Optimizing message-passing performance within symmetric multiprocessor systems

Xuehua Chen

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
https://lib.dr.iastate.edu/rtd/19929

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Optimizing message-passing performance within
symmetric multiprocessor systems

by

Xuehua Chen

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Suraj Kothari, Major Professor
Simanta Mitra
Manimaran Govindarasu

Iowa State University
Ames, Iowa
2003

Copyright © Xuehua Chen, 2003. All rights reserved.
This is to certify that the Master's thesis of

Xuehua Chen

Has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

## 1. OVERVIEW

1.1. Introduction to Parallel Computation  

1.2. Explicit Parallelization Models  
   1.2.1. Message-passing model  
   1.2.2. One-sided Communication Model  
   1.2.3. Shared Variable Model  
   1.2.4. Hybrid Model  

1.3. The Meaning of Efficient SMP Message Passing  

1.4. The Objective and Organization of This Thesis  

## 2. INTRODUCTION TO SMP MESSAGE PASSING

2.1. Introduction to MP_Lite  

2.2. The SMP Architecture  

2.3. MPI Libraries  
   2.3.1. MPICH and MPICH2  
   2.3.2. MPI/PRO  
   2.3.3. LAM/MPI  

2.4. Efficient SMP Message Passing of LAM/MPI  
   2.4.1. LAM/MPI Data Structures  
   2.4.2. Algorithms for the Send and Receive Operations
3. NEW SMP MESSAGE-PASSING IMPLEMENTATIONS

3.1. Introduction to MP_Lite
3.2. SMP Message Passing Implementations for MP_Lite
   3.2.1. Important Data Structures
   3.2.2. Algorithms for the Send and Receive Operations
   3.2.3. Synchronization and Lock Mechanisms
   3.2.4. Memory Copy Issues
3.3. Comparisons of Various Implementations

4. PERFORMANCE EVALUATION

4.1. Introduction to NetPIPE
4.2. Factors Affecting the Performance
4.3. Performance from Main Memory
4.4. Summary

5. PERFORMANCE ANALYSIS

5.1. Introduction to Performance Tools
   5.1.1. Introduction to PAPI
   5.1.2. Introduction to VTune
5.2. Timing Analysis of SMP message-passing Implementations
5.3. IA32 Architecture and the MESI Protocol
5.4. Event analysis of SMP message-passing
5.5. Discussion

6. PERFORMANCE IN A REAL APPLICATION

6.1. The CMD Code and Test Environment
6.2. Performance Results 68
6.3. Conclusions 70

7. CONCLUSIONS 71
7.1. Summary of Performance Optimization Techniques 71
7.2. Comparisons of Various SMP Message-passing Implementations 72
7.3. Future work 73

BIBLIOGRAPHY 75
CHAPTER 1  OVERVIEW

1.1 Introduction to Parallel Computation

Although the speed of serial computers keeps increasing at a very high rate, it is impossible to increase indefinitely due to many physical limitations. For many computation problems, we can still achieve a computation rate higher than the upper limit of serial computers by using parallel computers, which consist of multiple processors connected together with a high-speed network. In addition, from cost-performance point of view, the price of a parallel computer formed by connecting some cheap microprocessors is considerably lower than a serial computer with the same raw computation rate. For these reasons, parallel computers are widely used to solve computation intensive problems nowadays.

The two classes of architectures of parallel computers are based on the control mechanism: SIMD (single instruction stream, multiple data stream) and MIMD (multiple instruction stream, multiple data stream). In SIMD, a global control unit dispatches instructions to multiple processing units, and each instruction is executed by the processing units synchronously. This kind of architecture is synchronous and deterministic in that all processing units execute the same instruction in the same clock cycle. MIMD is more flexible than SIMD in that different processors can execute different instructions. Code on different processors can be the same, which is the SPMD (single program, multiple data), or different.

To utilize the power of parallel computers, some parallel programming techniques are needed. All the techniques can be classified into two parallel programming models: implicit and explicit parallel programming.

In implicit parallel programming, a sequential language is used to develop a program, and a parallelizing compiler will convert the sequential code into native parallel code by inserting the necessary instructs. This approach requires the least effort from programmers. The compiler takes up the time-consuming burden of dependency analysis and program
transformations. The runtime support system can also exploit some parallelism that can only be revealed during runtime. It is very difficult to automatically convert a sequential code to a very efficient parallel one. The speedup in performance by this approach is low compared with explicit programming.

In explicit programming, special language constructs, library functions and/or compiler directives are used to implement a parallel algorithm. The algorithm explicitly specifies the way that multiple processors cooperate with each other to solve a problem. Explicit parallel programming is widely used because it is more efficient and more portable than implicit programming.

1.2 Explicit Parallelization Models

With the development of parallel programming, many parallelization models have been proposed. There are three dominant parallelization models, message passing, shared-variable and hybrid techniques.

1.2.1 Message-Passing Model

In this model, data needs to be allocated explicitly. Programs of this type are multi-process and asynchronous, so explicit synchronization is needed. Each process has its own address space. Two widely used standard libraries are PVM [PVM03] and MPI-1 [MPI94] (old MPI standard). They are both two sided communication protocols. PVM and MPI are now supported on almost all the major parallel computers, which makes message-passing programs very portable.

PVM is the abbreviation for Parallel Virtual Machine. It is designed to utilize a network of heterogeneous computers as a single large parallel computer. MPI stands for Message Passing Interface. It is a standard specification developed by the MPI Forum for a message-passing library.

In [GL98], these two commonly used libraries are compared. PVM has the capability of dynamically managing processes. MPI-1 does not support this, but this feature has been included in the MPI-2 standard. PVM has been implemented on a wide range of platforms under Unix and Windows. MPI has more portability as it also considers systems other than
Unix and Windows. Both specifications support heterogeneity, which means the portability to a network of physically different machines. PVM has some functions to pack specific data types to a buffer while MPI has basic and derived data types. PVM is not a standard but a specific library. It is can be changed more easily and frequently than MPI. Another important feature of these two standards is interoperability, which is the ability for different implementations of a standard to communicate with each other. The feature is now in MPI-2, and is named IMPI.

1.2.2 One-sided Communication Model

In the message-passing model, data are transferred in a cooperative way. From [WER99] and [NLR03], this model has two disadvantages. The cooperative nature can limit performance by introducing an order in data delivery. In addition, programs with dynamic communication pattern and work distribution are hard to write in the message-passing model.

The one-sided communication model can solve these two problems. In this model, processes can access remote data asynchronously without the cooperation of the remote process. This model is especially useful for parallel programs that need to access remote data unpredictably. In [NLR03], it is mentioned that a class of applications, electronic structure computations, requires remote memory access.

A group of libraries exist that support one-sided communications, such as SHMEM on CRAY machines and SGI Origins, LAPI on the IBM SP, and active messages on the CM-5. These libraries support remote memory copies with contiguous data.

ARMCI [Cen03] (Aggregate Remote Memory Copy Interface) is a general purpose, portable and efficient one-sided communication library implemented on multiple platforms. Its implementation is platform specific in that it uses whatever mechanism is available on a specific platform to achieve the best performance. For example, the implementation of ARMCI on the IBM SP uses LAPI. ARMCI has been used to implement some high-level libraries such as Global Arrays and GPSHMEM. In addition, this library has another advantage over other one-sided communication libraries, its optimizations of noncontiguous data transfers and generalized scatter/gather operations.

Due to the advantages of this model, remote memory copy is included in MPI-2 as an
important new feature.

1.2.3 Shared Variable Model

In this model, there is only one process consisting of multiple threads. Every thread shares a single address space. Since all the data resides in this single shared address space, there is no need for data allocation. However, this feature makes it easy to make some subtle synchronization mistakes when reading and writing to shared variables. Programs of this type are also synchronous and require explicit synchronizations. Three existing shared variable standards are X3H5, Pthreads (Posix threads) and OpenMP.

X3H5 is an ANSI standard established by ANSI/X3 in 1993 based on the work done by the Parallel Computing Forum (PCF). Although its ideas are adopted by many shared memory systems, the standard has not been implemented completely. This standard was not adopted largely due to waning interest as distributed memory machines became popular.

Pthreads is the official IEEE POSIX 1003.1c thread standard established by IEEE. It only has a C interface. This standard allows users to create and join threads according to tasks. Explicit synchronization between threads is necessary. Since the user has good control of the threads, nested parallelism is possible using this approach. It is not easy to use this model for high performance computation as it targets task parallelism instead of data parallelism. In addition, it requires programming at a level lower than most technical developers would like.

OpenMP [Ope97] is a shared memory standard established by the OpenMP Standard Board in 1997. It specifies a set of compiler directives and library routines for parallel application programmers, which makes writing multi-threaded programs much easier. It has support for data parallelism but not task parallelism. C/C++ and Fortran interfaces are available for this standard. OpenMP uses the fork-join model of parallel execution. The API provides the ability to dynamically alter the number of threads, which may be used to execute different parallel regions. OpenMP also has the capability to incrementally parallelize a serial program and the capability to implement both coarse and fine grain parallelism. Unlike MPI or PVM that can be run on all architectures, OpenMP is designed specifically for the shared-memory architecture.
1.2.4 Hybrid Model

It is common to use clusters of SMPs (Symmetric Multiprocessor Systems). To optimize the performance for this kind of architecture, a new approach emerges that combines both message passing and shared variable approach to utilize the advantages of both models. MPI+Pthreads or MPI+openMP are representatives of this hybrid approach. The basic idea of this approach is to use MPI for inter-node communication and threads (OpenMP/Pthreads) within SMP nodes. Tests of many large-scale applications [CE00] [Hen00] have shown the negative aspects of this mixed approach compared with pure MPI while recent research [HD02] shows the positive aspects of it in some real applications.

Though the thread-safety feature is specified in the MPI-1 standard, thread issues are often ignored in some implementations. Some features of thread support have been added in the new standard of MPI-2. However, these features have not been implemented in some well-known MPI libraries yet.

This new model requires the programmer to understand both message-passing and shared-variable programming. Therefore, it is the most difficult from the user’s point of view. It is still in the research stage and has not been widely used yet. This approach could deliver optimal performance on SMP clusters.

1.3 The Meaning of Efficient SMP Message Passing

Programs written in MPI can run on any parallel computers. The portability of MPI makes it widely accepted by users and many parallel programs have been written using MPI. A better implementation of MPI library on SMP can make these programs achieve better performance on SMP systems.

In addition, with the trend of forming clusters of SMPs, a better implementation of MPI for SMP systems could deliver higher performance on this kind of architecture. For many large-scale computation problems, hybrid parallel programming using MPI+OpenMP has poorer performance than pure MPI and it is more difficult for the user to write. Unless significant progress in usability and efficiency of mixed programming happens, programs written in MPI will still dominate for a long time. With a more efficient implementation of message passing on SMPs, better performance from clusters of SMP systems could be
achieved and the scalability of many programs written in MPI could be enhanced.

Recently, a new technology called InfiniBand [Inf03] arises that could bring a huge change to the parallel programming. This technology greatly improves the communication bandwidth and latency between nodes. Some performance tests have shown that the bandwidth of InfiniBand is getting closer to the bandwidth of main memory. With this technology, the communication between nodes will no longer be a severe bottleneck for clusters of SMP systems as before, which also means that the efficiency of message passing within a SMP node will play a more important role than before.

1.4 The Objective and Organization of This Thesis

Message passing is widely used on SMP systems in real practice. In this thesis, our objectives are to explore different ways of implementing an efficient SMP message-passing library that could have better performance than current SMP modules in other MPI libraries, and to have a better understanding of the performance results of message passing on SMP systems. The remainder of the thesis is organized in the following way.

In Chapter 2, the MPI standard and the basic structure of the SMP architecture are introduced in the beginning. Then several common MPI libraries are introduced and brief descriptions of implementations are given for some of them. An efficient SMP message-passing implementation for one of the library LAM/MPI is described in detail.

Two new SMP message-passing implementations using the lock-based and lock-free approaches are elaborated in Chapter 3, each with its advantages and disadvantages. Efficient synchronization, lock mechanisms, and memory copy are critical factors affecting performance. Various ways to implement each are studied, and their efficiencies compared. Finally, various SMP message-passing implementations are compared.

Chapter 4 evaluates the performance of various implementations of SMP message passing from different aspects on several SMP systems. From results shown by a convenient and powerful tool, NetPIPE [Ame03], advantages of the new implementations over others are shown.

Chapter 5 delves into performance analysis of SMP implementations. Much effort has been spent to understand performance phenomena of different implementations on various
SMP systems. Performance analysis tools like PAPI [DLM01] and VTune [Int03] are used to gather useful runtime information such as various cache events of the test programs. With this data and hardware information on the SMPs, some performance phenomena are explained from an architectural point of view.

The best way to verify that an implementation is better is to compare it using real applications against other MPI implementations. In chapter 6, tests of a real application in physics have been conducted on several machines for various implementations and results show the merits of the new implementation.

Chapter 7 presents the conclusions and provides some suggestions for future work.
CHAPTER 2 INTRODUCTION TO SMP MESSAGE PASSING

This chapter focuses on message passing on SMP systems. The MPI standard that is the most widely message-passing model is introduced first. Then the basic structure of the SMP architecture is described. Some well-known MPI libraries together with their SMP message-passing implementations are briefly introduced, including MPICH, LAM/MPI, and MPI/PRO. LAM/MPI has an efficient transport layer, usysv, for SMP systems, which delivers very high throughput. The implementation of the usysv layer is described at the end for better understanding of the performance.

2.1 The MPI Standard

In the early 1990s, massively parallel processing (MPP) systems and networks of workstations became popular in high performance computing. Most of these used the message-passing computation model. Various message-passing libraries were developed for different systems. MPP vendors had their own libraries while some other libraries, such as TCGMSG [Wil03], PVM [Oak03], and p4 [Arg03], were used on networks of workstations. A unified standard that could run on any parallel environment did not exist and programs written were not portable at all.

The MPI Forum, a broadly based consortium of parallel computer vendors, library writers and application developers, was formed to solve this problem. It released the MPI-1 standard in the summer of 1994. The standard is very good for the following reasons.

It achieves great portability by providing a public domain, platform-independent standard library, specifies this library in a language-independent form, and provides C, C++ and Fortran bindings. The specification does not contain any feature that is specific to any particular vendor, operating system, or hardware.

It provides powerful functionality based on four concepts: Data types, communicators, communication operations, and a virtual topology. Many point-to-point and collective functions are specified and the concept of derived data type is introduced, which is a great
improvement over previous message-passing libraries.

Because the standard provides more features, more precise semantics, and more potential for highly optimized implementations, it was adopted shortly after its release by all major parallel computer vendors and widely accepted by users.

MPI-1 is based on a strict message-passing model in which all data is transferred as a cooperative operation among a fixed number of participating processes. Due to this strict model, it lacks some useful features. For example, it does not include one-sided communications, where one process can specify the communication and data transfer is separated from synchronization. One-sided communication provides some elements of a shared memory model in a message-passing environment. MPI-1 does not have the capability of dynamically changing the number of processes as PVM does. Some other features that are useful but not included in MPI-1 include parallel IO, the support for threads, and mixed model programming. All these new and useful features were included in the release of the MPI-2 standard [MPI97] in 1997 after the MPI forum revisited their work.

2.2 The SMP Architecture

There are two kinds of shared memory multiprocessor systems, symmetric multiprocessor (SMP) systems, and cache-coherent non-uniform memory access computers. Shared memory systems have very low latency for communications between processors compared to multi-computers connected by networks. Of the two shared-memory architectures, the SMP architecture has been more widely studied and adopted in most parallel servers. Figure 2-1 shows the basic structure of the SMP architecture.

An SMP system uses a shared bus or a crossbar switch to connect multiple CPUs to main memory and the I/O system. When a share bus is used, the bus arbitration logic grants the bus access one request at a time. There can be a bottleneck when processors and I/O controllers request the memory bus and the shared memory at the same time. Techniques such as split transactions and multiple non-blocking outstanding requests can be used to relieve this problem. An example of a shared bus is the recent NetBurst Micro-architecture system bus of Intel Pentium 4 computers. Currently, the available speeds of the bus are 400 MHz and 533 MHz, and both the split-transaction and pipeline techniques are used.
SMP systems using a shared bus include the IBM SP Winterhawk, and most Intel and AMD based multiprocessor computers. With comparatively low cost and fairly good performance, this kind of SMP has become very common.

![Diagram of SMP architecture]

Due to the contention of the shared bus and memory, the shared-bus SMP architecture is not very scalable. Therefore, the number of processors is limited for this architecture. SMP systems using crossbar interconnects are much more scalable. Normally, multiple memory banks are used in the architecture. Lower-order address bits are used to select banks, which can reduce bank conflicts. A crossbar-switched network connection can provide a much higher memory bandwidth by allowing multiple simultaneous memory accesses of different memory banks. One memory bank can only be accessed by one processor at a time. The crossbar switch needs to arbitrate and grant only one access when there are multiple requests for one memory bank. SMP systems that use a crossbar switch include the IBM SP Nighthawk and the Sun Ultra Enterprise 10000 system.

Though our MPI implementations can run on different types of SMP systems, the work here is mainly focused on the most commonly used shared-bus SMP systems.

2.3 MPI Libraries

There are various implementations of MPI made by different computer vendors and
research organizations. Some well-known MPI implementations are MPICH [WLS03], MPI/PRO [MST03], and LAM/MPI [Ope03]. MPICH and LAM/MPI are freely available and portable to a wide range of platforms. MPI/PRO is a commercial product supported on various clusters and some SMP systems.

2.3.1 MPICH and MPICH2

MPICH is the first complete implementation of the MPI-1 standard. It was originally built on p4 and Chameleon libraries. With its fast availability, portability, good performance, and stability, the library has been widely used in the area of parallel computation.

Since it was implemented, many modifications have been made to the library. MPICH achieve good portability and performance by the specification of the Abstract Device Interface (ADI) layer. All MPI functions are implemented on the ADI layer, which is portable and easy to implement. Channel interfaces are lower level implementations of ADI that support different methods of data transfer. Three data exchange mechanisms, *eager*, *rendezvous*, and *get*, have been implemented.

The *eager* protocol provides low latency for efficient transfer of small messages. In this protocol, data is send to the destination where it is buffered if the corresponding receive operation has not been initiated. It has the disadvantage that sometimes it could stress the buffer available and degrade performance.

The *rendezvous* protocol is more efficient for large messages since it bypasses the buffer stage. In this protocol, data are sent to the destination only when it is requested.

The *get* protocol delivers the highest performance, but special hardware support is needed. In this protocol, the data is sent directly from the address space of the source process to that of the destination process.

MPICH has incorporated the parallel IO feature in the MPI-2 standard and is planning to implement some other features such as dynamic process management. Recently, by using a better algorithm, the performance of collective calls for large messages was greatly improved for this library.

Three devices available for SMP systems are *p4_shared*, *ch_shmem*, and *ch_lfshmem*. The *p4_shared* device can run on clusters of SMPs, while *ch_shmem* targets only SMP
systems. Test results show that on a single SMP node, the ch_shmem device can offer much better performance than the p4_shared device. The ch_lfshmem device uses a lock-free buffer and could deliver very high performance. However, it currently only supports the NEC SX-4.

Specifically, the ch_shmem device uses three protocols for different message sizes. They are the short, eager, and rendezvous protocols. Short messages are transferred through a set of packets of a chosen size that are shared by all processes. A big shared memory pool is used for both medium-sized and long messages. The eager protocol is used for the former and the rendezvous protocol for the latter. Semaphores are used for protections against multiple simultaneous accesses of shared objects. Since the ch_shmem device is more efficient than the p4_shared device, our performance tests later are mainly on this device.

The MPICH group is currently working on a new version named MPICH2. The device for SMP systems in this version is ch3_shared. Currently the implementation is the alpha stage, and some application tests have shown bugs in it.

2.3.2 MPI/PRO

MPI/PRO is a commercial library developed by MPI Software Technology. It is a full implementation of MPI-1.2 for Windows, Linux, and MacOS clusters. Many industry standards such as TCP/IP, VIA, Myrinet and InfiniBand are supported in it. This library is good in performance, robustness, and portability, with some advanced features such as thread safety and overlapping of communication and computation. On SMP systems, two protocols are used for short and long messages respectively. Since no source code is available for this commercial product, little information is known about its implementation.

2.3.3 LAM/MPI

LAM/MPI was originally developed at Ohio State University. Now it is maintained by the Open Systems Laboratory at Indiana University. LAM/MPI is a full implementation of MPI 1.1 specification with some features of the MPI-2 specification such as dynamic process spawning, one-sided communications, and parallel IO. It provides fast client-to-client communication as well as extensive debugging support.
LAM/MPI has short and long protocols. For small messages, the process sends the data and its header to the destination immediately. A more complicated method is used to transfer long messages. The source process sends out a header, possibly containing some data, to the destination process and waits for a response. After the response arrives, the rest of the message is sent. This short/long protocol is used for communication between different nodes and within an SMP node.

The communication between different nodes is supported through a TCP transport layer, while either one of the two transport layers, sysv and usysv, can be used for communication within a node. Both sysv and usysv layers use a global shared memory pool for communication between processes. The difference between these two layers is the synchronization mechanism used. The sysv layer uses SYSV semaphores while usysv uses spinlocks with the sched_yield function for synchronization. Allocation and de-allocation of memory in the global pool are critical sections that need to be protected. Both layers use semaphores to prevent multiple simultaneous accesses to the shared pool. The usysv layer is much more efficient on multiprocessor computers while sysv is superior on uniprocessor computers.

Tests have shown that the usysv layer of LAM/MPI has a very low latency for small messages and a very high throughput for medium-size messages. For better understanding of its performance, the implementation of this layer is described in the next section.

### 2.4 Efficient SMP Message Passing of LAM/MPI

The usysv layer of LAM/MPI is a good implementation that delivers very high communication performance for message passing within SMP systems. The implementations of MPI_Send and MPI_Receive routines for this layer are of special interest since they are the most commonly used. Complicated collective communication routines can be built on them.

The core parts of MPI_Send and MPI_Receive routines are the shm_fast_send and shm_fastrecv routines in shm_ulow.c file. To begin with, some important data structures are introduced and their meaning explained. Then algorithms to implement the send and receive operations are described in detail.
2.4.1 LAM/MPI Data Structures

The most important data structures in the usysv layer are the “postboxes” owned by each process, the linked list in the shared memory pool, and the hash table to store out-of-order messages. Their meanings and functions are explained as follows.

Every process has \((nproc-1)\) outgoing and \((nproc-1)\) incoming postboxes associated with it, where \(nproc\) is the total number of processes in the run. Each postbox can be accessed by two processes. The relationships of the postboxes between any pair of processes \(i\) and \(j\) are: The outgoing postbox of process \(i\) for process \(j\) points to the same area as the incoming postbox of process \(j\) for process \(i\) does. A postbox contains a header consisting of some important member variables including a synchronization variable named \(lock\), the tag and size of a message, a flag indicating the current message type (\(short\) or \(long\)), and the address of the message in the shared pool when the pool is used. Right after the header is a region of LAM_SHMSHORTMSGLEN bytes. This space is used to send the messages below \((2*\text{LAM_SHMSHORTMSGLEN})\) bytes or to send the first LAM_SHMSHORTMSGLEN bytes of a long message. The synchronization variable \(lock\) has a different meaning according to its value. A process checks this variable before sending out a message. If it equals 0, the outgoing postbox is empty and can be used. Otherwise, the process needs to wait. Before receiving a message, a process also checks the \(lock\) variable of the incoming postbox. When it is 1, the postbox is filled with some useful information and the process can proceed. Otherwise, it needs to wait for the other process to fill the postbox. The \(lock\) variables of all postboxes are initialized to 0 in the beginning.

A shared memory pool is used for transfers of long messages. A circular list is used to track which space available. Each entry of the linked list consists of the address of the next entry and the size of the space associated with the current entry. When sending out a packet too large to store in the postbox, the linked list is checked through to find an appropriate space in the shared pool. If space is found, the corresponding entry is removed from the linked list and used for the transfer of the packet. After the destination process receives the entire message, it frees the space for reutilization. Sometimes, the space freed could be merged with the adjacent space and a larger contiguous space could be formed. The linked list will be modified correspondingly. There are two configurable parameters related to the
shared pool: LAM_MPI_SHMPOOLSIZE and LAM_MPI_SHMMAXALLOC. The former defines the size of the shared pool in bytes and the latter the size of the largest space that can be allocated in the shared pool in bytes. For a message larger than LAM_MPI_SHMMAXALLOC bytes, a space of LAM_MPI_SHMMAXALLOC bytes will be allocated and used multiple times to transfer the message. Handling long messages in this way prevents very long messages from slowing down transfer of other existing messages.

Each process has a hash table containing a list of out-of-order messages that have been buffered. If several messages are mapped to a key, they will appear in the linked list corresponding to that key in the order of arrival times.

2.4.2 Algorithms for the Send and Receive Operations

The usysv layer handles messages differently depending on their sizes. The parameters of LAM_SHMSHORTMSGLEN and LAM_MPI_SHMMAXALLOC decide how the message is transferred. Normally, long messages are sent out through the shared memory pool and short messages via postboxes. LAM_MPI_SHMMAXALLOC needs to be large enough, or the communication performance will degrade for long messages. The algorithms used to implement the send and receive operations are described in detail as follows.

The send operation can be divided into the following four steps, some of which may not be necessary for certain message sizes. Only very long messages require all these steps.

1. The source process waits for the outgoing postbox to be empty, and then sets the header of the postbox to tell the destination process some information about the outgoing message. The source process cannot write to the postbox until the lock variable becomes 0, which means that the previous contents of the postbox have been used by the destination process. This prevents outgoing messages from trampling each other. A spin-lock with the sched_yield function is used in a busy wait loop, ensuring a short latency. Some members of the postbox header are set, which include the size and tag of the outgoing message, and some other control information.

2. If the message size is no more than LAM_SHMSHORTMSGLEN bytes, the source process sets a flag to notify the destination process that the outgoing message is of short type, and then copies the message to the area after the header and sets lock to 1. The send
operation is completed for a short message. Otherwise, the source process sets the flag indicating that the message is a long one and copies the first LAM_SHMSHORTMSGLEN bytes of the message to the postbox. Then it sets the lock variable to 1 and waits for the acknowledgement from the corresponding destination process. The lock variable becomes 0 again when the destination process makes an acknowledgement.

3. After the acknowledgment arrives, the size of the remaining part of the message is checked to determine the transfer method. If the size is larger than LAM_SHMSHORTMSGLEN bytes, a space will be allocated from the shared pool as an intermediate buffer to hold a packet. Then the postbox header is set to indicate that a packet is sent out through the shared pool and to tell the destination process the address and size of the new packet in the pool. The second packet of the message is copied to the allocated space and the lock variable is set to 1. Since the space is less than LAM_MPI_SHMMAXALLOC bytes, some part can still be left if the remaining part is very large. If the remaining part has a size less than LAM_SHMSHORTMSGLEN bytes, it is sent out via the postbox.

4. For a very long message, there still can be some part left after step 3. Under this situation, the rest is sent out in a synchronous way. The source process waits until the lock variable becomes 0, then sends out another packet via the same buffer used in step 3. The same procedure will be repeated until the entire message is sent.

A receive operation cooperates with a send operation to get a message. A process receives a message in at most five steps as follows.

1. It checks whether there is a match in the hash table that stores the out-of-order messages already received and buffered. If a match is found, the message is copied to the destination buffer and removed from the hash table and the receive operation is complete.

2. It waits for the lock variable of the incoming postbox to become 1, which means that a new message has arrived. The header of the postbox is checked to decide whether the arriving message is a match. If not, buffer the out-of-order message and append a new entry to the hash table.

3. It checks the message type to decide where the incoming message is located. If the current message is of short type, the message is copied from the postbox to the destination
buffer. The *lock* variable is set to 0 to notify the corresponding source process that the current message was received. If the message type is *long*, the first packet of LAM_SHMSHORTMSGLEN bytes is copied from the postbox to the destination buffer. The *lock* variable is set to 0 and an acknowledgement is sent back to the source process.

4. This step applies only to long messages. The destination process waits for the *lock* variable to become 1 again, which means that the second packet of the message has arrived. Then the header of the postbox is checked to find the transfer method used, i.e., whether the buffer used is in the shared pool or in the postbox. The packet stored in the buffer is then copied to the destination buffer.

5. If there is still some part left, it will be received in a synchronous way with the cooperation of the source process. The destination process waits for the *lock* variable to become 1, and then receives a packet from the same buffer as used in the last step. The procedure is repeated until the entire message has been received.

In general, SMP message passing in LAM/MPI uses complicated algorithms for send and receive operations. Short messages and long messages are handled in different ways. Short messages are transferred through postboxes, while long messages are transferred in multiple packets through the shared pool or postboxes. Synchronization is needed for accesses of a postbox from the source and destination processes. In the *usysv* layer, this is achieved by polling the *lock* variable of the postbox with the *sched_yield* function, while a semaphore is used for the *sysv* layer. For long messages where the share pool is used, both send and receive operations will modify the linked list of the pool, which is a critical section problem to be protected. LAM/MPI uses a semaphore to solve the problem under most operating systems. On certain operating systems such as IRIX, a process-shared mutex can also be used to protect the shared pool.
CHAPTER 3  NEW SMP MESSAGE-PASSING IMPLEMENTATIONS

In the last chapter, various SMP message-passing implementations for different MPI libraries were introduced. This chapter will present two new implementations for MP_Lite, a compact and efficient MPI library.

3.1 Introduction to MP_Lite

MP_Lite is a compact message-passing library developed by Ames Laboratory. It is a lightweight and convenient tool developed to study different ways to achieve good performance in the message-passing model.

A small subset of the most common MPI functions has been implemented, including blocking and non-blocking send and receive operations, and some basic collective calls. These functions are fundamental and important in that many other MPI functions can be built on them easily. They are also widely used by parallel programs using MPI.

MP_Lite has very good portability and can deliver high performance. A clear view of the MP_Lite architecture is shown in Figure 3-1.

Two modes, the synchronous mode and the SIGIO mode, are supported when run on top of TCP on clusters of workstations. The former achieves high performance by minimizing the buffering, while the latter utilizes the SIGIO interrupts generated when the TCP buffers empty data and is more robust.

As the kernel is involved in the TCP functions, some overhead can be caused by context switch and some extra buffering involved in the kernel. To minimize the overhead, MP_Lite can be run on VIA to achieve a higher performance by bypassing the operating system. On CRAY T3E and SGI Origin systems, the SHMEM library is utilized to implement message passing, and most of its performance can be delivered to the application layer. Two shared memory modules, which will be described soon, enable MP_Lite to run on SMP systems
MPI Applications restricted to a subset of MPI functions

MP_Lite Syntax

MPI to retain portability for MP_Lite syntax

VIA OS-bypass
TCP workstations PCs
SMP shared-memory segment
SHMEM one-sided functions

Giganet hardware
M-VIA Ethernet

Mixed system clusters of SMPs

Cray T3E SGI Origin

Figure 3-2. The architecture of the MP_Lite library.

3.2 SMP Message-Passing Implementations for MP_Lite

Most existing message-passing implementations on SMP systems use a shared memory pool as an intermediate buffer to store massages. Normally, a semaphore-based lock is used to protect the pool from multiple simultaneous modifications. This approach is called the lock-based approach in this work due to the necessity of lock mechanisms in the implementation. A new implementation using this approach was developed for MP_Lite, which exploited some new techniques to optimize the communication performance.

The lock-based approach can cause contention among processes. When multiple different processes request to modify the shared pool simultaneously, only one of them is granted the right. All other processes need to wait until the chosen process finishes its work and leaves the critical section. This can cause high overhead if the lock mechanism
implemented is not efficient or the time spent in the critical section is long. Moreover, it would be expected that the overhead should be higher when the number of processes involved increases. With this consideration, a new approach was designed that can eliminate the critical section problem in the SMP message passing.

In the lock-free approach, the shared memory pool is further partitioned into \( n_{proc} \) sub-pools, where \( n_{proc} \) is the number of processes running on the SMP node. Each process uses its own sub-pool to store outgoing messages. Receive operations in this approach will not modify the structure of sub-pools. Therefore, modifications of one place in the shared pool by different processes are avoided and every process can send or receive a message without waiting for others.

The implementations of the lock-based and lock-free modules in MP_Lite are elaborated in the following way: Important data structures of these two modules are introduced. Then algorithms used in the send and receive operations are described in detail. Since both modules used similar data structures and algorithms, they are explained together in these two parts to avoid repetition. Various lock and synchronization mechanisms are then explored to minimize the overhead. A new method of memory copy other than the \texttt{memcpy} function of the \texttt{libc} library is implemented, significantly improving the performance for large messages. Finally, comparisons of the two new implementations with existing ones are made.

### 3.2.1 Important Data Structures

The lock-based and lock-free modules have slightly different data structures that are used for similar purposes. Their meaning and functions are described in detail as follows.

In the lock-based module, a linked list is used to manage the memory in the shared pool. This differs from LAM/MPI and MPICH. The linked list keeps track of all areas in use rather than the free areas available in the shared pool. Each entry of the linked list is a header that contains the size, tag, and status of a message. The message is located right after the header. A flag variable is used in the header to indicate the status of the message, which has various meanings according to different values. When its value is 0, a space in the pool has been allocated for the message while the entire message is not in the pool yet. A value of 1 indicates that the entire message is stored in the pool but has not been received yet. Finally,
it is set to $-1$ when the message is received. The status flag is used for synchronization and space recollection. For example, the destination process needs to wait for the flag to become 1 before receiving the message. When the source process finds an entry with the flag equaling $-1$, the space used for the message can be recollected.

Things are a little different for the lock-free modules. Messages are stored to different sub-pools according to the sources. Each sub-pool has a linked list to keep track of the messages. The structure of all link lists is the same as in the lock-based module.

FIFOs are used in both modules to transfer addresses of the message headers described above. After a new space is allocated for the outgoing message, the corresponding header is created at the beginning of the space and inserted in the linked list. The address of the header will be sent to the destination through a pair of FIFOs between the source and destination processes. Though most operating systems have already provided implementations of FIFOs (named pipes) and message queues for inter-process communication, the performance delivered is not very good. To achieve higher performance, the FIFO used here is an implementation based on SYSV shared memory.

Any two processes $i$ and $j$ have a pair of FIFOs, FIFO $i \rightarrow j$ and FIFO $j \rightarrow i$, to communicate with each other. There are $nproc*(nproc-1)$ FIFOs in total when $nproc$ processes run on a SMP system. The FIFO implemented here is a circular queue shared by two processes. The message addresses are stored contiguously in the queue by the source process and retrieved in the same order by the destination process. A polling method is used when the source process waits for space available in the queue and when the destination process waits for the next incoming message address.

In both modules, a linked list is used by each process that holds headers for incoming out-of-order messages stored in the order of arrival. The headers also contain the addresses of the entries corresponding to the messages. When a process receives a message header that does not match the requirements, it will append the message header to the end of the linked list. The linked list will be searched in the beginning of each operation to see whether there is a match of the current wanted message.
3.2.2 Algorithms for the Send and Receive Operations

The send and receive operations are the most commonly used functions in the MPI programs and the basis for more complicated collective operations. This section describes in detail how these two operations are implemented respectively. Since similar algorithms are used for the lock-based and lock-free modules, they will be explained together.

A send operation consists of four basic steps: Space recollection, space allocation, transmission of message address, and storing the message.

1. The source process collects those areas containing messages received in the shared pool for reutilization. For the lock-based module, the process needs to achieve the right to modify the linked list of the shared pool first. To prevent unpredicted results caused by simultaneous modifications of different processes, a lock mechanism is used to ensure that only one process at a time can have the right to modify the linked list. Therefore, if there is a process executing in the critical section already, the current process needs to wait until the previous one completes its work. After getting the right, the process checks through the whole the linked list, collecting memory by remove all the entries with the status flag of −1. This step is simpler for the lock-free module. The process checks the linked list of its sub-pool directly to recollect memory.

2. The source process allocates a new space in the shared pool to store the outgoing message. For both modules, the process checks through the linked list again to find a free space large enough for the outgoing message. Then a new header is created and inserted into the linked list, with contents set according to the information about the outgoing message. This step is completed for the lock-free module now. For the lock-based module, the process still needs to yield the modification right over the linked list.

3. The address of the new header is sent through a FIFO to the destination process. Before sending out the address, the FIFO is checked to see whether an empty entry is available. If no empty entries exist, the process needs to wait until the corresponding destination process consumes one entry and make it available. This should rarely occur if the number of entries is not too small.

4. The source process copies the message from the source buffer to the location right after the newly allocated entry, and sets the flag of the entry to 1 indicating that the message
is ready to receive.

The destination process must coordinate with the source process to get the message. The algorithms used to implement the receive operation are the same for both modules. The receive operation can be divided into the following steps.

1. The destination process checks its linked list of headers for out-of-order messages. As stated above, each process maintains a local linked list to store headers of the out-of-order messages. If a matching header is found, the address of the corresponding entry in the shared pool can be gotten from the header, and the destination process can retrieve the message.

2. The destination process obtains an address from a specific FIFO. If the destination process has a rank of j and is receiving a message from a process with rank i, it will obtain the message header from the FIFO \(i \rightarrow j\). It could happen that the receive operation occurs before the send operation. Under this situation, no message header addresses exist in the FIFO and the destination process would need to wait for one.

3. Once the header address is known, the process checks the contents of the header to see whether the incoming message is the one desired. If not, the destination process appends the header to the linked list that stores the headers of out-of-order messages, and goes back to step 2.

4. The destination process waits until the status flag of the message header becomes 1, which means the entire message is ready, then copies the message to its destination. Finally, the status flag of the message header in the pool is set to \(-1\) to indicate that the message is completed and the entry can be removed.

In the two algorithms for the send and receive operation, there are several places that require synchronization of the source and destination processes. For example, during a transfer of a message header address, if the FIFO used is full, the source process needs to wait until the destination process retrieves one address from the FIFO. On the other hand, the destination process cannot obtain a message header address from the FIFO until the source process stores it. Another example is that the destination process can start copying the message to the destination buffer only after the source process stores the whole message to the shared pool.

These synchronizations together with the locks to protect the linked list in the shared pool
can be expensive compared to the whole send and receive operations if they are not implemented efficiently. Different synchronization and lock mechanisms are explored and their performance is compared in the following section.

3.2.3 Synchronization and Lock Mechanisms

Three synchronization mechanisms have been implemented using signals, semaphores, and spinlocks respectively. Here they are illustrated using the example that the destination process must wait for the message.

The SYSV semaphores provide the simplest way for synchronization. A solution using a semaphore is as follows.

The source process:
- Store the message to the pool
  - `sem_signal(sem_id)`

The destination process:
- `sem_wait(sem_id)`
  - Receive the message from the pool

The value of the semaphore variable is initialized to 0. The destination process will go to sleep at the call to `sem_wait` if the call to `sem_signal` has not been executed in the source process, which ensures the correctness the implementation.

Signals can also be used to solve the problem. A pseudo implementation for the synchronization between the source process and the destination process is as follows.

The source process:
- Store the message to the pool
- Set the status flag of the message header to 1
- Send the signal SIGUSER1 to the destination process
The destination process:

\[
\text{while (the status flag of the message header is 0) }
\{
\hspace{1em}\text{pause} \quad // \text{Sleep until a signal is received}
\}
\]

Receive the message from the pool

This seemingly correct implementation will run successfully most of the time. However, the destination process can block forever if the time when the source process sets the status flag to 1 and sends a signal is between the test in the while statement and the call to `pause`. To solve the problem, the signal needs to be blocked during the initialization and is only unlocked it before the call to `pause`. Another solution for the destination process is present below.

\[
\text{when (the status flag of the message header is 0) }
\{
\hspace{1em}\text{Unblock SIGUSER1 and pause} \quad // \text{step 1}
\hspace{1em}\text{Block SIGUSER1} \quad // \text{step 2}
\}
\]

Receive the message from the pool

The step 1 above is atomic in that both or none of the operations in it will execute successfully. If the signal comes between the test of the flag and step 1, the destination process will be interrupted when it tries to unblock SIGUSER1. Thereafter, the signal SIGUSER1 is still blocked, and the call to `pause` is not executed. The destination process can proceed. If the signal comes after step 1 is executed successfully, the destination process will wake up and continue. The `sigsuspend` system call is used in the implementation that actually does the work of step 1 and 2 atomically.

A spinlock is another synchronization mechanism tried, which is efficient but can waste CPU time. In this method, the destination process keeps checking the lock variable until it is
The source process:
Store the message to the pool
Set the status flag of the message header to 1

The destination process:
when (the status flag of the message header is 0)
{
    Delay a short time
}
Receive the message from the pool

It is necessary to introduce a short delay between two checking by a call to `sched_yield`, `sleep`, or `nanosleep`, or by inserting a NOP or PAUSE instruction. The reason is fully explained in [Int01]. In modern computers, a spinlock loop that keeps checking the condition variable will cause multiple simultaneous read requests. These requests can be executed out-of-order. When the other process updates the condition variable, the time spent to exit the loop is very costly due to the efforts of the processor to maintain the memory order. The penalty is more severe for more deeply pipelined processors because of the greater number of read requests executed at the same time.

Currently, the `usleep` and `nanosleep` functions are still not implemented well on Linux systems. Tests on a dual-processor Pentium 4 Xeon computer shows that the execution time of both `usleep(1)` and `nanosleep(1)` is actually more than 10 ms, which is unacceptable for transfer of small messages.

A better way is to use a `sched_yield`. When this function is called, the running process gives up the processor without blocking, and another process will get a chance to execute. A NOP instruction that actually does nothing can also be inserted here. For Intel Pentium 4, the instruction PAUSE is introduced specifically for spinlock [Int01]. This instruction causes a very short delay in the loop. This delay slows the speed that read requests are issued. The
delay is approximately the time needed for the synchronization variable to be changed by another processor. As the change of the synchronization variable is a slow event, checking it faster will not be useful but it can cause very high overhead when exiting the loop. Using the PAUSE can maximize the performance of the spinlock. The instruction PAUSE is actually the same as NOP instruction for earlier Intel CPUs.

Among the above three mechanisms, normally signal is the slowest and spinlock is the fastest. The time between a process sending out a signal and the other process receiving the signal can be 9 µs. The efficiency of a semaphore varies in different systems, but it is always slower than spinlock.

In the lock-based module, lock mechanisms are used to protect the linked list of the shared pool from multiple simultaneous modifications by different processes. There are mainly three ways to implement lock mechanisms, which are discussed as follows.

The Peterson’s Algorithm [Tan92] and the bakery algorithm [SP97] are two implementations of locks using load and store instructions alone. However, [Cur94] points out that these algorithms require the sequential memory model to work. In the sequential model, memory operations of loads and stores are executed in the program order; the order in which the instructions appear in the instruction stream. From [Int02a], the Intel386 processor enforces such a model. While the Intel486, Pentium, Pentium 4, Intel Xeon, and P6 family processors use a total store ordering model, which allows reads to go ahead of buffered writes but all the writes still execute in the program order. Thus, special attention needs to be paid when implementing locks this way. For recent Intel processors with the total store ordering model, the SFENSE, LFENSE, and MFENSE instructions are provided to serialize the memory loads and stores. Both the Peterson’s algorithm and the bakery algorithms can be implemented correctly with the help of these instructions. In the lock-based module of MP_Lite, the latter was implemented as one of the options for lock mechanisms.

The implementation of locks is easier when special instructions are available that can test and modify a word atomically, or exchange the values of two words atomically. Such instructions are available on most modern processors, though their names may differ. On Intel processors, the instruction to exchange the values of two variables atomically is XCHG.
In addition, when the LOCK prefix is used together with a set of instructions, a signal of LOCK# will be asserted during execution of the accompanying instruction. With this signal, requests of all other bus agents will be blocked so that only the processor that issues the LOCK# can access the variable. The lock-based module also has an implementation of locks using the BTS (Bit Test and Set) instruction and LOCK prefix.

LOCK operations can cause different effects on different systems. On Intel 486 and Pentium processors, the LOCK# signal will always be asserted on the bus during a lock operation. Recent Intel processes have been optimized to improve the performance. On the Pentium 4, Intel Xeon, and P6 family processors, when write back memory is used and the memory being locked resides completely in a cache line of the processor performing the lock operation, the processor may not assert the LOCK#. Instead, the cache coherence mechanism will ensure the atomicity of the operation. This is called cache locking.

Cache locking is very beneficial to the performance. When multiple processes are waiting at the entrance of the critical section, they all need to execute the atomic operation. Without cache locking, this could prevent other processes from reading or writing to main memory causing high overhead.

Semaphores provide an easy and high-level locking solution. In Unix systems, two versions of semaphores have been developed, SYSV and Posix semaphores. In the lock-based module, the SYSV semaphores are used as the third option of locks. Semaphores can have various values for synchronization under different conditions, but for the implementation of a lock mechanism, a binary semaphore or mutex is enough. Most operating systems have implemented mutexes shared among different threads within a process. Some operating systems like IRIX have implemented the process shared mutex which is much more efficient than SYSV semaphores.

The various lock mechanisms differ in their efficiencies. On most systems, the semaphore-based implementation has much more overhead than the bakery algorithm and the implementation using the test-and-set instruction. A lock mechanism based on the process shared mutex has been implemented on IRIX, and its overhead is close to those of the latter two methods, and much lower than the semaphore-based implementation.

Some issues need attention when implementing the synchronization and lock mechanisms.
[Int01] mentioned some situations under which false sharing may occur that can degrade the performance significantly. One situation is that multiple processes compete for a lock to enter a critical section and a process entering the critical section, modifies some other data that resides on the same cache line as the synchronization variable. Intel processors use the MESI protocol to maintain cache coherence. When one process is modifying the data, it needs to gain the exclusive rights to the cache line by invalidating the cache line on other processors. On the other hand, processes on other CPUs keep checking the synchronization variables, which will get the cache line from the CPU where the first process is running and change the state to Shared. The cache line will change its state over and over and data will be transferred from one CPU to the others over and over. This can cause high overhead and needs to be avoided. Another situation is that multiple synchronization variables might reside on one cache line. This could cause more severe degradation of performance. The above false sharing situation can be avoided by making the synchronization variable take up a whole cache line. In the lock-based and lock-free modules, these situations will not occur because all synchronization variables are separated, each taking up one whole cache line (128 bytes on the Pentium 4 processor and 32 bytes on the Pentium III processor).

3.2.4 Memory Copy Issues

In SMP message passing, at least two memory copies are needed for transfer of each message. The source process needs to copy the message from the source buffer to the shared memory pool, and the destination process needs to copy the message from the pool to the destination buffer. The memory copies are the most time-consuming part for all but small messages. Therefore, the memory copy rate is an important factor affecting the communication performance.

Normally the memcpy function is used directly for memory copies, which is what MPICH and LAM/MPI are doing. The memcpy function loads a few bytes from source buffer to a register and then stores those bytes from register to the destination buffer. This procedure is repeated until all the data are moved from the source to the destination. On recent Intel and AMD computers, caches are involved in the procedure in the following way. When loading several bytes from the source to the register, if the source location is already in the local
cache, data goes directly into the register. Otherwise, a read miss occurs and one whole line will be loaded into cache either from some CPU’s cache or from main memory. When storing several bytes of data to the destination, if the destination location resides in the cache, the data is write to the cache directly without going to main memory due to the write back cache policy. But if the destination location does not reside in the local cache, a whole line will be read from some other CPU or from main memory first, and then the data is written to the cache line called a write allocate.

In [Int02b], another way of memory copy is described. This method uses non-temporal stores that will not disturb the cache hierarchy and is good for a copy between two large buffers. A non-temporal store writes data to main memory directly without invoking a cache line allocation. This method can prevent cache thrashing and save bus bandwidth from dirty write backs. A full implementation of memory copy using non-temporal stores has been developed based on a simple example given out by [Int02b]. The implementation consists of the following steps.

1. The destination address is checked to see if it is divisible by 16 or not. If not, the first few bytes are copied to make the remaining destination address divisible by 16. This step is necessary because the store instruction used in the step 3 requires the destination addresses to be 16-byte aligned. The Memcopy function can be used in this step to copy the small part in the beginning of the message.

2. The remaining part is transferred in blocks, all of which are divisible by a cache line size. A block of data is pre-fetched from the source buffer into L2 cache. The _mm_prefetch function is used which will prefetch a cache line (32 bytes on the Pentium III processor and 128 bytes on the Pentium 4 processor) at a time to fill the block. In order to prevent the pre-fetched data from being invalidated, the block size cannot be too large. The block is stored to the destination 16 bytes a time as follows: sixteen bytes of data are loaded from the L2 cache to an XMM register. The _mm_load_ps function is used for loads if the source address is divisible by 16. Otherwise, the _mm_load_ps1 function is used. Then the data is transferred to the destination via the _mm_stream_ps function, which bypass the cache. The procedure is repeated until all blocks of data are transferred.

3. There could still a small part with size less than a cache line left. If so, the remaining
part is copied to the destination using `memcpy`.

Some Pentium 4 computers have the hyper-threading feature, which can affect the performance of the non-temporal copy. Some figures in the next chapter show the effect of the hyper-threading feature on these two ways of copying memory.

### 3.3 Comparisons of the Various Implementations

Both MPICH and LAM/MPI handle short and large messages differently. For the `usysv` layer of LAM/MPI, a postbox is used to transfer a small message. Both send and receive operations require only two basic steps, which is much simpler than MPICH and the two modules in MP_Lite. This gives LAM/MPI the shortest latencies for small messages. However, it puts a limitation that a send operation may need to wait for the previous message to be received. A postbox and the shared pool are used together for large messages in LAM/MPI.

The `ch_shmem` device of MPICH uses a set of shared packets to transfer small messages and a shared pool for transferring large messages. Both modules in MP_Lite use a simpler algorithm in which all messages are transferred through the shared pool in the same way.

The way in which the `ch_shmem` device of MPICH and the `usysv` layer of LAM/MPI manage the shared pool is different from that in MP_Lite. Both MPICH and LAM/MPI keep a linked list of free areas while our implementation tracks a list of areas being used. Due to this difference, both send and receive operations need to modify the linked list in MPICH and LAM/MPI. Thus, the linked list needs to be protected by lock mechanisms twice for one message transfer. In the lock-based module in MP_Lite, the send operation modifies the linked list to collect reusable memory and allocate a new area for the outgoing message. The receive operation only needs to set a flag to signify that the message has been received and can be de-allocated. Modification of the linked list is not need. In this way, a pair of lock and unlock operations are saved in the lock-based implementation. In addition, the lock-free module eliminates all need for locks. In the module, the shared pool is divided into sub-pools; each has a linked list for maintaining messages from only one specific process. This prevents simultaneous modifications of a linked list during send operations.

Synchronization and lock mechanisms can be important factors affecting performance.
Spinlocks using the `sched_yield` function are used by both MPICH and LAM/MPI to coordinate a pair of source and destination processes. It is found that a replacement of the `sched_yield` function by the PAUSE instruction can be more efficient on Pentium 4 processors, so this is used in the two MP_Lite modules. Both MPICH and LAM/MPI use semaphores to lock and unlock the shared pool. On some operating systems (IRIX, for example), locks based on process shared mutexes are also available in LAM/MPI. In the lock-based module of MP_Lite, two more methods have been tried, the bakery algorithm and an implementation employing the `test-and-set` atomic operation. The semaphore-based solution is simple but less efficient than the two new methods used.

The above comparisons are solely based on the implementation techniques. Tests are still necessary to see the communication performance delivered in applications by different implementations, which are dealt with in the following chapters.
CHAPTER 4 PERFORMANCE EVALUATION

4.1 Introduction to NetPIPE

In this chapter, the NetPIPE [SMG97] [Ame03] program is used to evaluate and compare the performance of SMP message passing for the various MPI implementations. NetPIPE stands for Network Protocol Independent Performance Evaluator. It has been used to evaluate the performance of many different communication protocols [TC03] [OF00]. This tool can provide us with useful information such as the latency and throughput of communications across a network or within an SMP system, and present it in a friendly way.

NetPIPE measures the communication performance between two processes in the following way. Both processes are initialized for the chosen communication protocol. The message size is increased exponentially with small perturbations from one byte to 8 MB. For every message size, each process dynamically allocates a buffer to hold the message. Multiple ping-pong tests are performed between the processes and the total time is measured. The average time is then calculated and used to derive the communication performance. Multiple ping-pong tests are necessary to achieve accurate results for small messages, where the time of a single ping-pong test is close to the resolution of the timing function.

NetPIPE writes the results to a formatted output file, with each line containing the message size and corresponding throughput and transfer time. The results can be viewed conveniently by plotting a graph of throughput versus message size.

4.2 Factors Affecting the Performance

NetPIPE was originally designed to measure the performance of different communication protocols between two nodes, where the only important factor was the speed of the network card. This is no longer valid when communicating within a node. Because the media used for communication is the memory itself, more factors need to be considered.

For SMP message passing, factors affecting the performances include message size,
following parts, the effects of these factors are described in more detail.

1. The message size can affect the performance in two ways; messages with different size have different cache effects and message sizes not divisible by four can degrade the communication rate because of an inefficient C library.

Figure 4-1. The effect of message sizes on the performance of SMP message passing on a 1.7 GHz dual-processor Pentium 4 Xeon computer running RedHat 8.0.

Figure 4-1 shows the performance curves of three SMP message-passing implementations, the usysv transport layer of LAM/MPI, the ch_shmem device of MPICH and the lock-based shared memory module of MP_Lite on a 1.7 GHz dual-processor Pentium 4 Xeon computer using RedHat 8.0 and the gcc compiler. Peaks are shown in all three curves, which are caused by cache effects. The total communication time can be divided into two main parts, the time spent on the synchronization between two processes and the time spent on two memory copy operations. For small messages, memory copy
operations take up a small portion of the total time while the synchronization between processes is the dominant factor affecting performance. As the message size increases, the percentage of synchronization time drops and the memory copy time becomes more important and the throughput will increase accordingly. Because the part or all of the buffer holding the message resides in cache, the memory copy at this stage is heavily affected by the cache. For larger message sizes, most data reside in main memory, so cache effects are less significant. The throughput curve drops to half main memory speed since this is a two-copy transfer.

There are many spikes for the MPICH and LAM/MPI curves in Figure 4-1, which correspond to messages with sizes divisible by four bytes. The performance can drop by half when the message size is not divisible by four. The curve corresponding to MP_Lite is free of this problem. The memory copy routine in MP_Lite fixed this problem by splitting the copy into one big segment divisible by sixteen bytes and one or two very small segments on the ends.

Gcc-2.96-113 was used to compile the SMP message-passing libraries, and the library used for linkage was glibc-2.94-5.i686. The buffers of both processes are 8-byte aligned. When the Intel icc compiler is used for compilation, the problem disappears. It is also found that compiling with the -static flag or using the library of glibc-2.94-5.i386 (not the default glibc used for Intel processors after Pentium Pro) can solve the problem. Based on these facts, it seems that the cause of the problem is a poorly optimized memcpy routine in glibc-2.94-5.i686 on RedHat.

2. Poor buffer alignment can degrade the communication performance. For the send and receive operations, if the source and destination buffers are not properly aligned, the performance will drop noticeably.

Figure 4-2 shows the throughput curves for LAM/MPI, MPICH and MP_Lite when the source and destination buffers are cache line aligned with offsets of 2, 4 and 8 bytes respectively. For LAM and MPICH, it is shown that offset_of_8 > offset_of_4 > offset_of_2 for the throughput. Things are different for MP_Lite, it is noticed that offset_of_8 > offset_of_4 \approx offset_of_2. By using the wrapper function developed around memcpy, MP_Lite can achieve good performance even when the buffer alignment is bad.
3. Different memory copy routines can result in different performance in message passing. The `memcpy` function of the `libc` library has the advantage in utilizing cache for a certain range of message sizes, while the non-temporal copy is much more efficient for very large messages since it eliminates cache thrashing.

Figure 4-3 shows the throughputs of the new lock-based algorithm utilizing the `memcpy` function and the non-temporal copy respectively. When the message size is below half the L2 cache, 128 kB here, the implementation using the `memcpy` function performs much better than that using the non-temporal copy. However, the latter method enables the implementation to achieve 50 percent higher throughput for large messages. Moreover, tests have shown that performance degrades little for those messages bad aligned or of sizes not
divisible by 4 bytes when the non-temporal copy is used.

Figure 4-3. The performance of the lock-based module of MP_Lite using the `memcpy` function and the non-temporal copy on a 1.7 GHz dual-processor Pentium 4 Xeon computer running RedHat 8.0.

Based on the results in Figure 4-3, a combined memory copy method is implemented that uses the `memcpy` function for messages below 128 kB and the non-temporal copy for messages above 128 kB. The throughput curve of the new lock-based module using the combined copy method overlaps closely with the upper bounds of the two curves in Figure 4-3.

4. SMP message-passing implementations differ in both the algorithms and synchronization mechanisms, which cause the differences in the communication performance. Figure 4-4 provides a comprehensive view of the performance of various SMP message-passing implementations, such as the `sysv` and `usysv` transport layers of LAM/MPI.
the `ch_shmem` and `p4_shared` devices of MPICH-1.2.5, the `ch3_shared` device of MPICH2, MPI/PRO and the new lock-free and lock-based modules of MP_Lite. Big differences in the performance have been shown for various MPI libraries.

![Figure 4-4. The performance of various SMP message-passing implementations on a 1.7 GHz dual-processor Pentium 4 Xeon computer running RedHat 8.0.](image)

Several facts are noticed in this figure. The `usysv` transport layer of LAM/MPI achieves the best throughput for messages below 1000 bytes, while the lock-based algorithm of MP_Lite beat others for messages between 1000 bytes and 128 kB (half the L2 cache size). For large messages, both the lock-free and lock-based implementations perform 50 percent better than other MPI implementations due to the usage of the non-temporal copy. For LAM/MPI, the `usysv` transport layer performs better than `sysv` for all message sizes. This is because the former uses spinlocks for synchronization between processes, which is much more efficient than the semaphores used in the latter. For MPICH 1.2.5, the curve for the
The **p4_shared** device of MPICH 1.2.5 is far below those of other implementations. Its **ch_shmem** device is better than the **ch3_shared** device of MPICH 2 for messages between 20 kB and 128 kB, while the latter is much better for small messages below 20 kB. MPIPRO performs poor for short and medium messages, but it is about 15 percent faster than most other message passing libraries except MP_Lite for large messages.

There are some phenomena that deserve further analysis: The lock-based module in MP_Lite and the **usysv** layer of LAM/MPI are evidently the two best implementations. The former is the fastest for messages above 1000 bytes and the latter is the fastest for messages below 1000 bytes. For MP_Lite, the lock-free module performs much worse than the lock-based module for messages above 400 bytes, while they consist of similar steps. The throughput curve of the **usysv** layer of LAM/MPI jumps at 16 kB, where it switches from a short protocol to a long protocol. If the parameter of LAM_SHMSHORTMSGLEN is increased to half the cache size, 128 kB, the second jump in the curve disappears and a similar pattern like that of the lock-based module in MP_Lite is shown. It seems that the long protocol has advantages over the short protocol for a certain range of messages. In the next chapter, some performance tools will be used to analyze these three implementations at a lower level and to explain these phenomena.

5. The hardware configuration is also an important factor that affects performances. The same implementation of SMP message passing can show different performance patterns when run on various platforms.

This can be illustrated clearly by the test results of the lock-based and lock-free implementations of MP_Lite on three different Intel SMP systems. The platforms involved are a 450 MHz dual-processor Pentium III Katmai PC, a 1.7 GHz dual-processor Pentium 4 Xeon PC and a 1.4 GHz quad-processor Pentium 4 Xeon MP server. For the Pentium III PC, each processor has 32 kB L1 cache, 16 kB for code and 16 kB for data. Its unified L2 cache is 256 kB and the speed of the system bus is 100 MHz. Both Pentium 4 computers have 16 kB execution trace cache, 8 kB L1 data cache, and 256 kB unified L2 cache. The quad-processor Pentium 4 computer has extra 256 kB unified L3 cache and the hyper-threading feature. Both Pentium 4 computers have a 400 MHz Intel NetBurst Micro-architecture system bus, which is 3 times faster than that of Pentium III. The bus uses split-transaction,
pipeline techniques, and has 128-byte lines with 64-byte accesses.

On the Pentium III, both the lock-based and lock-free modules achieve similar performance. It has a small peak centered at 8 kB (half the L1 data cache size), and a slightly lower plateau that drops at about 128 kB (half the L2 cache size). After that, the curves become flat, and the throughput is determined by main memory speed. The non-temporal copy instead of the `memcpy` function is used here to achieve better performance for large messages.

Things are different on Pentium 4 computers. The lock-based and lock-free implementations differ significantly in the performance. On the dual-processor Pentium 4 computer, the curve of the lock-based implementation has only one peak that drops at the 128 kB (half the L2 cache size). The non-temporal copy is also used which achieves a
throughput of about 3800 Mbps. A wider peak appears in the throughput curve when running on the quad-processor Pentium 4 computer, which is caused by the extra 256 kB L3 cache. However, it is found that when the non-temporal copy is used, the bandwidth for large messages is much lower than that of the dual-processor Pentium 4. Tests have shown that the memcpy function is more efficient than the non-temporal copy at this time. From [KW02], it is found that the hyper-threading feature does not always speed up programs. Performance degradation caused by this feature has been noticed for some benchmarks. Some tests have run on a 2.4 GHz uni-processor Pentium 4 Xeon computer and results show that message passing using the non-temporal copy achieves better performance than that using the memcpy function when the hyper-threading feature is disabled, while the latter is better when the feature is turned on. Why lock-based and lock-free implementations show quite different patterns on Pentium III and Pentium 4 computers is still unclear, and will be analyzed in the next chapter using some performance tools.

4.3 Performance from Main Memory

In Section 3.1, the mechanism that NetPIPE uses to measure performance is described, in which messages are transferred back and forth multiple times between two buffers of the communicating processes. When the message size is small, messages reside fully in cache, which makes the communication very fast.

However, this situation will not always occur in practice. It is difficult to determine what portion of a message resides in cache in a real application. Consider a generic situation in parallel computing where every process calculates the values of an array and sends them to other processes. The pseudo code to represent this situation is as follows:

```
For i = j, k
    Calculate A[i]
Done
Send out A[j..k] to another process
```

Many modern processors use the write-allocate policy on cache writes. When A[i] is
calculated and assigned a value, it will reside in the cache line no matter whether A[i] is previously in cache or not. However, it might happen that some previous elements can be replaced out from the cache during the calculation of later elements. For example, if another array B is used when calculating the array A, reading the array B could replace some cache lines that hold some elements of array A calculated previously. A few processors, such as the Intel 80486 and the TI MicroSPARC use the write-around (no-write allocate) policy. If the element of A[i] is not in cache originally, it will not reside in cache when being assigned a value. Another factor to consider is that most L2 caches are unified, being used for both the data and the code. The L2 cache lines storing data could be replaced during instruction load misses. For these reasons, various numbers of elements in an array can reside in cache depending on different situations when the array is sent out.

For a more comprehensive view of SMP message passing, NetPIPE was extended to measure the communication performance when messages start in main memory. It would be expected that the performance of real applications should be within a boundary formed by two throughput curves. The upper boundary is the throughput curve when the message resides in cache and the lower boundary the curve when the message starts in main memory. In the next chapter, NetPIPE will be analyzed further by employing more cache theory and by considering the intermediate shared buffer holding the message. Certain situations still exist under which the throughput could still be a little higher than the upper bound or lower than the lower bound.

The mechanism used to measure the communication performance when the messages reside in memory is as follows. During initialization, both communicating processes allocate two large pools that are much larger than the total cache size. Each pool can be divided into multiple buffers to hold messages, among which no two buffers overlap with each other or sharing a cache line with each other. One pool consists of buffers holding outgoing messages and the other consists of buffers for incoming messages. In addition, another big chunk of empty memory is allocated to flush cache. Before measuring of the throughput at each message size, the big memory chunk is read to flush the previous data in cache to main memory by filling all cache lines with useless data. This step ensures that no previous messages reside in the cache before the test. Then ping-pong tests are run, in which both
processes send out messages at different locations in the send pool and receive messages to different locations in the receive pool in a round robin pattern. This ensures that every message resides in main memory before being sent out and every message being received is stored to different buffers not in cache.

There is a hidden problem worthy of notice when measuring the communication performance. On some systems (RedHat, for example), tests have shown the copy rate of the `memcpy` function when the source and destination buffers are not initialized is higher than that when the source and the destination are initialized. The latter should be used since real applications will not send and receive garbage. Therefore, each process must initialize the memory for the two large pools before the communication begins. Otherwise, the results measured may be misleading.

![Figure 4-6](image)

Figure 4-6. The performance of various SMP message-passing implementations when data reside in main memory on a 1.7 GHz dual-processor Pentium 4 Xeon running RedHat 8.0.
The performance results of various SMP message-passing implementations on the 1.7 GHz dual-processor Pentium 4 PC are shown in Figure 4-6. The following facts can be seen from the figure. The ussysv layer of LAM/MPI performs the best for messages below 1 kB. The lock-based module of MP_Lite using the non-temporal copy is much better than any other implementation for messages above 1 kB. Between 1 kB and 128 kB (half the L2 cache size), the module using the memcpy function also exceeds other SMP message-passing implementations. For large messages, MPI/PRO shows some advantages over other implementations except the locked-based module of MP_Lite using the non-temporal copy. For clarity of the figure, some interesting performance results are not shown but listed here. The lock-free module of MP_Lite has a lower peak than the lock-based module. When the parameter of LAM_SHMSHORTMSGLEN is increased to half the L2 cache size, the second jump in the throughput curve of LAM/MPI will disappear, and the curve will have a pattern like that of the lock-free module of MP_Lite. Just as the situation before when messages reside in cache, the long protocol has advantages over the short protocol in LAM/MPI, and the lock-based module is better than the lock-free module in MP_Lite for medium-sized messages.

The hardware configuration is also an important factor affecting performance when messages reside in main memory. Figure 4-7 shows the throughput curves of the lock-based module of MP_Lite employing the memcpy function and the non-temporal copy on the three Intel SMP systems described previously. On the quad-processor Pentium computer, the throughput when the non-temporal copy is used is worse than that when the memcpy function is used. This is caused by the hyper-threading feature of the computer. On the other two systems, the non-temporal copy performs better than the memcpy function. When the memcpy function is used, the throughput curve is almost flat on the Pentium III, while peaks are shown on two Pentium 4 computers for medium-sized messages. Again, some results interesting are not shown here for clarity of the figure. For MP_Lite, the lock-based module has performance similar to that of the lock-free module on the Pentium III, while the former is much better than on the two Pentium 4 computers.
Figure 4-7. The performance of the lock-based module in MP_Lite using two memory copy methods on different systems when data starts in main memory.

4.4 Summary

In this chapter, various SMP message-passing methods are evaluated using NetPIPE. Different factors that can affect the communication performance are described and illustrated by figures. The original design of NetPIPE measures the communication performance with cache effects. An extension is made to measure the performance when messages start in main memory. The two performance curves derived give an approximation to the upper and lower bounds of the communication performance in real applications.

The results of NetPIPE show the advantages and disadvantages of different SMP message-passing implementations. The lock-based module in MP_Lite performs much better than other methods for messages above 1000 bytes. Its latency for small messages is also relatively small. The \texttt{usysv} layer of LAM/MPI outperforms all other implementations for
small messages and is very good for medium-sized messages. When the non-temporal copy is used, the communication performance can be improved for large messages and when messages start in main memory. In addition, it is noticed that the hyper-threading feature affects the performance of the non-temporal copy.

Some phenomena are noticed that are not self-evident. When messages start in cache, a high peak is noticed in the throughput curves for both the lock-based module in MP_Lite and the long protocol of the usysv layer in LAM/MPI on Pentium 4 computers. This is apparently caused by the L2 cache effect. The lock-free module in MP_Lite and the short protocol of the usysv layer in LAM/MPI does not perform as well for medium-sized messages. In addition, results on the Pentium III computer are quite different. All four implementations have similar performance for the same range of messages. It is worth finding the factors that cause all these differences, which in turn could be helpful in understanding what really happens on those computers at a low level. In the next chapter, performance tools are used to gather some runtime information for the NetPIPE program. Moreover, some explanations of these phenomena are given based on the information collected, some cache theory and the hardware.
CHAPTER 5 PERFORMANCE ANALYSIS

5.1 Introduction to Performance Analysis Tools

Performance analysis of real applications has been a difficult problem for years. For the limited and imprecise information available. For example, the function of `gettimeofday` is the best timing routine provided by some operating systems. Its resolution is about four microseconds on a Pentium 4 Xeon computer using RedHat 7.3. This makes the time measurement within a code segment of which the execution time is shorter than four microseconds impossible. In addition, people were not able to get precise and detailed information related to different layers of the memory hierarchy.

Most major processors have hardware counters that can monitor and measure a selection of processor performance metrics without affect the performance of a program. The information obtained through these counters can be valuable in understanding and improving the performance of real applications. In recent years, some performance tools exploiting hardware counters have appeared, which provide users a friendly and convenient way to get much more performance information than before. A list of this kind of tools are Rabbit [Hel03], PAPI [DLMMT01], PCL [BZ03], VTune [Int03] and TAU [OLR03]. Among these tools, the Rabbit utility is comparatively old with support limited to old Pentium and AMD computers. PAPI and PCL are well-maintained libraries that support a wide range of platforms. The VTune utility is developed by the Intel Incorporation, targeting almost all generations of Pentium computers. The TAU utility is a higher-level tool in that it is built on top of PCL or PAPI.

In this chapter, the PAPI library is used to analyze the performance of various SMP message-passing implementations on different Intel computers. Because the support of the PAPI library (also true for the PCL library) on Pentium 4 Xeon computers is still very limited, the VTune utility is used to analyze the performance on this platform. All experimental results in this chapter are derived from the NetPIPE tests with cache effects. That is, the same buffer is used to hold the outgoing message and incoming messages for each process.
5.1.1 Introduction to PAPI

PAPI is an abbreviation for Performance Application Programming Interface. The library is developed by the Innovative Computing Laboratory of the University of Tennessee. It is an instrumentation tool in that some API functions need to be inserted into programs to get dynamic runtime information.

The PAPI architecture has two layers; the Portable Layer and the Computer Specific Layer. The Portable Layer consists of low-level and high-level computer independent API functions. The high-level API is easier to use but is not thread safe. It only supports limited PAPI events, and is only used for coarse-grained measurements. The low-level API is thread-safe, more efficient and supports more events. However, it is harder to use than the high-level API. The Portable Layer is built on the Computer Specific Layer, which in return is implemented by kernel extensions, operating system calls, or assembly language code. PAPI uses the most efficient way available to access the hardware performance counters.

PAPI is a very good performance tool in that it is cross-platform, standardized, has low-overhead and is very accurate. These features are explained as follows.

It is cross-platform. Currently, the systems that PAPI support include AIX on IBM Power3 and Power4, Linux on Itanium and Itanium II, Linux/x86 on Intel and AMD, IRIX 6.5 on MIPS, Solaris 8 on Ultrasparc, Unicos on Cray, True64 Unix and Linux on Alpha.

It is standardized. PAPI defines a set of standard performance metrics on all platforms and provides a standard API, which makes it easy to use.

It has low overhead. PAPI functions (especially those of the low-level API) are implemented in a very efficient way so that there is little overhead.

It is accurate. PAPI has implemented accurate timing functions with very small resolution. In addition, the low overhead of PAPI functions affects the performance of programs being tested very little. The resources used by PAPI are also minimized. These factors ensure accurate performance results.

PAPI provides some other very useful features including multiplexing and overflow handling. Multiplexing allows a large number of events to be counted in one run using a limited number of hardware counters. An overflow happens when the number of a hardware event exceeds a predefined threshold. A user-defined handler can be called to execute some
specified operations at this time. With all these advantages, PAPI has been widely used in performance analysis and in development of high-level performance tools.

5.1.2 Introduction to VTune

The VTune Performance Analyzer is a performance tool developed by Intel. VTune for Windows has been in existence for a while and is full-featured. It can collect, analyze and display performance data for a program from the system level down to the instruction level. The utility has been long used by software developers to optimize applications on Intel processors under Windows operating system. Recently, with the effort to boost the Linux operating system, Intel developed the VTune Performance Analyzer for Linux as well. VTune under Linux has fewer features than the Windows version at this time. As our SMP message-passing work is done on Unix and Linux, here introduce more about VTune for Linux.

VTune has two data collectors: sampling and call graph. The sampling collector measures the system-level performance data during a time interval. The performance data collected is the combined results caused by all active programs on the system including the kernel, device drivers, and user applications. The call graph collector can provide program flow information for an application. It instruments an application by inserting entry and exit points, which will slow down the application during runtime. Then it launches and profiles the application, recording the information of the caller and callee functions and the number of times each function is called.

In this chapter, the sampling collector is used to get performance data for analysis. Compared with PAPI, VTune is much easier to use since there is no need to modify the source code. However, VTune (under Linux) has several disadvantages. It cannot collect thread-specific performance information. Thus, it is impossible to get the performance data related to each process. The sample collector of VTune runs as a separate process, which will access the performance counter during the runtime of the application. For the mechanism used, the collector cannot get the performance data at the function and instruction level. Though VTune has some limitations, it is still used to analyze the performance of SMP message-passing systems For the current poor support of PAPI on Pentium 4 processors.
5.2 Timing Analysis of Message-Passing Implementations

Performance results of various SMP message-passing implementations have been shown in the last chapter. From the results, the following facts are noticed: When the `memcpy` function is used, the `usysv` layer of LAM/MPI has the shortest latency for small messages. The lock-based module of MP_Lite has the highest throughput for medium-sized message, while the performance of the lock-free module is much lower for the same range of messages. Timing analysis is done on the send and receive operations in NetPIPE, which could help to identify the hot spots that cause the performance differences.

To do the timing analysis, some instrumentation functions of PAPI were inserted within the source codes of LAM/MPI and MP_Lite. PAPI has very accurate routines for time measurements that are less intrusive compared with other existing timing routines. The functions of `PAPI_get_real_cyc` and `PAPI_get_real_usec` measure the real time in CPU cycles and microseconds respectively. Tests have shown that the resolutions of these two functions on the 1.4 GHz quad Pentium 4 Xeon server are 152 cycles and 1 microsecond respectively, which are much less than that of the `gettimeofday` function. In the following parts, the `PAPI_get_real_cyc` function is employed to measure the time spent on different parts of the send and receive operations in different SMP message-passing methods, which will produce the most accurate results possible.

Although the overhead of PAPI is very small, the instrumentation method still can be very intrusive for codes that execute in a very short time. For the correctness and accuracy of the timing results, the inserted PAPI routines should not be too intrusive to the application. For small messages below 500 bytes, it is noticed that instrumentation unavoidably degrades the communication performance significantly. For this reason, time measurements were not done for very small messages.

Time measurements were done for message sizes starting from 1000 bytes where the communication performance degrades by less than 10 percent. The chosen message sizes are 1 kB, 5 kB, 15 kB, 30 kB, 60kB, and 8 MB, which cover all communication regimes for LAM/MPI. Messages of 1 KB and 5 kB are small messages transferred in one packet via the postbox. The message of 15 kB is transferred in two packets through the postbox. Messages of 30 kB and 60 kB are transferred in packets through both the postbox and the shared memory.
pool. The message of 8 MB is transferred in multiple packets, the first via the postbox and all others via the shared memory pool. The time spent on different steps of the send and receive operations of the usysv layer of LAM/MPI is shown in Table 5-1. The meaning of each step is defined in Chapter 2.

Table 5-1. The time distribution of the usysv layer of LAM/MPI on the 1.4 GHz quad-processor Pentium 4 Xeon server.

<table>
<thead>
<tr>
<th>Size</th>
<th>Operation</th>
<th>1 kB</th>
<th>5 kB</th>
<th>15 kB</th>
<th>30 kB</th>
<th>60 kB</th>
<th>8 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spin</td>
<td>408</td>
<td>408</td>
<td>428</td>
<td>444</td>
<td>556</td>
<td>844</td>
</tr>
<tr>
<td></td>
<td>Send 1st packet</td>
<td>1532</td>
<td>6384</td>
<td>11056</td>
<td>12800</td>
<td>12772</td>
<td>23268</td>
</tr>
<tr>
<td></td>
<td>Wait for ACK</td>
<td>-</td>
<td>-</td>
<td>10528</td>
<td>10876</td>
<td>10852</td>
<td>21912</td>
</tr>
<tr>
<td></td>
<td>Allocate</td>
<td>-</td>
<td>-</td>
<td>164</td>
<td>4720</td>
<td>4680</td>
<td>5240</td>
</tr>
<tr>
<td></td>
<td>Send 2nd packet</td>
<td>-</td>
<td>-</td>
<td>7608</td>
<td>9780</td>
<td>21160</td>
<td>2501904</td>
</tr>
<tr>
<td></td>
<td>Send rest part</td>
<td>-</td>
<td>-</td>
<td>244</td>
<td>180</td>
<td>176</td>
<td>33750292</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>1940</td>
<td>6792</td>
<td>30028</td>
<td>38800</td>
<td>50196</td>
<td>36303460</td>
</tr>
<tr>
<td></td>
<td>Check buffer</td>
<td>224</td>
<td>192</td>
<td>172</td>
<td>184</td>
<td>172</td>
<td>2844</td>
</tr>
<tr>
<td></td>
<td>Spin</td>
<td>3888</td>
<td>13928</td>
<td>21744</td>
<td>41120</td>
<td>71592</td>
<td>1839428</td>
</tr>
<tr>
<td></td>
<td>Recv 1st packet</td>
<td>1396</td>
<td>5764</td>
<td>8828</td>
<td>8964</td>
<td>8980</td>
<td>17024</td>
</tr>
<tr>
<td></td>
<td>Send ACK</td>
<td>-</td>
<td>-</td>
<td>556</td>
<td>652</td>
<td>636</td>
<td>1408</td>
</tr>
<tr>
<td></td>
<td>Recv 2nd packet</td>
<td>-</td>
<td>-</td>
<td>18960</td>
<td>38100</td>
<td>79884</td>
<td>4766136</td>
</tr>
<tr>
<td></td>
<td>Recv rest part</td>
<td>-</td>
<td>-</td>
<td>148</td>
<td>148</td>
<td>136</td>
<td>33381000</td>
</tr>
<tr>
<td></td>
<td>Free allocation</td>
<td>-</td>
<td>-</td>
<td>188</td>
<td>5468</td>
<td>5660</td>
<td>13904</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>5508</td>
<td>19884</td>
<td>50596</td>
<td>94636</td>
<td>167060</td>
<td>40021744</td>
</tr>
</tbody>
</table>

The results in the table corresponds to the usysv layer of LAM/MPI using the default configuration, where the values of LAM_SHMSHORTMSGLEN, LAM_MPI_SHMPOOLSIZE, LAM_MPI_SHMMAXALLOC are 8192, 16777216 and 1048576 bytes respectively. All the time measured is in CPU cycles. The cell with a value of
"-" has a meaning that the corresponding step is not needed for the given message size. In the send operation, the time spent on sending the first packet is similar for when the message sizes are 15 kB, 30 kB, and 60 kB. This is because the first packet is of the same size (8192 bytes) and it resides in cache. The corresponding time for the message of 8 MB is much longer because the first packet resides in main memory. The time spent on some other steps for various message sizes can be explained in a similar way.

Correspondingly, the time spent on different steps in the lock-based and lock-free modules of MP_Lite for the same set of message sizes were measured. Table 5-2 shows the timing results of the lock-based module. A detailed description of these steps are in Chapter 2.

Table 5-2. The time distribution of the lock-based module of MP_Lite on the 1.4 GHz quad-processor Pentium 4 Xeon server.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Size Operation</th>
<th>1 kB</th>
<th>5 kB</th>
<th>15 kB</th>
<th>30 kB</th>
<th>60 kB</th>
<th>8 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SEND</strong></td>
<td>Allocate</td>
<td>1200</td>
<td>1056</td>
<td>1040</td>
<td>1004</td>
<td>1060</td>
<td>2256</td>
</tr>
<tr>
<td></td>
<td>Write to FIFO</td>
<td>456</td>
<td>440</td>
<td>208</td>
<td>208</td>
<td>472</td>
<td>1016</td>
</tr>
<tr>
<td></td>
<td>Send to pool</td>
<td>632</td>
<td>2032</td>
<td>5100</td>
<td>9884</td>
<td>18580</td>
<td>18013072</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>2288</td>
<td>3528</td>
<td>6348</td>
<td>11096</td>
<td>20112</td>
<td>18016344</td>
</tr>
<tr>
<td><strong>RECEIVE</strong></td>
<td>Check buffer</td>
<td>156</td>
<td>156</td>
<td>148</td>
<td>148</td>
<td>156</td>
<td>912</td>
</tr>
<tr>
<td></td>
<td>Read from FIFO</td>
<td>5380</td>
<td>10164</td>
<td>19608</td>
<td>34144</td>
<td>65236</td>
<td>17984488</td>
</tr>
<tr>
<td></td>
<td>Spin until ready</td>
<td>212</td>
<td>2676</td>
<td>4988</td>
<td>9712</td>
<td>21432</td>
<td>17960180</td>
</tr>
<tr>
<td></td>
<td>Receive to local</td>
<td>1564</td>
<td>5524</td>
<td>15400</td>
<td>31728</td>
<td>60520</td>
<td>18056156</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>7312</td>
<td>18520</td>
<td>40144</td>
<td>75732</td>
<td>147344</td>
<td>54001736</td>
</tr>
</tbody>
</table>

Table 5-1 and 5-2 show that LAM/MPI performs better for a message size of 1 KB. This is because there is no need to allocate an area in the shared pool for small messages for LAM/MPI and the read operations on the FIFO cause comparatively high overhead in the lock-based module of MP_Lite. An interesting phenomenon is noticed in Table 5-2: Both the "send to pool" and "receive to local" steps are memory copy operations of a same size, but the
former costs much less time than the latter. This is because the latter requires more accesses to main memory, which will be explained later in this chapter. For messages of 5 kB and 15 kB, the operation to copy the message to the postbox in LAM/MPI is apparently more time-consuming than the operation to copy the message to the shared pool in the lock-based module of MP_Lite, which contributes significantly to the disadvantages of LAM/MPI. For messages of 30 kB and 60 kB, the space allocation step in the shared pool of LAM/MPI takes longer time than that in the lock-based module. This is because the former uses semaphores to protect the shared pool when doing the memory allocation, which causes higher overheads than the lock mechanism built on the test-and-test instruction in the latter. In addition, LAM/MPI use semaphores again to free the space used, while the space recollection is done together with the space allocation in the lock-based module. The latter is evidently more efficient as a pair of lock and unlock operations are saved. Another factor that makes the usysv layer of LAM/MPI less efficient for the medium-sized messages is that a message is separated into two parts, and the total time spent on the two copy operations is longer than that spent on the single copy operation of the message in the lock-based module of MP_Lite. When the message size is 8 MB, LAM/MPI transfers the message in multiple packets in a synchronous way, thus the total time is split evenly on the send and receive operation. In the lock-based module of MP_Lite, much more time is spent on the spinning (when reading an address from the FIFO and waiting for the message) in the receive operation. Both ways have similar performance at this time.

Table 5-3 shows the timing results for the lock-free module in MP_Lite. Because a pair of lock operations is omitted, the memory allocation step in this module is much faster compared with the previous two implementations. On the other hand, the copy operation of the “send to pool” step and the spin operation take much more time than those in the lock-based algorithm. The time spent on the spin operation for the current process relates closely to the time spent on the copy operation of the “send to pool” step of the other process. It can be concluded that the slow copy operations of both communicating processes results in the low throughput of the lock-free algorithm for the medium-sized messages.
Table 5-3. The time distribution of the lock-free module of MP_Lite on the 1.4 GHz quad-processor Pentium 4 Xeon server.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Size</th>
<th>1 kB</th>
<th>5 kB</th>
<th>15 kB</th>
<th>30 kB</th>
<th>60 kB</th>
<th>8 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate</td>
<td>S</td>
<td>400</td>
<td>448</td>
<td>420</td>
<td>464</td>
<td>484</td>
<td>1420</td>
</tr>
<tr>
<td>Write to FIFO</td>
<td>E</td>
<td>492</td>
<td>464</td>
<td>208</td>
<td>468</td>
<td>468</td>
<td>1340</td>
</tr>
<tr>
<td>Send to pool</td>
<td>N</td>
<td>1732</td>
<td>5084</td>
<td>11348</td>
<td>23376</td>
<td>44604</td>
<td>18032548</td>
</tr>
<tr>
<td>Total</td>
<td>D</td>
<td>2624</td>
<td>5996</td>
<td>11976</td>
<td>24308</td>
<td>45556</td>
<td>18035308</td>
</tr>
<tr>
<td>Check buffer</td>
<td>R</td>
<td>156</td>
<td>152</td>
<td>152</td>
<td>152</td>
<td>152</td>
<td>592</td>
</tr>
<tr>
<td>Read from FIFO</td>
<td>E</td>
<td>3580</td>
<td>10168</td>
<td>21988</td>
<td>38820</td>
<td>76064</td>
<td>17989468</td>
</tr>
<tr>
<td>Spin until ready</td>
<td>C</td>
<td>3084</td>
<td>7696</td>
<td>28928</td>
<td>61212</td>
<td>125056</td>
<td>18145300</td>
</tr>
<tr>
<td>Recv to local</td>
<td>V</td>
<td>1632</td>
<td>5932</td>
<td>15540</td>
<td>30632</td>
<td>64336</td>
<td>17917492</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>8452</td>
<td>23948</td>
<td>66608</td>
<td>130816</td>
<td>265608</td>
<td>54052852</td>
</tr>
</tbody>
</table>

The timing analysis above explains why the lock-based module of MP_Lite has longer latencies for small messages and higher throughputs for medium sized messages than those of the usysv layer of LAM/MPI. The cause of the low throughput of the lock-free module is also found to be the inefficient copy operations in the send. In later sections, the phenomena are explained from another perspective. The architecture of the computers being used is described first, and information of various cache events is gathered, which explains what happens at the hardware level.

5.3 The IA32 Architecture and the MESI Protocol

Both the P6 family (including Pentium III) and the Pentium 4 belong to the Intel IA32 architecture. On these computers, L1 and L2 cache are on chip, with possibly external L3 cache. IA32 multiprocessors adopt a low-cost shared-bus SMP architecture. The L1 cache controller can communicate with the L2 cache but direct communication with the bus is not possible. Thus, all L1 misses will result in retrieving data from the L2 cache. The L2 cache controller is connected to the shared system bus and can get data from the caches of other
processors or main memory during a cache miss. As a comparison, the AMD K7 processor adopts a different organization in which the L1 cache controller is connected to the system bus, which enables cache-to-cache transfer at the L1 level.

On most SMPs, each processor has its own cache and multiple processors can keep the same memory segment in their cache lines. [HX97] mentioned three situations that can lead to cache incoherence, where different data corresponding to the same memory segment exist. A special scheme is needed to prevent this, which is called the cache coherence protocol.

There are various cache protocols and [Fly95] classifies them into two categories; snoopy protocols and directory based protocols. Snoopy protocols are widely used on shared bus SMPs. Two snoopy protocols widely used on various kinds of processors are MESI and MOESI protocols. Processors using MESI include Intel Pentium series and AMD K6 while UltraSparc I, II, IIi, IIIi, DEC Alpha 21164 and AMD K7 use MOESI protocols.

In the MESI protocol, a cache line can be in one of the following four states: Modified, Exclusive, Shared and Invalid. Cache lines in various states have different properties. A cache line in the Modified state is valid while the copy in main memory is stale. No extra copy exists in other processors. A cache line in the Exclusive state is the only valid copy existing in all processors and the copy in the memory is up-to-date. A cache line in the Shared state and the corresponding copy in main memory is valid, but valid copies may exist in other CPUs as well. Invalid cache lines do not contain useful data. Figures 5-1 and 5-2 illustrate cache state transitions given by [Fly95].

When a read miss occurs, the missing line can be retrieved from other caches, if a valid copy exists on any of the other processors. All copies of the cache line will be in the Shared state. In addition, if the cache line provided was in the Modified state, the copy in main memory needs to be updated. When no valid cache lines of the memory segment exist in other processors, the missing cache line gets data from main memory and enters the Exclusive state.

The write back caching method is normally used in the Intel P6 family, Pentium 4 and Intel Xeon processors. In this method, a write on a cache line will always put the cache line in the Modified state while different operations may happen during the process depending on various situations. A write hit on a Modified or Exclusive cache line only updates the cache line while a write hit on a Shared cache line may require an extra operation to invalidate the Shared cache line.
lines on other processors if they exist. A write miss can be divided in two steps. The missing line is first retrieved from other caches or from main memory in a way as if a read miss happens, entering the Shared state if data comes from other cache or the Exclusive state if data comes from main memory. Then a write hit updates the cache line and puts it into the Modified state.

![Diagram](image)

**Figure 5-1.** CPU-initiated cache state transitions for the MESI protocol.

![Diagram](image)

**Figure 5-2.** Bus-induced cache state transitions for the MESI protocol.
5.4 Event analysis of SMP Message-passing Implementations

The number of cache misses is an important factor that affects the performance of a memory copy operation, since accessing main memory is much more time-consuming than accessing cach. In section 4.2, it was found that the copy operation in the send costs much more time for the lock-free module than in the lock-based module. A reasonable speculation is that more cache misses occur when the lock-free module is used. Some hardware events such as cache misses can be measured to verify this speculation.

PAPI provides a way to measure the event count at the instruction level, in which events are counted using hardware counters. Pentium 4 Xeon processors have 16 performance counters that enable measuring many different events at the same time. However, events supported on Pentium 4 processors are very limited at this time, which is possibly because of a design error in hardware. A Pentium III processor has only four performance counters, but many events can be measured. Although Pentium 4 and Pentium III processors vary in the total cache size and the cache line size, the same cache replacement policy (a pseudo Least Recently Used policy) and the same cache coherence protocol (MESI) are used. Based on these facts, measurements of the cache events on a dual Pentium III SMP computer can give us useful information on both platforms.

By inserting some PAPI calls into NetPIPE code, various cache events during the memory copy of the send operation were measured. The dual-processor Pentium III Katmai computer where tests have been done before was used. The number of cache misses was similar for L1 but differ for L2 for the lock-based and the lock-free implementations. In addition, the L2 cache misses are mainly data cache misses. Numbers of the cache misses and cache line invalidations are listed in Table 5-4.

In Table 5-4, each entry contains two numbers separated by a slash, which represent the numbers of events happen in k in the two communicating processes respectively. Four message sizes were chosen, where 5 kB and 8 kB are less than the L1 data cache size of 16 kB, 80kB and 150 kB are between the L1 cache size and the L2 cache size of 512 kB. Results in Table 5-4 clearly show that the lock-based module has fewer L2 cache misses than the lock-free module. Another fact is that the multiplication of the numbers of cache line
invalidations and the cache line size (32 bytes) are very close to the message sizes, which means that valid cache lines exist on the other processor when storing the message.

Table 5-4. Cache events of the _memcpy_ function in the send operation measured on the 450 MHz dual-processor Pentium III Katmai using PAPI-3.4.1.

<table>
<thead>
<tr>
<th>Events Size</th>
<th>L2 Data Cache Miss</th>
<th>L2 Instruction Cache Miss</th>
<th>Cache Line Invalidation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lock</td>
<td>Lock-free</td>
<td>Lock</td>
</tr>
<tr>
<td>5 kB</td>
<td>1/1</td>
<td>144/133</td>
<td>0/0</td>
</tr>
<tr>
<td>8 kB</td>
<td>1/1</td>
<td>183/199</td>
<td>0/0</td>
</tr>
<tr>
<td>80 kB</td>
<td>3/1</td>
<td>199/211</td>
<td>0/0</td>
</tr>
<tr>
<td>150 kB</td>
<td>15/223</td>
<td>1220/1623</td>
<td>0/1</td>
</tr>
</tbody>
</table>

To further verify the speculation and to understand more, VTune was used to measure some other cache events. VTune supports a larger but different set of events. However, the number of events counted includes those of all running programs, the kernel and all applications. It is also impossible to get event counts at the instruction level for the current version of VTune for Linux. To get accurate results, the number of ping-pong tests of each message size in NetPIPE was set to a very large number (5000 here), which ensures that the events measured are mostly caused by the communication between the two processes. Moreover, all the other processes are daemons with very low CPU usage For a specific message size of n bytes, the total number of events during the ping-pong tests of messages with sizes of (n-3), n and (n+3) bytes are measured.

For the difference in architecture design, VTune supports different sets of events on various types of processors. Two available events related to the L2 cache misses are the L2 Read Miss and the L2 Write Miss. The results of these two events are shown in Table 5-5.

In Table 5-5, the total numbers of L2 read misses is similar for the two modules, whereas the total numbers for L2 write miss varies significantly. The lock-free module has more write misses, which again explains why it performs worse than the lock-based module. In the ping-pong tests, the sources of L2 cache write misses include allocation of a new space in the
shared pool, writing the address to the FIFO, copying the message to the shared pool and copy
the message from the shared pool to the destination buffer. The first two steps will not cause
many write misses, since there are few write operations in them. In addition, the timing results
do not show any disadvantages of the lock-free implementation in these two steps. Thus, the
write misses come from the latter two steps. Previous results have shown that the difference in
the copy step in the receive operation are small for the two modules. Tests using PAPI have
shown that L2 cache misses of the memory copy in receive differ very little when testing on the
Pentium III computer. Based on these facts, a conclusion can be made that the lock-free
module has more L2 write misses in the memory copy step of the send operation.

Table 5-5. Cache events of NetPIPE measured on the 450 MHz dual-processor Pentium III
Katmai using VTune 1.1.

<table>
<thead>
<tr>
<th>Size</th>
<th>L2 Read Miss</th>
<th>L2 Write Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lock</td>
<td>Lock-free</td>
</tr>
<tr>
<td>5 kB</td>
<td>15432k</td>
<td>15536k</td>
</tr>
<tr>
<td>8 kB</td>
<td>23908k</td>
<td>23986k</td>
</tr>
<tr>
<td>80 kB</td>
<td>226178k</td>
<td>226327k</td>
</tr>
<tr>
<td>150 kB</td>
<td>500636k</td>
<td>501871k</td>
</tr>
</tbody>
</table>

VTune counts cache misses in a different way on the Pentium 4 processors. Events of
cache write miss are not supported. Instead, they are counted as part of cache read misses.
Thus, the cache read misses include read misses caused by program loads and program stores
(Read For Ownership misses). Another factor further complicates the counting of cache
misses. The granularity of a read miss caused by a program load is 128 bytes while that of a
RFO miss is 64 bytes. For these facts, the number of cache misses on the Pentium 4 processors
needs to be treated carefully. Events of cache read misses at different levels are still useful
indicators of the effectiveness of cache usage and two levels of cache read misses on the quad
Pentium 4 Xeon Server can be tested in a similar way. A different set of message sizes was
chosen because of the different sizes of the L1 and L2 cache on the Pentium 4 Xeon processors.
The number of ping-pong tests was also changed to 10000. The results are presented in Table 5-6. More L2 and L3 cache read misses (including program load and RFO misses) are noticed in the lock-free implementation, which comply with the results measured on the Pentium III computer and further support the speculation proposed earlier.

Table 5-6. Cache events of NetPIPE measured on the 1.4 GHz quad-processor Pentium 4 Xeon using VTune 1.1.

<table>
<thead>
<tr>
<th>Size</th>
<th>L2 Read Miss</th>
<th>L3 Read Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lock</td>
<td>Lock-free</td>
</tr>
<tr>
<td>3 kB</td>
<td>8049k</td>
<td>11988k</td>
</tr>
<tr>
<td>6 kB</td>
<td>12102k</td>
<td>20823k</td>
</tr>
<tr>
<td>50 kB</td>
<td>74020k</td>
<td>144442k</td>
</tr>
<tr>
<td>100 kB</td>
<td>284334k</td>
<td>451991k</td>
</tr>
</tbody>
</table>

5.5 Discussion

In the previous sections, some information has been collected which is beneficial for a better understanding of the phenomena demonstrated in the NetPIPE tests. In this section, cache events occurring during the NetPIPE tests are described in details using some cache theories. Then an explanation of some interesting phenomena will be made based on the hardware information and the information collected using PAPI and VTune.

When the lock-based approach is used, normally a shared pool acts as the intermediate buffer for message transfer. In the NetPIPE tests, the two communicating processes use the same buffer in the pool for the message transfer. In each process, the shared buffer maps to a cache location associated with the processor that it runs on. Thus, two copies of the shared buffer can reside in the caches of different processes. Assume that two communicating processes A and B run on processors 1 and 2 respectively. Figure 5-3 illustrates the cache events happening during the ping-pong tests. In the figure, cache A is being mapped to buffer A that process A uses to send a message out and to receive a message in. Cache B has a similar
meaning. Cache C1 and C2 are the images of the shared buffer on processor 1 and 2 respectively.

Figure 5-3. Cache state transitions in the NetPIPE tests with cache effects using the lock-based approach.

When receiving a message, the store operation of process A puts Cache A into the Modified state. The load operation on buffer A in the send does not change the state of A. Thus Cache A always stays in the Modified state. So does Cache B. When process A stores the message to the shared buffer during a send, processor 1 sends out an invalidate signal to the shared bus first, as there is another copy of the segment in the cache of processor 2 for the previous receive operation of process B. Cache C2 will be invalidated after processor B receives the signal. Cache C1 becomes the only copy of the shared buffer and is dirty (the copy in main memory is invalid). After that, process B starts to receive the message from the shared buffer. Since there is a copy of buffer A residing in cache C1 on processor 1, cache-to-cache transfer will occur in which processor 1 provides data from Cache C1 to Cache C2 of processor 2 directly. Both Cache C1 and Cache C2 enter into the Shared state. Contents of Cache C2 are written to main memory by the memory controller. For this reason, the memory copy step in the send operation is much faster than in the receive operation. Similarly, cache line invalidations happen when process B stores a message to the shared buffer, and cache-to-cache
transfers with update of main memory take place while process A receives a message from the
shared buffer to buffer A.

For the lock-free approach, each of the communicating processes uses a buffer for message
transfer. The cache events that happen during the ping-pong tests are similar to those of the
lock-based approach. They are shown in Figure 5-4. In the figure, Cache C1 and C2 are
mapped to the same buffer used by process A to send a message out, and cache D1, D2 are the
images of the buffer used by process B for transmission of outgoing messages.

![Cache State Transitions Diagram]

Figure 5-4. Cache state transitions in the NetPIPE tests with cache effects using the
lock-free implementation.

From tests using PAPI and VTune, there are more cache misses during the send operation
of each process when the lock-free approach is used. When cache misses happen, cache lines
are received from copies on the other processors. Then copies on the other processors are
invalidated before the modifications occur in the write operations. This can explain why the
send operation in the lock-based implementation costs less time than those in the lock-free implementation on Pentium 4 computers. Things are different for the dual-processor Pentium III computer. Tests have shown that the differences in the time spent on the copy step in the send operation are small for the two approaches. An explanation of this phenomenon with some speculation is as follows. The 100 MHz front side bus used in the Pentium III computers is much slower than the 400 MHz bus used in the Pentium 4 computers. There are also some other enhancements (for example, pipelining) in the new system buses used by Pentium 4 computers. In the send operation, write hits on shared cache lines result in invalidation requests on the bus to the other processors. In the receive operations, cache-to-cache transfer and memory updates during load misses take up bus bandwidth. Therefore, the system bus plays a very important role in the performance of send operations. For the lock-based implementation, the memory copy in a send operation at message size 5 kB costs about 8850 cycles on the 450 MHz dual-processor Pentium III Katmai computer. From Table 5-2, the counterpart on the 1.4 GHz quad-processor Pentium 4 Xeon costs about 2032 cycles. Considering the frequencies of the CPUs, the memory copy step in the send operation on the Pentium 4 is \((\frac{8850}{450}) / (\frac{2032}{1400}) - 1\) = 12.5 times faster than on the Pentium III.

Different cache line sizes also contribute to this difference. The cache line size of the Pentium 4 processor is 64 bytes while that of the Pentium III processor is 32 bytes, which means more invalidation of cache lines are needed on Pentium III computers. For the big difference in performance, the effect of cache misses during the memory copy step on the Pentium 4 will be much larger that on the Pentium III. In fact, it is found that the time spent on the memory copy for the lock-free implementation is 8900 cycles. The effect of cache misses is very small. On the quad Pentium 4 server, Table 5-3 shows that the time spent on the memory copy step in the lock-free implementation is 5084 cycles, almost 2.5 times of that in the lock-based algorithm. The impact of extra cache misses is large on this platform, which explains the big differences in performance between the lock-based and the lock-free implementations.

From Figure 5-4, each processor holds two buffers in cache in the lock-based implementation. If the cache replacement policy was LRU, it would be expected that the peak throughput could be achieved when the message is half the L2 cache size, because the two
buffers could be held in the L2 cache at the same time. In fact, results show that the highest throughput is achieved at messages of about 60 kB, which is far less than half of the L2 cache size of 128 KB on the Pentium 4 computer. This is because cache misses increase rapidly after a certain point before half of the L2 cache size. In Table 5-6, when the message size is increased by more than 7 times from 6 kB to 50 kB, the number of cache misses increases by about 5 times from 12102k times to 74020k times. As a comparison, when message size is doubled from 50 kB to 100 kB, the increase in the L2 cache misses is about 300%.

It is still not known what causes the sudden increase in cache misses, and why some cache misses happen for very small messages during the memory copy step of the send operation for the lock-free implementation. Further research is still necessary.

It is worth mentioning that the NetPIPE tests for messages starting in main memory still have cache effects, since the intermediate shared buffer(s) is (are) cached for small messages in the lock-based (lock-free) module. This explains why there are small peaks for the throughput curve when messages reside in main memory. A flatter curve would be expected if the shared buffer also resides in main memory, which will be a better lower bound for a message transfer. On the other hand, there are situations that a message transfer can be faster than that in the NetPIPE tests when messages reside in cache. During the memory copy step in the send operation, cache invalidations do not happen if stores hit on shared cache lines that are the only copies existing in the cache. Message transfers under this situation will be a little faster. Of course, the discussions will not apply when the non-temporal copy is used since the memory copy method bypasses cache.

The short protocol of LA I can be classified as a lock-free approach. Two separate buffers (postboxes) are used during the ping-pong tests and no locks are needed. Thus, LAM/MPI will have a similar performance pattern like that of the lock-free module of MP_Lite if the protocol is used for all message sizes. The long protocol of LAM/MPI uses postboxes and a buffer in the shared pool. It can be treated as a combination of the two approaches. It performs better than the short protocol for some range of message sizes because of the buffer used in the shared pool.

As a comparison to Intel CPUs, tests on AMD K7 processors show different performance results. For NetPIPE tests with cache effects, the peak throughput happens at about 30 kB, half
the L1 cache size for the lock-based module and 20 kB or one third of the L1 cache size for the
lock-free module. This is because the L1 cache in the AMD processor is connected to the
system bus directly so that cache-to-cache transfers and cache invalidations can happen at the
L1 level without involving of the L2 cache.

NetPIPE tests do provide some useful information including the approximate upper and
lower bounds of a message transfer rates. However, many applications use bi-directional
communications. In the future, this communication pattern will be measured and analyzed.
CHAPTER 6 PERFORMANCE IN A REAL APPLICATION

In the previous two chapters, various SMP message-passing implementations have been evaluated using the NetPIPE program and analyzed using some performance tools. Although results have shown advantages of the lock-based module of MP_Lite over other implementations, it is still difficult to conclude that the implementation will beat others in real applications. The NetPIPE utility measures the performance of SMP message passing under two situations and provides the approximate upper and lower bounds of communication throughput. However, situations in real applications can differ and the communication performance can vary accordingly. Therefore, tests using real applications are still necessary and even more important for the performance evaluation.

In this chapter, a real physics application is tested using various SMP message-passing methods and its communication performance is measured. This chapter is organized in the following way: The background of the real application is introduced and the test environment is described. Then tests of the physics code are designed for a selected set of SMP message-passing implementations, and the corresponding results are shown in tables. Finally, conclusions are made based on the results.

6.1 The Test Code and Environment

The application chosen to test different ways of SMP message passing is the Classical Molecular Dynamics (CMD) code [MTH03]. It is developed by Ames Laboratory to calculate various classical atomic interactions. The code has been run efficiently on scalar computers and on a wide range of parallel computers using the MP_Lite library.

The parallel version of the CMD code utilizes a spatial decomposition approach, in which neighboring regions are assigned to adjacent nodes when possible. Communications among nodes are localized to make the code very scalable. Figure 5.1 shows an example of the decomposition of data and communications between nodes on an Intel Paragon.

The computer system used for the tests is the 1.7 GHz dual-processor Pentium 4
workstation where NetPIPE tests have been done before. The OS on this computer is RedHat 8.0 and the corresponding kernel version is 2.4.18-27.8-SMP. Under this test environment, only two processes are involved, each running on a different processor. The communication pattern is different from that in Figure 5-1. All the communications between the two processes are bidirectional, taking place at two places in each iterative cycle of the code. $N_a$ will denote the number of atoms stored in a process. In the first communication, the processes exchange messages of size $8*N_a$ bytes and $24*N_a$ bytes respectively. In the second communication, two messages of $32*N_a$ bytes are exchanged between the two processes.

**Classical Molecular Dynamics on the Paragon**

**Embedded-Atom Method**

![Diagram](image)

- **Spatial Decomposition**
  - neighboring regions are on neighboring nodes

- **Communication is Localized**
  - providing ideal scaling

- **Pair Interactions**
  - Each node's atomic positions are cycled through half of the interaction range while the positions from nodes to the left pass through the original node, allowing all pair interactions to be calculated.

Figure 6.1 The communication pattern of the Classical Molecular Dynamics code on the 44-node Intel Paragon showing the 2-dimensional spatial decomposition of the simulation space.
The SMP message-passing implementations evaluated here include the \textit{ch\_shmem} device of MPICH, the \textit{usysv} transport layer of LAM/MPI, and the two modules of MP\_Lite. When MPI/PRO is used, the CMD code runs successfully on RedHat 7.3, but segmentation faults happen when running on RedHat 8.0. The \textit{ch3\_shared} device of MPICH2 is still an alpha version and the CMD code stops in the middle with errors when the device is used. Therefore MPI/PRO and MPICH2 results are not presented.

6.2 Performance Results

The time distribution of the CMD program was measured for the process with rank 0. The time spent on the two communication phases are of special interest, representing the efficiency of the SMP message passing method being tested. Test cases with various numbers of atoms are run to give a comprehensive view of the performance.

There will always be some randomness in the tests due to various factors. Considering this, every test case is run 10 times on an idle system and the average time is reported. The average time and the corresponding standard deviation in the two communication phases are shown in Tables 5-1 and 5-2 respectively. To see the effects of different memory copy routines, both the lock-based and the lock-free modules in MP\_Lite were tested using the \textit{memcpy} function of the \textit{libc} library and the non-temporal copy routine respectively.

Table 6-1. The communication time in the charge density calculation routine for various SMP message-passing implementations for different system sizes.

<table>
<thead>
<tr>
<th>Number of atoms</th>
<th>64 Mean</th>
<th>64 Std</th>
<th>2k Mean</th>
<th>2k Std</th>
<th>10k Mean</th>
<th>10k Std</th>
<th>100k Mean</th>
<th>100k Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPICH</td>
<td>0.6474</td>
<td>0.0396</td>
<td>0.3159</td>
<td>0.0037</td>
<td>1.0214</td>
<td>0.0052</td>
<td>0.9698</td>
<td>0.0036</td>
</tr>
<tr>
<td>LAM</td>
<td>0.4445</td>
<td>0.0411</td>
<td>0.2999</td>
<td>0.0074</td>
<td>1.0351</td>
<td>0.0073</td>
<td>0.9761</td>
<td>0.0410</td>
</tr>
<tr>
<td>Lock</td>
<td>0.4015</td>
<td>0.0110</td>
<td>0.2399</td>
<td>0.0049</td>
<td>0.9591</td>
<td>0.0324</td>
<td>0.9584</td>
<td>0.0306</td>
</tr>
<tr>
<td>Lock/NT</td>
<td>0.4113</td>
<td>0.0334</td>
<td>0.2401</td>
<td>0.0410</td>
<td>0.8972</td>
<td>0.0342</td>
<td>0.7751</td>
<td>0.0025</td>
</tr>
<tr>
<td>Free</td>
<td>0.6367</td>
<td>0.0157</td>
<td>0.2985</td>
<td>0.0031</td>
<td>1.0335</td>
<td>0.0407</td>
<td>0.9689</td>
<td>0.0297</td>
</tr>
<tr>
<td>Lock/NT</td>
<td>0.6469</td>
<td>0.0113</td>
<td>0.2679</td>
<td>0.0052</td>
<td>0.9103</td>
<td>0.0469</td>
<td>0.7451</td>
<td>0.0089</td>
</tr>
</tbody>
</table>
Table 6-2. The communication time in the atomic force calculation routine for various SMP message-passing implementations at different atom numbers.

<table>
<thead>
<tr>
<th>Number of atoms</th>
<th>64 Mean</th>
<th>64 Std</th>
<th>2k Mean</th>
<th>2k Std</th>
<th>10k Mean</th>
<th>10k Std</th>
<th>100k Mean</th>
<th>100k Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPICH</td>
<td>0.9442</td>
<td>0.0101</td>
<td>0.5059</td>
<td>0.0270</td>
<td>2.1443</td>
<td>0.0414</td>
<td>1.9390</td>
<td>0.0066</td>
</tr>
<tr>
<td>LAM</td>
<td>0.4571</td>
<td>0.0054</td>
<td>0.5091</td>
<td>0.0074</td>
<td>2.0842</td>
<td>0.0106</td>
<td>1.9303</td>
<td>0.0015</td>
</tr>
<tr>
<td>Lock</td>
<td>0.4351</td>
<td>0.0139</td>
<td>0.4350</td>
<td>0.0319</td>
<td>1.9905</td>
<td>0.0463</td>
<td>1.9360</td>
<td>0.0717</td>
</tr>
<tr>
<td>Lock/NT</td>
<td>0.4565</td>
<td>0.0604</td>
<td>0.3631</td>
<td>0.0038</td>
<td>1.8134</td>
<td>0.0455</td>
<td>1.5899</td>
<td>0.0353</td>
</tr>
<tr>
<td>Free</td>
<td>0.6882</td>
<td>0.0513</td>
<td>0.4791</td>
<td>0.0082</td>
<td>2.0804</td>
<td>0.0628</td>
<td>1.9177</td>
<td>0.0021</td>
</tr>
<tr>
<td>Lock/NT</td>
<td>0.6896</td>
<td>0.0199</td>
<td>0.4220</td>
<td>0.0300</td>
<td>1.7687</td>
<td>0.0326</td>
<td>1.4833</td>
<td>0.0107</td>
</tr>
</tbody>
</table>

In the tables above, all the times are in seconds. "Lock" and "Free" are used to represent the lock-based and lock-free modules in MP_Lite respectively. In addition, the modules using the non-temporal copy are named with a slash followed by "NT" at the end. The numbers of iterations for the four system sizes are 10000, 1000, 1000 and 100 respectively.

Both Table 6-1 and 6-2 shows the following facts. The lock-based module of MP_Lite and LAM/MPI achieve good performance for short messages when the total number of atoms is 64, while MPICH and the lock-free module of MP_Lite have high latency at this time. The lock-based module of MP_Lite is the best for the medium system size (2k). For large system sizes (10k and 100k), the memory copy method used becomes the most important factor. All the implementations using the non-temporal copy achieve higher throughput than those implementations using the *memcpy* function.

The total execution time of the CMD program consists of the communication time and the calculation time. When the total atom number is small (64), the communication time is comparable to the calculation time and the lock-based module of MP_Lite does show a lower total execution time than with the other libraries. However, no apparent advantages are found for any specific implementation when the total atom number is larger. For these testing cases, the time spent on communication takes a very small portion of the total execution time. Considering the randomness of the computation time, comparisons of the
total execution time among different implementations are not appropriate here.

6.3 Conclusions

In this chapter, various SMP message-passing implementations have been evaluated using a real application, the CMD code in physics. From the results above, the lock-based module of MP_Lite shows advantages over other implementations, while the lock-free module was not as efficient for small and medium-sized messages. In addition, it was found that the non-temporal copy can contribute to a more efficient SMP message-passing implementation for long messages. The performance of MPICH is not as good when messages are small due to its poor latency. MPICH2 has much shorter latency and should resolve this. Most of the results here comply with those of the previous tests using NetPIPE. All of these further strengthen the conclusion that the lock-based module of MP_Lite does have advantages over other implementations.

The communication situation that happens on a dual-processor computer is still comparatively simple. Things can be more complicated when more processes run on an SMP node with more than two processors. Currently, further work to analyze and optimize the performances is still in progress and comprehensive tests on the 1.4 GHz quad-processor Pentium 4 server will be done in near future.
CHAPTER 7 CONCLUSIONS

Previous chapters have discussed the development and optimization of an SMP message-passing system. This chapter sums up the techniques used to optimize the performance, compares various MPI implementations, and speculates on future directions for research in this area.

7.1 Summary of Performance Optimization Techniques

Due to its advantages in portability, scalability, and ease of use, MPI has been widely accepted and will continue to dominate in the area of scientific computing for a long time. Various MPI libraries implement SMP message passing, which enables parallel programs to run efficiently on SMP systems, or clusters of SMP systems when combined with other ways of communication such as TCP/IP. Most SMP message-passing methods use a shared memory pool as the intermediate buffer to hold messages, lock mechanisms for protection of the pool, and synchronization mechanisms for coordination among processes. However, the performance varies significantly for different implementations. Two different modules of SMP message passing were implemented for MP_Lite using the lock-based and the lock-free approaches.

In the lock-based module, every process stores its outgoing messages in a large shared pool. The pool must be protected to prevent modifications by multiple processes simultaneously. For most MPI implementations, receive operations recollect memory used for message transfers, and a pair of lock and unlock operations are needed for the memory recollection step. The lock-based module takes a new approach. A receive operation only needs to set a flag to indicate that the memory space can be used again, while the recollection happens in a later send operation. This saves some overhead time for the receive operation. In addition, a faster lock mechanism based on the test-and-set atomic instruction is used, which is more efficient than semaphores used in other implementations. To minimize the time spent on synchronization among processes, various ways are tried and their advantages
and disadvantages are compared. A spin-lock with the sched_yield function, which is being used in various MPI implementations, can achieve very good latency for most systems. When the number of processes is less than the number of processors, polling using the NOP or the PAUSE instruction can perform better. Specifically, the PAUSE instruction is best suited for synchronizations on the Pentium 4 processors. It is introduced in this platform to maximize the performance of the spin-lock and to reduce the power consumed. Finally, shared memory based FIFOs are used to efficiently transfer message addresses from sending processes to receiving processes, which are faster than the FIFOs and message queues provided by the operating system.

The lock-free module takes a quite different approach from most other implementations. It divides the shared pool into sub-pools, each process owning a separate pool to store outgoing messages. Every process stores messages to different places maintained by different linked list, so no protection is needed during the send operation. Other than this difference, the lock-free module is very similar to the lock-based module.

Two places in SMP message-passing implementations involve memory copies: storing messages to a shared buffer and receiving messages from a shared buffer to the destination buffer. Thus, the memory copy rate can be a significant factor affecting performance. A non-temporal copy is used in both modules, which enhances the communication rate of messages that reside in main memory.

7.2 Comparisons of Various SMP Message-passing Implementations

Various tests were performed to evaluate the performances of all the SMP message-passing libraries. The results show that four implementations have good performance for small messages: the usysv layer of LAM/MPI, the lock-based and lock-free modules of MP_Lite, and the ch3_shared device of MPICH2. For the intermediate region, the lock-based module of MP_Lite performs better than other implementations. The lock-free module is less efficient at this time because it does not take advantage of cache as well. However, it could perform better than others on SMP systems with more processors, where the lock overheads of other implementations will be higher. The non-temporal copy is used in both modules, which produces much higher throughputs for messages reside in main memory.
The *ch_shmem* device of MPICH shows better performance than the *usysv* layer of LAM/MPI for long messages in the CMD code. The *ch3_shared* device of MPICH2 performs well for small messages, while its throughput of medium-sized messages is not as good. It is still an alpha version and errors occur when running some MPI programs. MPI/PRO has very high latency for small messages, and poorer throughput for medium-sized messages. However, its performance for long messages is better than other implementations except for MP_Lite. It works well on RedHat 7.3 while segmentations happen when running some MPI programs on RedHat 8.0.

### 7.3 Future Directions

More research still needs to be done in the future in several areas. Performance tests have been done using NetPIPE for various SMP message-passing implementations. Analysis of NetPIPE shows that it is still limited in performance evaluation in that it only measures the performance under certain conditions. Future work to evaluate the performance under other conditions still needs to be done to provide a more comprehensive evaluation of the different implementations. Tests of only one real application have been done on a dual-processor SMP system. In the future, tests of more real applications on SMPs with more processors will be needed.

The snoopy based MESI protocol used on Intel CPUs and AMD CPUs before AMD K7 has been studied and its effect on communication performance was analyzed. Some SMP architectures use other cache coherence protocols. For example, the Alpha 21264 processor, some UltraSparc processors, and the AMD K7 processor use the MOESI protocol. The newest Alpha 21364 processor uses a directory-based protocol. It would be interesting to explore how these different architectures and cache coherence protocols affect the performance of MPI programs.

The work is focused on optimizing message passing on the SMP systems with shared buses, while some other SMP systems have crossbar switches and interleaved memory banks. Further research could be done to optimize the performance on these architectures. Some test results have shown that the vendor provided MPI library on the IBM SP2 has twice of the
throughput of other implementations for medium and long message sizes. Further research needs to be done to understand the mechanism used.

Currently, the lock-based and lock-free modules are still standalone SMP modules in MP_Lite. The next step is to integrate the SMP modules with the TCP/IP module into a new mixed module that can run on clusters of SMP systems, where the shared memory will be used in communications within a SMP node and the TCP/IP module will be used for communications between nodes.

As is mentioned in the first chapter, some mixed programming methods on SMP clusters using OpenMP and MPI have shown that its scalability is worse than the programs using pure MPI. In this work, the performance issues of MPI are studied in detail. Some future research in performance analysis of OpenMP from an architectural point of view might uncover the underlying causes of this phenomenon. Studying these two models will contribute to better explanations of various performance phenomena and a better understanding of their merits and defects.
BIBLIOGRAPHY


ftp://download.intel.com/design/perftool/cbts/appnotes/sse2/w_spinlock.pdf

ftp://download.intel.com/design/Pentium4/manuals/24547210.pdf
(date accessed Mar. 2003).

ftp://download.intel.com/design/Pentium4/manuals/24896607.pdf
(date accessed Mar. 2003).

ftp://download.intel.com/design/Pentium4/manuals/24896607.pdf
(date accessed Mar. 2003).

[Int03] Intel Corporation. Intel® VTune(TM) Performance Analyzer
http://www.intel.com/software/products/vtune/

http://www.dell.com/us/en/esg/topics/power_ps3q02-khalid.htm
(date accessed May. 2003).
http://www.mpi-forum.org/docs/mpi10.ps  


[MST03] MPI Software Technology, Inc. Introduction of MPI products.  

http://www.cmp.ameslab.gov/cmp/CMP_Theory/cmd/cmd.html  

http://www.osl.iu.edu/download/mpidc95/papers/html/nieplocha/  
(date accessed: April, 2003).


http://www.scl.ameslab.gov/Projects/MP_Lite/MP_Lite.html  
(date accessed: Jan. 2003)


