specification*, C/C++ version 1.0, October 1998
- [15] OpenMP Architecture Review Board, *OpenMP specification*, C/C++ version 2.0, March 2002
- [16] Z. Radovic, and E. Hagersten, Hierarchical Backoff Locks for Nonuniform Communication Architectures, in *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, February 2003
- [17] A. Rodman, and M. Brorsson, Programming Effort vs. Performance with a Hybrid Programming Model for Distributed Memory Parallel Architectures, in *Proceedings of Euro-Par 1999*, September 1999, pp. 888-898
- [18] Y. Zhou, L. Iftode, and K. Li., Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. in *Proceedings of the 2nd Operating Systems Design and Implementation Symposium*, October 1996