Sun HPC ClusterTools™ 6 Software Performance Guide

Sun Microsystems, Inc.
www.sun.com

Part No. 819-4134-10
March 2006, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Please
Recycle

Adobe PostScript™

Contents

Figures

Tables

Code Samples

Preface

This manual presents techniques that application programmers can use to get top performance from message-passing programs running on Sun™ servers and clusters of servers.

Before You Read This Book

This manual assumes that the reader has basic knowledge of:

- Developing parallel applications with the Sun MPI libraries
- Executing parallel applications with the Sun Cluster Runtime Environment (CRE) and a Distributed Resource Manager, such as Sun N1 Grid Engine (N1GE) Version 6, Load Sharing Facility (LSF) HPC Version 6.2 from Platform Computing, and OpenPBS Portable Batch System (PBS) 2.3.16 and Altair PBS Professional 7.1.
- Debugging parallel applications

How This Book Is Organized

This manual covers the following topics.

- Chapter 1 – Quick Reference - A summary of performance tips
- Chapter 2 – Introduction: The Sun HPC ClusterTools Solution
- Chapter 3 – Choosing Your Programming Model and Hardware
- Chapter 4 – Performance Programming with the Sun MPI (message-passing) library
- Chapter 5 – One-sided Communications
- Chapter 6 – Compilation and Linking for top performance

- Chapter 7 – Runtime Considerations and Tuning
- Chapter 8 – Profiling tools and techniques
- Appendix A – Sun MPI Implementation and how it affects performance
- Appendix B – Sun MPI Environment Variables and how to use them

Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:

  `http://docs.sun.com`

Typographic Conventions

Typeface[*]	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

* The settings on your browser might differ from these settings.

Shell Prompts

Shell	Prompt
C shell	*machine–name*`%`
C shell superuser	*machine–name*`#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	`#`

Related Documentation

The following materials provide useful background about using Sun MPI and Sun HPC ClusterTools.

Application	Title	Part Number
Sun HPC ClusterTools Documentation	*Read Me First: Guide to Sun HPC ClusterTools Software Documentation*	819-4136-10
Sun HPC ClusterTools Software	*Sun HPC ClusterTools 6 Software Release Notes*	819-4129-10
	Sun HPC ClusterTools 6 Software Installation Guide	819-4130-10
	Sun HPC ClusterTools 6 Software User's Guide	819-4131-10
	Sun HPC ClusterTools 6 Software Administrator's Guide	819-4132-10
Sun MPI Programming	*Sun MPI 7.0 Software Programming and Reference Guide*	819-4133-10

In addition, if you are using Platform Computing's Load Sharing Facility (LSF) Suite, consult the documentation available from their website:

`http://www.platform.com`

Sun HPC ClusterTools documentation is available online at:

`http://www.sun.com/documentation`

Documentation, Support, and Training

Sun Function	URL
Documentation	`http://www.sun.com/documentation/`
Support	`http://www.sun.com/support/`
Training	`http://www.sun.com/training/`

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

`http://www.sun.com/hwdocs/feedback`

Please include the title and part number of your document with your feedback:

Sun HPC ClusterTools 6 Software Performance Guide, part number 819-4134-10

CHAPTER 1

Quick Reference

This list is a summary of the key performance tips found in this document. They are organized under the following categories:

- "Compilation and Linking" on page 1
- "MPProf" on page 2
- "Analyzer Profiling" on page 3
- "Job Launch on a Multinode Cluster" on page 5
- "MPI Programming Tips" on page 7

Compilation and Linking

Compilation and linking are discussed in Chapter 6.

- Use Sun Studio Compiler Collection compilers for best performance. Sun HPC ClusterTools 6 supports versions 8, 9, 10, and 11 of the Sun Studio compilers for C, C++, and Fortran.

  See "Compiler Version" on page 73.

- Use the `mpf77`, `mpf90`, `mpcc`, and `mpCC` utilities where possible. Link with `-lmpi`. For example:

```
% mpf90 -fast -g a.f -lmpi
```

  See "The `mp*` Utilities" on page 74.

- Compile with `-fast`.

  See "The `-fast` Switch" on page 74.

- As appropriate, add the following -xarch setting after -fast:

	32-bit binary	64-bit binary
UltraSPARC II (will also run on UltraSPARC III)	-xarch=v8plusa	-xarch=v9a
UltraSPARC III (will not run on UltraSPARC II)	-xarch=v8plusb	-xarch=v9b
UltraSPARC IV and IV+	-xarch=v8plusb	-xarch=v9
AMD Opteron processors	-xarch=i386	-xarch=amd64

  See "The -xarch Switch" on page 75.

- Compile with –xalias=actual due to Fortran binding issues in the MPI standard. See "The -xalias Switch" on page 75.
- Compile and link with -g.

  See "The -g Switch" on page 76.

- Link with -lopt for C programs.
- Compile and link with -xvector if math library intrinsics (logarithms, exponentials, or trigonometric functions) appear inside long loops.
- Compile with -xprefetch selectively.
- Compile with -xrestrict and -xalias_level, as appropriate, for C programs.
- Compile with -xsfpconst, as appropriate, for C programs.
- Compile with -stackvar, as appropriate, for Fortran programs.

  See "Other Useful Switches" on page 76.

MPProf

- Before running your Sun MPI program, set the MPI_Profile environment variable to 1.

```
% setenv MPI_PROFILE 1
```

- After running your Sun MPI program, you will find a file of the form mpprof.index.*rm.jid* in your working directory. Type the following command:

```
% mpprof mpprof.index.rm.jid
```

- To archive profiling results, type the following command:

```
% mpprof -r -g archive_directory mpprof.index.rm.jid
```

- To clean up files, type the following command:

```
% mpprof -r mpprof.index.rm.jid
```

Analyzer Profiling

Use of the Performance Analyzer with Sun MPI programs is discussed in Chapter 7.

- Set your path to include the most recent compiler software, usually `/opt/SUNWspro/bin`
- The following examples show basic usage to collect performance data and analyze results:

```
% mprun -np 16 collect a.out 3 5 341
% analyzer test.*.er
```

- For more advanced data collection, use scripts. See the following example:

```
% cat csh-script
#!/bin/csh
if ( $MP_RANK == 0 ) then
  mkdir myrun
endif
if ( $MP_RANK < 4 ) then
  collect -p 20 -m on -o /tmp/proc-$MP_RANK.er $*
  er_mv /tmp/proc-$MP_RANK.er myrun
else
  $*
endif
% mprun -np 16 csh-script a.out 3 5 341
```

Here, the following techniques have been used:

- Data volumes have been reduced by profiling only a subset of the processes.
- Data volumes have been reduced by increasing the profiling frequency with the `collect -p` switch.
- Data volumes have been handled by collecting to the local filesystem /tmp. Other fast file systems can be identified by your system administrator or with the command `/usr/bin/df -lk`
- MPI wait tracing data is activated with the collect `-m` switch.
- Experiments have been named by process rank.
- Experiments have been gathered to a centralized location, directory myrun, after the MPI job finished.
- Analyzing data
  - Basic view shows how much time is spent per function.
  - Click on the Source button to see how much time is spent per source-code line. This requires that the code was compiled and linked with `-g`, which is compatible with high levels of optimization and parallelization.
  - Click on Callers-Callees to see caller-callee relationships.
  - Click on Timeline to see a timeline view.
  - Select other metrics with the Metrics button.
  - Use `er_print` to bypass the graphical interface:

```
% er_print -functions       proc-0.er
% er_print -callers-callees proc-0.er
% er_print -source lhsx_ 1 proc-0.er
```

- Look at inclusive time for high-level MPI functions to filter out internal software layers of the Sun MPI library:

```
% er_print -function proc-0.er | grep PMPI_
```

- To ensure that MPI wait times are profiled, select wall-clock time, instead of CPU time, as the profiling metric. Or, to collect data in the first place, type the following command:

```
% setenv MPI_COSCHED 0
% setenv MPI_SPIN    1
```

- Loading data:
  - The Performance Analyzer accepts experiment names on the command line, such as the following:

```
% analyzer
% analyzer proc-0.er
% analyzer run1/proc-*.er
```

  - After the Performance Analyzer has been started, use the Experiment menu to Add and Drop individual experiments.

Job Launch on a Multinode Cluster

- Checking Load (see the following example for CRE and UNIX commands useful for checking load)

	CRE	UNIX
How high is the load?	% **mpinfo -N**	% **uptime**
What is causing the load?	% **mpps -e**	% **ps -e**

See "Running on a Dedicated System" on page 79.

- Objectives for Job Launch
  - Minimize internode communication.
    - Run on one node if possible.
    - Place heavily communicating processes on the same node as one another.

See “Minimizing Communication Costs” on page 83.

- Maximize bisection bandwidth.

  - Run on one node if possible.

  - Otherwise, spread over many nodes.

  - For example, spread jobs that use multiple I/O servers.

  See “Controlling Bisection Bandwidth” on page 84.

- Examples of Job Launch with CRE as the Resource Manager
  - To run jobs in the background, perhaps from a shell script, use the `-n` option:

```
% mprun -n -np 4 a.out &
```

or use the following commands:

```
% cat a.csh
#!/bin/csh
mprun -n -np 4 a.out
% a.csh
```

See “Running Jobs in the Background” on page 85.

- To eliminate core dumps, do so in the parent shell.

  `% limit coredumpsize 0` (for `csh`)

  `$ ulimit -c 0` (for `sh`)

  See “Limiting Core Dumps” on page 86.

- To run 32 processes, with each block of consecutive 4 processes mapped to a node:

  `%` **`mprun -np 32 -Zt 4 a.out`**

  or

  `%` **`mprun -np 32 -Z  4 a.out`**

  See “Collocal Blocks of Processes” on page 88.

- To run 16 processes, with no two mapped to the same node:

```
% mprun -Ns -np 16 a.out
```

See “Multithreaded Job” on page 88.

- To map 32 processes in round-robin fashion to the nodes in the cluster, with possibly multiple processes per node:

```
% mprun -Ns -W -np 32 a.out
```

  See "Round-Robin Distribution of Processes" on page 89.

- To map the first 4 processes to `node0`, the next 4 to `node1`, and the next 8 to `node2`, type the following:

```
% cat nodelist
node0 4
node1 4
node2 8
% mprun -np 16 -m nodelist a.out
```

  See "Detailed Mapping" on page 89.

MPI Programming Tips

- Minimize number and volume of messages.

  See "Reducing Message Volume" on page 29.

- Reduce serialization and improve load balancing.

  See "Reducing Serialization" on page 29 and "Load Balancing" on page 29.

- Minimize synchronizations:
  - Generally reduce the amount of message passing.
  - Reduce the amount of explicit synchronization (such as `MPI_Barrier()`, `MPI_Ssend()`, and so on).
  - Post sends well ahead of when a receiver needs data.
  - Ensure sufficient system buffering.

  See "Synchronization" on page 30.

- Pay attention to buffering:
  - Do not assume unlimited internal buffering by Sun MPI.
  - Use nonblocking calls such as `MPI_Isend()` for finest control over user-specified buffering.
  - Post receives early to relieve pressure on system buffers.

  See "Buffering" on page 31.

- Replace blocking operations with nonblocking operations:

  - Initiate nonblocking operations as soon as possible.
  - Complete nonblocking operations as late as possible.
  - Test the status of nonblocking operations periodically with `MPI_Test()` calls.

  See "Nonblocking Operations" on page 32.

- Pay attention to polling:
  - Match message-passing calls (receives to sends, collectives to collectives, and so on).
  - Post `MPI_Irecv()` calls ahead of arrivals.
  - Avoid `MPI_ANY_SOURCE`.
  - Avoid `MPI_Probe()` and `MPI_Iprobe()`.
  - Set the environment variable `MPI_POLLALL` to 0 at run time.

  See "Polling" on page 33.

- Take advantage of MPI collective operations.

  See "Sun MPI Collectives" on page 34.

- Use contiguous data types:
  - Send some unnecessary padding if necessary.
  - Pack your own data if you can outperform generalized `MPI_Pack()`/`MPI_Unpack()` routines.

  See "Contiguous Data Types" on page 34.

- Avoid congestion if you're going to run over TCP:
  - Avoid "hot receivers."
  - Use blocking point-to-point communications.
  - Use synchronous sends (`MPI_Ssend()` and related calls).
  - Use MPI collectives such as `MPI_Alltoall()`, `MPI_Alltoallv()`, `MPI_Gather()`, or `MPI_Gatherv()`, as appropriate.
  - At run time, set `MPI_EAGERONLY` to 0, and possibly lower `MPI_TCP_RENDVSIZE`.

  See "Special Considerations for Message Passing Over TCP" on page 35.

CHAPTER **2**

Introduction: The Sun HPC ClusterTools Solution

The Sun HPC ClusterTools suite is a solution for high-performance computing. It provides the tools you need to develop and execute parallel (message-passing) applications. These programs can run on any Sun system, from a single workstation up to a cluster of high-end symmetric multiprocessors (SMPs).

This chapter presents an overview of the hardware and software products that Sun Microsystems provides for high-performance computing, with emphasis on the components of the Sun HPC ClusterTools software suite.

Sun HPC Hardware

Programs written with Sun HPC ClusterTools software run on the whole line of Sun servers and workstations. This feature enables you to exploit all available hardware in achieving performance.

For detailed information on UltraSPARC-based computing see:

`http://www.sun.com/sparc`

`http://www.sun.com/desktop`

`http://www.sun.com/servers`

For detailed information on AMD Opteron x64-based computing, see:

`http://www.sun.com/x64/index.html`

This section notes the performance-related features of Sun SMPs and clusters. These will be important in the first step of performance programming, choosing your tools and hardware, discussed in Chapter 3.

Processors

UltraSPARC microprocessors are a full implementation of the SPARC V9 architecture, which provides high-performance, 64-bit computing in a Solaris Operating System.

The UltraSPARC-I processor introduced this family in 1995.

The UltraSPARC-II processor supports CPU clock speeds in the range of 250-480 MHz and L2 cache sizes up to 8 bytes.

The UltraSPARC-III processor, introduced in 2000, has complete binary compatibility with older applications, while introducing new performance enhancements for programs that target only the latest processors. Clock speeds of up to 1.2 GHz are supported. L2 cache sizes are up to 8 Mbytes.

The UltraSPARC IV processor has clock speeds up to 1350MHz, with L2 cache sizes of up to 16 Mbytes. Its successor, the UltraSPARC IV+ processor supports initial clock speeds of up to 1.5 MHz. The UltraSPARC IV+ contains a different caching scheme than its predecessor, which allows it to support 2 MB of L2 cache on-chip as well as 32 Mbytes of external L3 cache.

The AMD Opteron 200 Series processors in Sun Fire ™ network servers have clock speeds of up to 2.4 GHz with 1 Mbytes of L2 cache per core on single- or dual-core servers.

Nodes

Nodes (the units of a cluster) might be as small as a single workstation or server, or as large as a high-end server containing multiple processors.

Clusters

SMPs might be clustered by means of any Sun-supported TCP/IP interconnect, such as Gigabit Ethernet or Infiniband.

Individual Sun HPC ClusterTools message-passing applications can have up to 2048 processes running on as many as 256 nodes of a cluster. The programmer must manage the location of data in the distributed memory and its transfers between nodes.

Sun HPC ClusterTools Software

Sun's HPC message-passing software supports applications designed to run on single systems and clusters of SMPs. Called Sun HPC ClusterTools software, it provides the tools for developing distributed-memory parallel applications and for managing distributed resources in the execution of these applications.

Sun HPC ClusterTools 6 software runs under the Solaris 10 (32-bit or 64-bit) Operating System (Solaris OS).

The Sun HPC ClusterTools suite is layered on top of the Sun compilers. Specifically, HPC ClusterTools 6 software is supported with the Sun Studio Compiler Collection 8, 9, 10, and 11 release. The compiler collections include Fortran, C, and C++ compilers, Sun Performance Library software, and the Performance Analyzer, all of which can be used to develop single-process or multithreaded programs. The HPC ClusterTools software extends such shared-memory programming with message passing, so that parallel programs can run distributed over multiple processes, potentially on multiple nodes of a cluster. The relationship between HPC ClusterTools software and the compiler collection are explored further in Chapter 3.

The remainder of this chapter describes some HPC ClusterTools features that extend shared-memory programming:

- Sun MPI library of message-passing and I/O routines
- Sun CRE, a runtime environment that manages the resources of a server or cluster to execute message-passing programs

Sun MPI

Sun MPI is a highly optimized version of the Message-Passing Interface (MPI) communications library. This dynamic library is the basis of distributed-memory programming, as it enables the programmer to create distributed data structures and to manage interprocess communications explicitly.

MPI is the de facto industry standard for message-passing programming. You can find more information about it on the MPI web page and the many links it provides:

`http://www.mpi-forum.org`

Sun MPI is a complete library of message-passing routines, including all MPI 1.2 compliant and MPI 2 compliant routines. In addition, Sun MPI provides the following features:

- Multiprotocol support. Sun MPI provides these protocol modules:
  - SHM (Shared Memory) for ranks on the same shared-memory node
  - TCP (Transmission Control Protocol) for nodes connected by any commodity interconnect supporting TCP/IP.

  In addition, any party can provide additional protocol modules to interoperate with Sun MPI and support high-performance message passing over any interconnect. At run time, Sun MPI chooses from among all loaded protocol modules to effect best performance.
- Threadsafe support.
- Finely tunable shared-memory communication.
- Optimized collectives for SMPs, for long messages, for clusters, etc.
- Parallel I/O to the ClusterTools Parallel (distributed) File System, as well as single-stream I/O to a standard Solaris file system (UFS).

Sun MPI programs are compiled on Sun Studio compilers. MPI provides full support for Fortran 77, C, and C++, and basic support for Fortran 90.

Chapter 4 and Appendix A of this manual provide more information about Sun MPI features, as well as instructions for getting the best performance from an MPI program.

Cluster Runtime Environment

The Cluster Runtime Environment (CRE) component of Sun HPC ClusterTools software serves as the runtime resource manager for message-passing programs. It supports interactive execution of Sun HPC applications on single SMPs, or on clusters of SMPs.

CRE is layered on the Solaris OS, but enhanced to support multiprocess execution. It provides the tools for configuring and managing clusters, nodes, and logical sets of processors (*partitions*).

Alternatively, Sun HPC message-passing programs can be executed by third-party resource-management software, such as Sun N1 Grid Engine (N1GE) Version 6, Platform Computing LSF HPC Version 6.2, and OpenPBS Portable Batch System (PBS) 2.3.16 and Altair PBS Professional 7.1.

CHAPTER 3

Choosing Your Programming Model and Hardware

This chapter outlines some points to consider in planning how to develop or port an HPC application. It provides a high-level overview of how to compare and assess programming models for use on Sun parallel hardware.

Starting Out

The first step in developing a high-performance application is to settle upon your basic approach. To make the best choice among the Sun HPC tools and techniques, you need to:

- Set goals for program performance and scalability
- Determine the amount of time and effort you can invest
- Select a programming model
- Assess the available computing resources

There are two common models of parallel programming in high performance computing: shared-memory programming and distributed-memory programming. These models are supported on Sun hardware with Sun compilers and with Sun HPC ClusterTools software, respectively. Issues in choosing between the models might include existing source-code base, available software development resources, desired scalability, and target hardware.

The basic Sun HPC ClusterTools programming model is distributed-memory message passing. Such a program executes as a collection of Solaris OS processes with separate address spaces. The processes compute independently, each on its own local data, and share data only through explicit calls to message-passing routines.

You might choose to use this model regardless of your target hardware. That is, you might run a message-passing program on an SMP cluster or run it entirely on a single, large SMP server. Or, you might choose to forgo ClusterTools software entirely and use only multithreaded parallelism, running on a single SMP server. It is also possible to combine the two approaches.

Programming Models

A high-performance application will almost certainly be parallel, but parallelism comes in many forms. The form you choose depends partly on your target hardware (server versus cluster) and partly on the time you have to invest.

Sun provides development tools for several widely used HPC programming models. These products are categorized by memory model: Sun Studio developer tools for shared-memory programming and Sun HPC ClusterTools for distributed-memory programming.

Shared memory means that all parts of a program can access one another's data freely. This might be because they share a common address space, which is the case with multiple threads of control within a single process. Or, it might result from employing a software mechanism for sharing memory.

Parallelism that is generated by the Sun compilers or is programmed using Solaris or POSIX threads requires a shared address space running on a single Solaris image.

Distributed memory means that multiple processes exchange data only through explicit message passing.

Message-passing programs, where the programmer inserts calls to the MPI library, are the only programs that can run across a cluster of computers. They can also, of course, run on a single computer or even on a serial processor.

TABLE 3-1 summarizes these two product suites.

TABLE 3-1 Comparison of Sun Compiler Suite and Sun HPC ClusterTools Software

	Sun Studio Compiler Collection	**Sun HPC ClusterTools Suite**
Target hardware	Any Sun workstation or SMP	Any Sun workstation, SMP, or cluster

TABLE 3-1 Comparison of Sun Compiler Suite and Sun HPC ClusterTools Software

	Sun Studio Compiler Collection	**Sun HPC ClusterTools Suite**
Memory model	Shared memory	Distributed memory
Runtime resource manager	Solaris OS	CRE (Cluster Runtime Environment) or third-party product
Parallel execution	Multithreaded	Multiprocess with message passing

Thus, available hardware does not necessarily dictate programming model. A message-passing program can run on any configuration, and a multithreaded program can run on a parallel server (SMP). The only constraint is that a program without message passing cannot run on a cluster.

The choice of programming model usually depends more on software preferences and available development time. Only when your performance goals demand the combined resources of a cluster of servers is the message-passing model necessarily required.

A closer look at the differences between shared-memory model and the distributed memory model as they pertain to parallelism reveals some other factors in the choice. The differences are summarized in TABLE 3-2.

TABLE 3-2 Comparison of Shared-Memory and Distributed-Memory Parallelism

	Shared Memory	**Distributed Memory**
Parallelization unit	Loop	Data structure
Compiler-generated parallelism	Available in Fortran 77, Fortran 90, and C via compiler options, directives/pragmas, and OpenMP	No established solution; options include HPF, split-C, UPC, OpenMP compiled for distributed memory, Co-Array Fortran, and various research projects. None of these are part of the Sun HPC ClusterTools software suite.
Explicit (hand-coded) parallelism	C/C++ and threads (Solaris or POSIX)	Calls to MPI library routines from Fortran 77, Fortran 90, C, or C++

Note – Nonuniform memory architecture (NUMA) is starting to blur the lines between shared- and distributed-memory architectures. That is, the architecture functions as shared memory, but typically the difference in cost between local and remote memory accesses is so great that it might be desirable to manage data locality explicitly. One way to do this is to use message passing.

Even without a detailed look, it is obvious that more parallelism is available with less investment of effort in the shared-memory model.

To illustrate the difference, consider a simple program that adds the values of an array (a global sum). In serial Fortran, the code is:

```
REAL A(N), X
X = 0.
DO I = 1, N
   X = X + A(I)
END DO
```

Compiler-generated parallelism requires little change. In fact, the compiler might well parallelize this simple example automatically. At most, the programmer might need to add a single directive:

```
      REAL A(N), X
      X = 0.

C$OMP DO REDUCTION(+:X)

      DO I = 1, N
         X = X + A(I)
      END DO
```

To perform this operation with an MPI program, the programmer needs to parallelize the data structure as well as the computational loop. The program would look like this:

```
      REAL A(NLOCAL), X, XTMP

      XTMP = 0.
      DO I = 1, NLOCAL
         XTMP = XTMP + A(I)
      END DO
      CALL MPI_ALLREDUCE
     & (XTMP,X1,MPI_REAL,MPI_SUM,MPI_COMM_WORLD,IERR)
```

When this program executes, each process can access only its own (*local*) share of the data array. Explicit message passing is used to combine the results of the multiple concurrent processes.

Clearly, message passing requires more programming effort than shared-memory parallel programming. But this is only one of several factors to consider in choosing a programming model. The trade-off for the increased effort can be a significant increase in performance and scalability.

In choosing your programming model, consider the following factors:

- If you are updating an existing code, what programming model does it use? In some cases, it is reasonable to migrate from one model to another, but this is rarely easy. For example, to go from shared memory to distributed memory, you must parallelize the data structures and redistribute them throughout the entire source code.
- What time investment are you willing to make? Compiler-based multithreading (using Sun Studio developer tools) might allow you to port or develop a program in less time than explicit message passing would require.
- What is your performance requirement? Is it within or beyond the computing capability associated with a single, uniform memory? Sun SMP servers can be very large—up to 106 processors and 576 Gbytes of memory in the current generation. For other purposes, a cluster—and thus distributed-memory programming—will be required.
- Is your performance requirement (including problem size) likely to increase in the future? If so, it might be worth choosing the message-passing model even if a single server meets your current needs. You can then migrate easily to a cluster in the future. In the meantime, the application might run faster than a shared-memory program on a single SMP because of the MPI discipline of enforcing data locality.

Mixing models is generally possible, but not common.

Scalability

A part of setting your performance goals is to consider how your application will scale.

The primary purpose of message-passing programming is to introduce explicit data decomposition and communication into an application, so that it will scale to higher levels of performance with increased resources. The appeal of a cluster is that it increases the range of scalability: a potentially limitless amount of processing power might be applied to complex problems and huge data sets.

The degree of scalability you can realistically expect is a function of the algorithm, the target hardware, and certain laws of scaling itself.

Amdahl's Law

Unfortunately, decomposing a problem among more and more processors ultimately reaches a point of diminishing returns. This idea is expressed in a formula known as Amdahl's Law.[1] Amdahl's Law assumes (quite reasonably) that a task has only some fraction `f` that is parallelizable, while the rest of the task is inherently serial. As the number of processors `NP` is increased, the execution time `T` for the task decreases as

```
T = (1-f) + f / NP
```

For example, consider the case in which 90 percent of the workload can be parallelized. That is, `f = 0.90`. The speedup as a function of the number of processors is shown in TABLE 3-3

TABLE 3-3 Speedup with Number of Processors

Processors (NP)	runtime (T)	Speedup (1/T)	Efficiency
1	1.000	1.0	100%
2	0.550	1.8	91%
3	0.400	2.5	83%
4	0.325	3.1	77%
6	0.250	4.0	67%
8	0.213	4.7	59%
16	0.156	6.4	40%
32	0.128	7.8	24%
64	0.114	8.8	14%

As the parallelizable part of the task is more and more subdivided, the non-parallel 10 percent of the program (in this example) begins to dominate. The maximum speedup achievable is only 10-fold, and the program can actually use only about three or four processors efficiently.

1. G.M. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities. In AFIPS Conference Proceedings, vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485.

Keep Amdahl's Law in mind when you target a performance level or run prototypes on smaller sets of CPUs than your production target. In the preceding example, if you had started measuring scalability on only two processors, the 1.8-fold speedup would have seemed admirable, but it is actually an indication that scalability beyond that might be quite limited.

In another respect, the scalability story is even worse than Amdahl's Law suggests. As the number of processors increases, so does the overhead of parallelization. Such overhead might include communication costs or interprocessor synchronization. So, observation will typically show that adding more processors will ultimately cause not just diminishing returns but negative returns: eventually, execution time might increase with added resources.

Still, the news is not all bad. With the high-speed interconnects within and between nodes and with the programming techniques described in this manual, your application might well achieve high, and perhaps near linear, speedups for some number of processors. And, in certain situations, you might even achieve superlinear scalability, because adding processors to a problem also provides a greater aggregate cache.

Scaling Laws of Algorithms

Amdahl's Law assumes that the work done by a program is either serial or parallelizable. In fact, an important factor for distributed-memory programming that Amdahl's Law neglects is communication costs. Communication costs increase as the problem size increases, although their overall impact depends on how this term scales vis-a-vis the computational workload.

When the local portion (the *subgrid*) of a decomposed data set is sufficiently large, local computation can dominate the runtime and amortize the cost of interprocess communication. TABLE 3-4 shows examples of how computation and communication scale for various algorithms. In the table, `L` is the linear extent of a subgrid while `N` is the linear extent of the global array.

TABLE 3-4 Scaling of Computation and Communication Times for Selected Algorithms

Algorithm	Communication Type	Communication Count	Computation Count
2-dimensional stencil	nearest neighbor	L	L^2
3-dimensional stencil	nearest neighbor	L^2	L^3
matrix multiply	nearest neighbor	N^2	N^3
multidimensional FFT	all-to-all	N	N log(N)

With a sufficiently large subgrid, the relative cost of communication can be lowered for most algorithms.

The actual speed-up curve depends also on cluster interconnect speed. If a problem involves many interprocess data transfers over a relatively slow network interconnect, the increased communication costs of a high process count might exceed the performance benefits of parallelization. In such cases, performance might be better with fewer processes collocated on a single SMP. With a faster interconnect, on the other hand, you might see even superlinear scalability with increased process counts because of the larger cache sizes available.

Characterizing Platforms

To set reasonable performance goals, and perhaps to choose among available sets of computing resources, you need to be able to assess the performance characteristics of hardware platforms.

The most basic picture of message-passing performance is built on two parameters: *latency* and *bandwidth*. These parameters are commonly cited for point-to-point message passing, that is, simple sends and receives.

- Latency is the time required to send a null-length message.
- Bandwidth is the rate at which very long messages are sent.

In this somewhat simplified model, the time required for passing a message between two processes is

```
time = latency + message-size / bandwidth
```

Obviously, short messages are latency-bound and long messages are bandwidth-bound. The crossover message size between the two is given as

```
crossover-size = latency x bandwidth
```

Another performance parameter is *bisection bandwidth*, which is a measure of the aggregate bandwidth a system can deliver to communication-intensive applications that exhibit little data locality. Bisection bandwidth might not be related to point-to-point bandwidth, because the performance of the system can degrade under load (many active processes) or since multiple CPUs are required to take advantage of the interconnect.

To suggest orders of magnitude, TABLE 3-5 shows sample values of these parameters for two Sun HPC platforms: a large SMP and a 64-node cluster.

TABLE 3-5 Sample Performance Values for MPI Operations on Two Sun Platforms

Platform	**Latency (microseconds)**	**Bandwidth (Mbyte/sec)**	**Crossover size = lat x bw (bytes)**	**Platform Bisection bandwidth (Mbyte/sec)**
SMP Enterprise 10000 server	~ 2	~ 200	~ 400	~ 2500
Cluster of 64 nodes connected with TCP network	~ 150	~ 40	~ 6000	~ 2000

The best performance is likely to come from a single server. With Sun servers, this means up to 72 CPUs.

For clusters, these values indicate that the TCP cluster is latency-bound. A smaller cluster using a faster interconnect would be less so. On the other hand, many nodes are needed to match the bisection bandwidth of single node.

Basic Hardware Factors

Typically, you work with a fixed set of hardware factors: your system is what it is. From time to time, however, hardware choices might be available, and, in any case, you need to understand the ways in which your system affects program performance. This section describes a number of basic hardware factors.

Processor speed is directly related to the peak floating-point performance a processor can attain. Because an UltraSPARC processor can execute up to one floating-point add and one floating-point multiply per cycle, peak floating-point performance is twice the processor clock speed. For example, a 1.8 GHz processor would have a peak floating-point performance of 3600 Mflops. In practice, achieved floating-point performance will be less, due to imbalances of additions and multiplies and the necessity of retrieving data from memory rather than cache. Nevertheless, some number of floating-point intensive operations, such as the matrix multiplies that provide the foundation for much of dense linear algebra, can achieve a high fraction of peak performance, and typically increasing processor speed has a positive impact on most performance metrics.

Large L2 (or external) caches can also be important for good performance. While it is desirable to keep data accesses confined to L1 (or on-chip) caches, UltraSPARC processors run quite efficiently from L2 caches as well. When you go beyond L2 cache to memory, however, the drop in performance can be significant. Indeed, though Amdahl's Law and other considerations suggest that performance should scale at best linearly with processor counts, many applications see a range of superlinear scaling, because an increase in processor count implies an increase in aggregate L2 cache size.

The *number of processors* is, of course, a basic factor in performance because more processors deliver potentially more performance. Naturally, it is not always possible to utilize many processors efficiently, but it is vital that enough processors be present. This means not only that there should be one processor per MPI process, but ideally there should also be a few extra processors per node to handle system daemons and other services.

System speed is an important determinant of performance for memory-access-bound applications. For example, if a code goes often out of its caches, then it might well perform better on 300-MHz processors with a 100-MHz system clock than on 333-MHz processors with an 83-MHz system clock. Similarly, performance speedup from 900-MHz processors to 1200-MHz processors, both with a 150-MHz system clock, is likely to be less than the 4/3 factor suggested by the processor speedup, since the memory is at the same speed in both cases.

Memory latency is influenced not only by memory clock speed, but also by system architecture. As a rule, as the maximum size of an architecture expands, memory latency goes up. Hence, applications or workloads that do not require much interprocess communication might well perform better on a cluster of 4-CPU workgroup servers than on a 64-CPU Sun E25K server.

Memory bandwidth is the rate at which large amounts of data can be moved between CPU and memory. It is related not only to cacheline size (amount of data moved per memory operation) and memory latency (amount of time to move one cacheline), but also to the system's ability to prefetch data and have multiple outstanding memory operations at any time.

Memory size is required to support large applications efficiently. While the Solaris OS will run applications even when there is insufficient physical memory, using virtual memory will degrade performance dramatically.

When many processes run on a single node, the *backplane bandwidth* of the node becomes an issue. Large Sun servers scale very well with high processor counts, but MPI applications can nonetheless tax backplane capabilities either due to local memory operations (within an MPI process) or due to interprocess communications via shared memory. MPI processes located on the same node exchange data by copying into and then out of shared memory. Each copy entails two memory operations: a load and a store. Thus, a two-sided MPI data transfer undergoes four memory operations.

On a 24-CPU Sun Fire 6800 server, with a 9.6-Gbyte/s backplane, this means that a large MPI all-to-all operation can run at about 2.4 Gbyte/s aggregate bandwidth. Here, MPI bandwidth is the rate at which bytes are sent.

For cluster performance, the *interconnect* between nodes is typically characterized by its latency and bandwidth. Choices include any network that supports TCP, such as Gigabit Ethernet or Infiniband.

Importantly, there will often be wide gaps between the performance specifications of the raw network and what an MPI application will achieve in practice. Notably:

- Latency might be degraded by software layers, especially operating system interactions in the case of TCP message passing.
- Bandwidth might be degraded by the network interface (such as PCI).
- Bandwidth might further be degraded on a loss-prone network if data is dropped under load.

A cluster's bisection bandwidth might be limited by its switch or by the number of network interfaces that tap nodes into the network. In practice, the number of network interfaces is typically the bottleneck. Thus, increasing the number of nodes might or might not increase bisection bandwidth.

Other Factors

At other times, even other parameters enter the picture. Seemingly identical systems can result in different performance because of the tunable system parameters residing in `/etc/system`, the degree of memory interleaving in the system, mounting of file systems, and other issues that might be best understood with the help of your system administrator. Further, some transient conditions, such as the operating system's free-page list or virtual-to-physical page mappings, might introduce hard-to-understand performance issues.

For the most part, however, the performance of the underlying hardware is not as complicated an issue as this level of detail implies. As long as your performance goals are in line with your hardware's capabilities, the performance achieved will be dictated largely by the application itself. This manual helps you maximize that potential for MPI applications.

CHAPTER 4

Performance Programming

This chapter discusses approaches to consider when you are writing new message-passing programs.

The general rules of good programming apply to any code, serial or parallel. This chapter therefore focuses primarily on optimizing MPI interprocess communications and concludes with an extended example.

When you are working with legacy programs, you need to consider the costs of recoding in relation to the benefits.

General Good Programming

The general rules of good programming apply when your goal is to achieve top performance along with robustness and, perhaps, portability.

Clean Programming

The first rule of good performance programming is to employ “clean” programming. Good performance is more likely to stem from good algorithms than from clever “hacks.” While tweaking your code for improved performance might work well on one hardware platform, those very tweaks might be counterproductive when the same code is deployed on another platform. A clean source base is typically more useful than one laden with many small performance tweaks. Ideally, you should emphasize readability and maintenance throughout the code base. Use performance profiling to identify any hot spots, and then do low-level tuning to fix the hot spots.

One way to garner good performance while simplifying source code is to use library routines. Advanced algorithms and techniques are available to users simply by issuing calls to high-performance libraries. In certain cases, calls to routines from one library might be speeded up simply by relinking to a higher-performing library. The following table shows examples of selected operations and suggests how these operations might be speeded up,

Operations...	might be speeded up by...
BLAS routines	linking to Sun Performance Library software
Collective MPI operations	formulating in terms of MPI collectives and using Sun MPI

Optimizing Local Computation

The most dramatic impact on scalability in distributed-memory programs comes from optimizing the data decomposition and communication. Aside from parallelization issues, a great deal of performance enhancement can be achieved by optimizing local (on-node) computation. Common techniques include loop rewriting and cache blocking. Compilers can be leveraged by exploring compilation switches (see Chapter 6).

For the most part, the important topic of optimizing serial computation within a parallel program is omitted here. To learn more about this and other areas of performance optimization, consult *Techniques For Optimizing Applications: High Performance Computing*, by Rajat Garg and Ilya Shapov, Prentice-Hall, 2001, ISBN: 0-13-093476-3. That volume covers serial optimization and various parallelization models. It deals with programming, compilation, and runtime issues and provides numerous concrete examples.

Optimizing MPI Communications

The default behavior of Sun MPI accommodates many programming practices efficiently. Tuning environment variables at runtime can result in even better performance. However, best performance will typically stem from writing the best programs. This section describes good programming practices under the following headings:

- "Reducing Message Volume" on page 29
- "Reducing Serialization" on page 29
- "Load Balancing" on page 29
- "Synchronization" on page 30
- "Buffering" on page 31
- "Nonblocking Operations" on page 32
- "Polling" on page 33

- “Sun MPI Collectives” on page 34
- “Contiguous Data Types” on page 34

These topics are all interwoven. Clearly, reducing the number and volume of messages can reduce communication overheads, but such overheads are inherent to parallelization of serial computation. Serialization is one extreme of load balancing. Load imbalances manifest themselves as performance issues only because of synchronization. Synchronization, in turn, can be mitigated with message buffering, nonblocking operations, or general polling.

Following the general discussion of these issues, this chapter illustrates them in a case study.

Reducing Message Volume

An obvious way to reduce message-passing costs is to reduce the amount of message passing. One method is to reduce the total amount of bytes sent among processes. Further, because a latency cost is associated with each message, aggregating short messages can also improve performance.

Reducing Serialization

Serialization can take many different forms. In multithreaded programming, contention for a lock might induce serialization. In multiprocess programming, serialization might be induced, for example, in I/O operations through a particular process that gathers or scatters data accordingly.

Serialization can also appear as operations that are replicated among all the processes.

Load Balancing

Generally, the impediment to great scalability is not as blatant as serialization, but simply a matter of poor work distribution or load balancing among the processes. A multiprocess job completes only when the process with the most work has finished.

More so than for multithreaded programming, load balancing is an issue in message-passing programming because work distribution or redistribution is expensive in terms of programming and communication costs.

Temporally or spatially localized load imbalances sometimes balance against one another. Imagine, for example, a weather simulation in which simulation of daytime weather typically is more computationally demanding than that of nighttime weather because of expensive radiation calculations. If different processes compute on different geographical domains, then over the course of a simulation day each process should see daytime and nighttime. Such circadian imbalances would average out.

As the degree of synchronization in the simulation is increased, however, the extent to which localized load imbalances degrade overall performance magnifies. In our weather example, this means that if MPI processes are synchronized many times over the course of a simulation day, then all processes will run at the slower, day-time rate, even if this forces night-time processes to sit idle at synchronization points.

Synchronization

The cost of interprocess synchronization is often overlooked. Indeed, the cost of interprocess communication is often due not so much to data movement as to synchronization. Further, if processes are highly synchronized, they tend to congest shared resources such as a network interface or SMP backplane, at certain times and leave those resources idle at other times. Sources of synchronization can include:

- `MPI_barrier` calls.
- Other MPI collective operations, such as `MPI_Bcast` and `MPI_Reduce`.
- Synchronous MPI point-to-point calls, such as `MPI_Ssend`.
- Implicitly synchronous transfers for messages that are large compared with the interprocess buffering resources.

  For example, the Sun MPI cyclic and rendezvous message-passing protocols induce extra synchronization between senders and receivers in order to reduce use of buffers. Use of such protocols and the size of internal buffering might be changed at runtime with Sun MPI environment variables, which are discussed in Chapter 7.

- Data dependencies, in which one process must wait for data that is being produced by another process.

  For example, a receiver must wait if it issues an `MPI_Recv` before its partner issues the corresponding `MPI_Send`.

Typically, synchronization should be minimized for best performance. You should:

- Generally reduce the number of message-passing calls.
- Specifically reduce the amount of explicit synchronization.
- Post sends as early as possible and receives as late as possible.
- Ensure sufficient system buffering.

If a send can be posted very early and the corresponding receive much later, then there would be no problem with data dependency, because the data would be available before it was needed. If internal system buffering is not provided to hold the in-transit message, however, the completion of the send will in some way become synchronized with the receive. This consideration brings up the topics of buffering and nonblocking operations.

Buffering

In most MPI point-to-point communication, for example, using `MPI_Send` and `MPI_Recv` calls, data starts in a user buffer on the sending process and ends up in a user buffer on the receiving process. In transit, that data might also be buffered internally multiple times by an MPI implementation.

There are performance consequences to such buffering. Among them:

- Some degree of synchronization might be induced between the sender and the receiver if the message exceeds the internal buffering that is available to it. That is, a send cannot complete before the correspond receive has been posted if there is nowhere else for the message to be stored.
- Data must be copied from one buffer to another. This is noteworthy, but typically not as important as synchronization effects.
- Buffers might have to be allocated and deallocated. Under most conditions, this is not important with Sun MPI.

The MPI standard does not require any particular amount of internal buffering for standard `MPI_Send` operations. Specifically, the standard warns against issuing too many `MPI_Send` calls without receiving any messages, as this can lead to deadlock. (See Example 3.9 in the MPI 1.1 standard.) MPI does, however, allow users to provide buffering with `MPI_Buffer_attach` and then to use such buffering with `MPI_Bsend` or `MPI_Ibsend` calls.

Sun MPI, as a particular implementation of the standard, allows users to increase internal buffering in two ways. One way, of course, is with the standard, portable `MPI_Buffer_attach` call. Another is with Sun MPI-specific runtime environment variables, as discussed in Chapter 7.

There are several drawbacks to using `MPI_Buffer_attach`. They stem from the fact that a buffered send copies data out of the user buffer into a hidden buffer and then issues a non-blocking send (like `MPI_Isend`) without giving the user access to the request handle. Non-blocking sends (like `MPI_Isend`) should be used in preference to buffered sends (like `MPI_Bsend`) because of these effects of buffered sends:

- Senders and receivers are not decoupled any more than with non-blocking sends.
- Another level of buffering and copying is involved.
- The status of the message cannot be queried (for instance, to determine when the hidden buffer allocated by `MPI_Buffer_attach` is free).
- The completion of the send cannot easily be forced.

Typically, performance will benefit more if internal buffering is increased by setting Sun MPI environment variables. This is discussed further in Chapter 7.

Sun MPI environment variables might not be a suitable solution in every case. For example, you might want finer control of buffering or a solution that is portable to other systems. (Beware that the MPI standard provides few, if any, performance portability guarantees.) In such cases, it might be preferable to using nonblocking `MPI_Isend` sends in place of buffered `MPI_Bsend` calls. The nonblocking calls give finer control over the buffers and better decouple the senders and receivers.

For best results:

- Do not assume unlimited internal buffering by Sun MPI.
- Use buffered calls, such as `MPI_Bsend` and the like, sparingly.
- Tune Sun MPI environment variables at runtime to increase system buffering.
- Use nonblocking calls such as `MPI_Isend` for finest control over user-specified buffering.
- Post nonblocking receives (like `MPI_Irecv`) early to relieve pressure on system buffers.

Other examples of internal MPI buffering include `MPI_Sendrecv_replace` calls and unexpected in-coming messages (that is, messages for which no receive operation has yet been posted).

Nonblocking Operations

The MPI standard offers blocking and nonblocking operations. For example, `MPI_Send` is a blocking send. This means that the call will not return until it is safe to reuse the specified send buffer. On the other hand, the call might well return before the message is received by the destination process.

Nonblocking operations enable you to make message passing concurrent with computation. Basically, a nonblocking operation might be initiated with one MPI call (such as `MPI_Isend`, `MPI_Start`, `MPI_Startall`, and so on) and completed with another (such as `MPI_Wait`, `MPI_Waitall`, and so on). Still other calls might be used to probe status, for example, `MPI_Test`.

Nonblocking operations might entail a few extra overheads. Indeed, use of a standard `MPI_Send` and `MPI_Recv` provides the best performance with Sun MPI for highly synchronized processes, such as in simple ping-pong tests. Generally, however, the benefits of nonblocking operations far outweigh their performance shortcomings.

The way these benefits derive, however, can be subtle. Though nonblocking communications are logically concurrent with user computation, they do not necessarily proceed in parallel. That is, typically, either computation or else communication is being affected at any instant by a CPU. How performance benefits derive from nonblocking communications is discussed further in the case study at the end of this chapter

To maximize the benefits of nonblocking operations:

- Replace blocking operations with nonblocking operations.
- Initiate nonblocking operations as soon as possible.
- Complete nonblocking operations as late as possible.
- Test the status of nonblocking operations periodically with `MPI_Test` calls.

Polling

Polling is the activity in which a process searches incoming connections for arriving messages whenever the user code enters an MPI call. Two extremes are:

- *General polling*, in which a process searches all connections, regardless of the MPI calls made in the user code. For example, an arriving message will be read if the user code enters an `MPI_Send()` call.
- *Directed polling*, in which a process searches only connections specified by the user code. For example, a message from process 3 will be left untouched by an `MPI_Recv()` call that expects a message from process 5.

General polling helps deplete system buffers, easing congestion and allowing senders to make the most progress. On the other hand, it requires receiver buffering of unexpected messages and imposes extra overhead for searching connections that might never have any data.

Directed polling focuses MPI on user-specified tasks and keeps MPI from rebuffering or otherwise unnecessarily handling messages the user code has not yet asked to receive. On the other hand, it does not aggressively deplete buffers, so improperly written codes might deadlock.

Thus, user code is most efficient when the following criteria are all met:

- Receives are posted in the same order as their sends.
- Collectives and point-to-point operations are interleaved in an orderly manner.
- Receives such as `MPI_Irecv()` are posted ahead of arrivals.
- Receives are specific and the program avoids `MPI_ANY_SOURCE`.
- Probe operations such as `MPI_Probe()` and `MPI_Iprobe()` are used sparingly.
- The Sun MPI environment variable `MPI_POLLALL` is set to 0 at runtime to suppress general polling.

Sun MPI Collectives

Collective operations, such as `MPI_Barrier()`, `MPI_Bcast()`, `MPI_Reduce()`, `MPI_Alltoall()`, and the like, are highly optimized in Sun MPI for UltraSPARC servers and clusters of servers. User codes can benefit from the use of collective operations, both to simplify programming and to benefit automatically from the optimizations, which include:

- Alternative algorithms depending on message size.
- Algorithms that exploit cheap on-node data transfers and minimize expensive internode transfers.
- Independent optimizations for shared-memory and internode components of algorithms.
- Sophisticated runtime selection of the optimal algorithm.
- Special optimizations to deal with hot spots within shared memory, whether cache lines or memory pages.

For Sun MPI programming, you need only keep in mind that the collective operations are optimized and that you should use them. The details of the optimizations used in Sun MPI to implement collective operations are available in Appendix A.

Contiguous Data Types

While interprocess data movement is considered expensive, data movement within a process can also be costly. For example, interprocess data movement via shared memory consists of two bulk transfers. Meanwhile, if data has to be packed at one end and unpacked at the other, then these steps entail just as much data motion, but the movement will be even more expensive because it is slow and fragmented.

You should consider:

- Using only contiguous data types.
- Sending a little unnecessary padding instead of trying to pack data that is only mildly fragmented.
- Incorporating special knowledge of the data types to pack data explicitly, rather than relying on the generalized routines `MPI_Pack()` and `MPI_Unpack()`.

Special Considerations for Message Passing Over TCP

Sun MPI supports message passing over any network that runs TCP. While TCP offers reliable data flow, it does so by retransmitting data as necessary. If the underlying network becomes loss-prone under load, TCP might retransmit a runaway volume of data, causing MPI performance to suffer.

For this reason, applications running over TCP might benefit from throttled communications. The following suggestions are likely to increase synchronization and degrade performance. Nonetheless, you might want to try these suggestions if the underlying network is losing too much data.

To throttle data transfers, you might:

- Avoid "hot receivers" (too many messages expected at a node at any time).
- Use blocking point-to-point communications (`MPI_Send()`, `MPI_Recv()`, and so on).
- Use synchronous sends (such as `MPI_Ssend()`).
- Use MPI collectives, such as `MPI_Alltoall()`, `MPI_Alltoallv()`, `MPI_Gather()`, or `MPI_Gatherv()`, as appropriate, because these routines account for loss-prone networks.
- Set the Sun MPI environment variable `MPI_EAGERONLY` to 0 at runtime and possibly lower `MPI_TCP_RENDVSIZE`, causing Sun MPI to use a rendezvous mode for TCP messages. See Appendix A and the *Sun MPI Programming and Reference Guide* for more details.

MPI Communications Case Study

The following examples illustrate many of the issues raised in the preceding conceptual discussion. These examples use a highly artificial test code to look at the performance of MPI communication and its interplay with computational load imbalance.

The main lessons to draw from this series of example are:

- A key performance metric in MPI programs is not the rate at which data is transferred, but the amount of idle time processes spend waiting to send or receive data. You should try to reduce the costs of interprocess synchronization that result from computational load imbalances.
- Sun MPI buffering, adjusted with environment variables, should be made sufficient for all messages that might be in transit at any one time.

- In the event that buffering is insufficient, use nonblocking operations, such as `MPI_Isend` and `MPI_Irecv`. This overlaps computation with communication. It does not overlap computation with data transfer, but it does help overlap computation with the wait times associated with communication.
- Post nonblocking operations such as `MPI_Isend` and `MPI_Irecv` as early as possible, and complete them with operations like `MPI_Waitall` as late as possible.
- In conjunction with nonblocking operations, `MPI_Testall` operations can be made during otherwise large computational blocks if there are messages in transit.

Algorithms Used

In these examples, each MPI process computes on some data and then circulates that data among the other processes in a ring pattern. That is, 0 sends to 1, 1 sends to 2, 2 sends to 3, and so on, with process np-1 sending to 0. An artificial load imbalance is induced in the computation.

The basic algorithm of this series of examples is illustrated in FIGURE 4-1 for four processes.

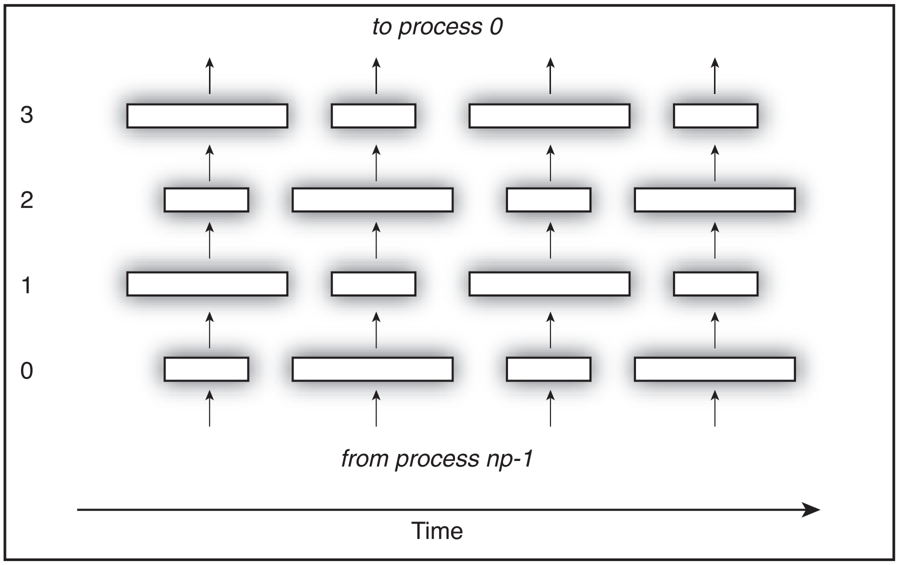


FIGURE 4-1 Basic Ring Sending Algorithm

In this figure, time advances to the right, and the processes are labeled vertically from 0 to 3. Processes compute, then pass data in a ring upward. There are temporal and spatial load imbalances, but in the end all processes have the same amount of work on average.

Even though the load imbalance in the basic algorithm averages out over time, performance degradation results if the communication operations are synchronized, as illustrated in FIGURE 4-2.

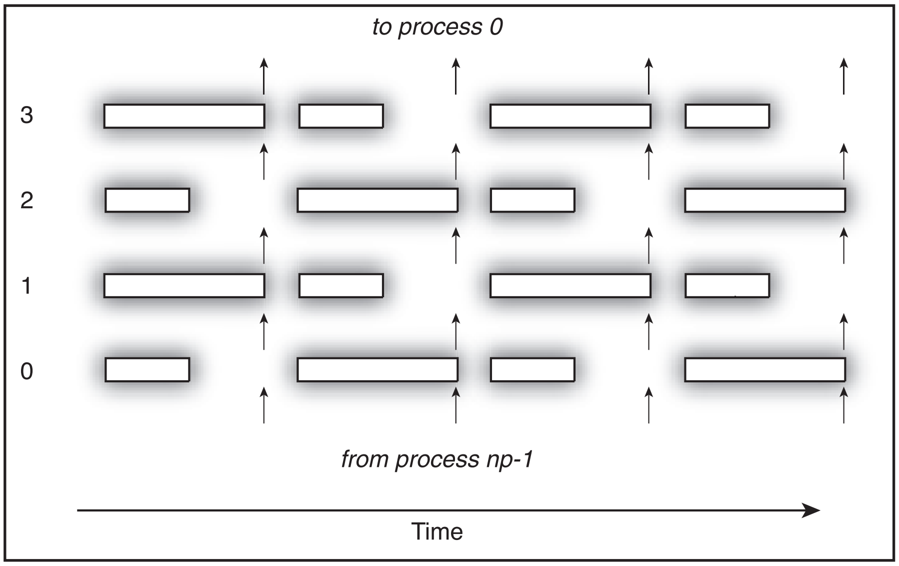


FIGURE 4-2 Basic Ring Sending Algorithm With Synchronization

Several variations on this basic algorithm are used in the following timing experiments, each of which is accompanied by a brief description.

Algorithm 1

- Phase 1: compute on buffer
- Phase 2: send buffer with `MPI_Send`
- Phase 3: receive buffer with `MPI_Recv`

This algorithm causes all processes to send data and then all to receive data. Because no process is receiving when they are all sending, the `MPI_Send` call must buffer the data to prevent code deadlock. This buffering requirement explicitly violates the MPI 1.1 standard. See Example 3.9, along with associated discussion, in the MPI 1.1 standard.

Nevertheless, Sun MPI can *progress* messages and avoid deadlock if the messages are sufficiently small or if the Sun MPI environment variable MPI_POLLALL is set to 1, which is the default. (See Appendix A for information on progressing messages.)

CODE EXAMPLE 4-1 Algorithm 1 Implemented in Fortran 90

```
subroutine compute(lda,n,x,ncompute,me,iup,idown,sum)
include 'mpif.h'
real(8) x(lda,*), sum

! phase 1
call compute_kernel(ncompute,n,x(:,1),sum)

! phase 2
call MPI_Send(x(:,1),n,MPI_REAL8,iup  ,1,MPI_COMM_WORLD,ier)

! phase 3
call MPI_Recv(x(:,1),n,MPI_REAL8,idown,1,MPI_COMM_WORLD,MPI_STATUS_IGNORE,ier)

end
```

The amount of computation to be performed in any iteration on any MPI process is dictated by the variable ncompute and is passed in by the parent subroutine. The array x is made multidimensional because subsequent algorithms will use multibuffering.

Algorithm 2

- Phase 1: compute on buffer
- Phase 2: perform communication with MPI_Sendrecv_replace

This algorithm should not deadlock on any compliant MPI implementation, but it entails unneeded overheads for extra buffering and data copying to “replace” the user data.

CODE EXAMPLE 4-2 Algorithm 2 Implemented in Fortran 90

```
subroutine compute(lda,n,x,ncompute,me,iup,idown,sum)
include 'mpif.h'
real(8) x(lda,*), sum

! phase 1
call compute_kernel(ncompute,n,x(:,1),sum)

! phase 2
call MPI_Sendrecv_replace(x(:,1),n,MPI_REAL8,iup  ,1, &
                                             idown,1, &
            MPI_COMM_WORLD,MPI_STATUS_IGNORE,ier)

end
```

Algorithm 3

- Phase 1: compute on buffer
- Phase 2: perform communication with `MPI_Sendrecv`

This algorithm removes the “replace” overheads by introducing double buffering.

CODE EXAMPLE 4-3 Algorithm 3 Implemented in Fortran 90

```
subroutine compute(lda,n,x,ncompute,me,iup,idown,sum)
include 'mpif.h'

real(8) :: x(lda,*), sum
integer ibufsend, ibufrecv
save    ibufsend, ibufrecv
data    ibufsend, ibufrecv / 1, 2 /

! phase 1
call compute_kernel(ncompute,n,x(:,ibufsend),sum)

! phase 2
call MPI_Sendrecv(x(:,ibufsend),n,MPI_REAL8,iup  ,1, &
                  x(:,ibufrecv),n,MPI_REAL8,idown,1, &
                  MPI_COMM_WORLD,MPI_STATUS_IGNORE,ier)

! toggle buffers
ibufsend = 3 - ibufsend
ibufrecv = 3 - ibufrecv

end
```

Algorithm 4

- Phase 1: nonblocking communication with `MPI_Isend` and `MPI_Irecv`
- Phase 2: compute on buffer
- Phase 3: `MPI_Waitall` to complete send and receive operations

This algorithm attempts to overlap communication with computation. That is, nonblocking communication is initiated before the computation is started, then the computation is performed, and finally the communication is completed. It employs three buffers: one for data being sent, another for data being received, and another for data used in computation.

Sun MPI does not actually overlap communication and computation, as the ensuing discussion makes clear. The real benefit of this approach is in decoupling processes for the case of computational load imbalance.

CODE EXAMPLE 4-4 Algorithm 4 Implemented in Fortran 90

```
subroutine compute(lda,n,x,ncompute,me,iup,idown,sum)

include 'mpif.h'

real(8) :: x(lda,*), sum
integer requests(2)

integer ibufsend, ibufrecv, ibufcomp
save    ibufsend, ibufrecv, ibufcomp
data    ibufsend, ibufrecv, ibufcomp / 1, 2, 3 /

! phase 1
call MPI_Isend &
  (x(:,ibufsend),n,MPI_REAL8,iup ,1,MPI_COMM_WORLD,requests(1),ier)
call MPI_Irecv &
  (x(:,ibufrecv),n,MPI_REAL8,idown,1,MPI_COMM_WORLD,requests(2),ier)

! phase 2
call compute_kernel(ncompute,n,x(:,ibufcomp),sum)

! phase 3
call MPI_Waitall(2,requests,MPI_STATUSES_IGNORE,ier)

! toggle buffers
ibuffree = ibufsend  ! send buffer is now free
ibufsend = ibufcomp  ! next, send what you just computed on
ibufcomp = ibufrecv  ! next, compute on what you just received
ibufrecv = ibuffree  ! use the free buffer to receive next

end
```

Algorithm 5

- Phase 1: nonblocking communication with `MPI_Isend` and `MPI_Irecv`
- Phase 2: compute on buffer, with frequent calls to `MPI_Testall`
- Phase 3: `MPI_Waitall` to complete send and receive operations

This algorithm is like Algorithm 4 except that it includes calls to `MPI_Testall` during computation. (The purpose of this is explained in "Use of `MPI_Testall`" on page 54.)

CODE EXAMPLE 4-5 Algorithm 5 Implemented in Fortran 90

```
subroutine compute(lda,n,x,ncompute,me,iup,idown,sum)
include 'mpif.h'

real(8) :: x(lda,*), sum
integer requests(2)
logical flag
integer ibufsend, ibufrecv, ibufcomp
save    ibufsend, ibufrecv, ibufcomp
data    ibufsend, ibufrecv, ibufcomp / 1, 2, 3 /
integer nblock0
save    nblock0
data    nblock0 / -1 /
character*20 nblock0_input
integer(4) iargc

! determine nblock0 first time through
if ( nblock0 .eq. -1 ) then
  nblock0 = 1024                   ! try 1024
  if ( iargc() .ge. 3 ) then       ! 3rd command-line argument overrides
    call getarg(3,nblock0_input)
    read(nblock0_input,*) nblock0
  endif
endif

! phase 1
call MPI_Isend &
  (x(:,ibufsend),n,MPI_REAL8,iup ,1,MPI_COMM_WORLD,requests(1),ier)
call MPI_Irecv &
  (x(:,ibufrecv),n,MPI_REAL8,idown,1,MPI_COMM_WORLD,requests(2),ier)

! phase 2
do i = 1, n, nblock0
  nblock = min(nblock0,n-i+1)
  call compute_kernel(ncompute,nblock,x(i,ibufcomp),sum)
  call MPI_Testall(2,requests,flag,MPI_STATUSES_IGNORE,ier)
end do

! phase 3
call MPI_Waitall(2,requests,MPI_STATUSES_IGNORE,ier)

! toggle buffers
ibuffree = ibufsend  ! send buffer is now free
ibufsend = ibufcomp  ! next, send what you just computed on
ibufcomp = ibufrecv  ! next, compute on what you just received
ibufrecv = ibuffree  ! use the free buffer to receive next

end
```

Making a Complete Program

To make a functioning example, one of the preceding subroutines should be combined with other source code and compiled using

% **mpf90 -fast** *source-files* **-lmpi**

CODE EXAMPLE 4-6 shows a sample Fortran 90 program that serves as the driver.

CODE EXAMPLE 4-6 Driver Program for Example Algorithms

```
program driver

include 'mpif.h'
character*20 arg
integer(4), parameter :: maxn = 500000
integer(4), parameter :: maxnbuffers = 3
integer(4) iargc
real(8) x(maxn,maxnbuffers), t

! initialize the buffers

x = 0.d0

! get the number of compute iterations from the command line

ncompute_A = 0
if ( iargc() .ge. 1 ) then
  call getarg(1,arg)
  read(arg,*) ncompute_A
endif

ncompute_B = ncompute_A
if ( iargc() .ge. 2 ) then
  call getarg(2,arg)
  read(arg,*) ncompute_B
endif

! initialize usual MPI stuff

call MPI_Init(ier)
call MPI_Comm_rank(MPI_COMM_WORLD, me, ier)
call MPI_Comm_size(MPI_COMM_WORLD, np, ier)
if ( mod(np,2) .ne. 0 ) then
  print *, "expect even number of processes"
  call MPI_Finalize(ier)
  stop
endif
```

```
! pump a lot of data through to warm up buffers
call warm_up_buffers(maxn,x)

! iterations

if ( me .eq. 0 ) write(6,'("    bytes/msg         sec/iter
Mbyte/sec")')
niter = 10
n = 0
do while ( n .le. maxn )

  ! make measurement and report
  call sub(maxn,n,x,niter,ncompute_A,ncompute_B,t)
  t = t / niter
  if ( me .eq. 0 ) write(6,'(i15,2f20.6)') 8 * n, t, 16.d-6 * n / t

  ! bump up n
  n = max( nint(1.2 * n), n + 1 )

enddo

! shut down

call MPI_Finalize(ier)

end

subroutine sub(lda,n,x,niter,ncompute_A,ncompute_B,t)

include 'mpif.h'
real(8) :: x(lda,*), sum, t

! figure basic MPI parameters
call MPI_Comm_rank(MPI_COMM_WORLD, me, ier)
call MPI_Comm_size(MPI_COMM_WORLD, np, ier)

! initialize sum
sum = 0.d0

! figure nearest neighbors
idown = me - 1
iup   = me + 1
if ( idown .lt.  0 ) idown = np - 1
if ( iup   .ge. np ) iup   = 0

! start timer
call MPI_Barrier(MPI_COMM_WORLD,ier)
t = MPI_Wtime()
```

```
! loop
do iter = 1, niter

  ! induce some load imbalance
  if ( iand(iter+me,1) .eq. 0 ) then
    ncompute = ncompute_A
  else
    ncompute = ncompute_B
  endif

  ! computation (includes communication)
  call compute(lda,n,x,ncompute,me,iup,idown,sum)

enddo

! stop timer
call MPI_Barrier(MPI_COMM_WORLD,ier)
t = MPI_Wtime() - t

! dummy check to keep compiler from optimizing all "computation"
away
if ( abs(sum) .lt. -1.d0 ) print *, "failed dummy check"

end


subroutine compute_kernel(ncomplexity,n,x,sum)
real(8) x(n), sum, t
if ( ncomplexity .eq. 0 ) return


! sweep over all data
do i = 1, n

  ! some elemental operation of particular complexity
  t = 1.d0
  do iloop = 1, ncomplexity
    t = t * x(i)
  enddo
  x(i) = t
  sum = sum + t

enddo

end
```

```
subroutine warm_up_buffers(n,x)
include 'mpif.h'
real(8) x(n,*), t

! pump a lot of data through to warm up buffers
! (ideally, use the same traffic pattern as in rest of code)
! (all this wouldn't be necessary if MPI_WARMUP were supported)

! usual MPI stuff
call MPI_Comm_rank(MPI_COMM_WORLD, me, ier)
call MPI_Comm_size(MPI_COMM_WORLD, np, ier)

! figure nearest neighbors
idown = me - 1
iup   = me + 1
if ( idown .lt.  0 ) idown = np - 1
if ( iup   .ge. np ) iup   = 0

! figure number of iterations
nMB   = 500                ! MB that should be enough to run through
all buffers
niter = nMB * 1024 * 1024  ! convert to byte
niter = niter / ( 8 * n )  ! convert to number of iterations

! iterate
if ( me .eq. 0 ) write(6,'("            Mbyte         Mbyte/sec")')
if ( me .eq. 0 ) write(6,'("            (to date)        (this
round)")')
do i = 1, niter
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t = MPI_Wtime()
  call MPI_Sendrecv(x(:,1),n,MPI_REAL8,iup  ,1, &
                    x(:,2),n,MPI_REAL8,idown,1, &
                    MPI_COMM_WORLD,MPI_STATUS_IGNORE,ier)
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t = MPI_Wtime() - t
  if ( me .eq. 0 ) write(6,'(2f20.6)') 8.d-6 * i * n, 8.d-6 * n / t
end do

end
```

Note the following features of the driver program in CODE EXAMPLE 4-6:

- *Load Imbalance*. The driver introduces an artificial computational load imbalance. On average, the computational load is balanced, that is, each process performs the same total amount of work as every other process. On any one iteration, however, the processes will have different amounts of work. In particular, on one iteration, every other process will do less work and the remaining processes will do more work. The processes switch roles on every iteration.

- *Multiple Buffering*. Some of the algorithms use multiple buffering. To keep the subroutine interfaces all the same, all the code examples support multiple buffers, even for the algorithms that do not use the additional buffers.
- *Bandwidth Reporting*. The code reports a Mbyte/sec bandwidth, but this figure also includes time for computation and is not, strictly speaking, just a measurement of communication performance.
- *Buffer Warmup*. The subroutine `warm_up_buffers` passes a series of messages to make sure that MPI internal buffers are touched and ready for fast reuse. Otherwise, spurious performance effects can result when particular buffers are used for the first time.

Timing Experiments With the Algorithms

You can construct a functioning test code by choosing one of the preceding algorithms and then compiling and linking it together with the driver code.

This section shows sample results for the various algorithms running as 4 MPI processes on a Sun Fire 6800 server with 900-MHz CPUs and a 8-Mbyte L2 cache. The command line for program execution is:

```
% mprun -np 4 a.out
```

Baseline Results

FIGURE 4-3 shows bandwidth as a function of message size for Algorithms 1 and 2.

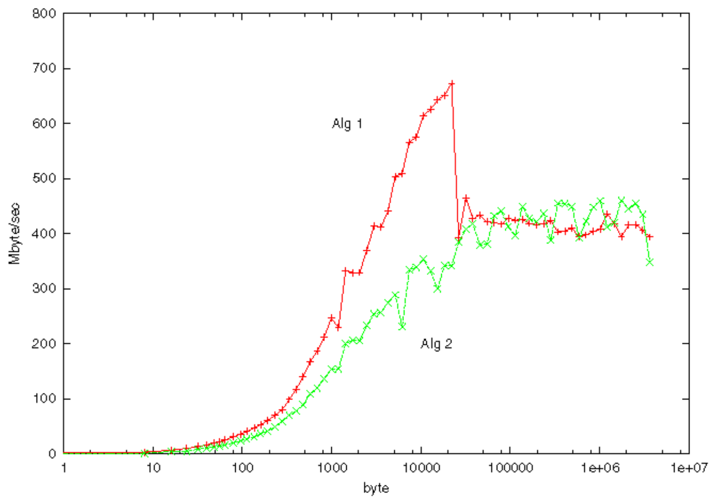


FIGURE 4-3 Bandwidth as a function of message size for Algorithms 1 and 2

Algorithm 2 proves to be slower than Algorithm 1. This is because `MPI_Sendrecv_replace` entails extra copying to “replace” data into a single buffer. Further, Sun MPI has special optimizations that cut particular overheads for standard `MPI_Send` and `MPI_Recv` calls, which is especially noticeable at small message sizes.

For Algorithm 1, the program reports about 700 Mbytes/s bandwidth for reasonably long messages. Above roughly 24 Kbyte, however, messages complete only due to general polling and the performance impact of processing unexpected messages can be seen in the figure.

Directed Polling

Now, let us rerun this experiment with directed polling. This is affected by turning general polling off:

```
% setenv MPI_POLLALL 0
% mprun -np 4 a.out
% unsetenv MPI_POLLALL
```

It is generally good practice to unset environment variables after each experiment so that settings do not persist inadvertently into subsequent experiments.

FIGURE 4-4 shows the resulting bandwidth as a function of message size.

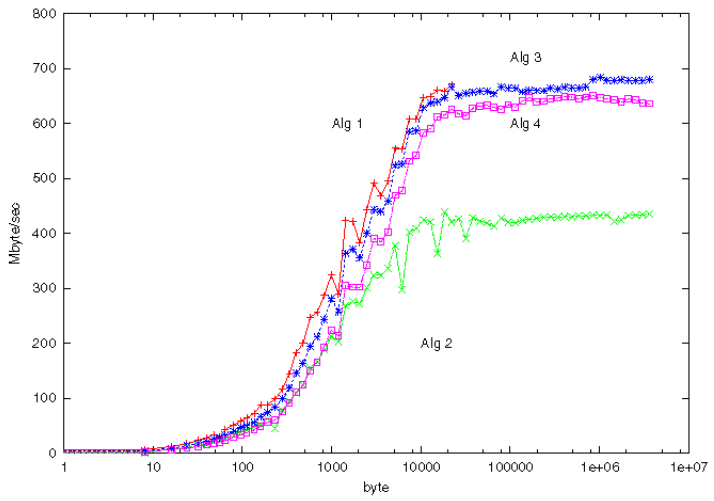


FIGURE 4-4 Bandwidth as a function of message size with directed polling

Although it is difficult to make a direct comparison with the previous figure, it is clear that direct polling has improved the bandwidth values across the range of message sizes. This is because directed polling leads to more efficient message passing. Time is not used up in searching all connections needlessly.

The highest bandwidth is delivered by Algorithm 1, but it deadlocks when the message size reaches 24 Kbytes. At this point, the standard send `MPI_Send` no longer is simply depositing its message into internal buffers and returning. Instead, the receiver is expected to start reading the message out of the buffers before the sender can continue. With general polling (see "Baseline Results" on page 48), processes drained the buffers even before receives were posted.

Algorithm 2 also benefits in performance from directed polling, and it provides an MPI-compliant way of passing the messages. That is, it proceeds deadlock-free even as the messages are made very large. Nevertheless, due to extra internal buffering and copying to effect the "replace" behavior of the `MPI_Sendrecv_replace` operation, this algorithm has the worst performance of the four.

Algorithm 3 employs `MPI_Sendrecv` and double buffering to eliminate the extra internal buffering and copying. Algorithm 4 employs nonblocking operations. These are fast and avoid deadlock..

Now, let us examine how Algorithm 4, employing nonblocking operations, fares once processes have drifted out of tight synchronization because of computational load imbalances.

Here, we run:

```
% setenv MPI_POLLALL 0
% mprun -np 4 a.out 1 200
% unsetenv MPI_POLLALL
```

The driver routine shown earlier (CODE EXAMPLE 4-6) picks up the command-line arguments (**`1 200`**) to induce an artificial load imbalance among the MPI processes.

CODE EXAMPLE 4-5 shows bandwidth as a function of message size when nonblocking operations are used.

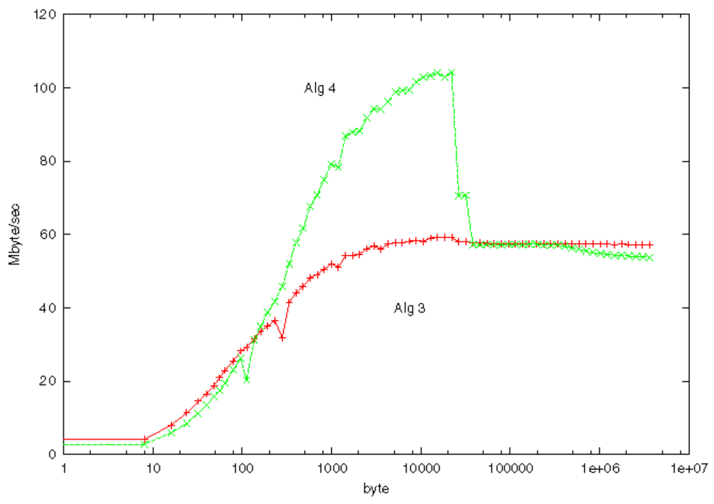


FIGURE 4-5 Bandwidth as a function of message size with nonblocking operations

While Algorithm 3 (`MPI_Sendrecv`) is comparable to Algorithm 4 (`MPI_Isend`, `MPI_Irecv`, `MPI_Waitall`) for synchronized processes, the nonblocking operations in Algorithm 4 offer the potential to decouple the processes and improve performance when there is a computational load imbalance.

The reported bandwidths are substantially decreased because they now include non-negligible computation times.

The internal buffering of Sun MPI becomes congested at longest message sizes, however, making the two algorithms perform equally. This behavior sets in at 24 Kbytes.

Increasing Sun MPI Internal Buffering

Twice now, we have seen that the internal buffering becomes congested at 24 Kbytes. This leads to deadlock in the case of Algorithm 1 and directed polling. It also leads to synchronization of processes with Algorithm 4, even though that algorithm employs nonblocking operations.

Note that Sun MPI has access to multiple protocol modules (PMs) to send messages over different hardware substrates. For example, two processes on the same SMP node exchange messages via the SHM (shared-memory) PM. If the two processes were on different nodes of a cluster interconnected by some commodity network, they would exchange messages via the TCP (standard Transmission Control Protocol) PM. The 24-Kbyte limit we are seeing is specific to the SHM PM.

However, the 24-Kbyte SHM PM buffers can become congested. Buffer sizes might be controlled with `MPI_SHM_CPOOLSIZE`, whose default value is 24576, or `MPI_SHM_SBPOOLSIZE`. More information about cyclic message passing and SHM PM buffers might be found in Appendix A. More information about the associated environment variables might be found in Appendix B.

Now, we rerun with:

```
% setenv MPI_POLLALL 0
% setenv MPI_SHM_SBPOOLSIZE 20000000
% setenv MPI_SHM_NUMPOSTBOX 2048
% mprun -np 4 a.out
% mprun -np 4 a.out 1 200
% unsetenv MPI_POLLALL
% unsetenv MPI_SHM_SBPOOLSIZE
% unsetenv MPI_SHM_NUMPOSTBOX
```

Here, we have set the environment variables not only to employ direct polling, but also to increase internal buffering.

FIGURE 4-6 shows bandwidth as a function of message size for the case of highly synchronized processes (the command line specifying a.out without additional arguments).

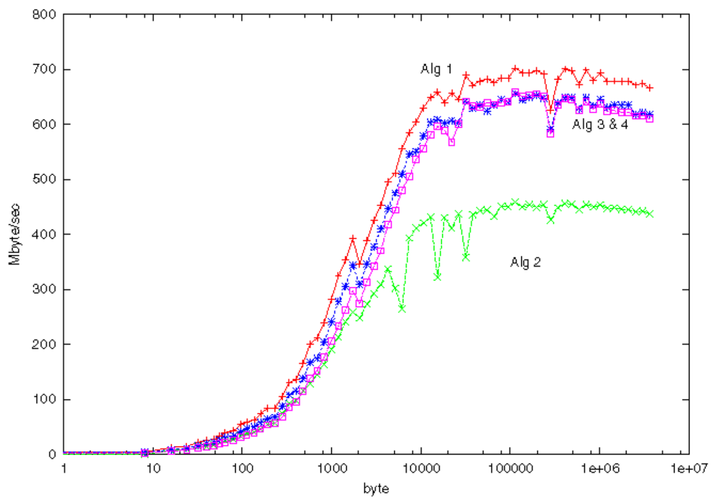


FIGURE 4-6 Bandwidth as a function of message size with highly synchronized processes

Having increased the buffering, we see that our illegal Algorithm 1 no longer deadlocks and is once again the performance leader. Strictly speaking, of course, it is still not MPI-compliant and its use remains nonrobust. Algorithm 2, using `MPI_Sendrecv_replace`, remains the slowest due to extra buffering and copying.

FIGURE 4-7 shows bandwidth as a function of message size for the case of load imbalance (the command line specifying a.out 1 200). Once a computational load imbalance is introduced, Algorithm 4, employing nonblocking operations, becomes the clear leader. All other algorithms are characterized by imbalanced processes advancing in lockstep.

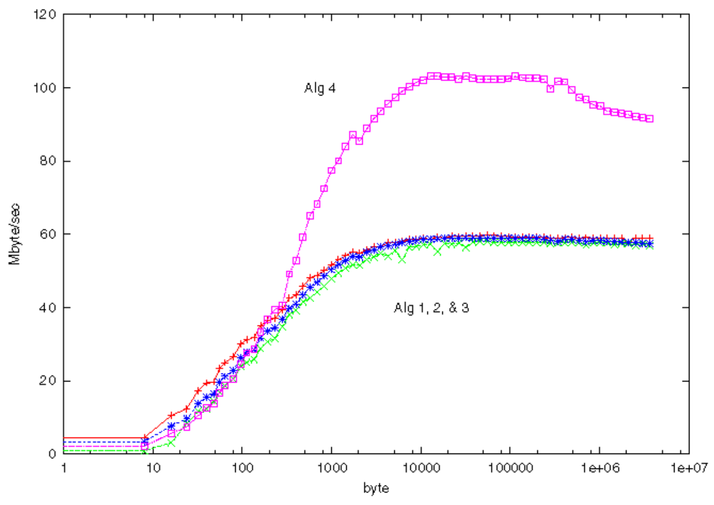


FIGURE 4-7 Bandwidth as a function of message size with load imbalance

Use of `MPI_Testall`

In some cases, for whatever reason, it is not possible to increase Sun MPI internal buffering sufficiently to hold all in-transit messages. For such cases, we can use Algorithm 5, which employs `MPI_Testall` calls to *progress* these messages. (For more information on progressing messages, see Progress Engine in Appendix A.)

Here, we run with:

```
% setenv MPI_POLLALL 0
% mprun -np 4 a.out 1 200     128
% mprun -np 4 a.out 1 200     256
% mprun -np 4 a.out 1 200     384
% mprun -np 4 a.out 1 200     512
% mprun -np 4 a.out 1 200    1024
% mprun -np 4 a.out 1 200    2048
% mprun -np 4 a.out 1 200    4096
% mprun -np 4 a.out 1 200    8192
% mprun -np 4 a.out 1 200   10240
% mprun -np 4 a.out 1 200   12288
% mprun -np 4 a.out 1 200   16384
% unsetenv MPI_POLLALL
```

The third command-line argument to `a.out` specifies, in some way, the amount of computation to be performed between `MPI_Testall` calls.

This is a slightly unorthodox use of `MPI_Testall`. The standard use of `MPI_Test` and its variants is to test whether specified messages have completed. The use of `MPI_Testall` here, however, is to progress all in-transit messages, whether specified in the call or not.

FIGURE 4-8 plots bandwidth against message size for the various frequencies of MPI_Testall calls.

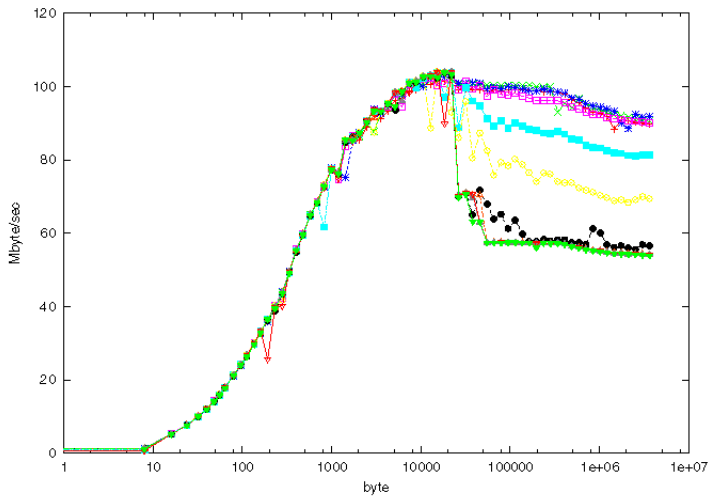


FIGURE 4-8 Bandwidth as a function of message size with `MPI_Testall` calls

There are too many curves to distinguish individually, but the point is clear. While performance used to dip at 24 Kbytes, introducing `MPI_Testall` calls in concert with nonblocking message-passing calls has maintained good throughput, even as messages grow to be orders of magnitude beyond the size of the internal buffering. Below the 24-Kbyte mark, of course, the `MPI_Testall` calls are not needed and do not impact performance materially.

Another view of the data is offered in FIGURE 4-9. This figure plots bandwidth as a function of the amount of computation performed between `MPI_Testall` calls.

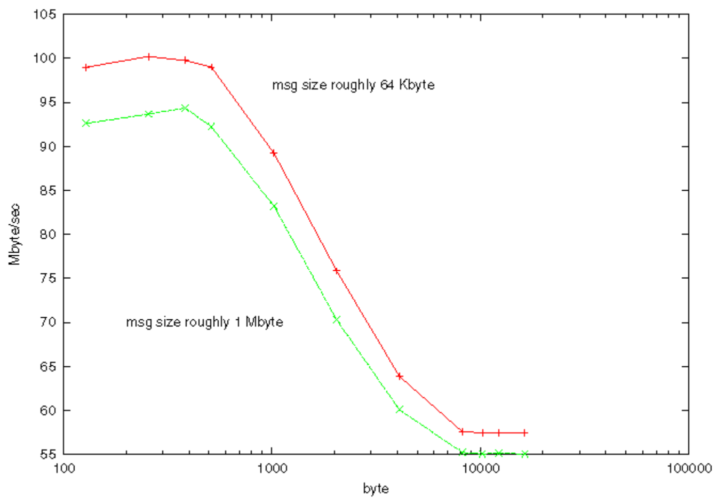


FIGURE 4-9 Bandwidth as a function of computation between `MPI_Testall` calls

For clarity, FIGURE 4-9 shows only two message sizes: 64 Kbyte and 1 Mbyte. We see that if too little computation is performed, then slight inefficiencies are introduced. More drastic is what happens when too much computation is attempted between `MPI_Testall` calls. Then, messages are not progressed sufficiently and long wait times lead to degraded performance.

To generalize, if `MPI_Testall` is called too often, it becomes ineffective at progressing messages. So, the optimal amount of computation between `MPI_Testall` calls should be large compared with the cost of an ineffective `MPI_Testall` call, which is on order of roughly 1 microsecond.

When `MPI_Testall` is called too seldom, interprocess synchronization can induce a severe degradation in performance. As a rule of thumb, the time it takes to fill or deplete MPI buffers sets the upper bound for how much computation to perform between `MPI_Testall` calls. These buffers are typically on order of tens of Kbytes, memory bandwidths are on order of hundreds of Mbyte/sec. Thus, the upper bound is some fraction of a millisecond.

These are rough rules of thumb, but they indicate that there is a wide range of nearly optimal frequencies for `MPI_Testall` calls.

Nevertheless, such techniques can be difficult to employ in practice. Challenges include restructuring communication and computation to post nonblocking sends and receives as early as possible while completing them as late as possible and injecting progress-inducing calls effectively.

CHAPTER 5

One-Sided Communication

This chapter describes performance issues related to MPI-2 standard one-sided communication:

Introducing One-Sided Communication

The most common use of MPI calls is for two-sided communication. That is, if data moves from one process address space to another, the data movement has to be specified on both sides: the sender's and the receiver's. For example, on the sender's side, it is common to use `MPI_Send()` or `MPI_Isend()` calls. On the receiver's side, it is common to use `MPI_Recv()` or `MPI_Irecv()` calls. An `MPI_Sendrecv()` call specifies both a send and a receive.

Even collective calls, such as `MPI_Bcast()`, `MPI_Reduce()`, and `MPI_Alltoall()`, require that every process that contributes or receives data must explicitly do so with the correct MPI call.

The MPI-2 standard introduces *one-sided* communication. Notably, `MPI_Put()` and `MPI_Get()` allow a process to access another process address space without any explicit participation in that communication operation by the remote process.

Comparing Two-Sided and One-Sided Communications

In selected circumstances, one-sided communication offers several advantages:

- *They can reduce synchronization and so improve performance.* Note that two-sided communication implies some degree of synchronization. For example, a receive operation cannot complete before the corresponding send has started.

  Further, because a sender cannot usually write directly into a receiver's address space, some intermediate buffer is likely to be used. If such buffering is small compared to the data volume, additional synchronization occurs whenever the sender must wait for the receiver to free up buffering. With one-sided communication, you must still specify synchronization to order memory operations, but you can amortize a single synchronization over many data movements.
- *They can reduce data movement.* As noted, two-sided communication sometimes introduces extra intermediate buffering, incurring extra data movement. An example of this would be an MPI program running on a large, shared-memory server. Interprocess communication usually depends upon having senders write to a shared-memory area and having receivers read from that area. This action requires twice as much data movement compared to the case where processes write directly to one another's address spaces. That extra data movement can affect aggregate backplane bandwidth, a critical resource in larger shared-memory servers.
- *They can simplify programming.* Note that information about a data transfer must be known and specified on only one side of the transfer, instead of two. This information is especially useful for dynamic, unstructured computations, where the traffic patterns change over the course of the computation.

Basic Sun MPI Performance Advice

Observe two principles to get good performance with one-sided communication with Sun MPI:

1. When creating a window (choosing what memory to make available for one-sided operations), use memory that has been allocated with the `MPI_Alloc_mem` routine. This suggestion in the MPI-2 standard benefits Sun MPI users:

   `http://www.mpi-forum.org/docs/mpi-20-html/node119.htm#Node119`

2. Use one-sided communication over the shared memory (SHM) protocol module. That is, use one-sided communication between MPI ranks that share the same shared-memory node. Protocol modules are chosen by Sun MPI at runtime and can be checked by setting the environment variable `MPI_SHOW_INTERFACES=2` before launching your MPI job.

Further one-sided performance considerations for Sun MPI are discussed in Appendix A and Appendix B.

Case Study: Matrix Transposition

This section illustrates some of the advantages of one-sided communication with a particular example: matrix transposition. While one-sided communication is probably best suited for dynamic, unstructured computations, matrix transposition offers a relatively simple illustration.

Matrix transposition refers to the exchange of the axes of a rectangular array, flipping the array about the main diagonal. For example, in FIGURE 5-1, after a transposition the array shown on the left side becomes the array shown on the right:

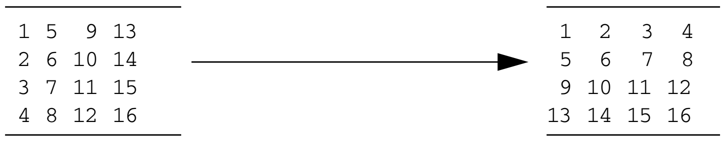
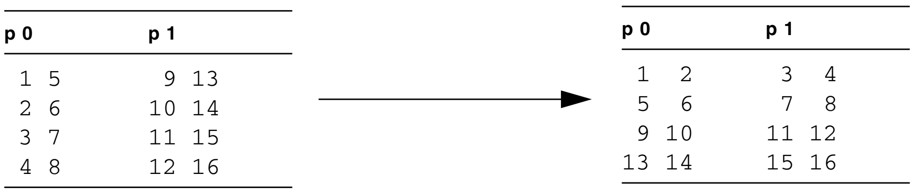


FIGURE 5-1 Matrix Transposition

There are many ways of distributing elements of a matrix over the address spaces of multiple MPI processes. Perhaps the simplest is to distribute one axis in a *block* fashion. For example, our example transposition might appear distributed over two MPI processes (process 0 (p 0), and process 1 (p 1)) such as in FIGURE 5-2:

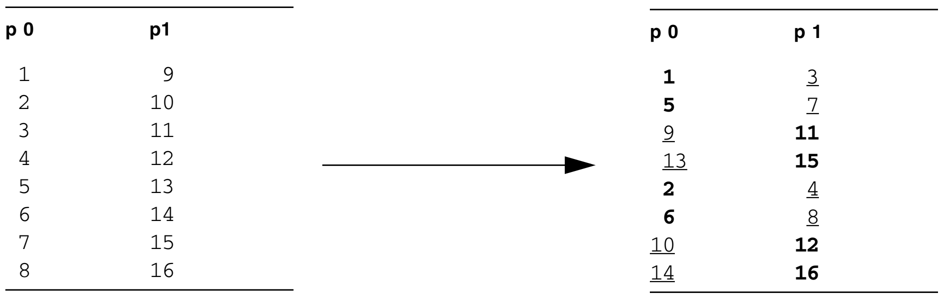


p 0		p 1	
1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

p 0		p 1	
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

FIGURE 5-2 Matrix Transposition, Distributed Over Two Processes

The ordering of multidimensional arrays within a linear address space varies from one programming language to another. A Fortran programmer, though, might think of the preceding matrix transposition as looking like FIGURE 5-3:



p 0	p1
1	9
2	10
3	11
4	12
5	13
6	14
7	15
8	16

p 0	p 1
1	$\underline{3}$
5	$\underline{7}$
$\underline{9}$	**11**
$\underline{13}$	**15**
2	$\underline{4}$
6	$\underline{8}$
$\underline{10}$	**12**
$\underline{14}$	**16**

FIGURE 5-3 Matrix Transposition, Distributed Over Two Processes (Fortran Perspective)

In the final matrix, as shown in FIGURE 5-4, elements that have stayed on the same process show up in bold font, while elements that have moved between processes are underlined.

These transpositions move data between MPI processes. Note that no two matrix elements that start out contiguous end up contiguous. There are several strategies to **effect**ing the interprocess data movement:

- Move one matrix element at a time. This strategy tends to introduce a lot of overhead and, therefore, performance penalties.
- Aggregate data locally within each MPI process, then move big blocks of matrix elements between processes, and finally rearrange data locally within each MPI process afterwards. This method makes the interprocess data movement relatively efficient, but entails a great deal of local data movement before and after the interprocess step.

- Move intermediate-size blocks of data between processes and do a moderate amount of local data rearrangement. This in-between approach is possible because there is some contiguity among data elements that travel between common MPI processes.

FIGURE 5-4 is an example of maximal aggregation for our 4x4 transposition:

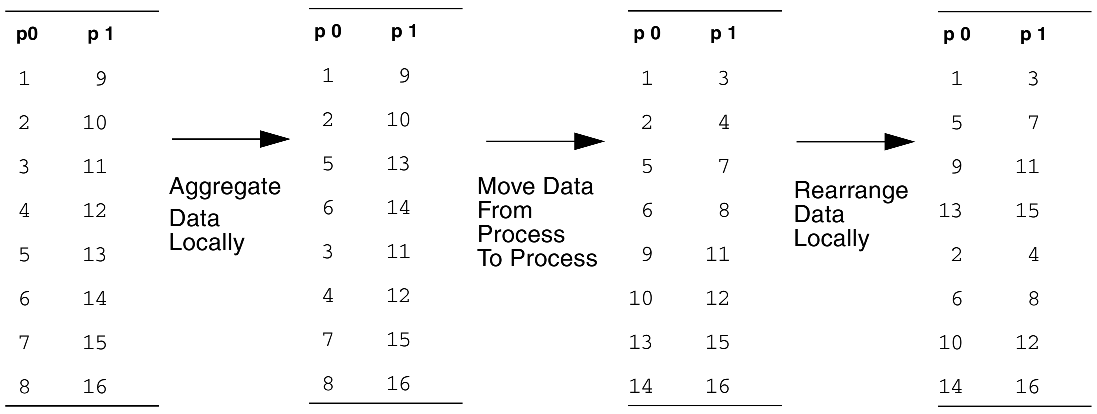


FIGURE 5-4 Matrix Transposition, Maximal Aggregation for 4X4 Transposition

Test Program A

Program A, shown in CODE EXAMPLE 5-1:

- Aggregates data locally.
- Establishes interprocess communication using a two-sided collective call `MPI_Alltoall()`.
- Rearranges data locally using the call `DTRANS()`, the optimized transpose in the Sun Performance Library.

CODE EXAMPLE 5-1 Test Program A

```
include "mpif.h"

real(8), allocatable, dimension(:) :: a, b, c, d
real(8) t0, t1, t2, t3
```

CODE EXAMPLE 5-1 Test Program A *(Continued)*

```
! initialize parameters
call init(me,np,n,nb)

! allocate matrices
allocate(a(nb*np*nb))
allocate(b(nb*nb*np))
allocate(c(nb*nb*np))
allocate(d(nb*np*nb))

! initialize matrix
call initialize_matrix(me,np,nb,a)

! timing
do itime = 1, 10
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t0 = MPI_Wtime()

  ! first local transpose
  do k = 1, nb
  do j = 0, np - 1
    ioffa = nb * (   j   + np * (k-1) )
    ioffb = nb * ( (k-1) + nb *   j   )
    do i = 1, nb
      b(i+ioffb) = a(i+ioffa)
    enddo
  enddo
  enddo

  t1 = MPI_Wtime()

  ! global all-to-all
call MPI_Alltoall(b, nb*nb, MPI_REAL8, &
                  c, nb*nb, MPI_REAL8, MPI_COMM_WORLD, ier)
  t2 = MPI_Wtime()

  ! second local transpose
  call dtrans('o', 1.d0, c, nb, nb*np, d)
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t3 = MPI_Wtime()

if ( me .eq. 0 ) &
  write(6,'(f8.3," seconds; breakdown on proc 0 =  ",3f10.3)') &
  t3 - t0, t1 - t0, t2 - t1, t3 - t2
enddo

! check
call check_matrix(me,np,nb,d)
```

CODE EXAMPLE 5-1 Test Program A *(Continued)*

```
deallocate(a)
deallocate(b)
deallocate(c)
deallocate(d)

call MPI_Finalize(ier)
end
```

Test Program B

Program B, shown in CODE EXAMPLE 5-2:

- Aggregates data locally.
- Establishes interprocess communication using a series of one-sided `MPI_Put()` calls.
- Rearranges data locally using a `DTRANS()` call.

Test program B should outperform test program A because one-sided interprocess communication can write more directly to a remote process address space.

CODE EXAMPLE 5-2 Test Program B

```
include "mpif.h"

integer(kind=MPI_ADDRESS_KIND) nbytes
integer win
real(8) c(*)
pointer (cptr,c)

real(8), allocatable, dimension(:) :: a, b, d
real(8) t0, t1, t2, t3

! initialize parameters
call init(me,np,n,nb)
! allocate matrices
allocate(a(nb*np*nb))
allocate(b(nb*nb*np))
allocate(d(nb*np*nb))

nbytes = 8 * nb * nb * np
call MPI_Alloc_mem(nbytes, MPI_INFO_NULL, cptr, ier)
if ( ier .eq. MPI_ERR_NO_MEM  ) stop

! create window
call MPI_Win_create(c, nbytes, 1, MPI_INFO_NULL, MPI_COMM_WORLD, win, ier)
```

CODE EXAMPLE 5-2 Test Program B *(Continued)*

```
! initialize matrix
call initialize_matrix(me,np,nb,a)

! timing
do itime = 1, 10
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t0 = MPI_Wtime()

  ! first local transpose
  do k = 1, nb
  do j = 0, np - 1
    ioffa = nb * (    j    + np * (k-1) )
    ioffb = nb * ( (k-1) + nb *    j    )
    do i = 1, nb
      b(i+ioffb) = a(i+ioffa)
    enddo
  enddo
  enddo
  t1 = MPI_Wtime()

  ! global all-to-all
  call MPI_Win_fence(0, win, ier)
  do ip = 0, np - 1
    nbytes = 8 * nb * nb * me
    call MPI_Put(b(1+nb*nb*ip), nb*nb, MPI_REAL8, ip, nbytes, &
                              nb*nb, MPI_REAL8, win, ier)
  enddo
  call MPI_Win_fence(0, win, ier)
  t2 = MPI_Wtime()

  ! second local transpose
  call dtrans(‘o’, 1.d0, c, nb, nb*np, d)

  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t3 = MPI_Wtime()
if ( me .eq. 0 ) &
  write(6,’(f8.3,” seconds; breakdown on proc 0 = ”,3f10.3)’) &
  t3 - t0, t1 - t0, t2 - t1, t3 - t2

enddo

! check
call check_matrix(me,np,nb,d)

! deallocate matrices and stuff
call MPI_Win_free(win, ier)
```

CODE EXAMPLE 5-2 Test Program B *(Continued)*

```
deallocate(a)
deallocate(b)
deallocate(d)
call MPI_Free_mem(c, ier)

call MPI_Finalize(ier)
end
```

Test Program C

Program C, shown in CODE EXAMPLE 5-3:

- Performs no data aggregation before the interprocess communication.
- Establishes interprocess communication using numerous small `MPI_Put()` calls.
- Rearranges data locally using a `DTRANS()` call.

Test program C should outperform test program B because program C eliminates aggregation before the interprocess communication. Such a strategy would be more difficult to implement with two-sided communication, which would have to make trade-offs between programming complexity and increased interprocess synchronization.

CODE EXAMPLE 5-3 Test Program C

```
include “mpif.h”

integer(kind=MPI_ADDRESS_KIND) nbytes
integer win
real(8) c(*)
pointer (cptr,c)

real(8), allocatable, dimension(:) :: a, b, d
real(8) t0, t1, t2, t3

! initialize parameters
call init(me,np,n,nb)

! allocate matrices
allocate(a(nb*np*nb))
allocate(b(nb*nb*np))
allocate(d(nb*np*nb))

nbytes = 8 * nb * nb * np
call MPI_Alloc_mem(nbytes, MPI_INFO_NULL, cptr, ier)
if ( ier .eq. MPI_ERR_NO_MEM  ) stop
```

CODE EXAMPLE 5-3 Test Program C *(Continued)*

```
! create window
call MPI_Win_create(c, nbytes, 1, MPI_INFO_NULL, MPI_COMM_WORLD, win, ier)

! initialize matrix
call initialize_matrix(me,np,nb,a)

! timing
do itime = 1, 10
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t0 = MPI_Wtime()
  t1 = t0

  ! combined local transpose with global all-to-all
  call MPI_Win_fence(0, win, ier)
  do ip = 0, np - 1
  do ib = 0, nb - 1
    nbytes = 8 * nb * ( ib + nb * me )
    call MPI_Put(a(1+nb*ip+nb*np*ib), nb, MPI_REAL8, ip, nbytes, &
                                    nb, MPI_REAL8, win, ier)
  enddo
  enddo
  call MPI_Win_fence(0, win, ier)
  t2 = MPI_Wtime()

  ! second local transpose
  call dtrans('o', 1.d0, c, nb, nb*np, d)

  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t3 = MPI_Wtime()

  if ( me .eq. 0 ) &
    write(6,'(f8.3," seconds; breakdown on proc 0 = ",3f10.3)') &
    t3 - t0, t1 - t0, t2 - t1, t3 - t2
enddo

! check
call check_matrix(me,np,nb,d)

! deallocate matrices and stuff
call MPI_Win_free(win, ier)
deallocate(a)
deallocate(b)
deallocate(d)
call MPI_Free_mem(c, ier)
```

CODE EXAMPLE 5-3 Test Program C *(Continued)*

```
call MPI_Finalize(ier)
end
```

Test Program D

Program D, shown in CODE EXAMPLE 5-4:

- Performs no data aggregation before the interprocess communication.
- Establishes interprocess communication using very numerous small `MPI_Put()` calls.
- Rearranges no data after the interprocess communication call.

Test program D eliminates all local data movement before and after the interprocess step, but it is slow because it moves all the data one matrix element at a time.

CODE EXAMPLE 5-4 Test Program D

```
include “mpif.h”

integer(kind=MPI_ADDRESS_KIND) nbytes
integer win
real(8) c(*)
pointer (cptr,c)

real(8), allocatable, dimension(:) :: a, b, d
real(8) t0, t1, t2, t3

! initialize parameters
call init(me,np,n,nb)

! allocate matrices
allocate(a(nb*np*nb))
allocate(b(nb*nb*np))
allocate(d(nb*np*nb))

nbytes = 8 * nb * nb * np
call MPI_Alloc_mem(nbytes, MPI_INFO_NULL, cptr, ier)
if ( ier .eq. MPI_ERR_NO_MEM  ) stop

! create window
call MPI_Win_create(c, nbytes, 1, MPI_INFO_NULL, MPI_COMM_WORLD, win, ier)

! initialize matrix
call initialize_matrix(me,np,nb,a)
```

CODE EXAMPLE 5-4 Test Program D *(Continued)*

```
! timing
do itime = 1, 10
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t0 = MPI_Wtime()
  t1 = t0

  ! combined local transpose with global all-to-all
  call MPI_Win_fence(0, win, ier)
  do ip = 0, np - 1
  do ib = 0, nb - 1
  do jb = 0, nb - 1
    nbytes = 8 * ( ib + nb * ( me + np * jb ) )
    call MPI_Put(a(1+jb+nb*(ip+np*ib)), 1, MPI_REAL8, ip,nbytes, &
                                       1, MPI_REAL8, win, ier)
  enddo
  enddo
  enddo
  call MPI_Win_fence(0, win, ier)
  call MPI_Barrier(MPI_COMM_WORLD,ier)
  t2 = MPI_Wtime()
  t3 = t2

  if ( me .eq. 0 ) &
    write(6,'(f8.3," seconds; breakdown on proc 0 = ",3f10.3)') &
    t3 - t0, t1 - t0, t2 - t1, t3 - t2
enddo

! check
call check_matrix(me,np,nb,c)

! deallocate matrices and stuff
call MPI_Win_free(win, ier)
deallocate(a)
deallocate(b)
deallocate(d)
call MPI_Free_mem(c, ier)

call MPI_Finalize(ier)
end
```

Utility Routines

Test programs A, B, C, and D use the utility routines shown in CODE EXAMPLE 5-5, CODE EXAMPLE 5-6, and CODE EXAMPLE 5-7 to initialize parameters, initialize the test matrix, and check the transposition.

CODE EXAMPLE 5-5 The `init` Subroutine

```
subroutine init(me,np,n,nb)

include “mpif.h”

! usual MPI preamble
call MPI_Init(ier)
call MPI_Comm_rank(MPI_COMM_WORLD, me, ier)
call MPI_Comm_size(MPI_COMM_WORLD, np, ier)

! get matrix rank n
if ( me .eq. 0 ) then
  write(6,*) “matrix rank n?”
  read(5,*) n
  if ( mod(n,np) .ne. 0 ) then
    n = np * ( n / np )
    write(6,*) “specified matrix rank not a power of np =”, np
    write(6,*) “using n =”, n
  endif
endif
call MPI_Bcast(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ier)
nb = n / np

end
```

CODE EXAMPLE 5-6 The `initialize_matrix` Subroutine

```
subroutine initialize_matrix(me,np,nb,x)

real(8) x(*)

n = nb * np

do k = 1, nb
do j = 0, np - 1
do i = 1, nb
  l2 = k + nb * me
```

CODE EXAMPLE 5-6 The `initialize_matrix` Subroutine *(Continued)*

```
  l1 = i + nb * j
  x(i+nb*(j+np*(k-1))) = l1 + n * ( l2 - 1 )
enddo
enddo
enddo

end
```

CODE EXAMPLE 5-7 The check_matrix Subroutine

```
subroutine check_matrix(me,np,nb,x)

include "mpif.h"

real(8) x(*), error_local, error_global

n = nb * np

error_local = 0
do k = 1, nb
do j = 0, np - 1
do i = 1, nb
l2 = k + nb * me
  l1 = i + nb * j
  error_local = error_local + &
    abs( x(i+nb*(j+np*(k-1))) - ( l2 + n * ( l1 - 1 ) ) )
enddo
enddo
enddo
call MPI_Allreduce(error_local,error_global,1, &
  MPI_REAL8,MPI_SUM,MPI_COMM_WORLD,ier)
if ( me .eq. 0 ) write(6,*) "error:", error_global

end
```

Timing

Here are sample timings for the three factors on an (older-generation) Sun Enterprise 6000 server. An 8192x8192 matrix of 8-byte (64-bit) elements is transposed using 16 CPUs (that is, distributed over MPI processes).

Program	Total (seconds)	Local Aggregation	Interprocess Communication	Local `DTRANS()` Call
A	2.109	0.585	0.852	0.673
B	1.797	0.587	0.466	0.744
C	1.177	0.000	0.430	0.747
D	4.908	0.000	4.908	0.000

Note that test program B is twice as fast in interprocess communication as test program A. This increased speed is because two-sided communication on a shared-memory server, while efficient, still moves data into a shared-memory area and then out again. In contrast, one-sided communication moves data directly from the address space of one process into that of the other. The aggregate bandwidth of the interprocess communication step for the one-sided case is:

```
8192 x 8192 x 8 byte / 0.466 seconds = 1.1 Gbyte/second
```

which is very close to the maximum theoretical value supported by this server (half of 2.6 Gbyte/second).

Further, the cost of aggregating data before the interprocess communication step can be eliminated, as demonstrated in test program C. This change adds a modest amount of time to the interprocess communication step, presumably due to the increased number of `MPI_Put()` calls that must be made. The most important advantage of this technique is that it eliminates the task of balancing the extra programming complexity and interprocess synchronization that two-sided communication requires.

Test program D allows complete elimination of the two local steps, but at the cost of moving data between processes one element at a time. The huge increase in runtime is evident in the timings. The issue here is not just overheads in interprocess data movement, but also the prohibitive cost of accessing single elements in memory rather than entire cache lines.

CHAPTER 6

Compilation and Linking

This chapter describes the Sun compiler switches that typically give the best performance for Sun MPI programs:

For more detailed information on compilation, refer to the following:

- The documentation and man pages that accompany your compiler
- The man pages for the Sun HPC ClusterTools utilities `mpf90`, `mpcc`, and `mpCC`
- *Techniques For Optimizing Applications: High Performance Computing*, by Rajat Garg and Ilya Shapov, Prentice-Hall, 2001, ISBN: 0-13-093476-3

Compiler Version

The simplest way to get the best performance from a compiler and associated libraries is to use the latest available version. The Sun Studio Compiler Collection software releases 8, 9, 10, and 11 are supported for the Sun HPC ClusterTools 6 suite.

The `mp*` Utilities

Sun HPC ClusterTools programs can be written for and compiled by the Fortran 77, Fortran 90, C, or C++ compilers. Although you can invoke these compilers directly, you might prefer to use the convenience scripts `mpf77`, `mpf90`, `mpcc`, and `mpCC`, provided with Sun HPC ClusterTools software.

This chapter describes the basic compiler switches that typically give best performance. The discussion centers around the `mpf90` and `mpcc` scripts, but it applies equally to the various scripts and aliases just mentioned. For example, you can use:

```
% mpf90 -fast -xalias=actual -g a.f -lmpi
```

to compile a Fortran program that uses Sun MPI.

For more detailed information, refer to the *Sun HPC ClusterTools User's Guide*.

The `-fast` Switch

The single most useful compilation switch for performance, is `-fast`. This macro expands to settings that are appropriate for high performance for a general set of circumstances. Because its expansion varies from one compiler release to another, you might prefer to specify the underlying switches explicitly. To see what the `-fast` switch expands to in the current release, use the `-v` option with Fortran or the `-#` option with C for verbose compilation output.

Part of the `-fast` switch is `-xtarget=native`, which directs the compiler to try to produce optimal code for the platform on which compilation is taking place. If you compile on the same type of platform that you expect to run on, then this setting is appropriate. (A compile-time warning might remind you that the resulting binary will not be compatible with older processors.)

Otherwise, specify the target platform with the `-xtarget` switch. The compiler man page (`f90`, `cc`, or `CC`) gives the legal values of the `-xtarget` switch.The `-xtarget` macro then expands into appropriate values of the `-xarch`, `-xchip`, and `-xcache` switches. It might suffice simply to specify the target instruction set architecture with the `-xarch` switch, as discussed next.

If you compile with the `-fast` switch and link in a separate step, be sure to link with the `-fast` switch.

If a Fortran program makes calls to the Sun MPI library, all its objects must have been compiled with the `-dalign` switch. This requirement is automatically satisfied when you compile with the `-fast` switch.

The `-xarch` Switch

The second most important compiler switch for maximizing performance is `-xarch`. While the `-fast` switch picks many performance-oriented settings by default, you should specify a value for the `-xarch` switch if you are compiling for a processor type that is different from the compilation system. Further, if you want 64-bit addressing for large-memory applications, then the `-xarch` argument is required to specify the format of the executable.

- Specify `-xarch=v8plusa` for 32-bit object binaries for UltraSPARC II processors.
- Specify `-xarch=v9a` for 64-bit object binaries for UltraSPARC II processors.
- Specify `–xarch=v8plusb` for 32-bit binaries for UltraSPARC III processors.
- Specify `-xarch=v9b` for 64-bit binaries for UltraSPARC III processors.
- Specify `-xarch=v9` for 64-bit binaries for UltraSPARC IV and IV+ processors.
- Specify `-xarch=amd64` for 64-bit binaries for the AMD Opteron x64 processors.

Note when using the `-xarch` switch, object files in 64-bit format can be linked only with other object files in the same format.

The `-fast` switch should appear before the `-xarch` switch on the compile or link line, as shown in the examples in this chapter. If you compile with the `-xarch` switch and then link in a separate step, be sure to link with the same setting.

The `-xalias` Switch

Sun MPI programs compiled using the Sun Studio Compiler Collection, Fortran compiler should be compiled with `-xalias=actual`.

This recommendation arises because the MPI Fortran binding is inconsistent with the Fortran 90 standard in several respects. This is documented in the MPI 2 standard,

```
http://www-unix.mcs.anl.gov/mpi/mpi-standard/
mpi-report-2.0/node19.htm#Node19
```

Specifically, see the discussion of "A Problem with Register Optimization."

This recommendation applies to the use of high levels of compiler optimization. A highly optimizing Fortran compiler could break MPI codes that use nonblocking operations.

While failures are unlikely, they can occur. The failure modes can be varied and insidious:

- Silently incorrect answers
- Intermittent and mysterious floating-point exceptions
- Intermittent and mysterious hangs

The –g Switch

With most compilers, the –g switch is not thought of as a performance switch. On the contrary, the –g switch has traditionally inhibited compiler optimizations.

With the Sun compilers, however, there is virtually no loss of performance with this switch. Further, –g compilation enables source-code annotation by the Performance Analyzer, which provides important performance-tuning information. Thus, the –g switch might be considered one of the basic switches to use in performance-tuning work.

Other Useful Switches

Performance benefits from linking in the optimized math library. For Fortran, the –fast switch invokes –xlibmopt automatically. For C, be sure to add the –lmopt switch to your link line (as shown in the following example):

```
% mpcc -fast -g -o a.out a.c -lmpi -lmopt
```

Include the argument –xvector[=yes] if math library intrinsics, such as logarithm, exponentiation, or trigonometric functions, appear inside long loops. This will make calls to the optimized vector math library. If you compile with the –xvector[=yes] argument, then include this switch on your link line to link in the vector library. The –fast switch might already include –xvector for Fortran compilation, but not for C.

The use of data prefetch can help hide the cost of loading data from memory. Compile with the –xprefetch switch to enable compiler generation of prefetch instructions. The –fast switch (typically) already includes –xprefetch for Fortran compilation, but not for C. Sometimes, the `-xprefetch` switch can slow performance, so it might best be used selectively. For example, you can compile some files with the `-xprefetch[=yes]` argument and some with `-xprefetch=no`. Or, for even greater selectivity, annotate your source code with prefetch pragmas or directives. For more information, see the compiler user guides.

C programmers should consider using the `-xrestrict` switch, which causes the compiler to treat pointer-valued function parameters as restricted pointers. Other information about pointer aliasing can be provided to the compiler by using the argument `-xalias_level`. Refer to the *C User's Guide* for more details.

C programmers should also consider the switch `-xsfpconst` if they largely perform floating-point arithmetic to 32-bit precision. Note that in C, floating-point constants are treated as double-precision values unless they are explicitly declared as floats. For example, in the expression `a=1.0/b`, the constant is treated as a double precision value, regardless of the types of `a` and `b`. This condition might lead to unintended numeric conversions and other performance implications. You can rewrite the expression as `a=1.0f/b`. Alternatively, you can compile with the `-xsfpconst` switch to treat unsuffixed floating-point constants as single-precision quantities.

Fortran codes written so that the values of local variables are not needed for subsequent calls might benefit from the argument `-stackvar`.

CHAPTER 7

Runtime Considerations and Tuning

To understand runtime tuning, you need to understand what happens on your cluster at runtime—that is, how hardware characteristics can impact performance and what the current state of the system is.

This chapter discusses the performance implications of:

Running on a Dedicated System

The primary consideration in achieving maximum performance from an application at runtime is giving it dedicated access to the resources. Useful commands include:

	CRE	UNIX
How high is the load?	% **mpinfo -N**	% **uptime**
What is causing the load?	% **mpps -e**	% **ps -e**

The UNIX commands give information only for the node where the command is issued. The CRE commands return information for all nodes in a cluster.

CRE's `mpps` command shows only those processes running under the resource manager. For more complete information, try the UNIX `ps` command. For example, either

```
% /usr/ucb/ps augx
```

or

```
% /usr/bin/ps -e -o pcpu -o pid -o comm | sort -n
```

will list most busy processes for a particular node.

Note that small background loads can have a dramatic impact. For example, the `fsflush` daemon flushes memory periodically to disk. On a server with a lot of memory, the default behavior of this daemon might cause a background load of only about 0.2, representing a small fraction of 1 percent of the compute resource of a 64-way server. Nevertheless, if you attempted to run a “dedicated” 64-way parallel job on this server with tight synchronization among the processes, this background activity could potentially disrupt not only one CPU for 20 percent of the time, but in fact all CPUs, because MPI processes are often very tightly coupled. (For the particular case of the `fsflush` daemon, a system administrator should tune the behavior to be minimally disruptive for large-memory machines.)

In short, it is desirable to leave at least one CPU idle per cluster node. In any case, it is useful to realize that the activity of background daemons is potentially very disruptive to tightly coupled MPI programs.

Setting Sun MPI Environment Variables

Sun MPI uses a variety of techniques to deliver high-performance, robust, and memory-efficient message passing under a wide set of circumstances. In most cases, performance will be good without tuning any environment variables. In certain situations, however, applications will benefit from nondefault behaviors. The Sun MPI environment variables discussed in this section enable you to tune these default behaviors.

If you need a quick and approximate evaluation of your environment variable settings, you can skip this section entirely and rely on the MPProf profiling tool, described further in Chapter 8, to recommend Sun MPI environment variable settings based on collected profiling data.

For greater detail, more information is available in Appendix A and Appendix B.

Are You Running on a Dedicated System?

If your system’s capacity is sufficient for running your Sun MPI job, you can commit processors aggressively to your job. Your CPU load should not exceed the number of physical processors. Load is basically defined as the number of MPI processes in your job, but it can be greater if other jobs are running on the system or if your job is multithreaded. Load can be checked with the `uptime` or `mpinfo` command, as discussed at the beginning of this chapter.

To run more aggressively, use either of these settings:

- `% `**`setenv MPI_SPIN 1`**

  This setting causes Sun MPI to "spin" aggressively, regardless of whether it is doing any useful work. If you use this setting, you should leave at least one idle processor per node to service system daemons. If you intend to use all processors on a node, setting this aggressive spin behavior can slow performance, so some experimentation is needed.

- `% `**`setenv MPI_PROCBIND 1`**

  While the Solaris OS schedules processes in generally optimal ways, performance in a dedicated environment is sometimes improved by binding processes to processors. This can be effected by setting the `MPI_PROCBIND` variable to 1 (one). Detailed control over the binding can be achieved by listing specific processors for binding. See the MPI man page for more details on `MPI_PROCBIND`.

  Performance can deteriorate dramatically with `MPI_PROCBIND` if multiple processes are bound to the same processor or if the processes are multithreaded.

Does the Code Use System Buffers Safely?

In some MPI programs, processes send large volumes of data with blocking sends before starting to receive messages. The MPI standard specifies that users must explicitly provide buffering in such cases, such as by using `MPI_Bsend()` calls. In practice, however, some users rely on the standard send (`MPI_Send()`) to supply unlimited buffering. By default, Sun MPI prevents deadlock in such situations through general polling, which drains system buffers even when no receives have been posted by the user code.

For best performance on typical, safe programs, general polling should be suppressed by using the setting shown in the following example:

```
% setenv MPI_POLLALL 0
```

If deadlock results from this setting, you might nonetheless use the setting for best performance if you resolve the deadlock with increased buffering, as discussed in the next section.

Are You Willing to Trade Memory for Performance?

Messages traveling from one MPI process to another are staged in intermediate buffers, internal to Sun MPI. If this buffering is insufficient, senders can stall unnecessarily while receivers drain the buffers.

One alternative is to increase the internal buffering using Sun MPI environment variables. For example, try this setting before you run:

```
% setenv MPI_SHM_SBPOOLSIZE 8000000
% setenv MPI_SHM_NUMPOSTBOX 256
```

Another alternative is to run your program with the MPProf tool, which suggests environment variable settings if it detects internal buffer congestion. See Chapter 8 for more information on MPProf.

For a more detailed understanding of these environment variables, see Appendix A and Appendix B.

Do You Want to Initialize Sun MPI Resources?

Use of certain Sun MPI resources might be relatively expensive when they are first used. This use can disrupt performance profiles and timings. While it is best, in any case, to ensure that performance has reached a level of equilibrium before profiling starts, a Sun MPI environment variable might be set to move some degree of resource initialization to the `MPI_Init()` call. Use:

```
% setenv MPI_FULLCONNINIT 1
```

Note that this setting does *not* tend to improve overall performance. However, it might improve performance and enhance profiling in most MPI calls, while slowing down the `MPI_Init()` call. The initialization time, in extreme cases, can take minutes to complete.

Is More Runtime Diagnostic Information Needed?

Some environment variable settings are advisory and will be ignored due to system administration policies or system resource limitations. Or, some settings may be ignored because a variable name was misspelled. To confirm what Sun MPI environment variable values are being used, set the `MPI_PRINTENV` environment variable:

```
% setenv MPI_PRINTENV 1
```

When multiple interconnects are available on your cluster, you can check which interconnects are actually used by your program with by setting the MPI_SHOW_INTERFACES environment variable:

```
% setenv MPI_SHOW_INTERFACES 2
```

Launching Jobs on a Multinode Cluster

In a cluster configuration, the mapping of MPI processes to nodes in a cluster can impact application performance significantly. This section describes some important issues:

- "Minimizing Communication Costs" on page 83
- "Load Balancing" on page 84
- "Controlling Bisection Bandwidth" on page 84
- "Considering the Role of I/O Servers" on page 85
- "Running Jobs in the Background" on page 85
- "Limiting Core Dumps" on page 86
- "Using Line-Buffered Output" on page 86

Minimizing Communication Costs

Communication between MPI processes on the same shared-memory node is much faster than between processes on different nodes. Thus, by collocating processes on the same node, application performance can be increased. Indeed, if one of your servers is very large, you might want to run your entire "distributed-memory" application on a single node.

Meanwhile, not all processes within an MPI job need to communicate efficiently with all others. For example, the MPI processes might logically form a square "process grid," in which there are many messages traveling along rows and columns, or predominantly along one or the other. In such a case, it might not be essential for all processes to be collocated, but only for a process to be collocated with its partners within the same row or column.

Load Balancing

Running all the processes on a single node can improve performance if the node has sufficient resources available to service the job, as explained in the preceding section. At a minimum, it is important to have no more MPI processes on a node than there are CPUs. It might also be desirable to leave at least one CPU per node idle (see "Running on a Dedicated System" on page 79). Additionally, if bandwidth to memory is more important than interprocess communication, you might prefer to underpopulate nodes with processes so that processes do not compete unduly for limited server backplane bandwidth. Finally, if the MPI processes are multithreaded, it is important to have a CPU available for each lightweight process (LWP) within an MPI process. This last consideration is especially tricky because the resource manager (CRE or LSF) might not know at job launch that processes will spawn other LWPs.

Controlling Bisection Bandwidth

Clusters configured with commodity interconnects typically provide little internodal bandwidth per node. Meanwhile, bisection bandwidth might be the limiting factor for performance on a wide range of applications. In this case, if you must run on multiple nodes, you might prefer to run on more nodes rather than on fewer.

This point is illustrated qualitatively in FIGURE 7-1. The high-bandwidth backplanes of large Sun servers provide excellent bisection bandwidth for a single node. Once you have multiple nodes using a commodity interconnect, however, the interface between each node and the network will typically become the bottleneck. Bisection bandwidth starts to recover again when the number of nodes—actually, the number of network interfaces—increases.

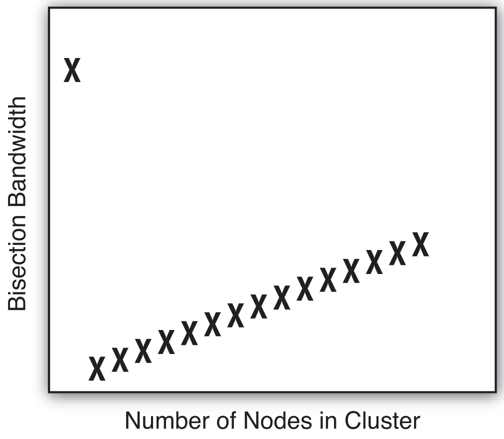


FIGURE 7-1 Relationship Between Bisection Bandwidth and Number of Nodes

In practice, every application benefits at least somewhat from increased locality, so collocating more processes per node by reducing the number of nodes has some positive effect. Nevertheless, for codes that are dominated by all-to-all types of communication, increasing the number of nodes can improve performance.

Considering the Role of I/O Servers

The presence of I/O servers in a cluster affects the other issues we have been discussing in this section. If, for example, a program will make heavy use of a particular I/O server, executing the program on that I/O node might improve performance. If the program makes scant use of I/O, you might prefer to avoid I/O nodes, since they might consume nodal resources. If multiple I/O servers are used, you might want to distribute MPI processes in a client job to increase aggregate ("bisection") bandwidth to I/O.

Running Jobs in the Background

Performance experiments conducted in the course of tuning often require multiple runs under varying conditions. It might be desirable to run such jobs in the background.

To run jobs in the background, perhaps from a shell script, use the -n switch with the CRE mprun command when the standard input is not being used. Otherwise, the job could block. The following example shows the use of this switch:

```
% mprun -n -np 4 a.out &
% cat a.csh
#!/bin/csh
mprun -n -np 4 a.out
% a.csh
```

Limiting Core Dumps

Core dumps can provide valuable debugging information, but they can also induce stifling repercussions for silly mistakes. In particular, core dumps of Sun HPC processes can be very large. For multiprocess jobs, the problem can be compounded, and the effect of dumping multiple large core files over a local network to a single, NFS-mounted file system can be crippling.

To limit core dumps for jobs submitted with the CRE mprun command, simply limit core dumps in the parent shell before submitting the job. If the parent shell is csh, use the command `limit coredumpsize 0`. If the parent shell is sh, use the `ulimit -c 0` command.

Using Line-Buffered Output

When multiple MPI ranks are writing to the same output device, the multiple output streams may interfere with one another, such that output from different ranks can be interleaved in the middle of an output line.

One way of handling this is to specify to CRE that it should use line-buffered output. For example, one may use the -o or -I switches to the mprun command.

The `-I` syntax is not simple but allows detailed control over a job's I/O streams. For example, consider the sample Fortran MPI code:

```
include "mpif.h"

call MPI_Init(ier)
call MPI_Comm_rank(MPI_COMM_WORLD,me,ier)
call MPI_Barrier(MPI_COMM_WORLD,ier)
do i = 1, 1000
  write(6,'("rank",i4,";  iteration", i6)') me, i
enddo
call MPI_Finalize(ier)
end
```

Executing the job without line buffering can lead to output lines from different ranks being combined (as shown in this example):

```
% mprun -np 16 a.out
[...]
rank   2;  iteration    34
rank   2;  iteration    35
rank   2;  iteration    36
rank   2;  iteration    37
rank   2;  iteration    3rank   7;  iteration     1
rank   7;  iteration     2
rank   7;  iteration     3
rank   7;  iteration     4
[...]
```

In contrast, you can use the `-I` switch:

```
% mprun -np 16 -I 0r=/dev/null,1wl,2w=errorfile a.out
```

Using this switch directs the job:

- To read `stdin` from `/dev/null`
- To use line buffering for `stdout`
- To direct `stderr` to *errorfile*

For more information on this syntax, see the section of the `mprun` man page that covers file descriptor strings.

Multinode Job Launch Under CRE

CRE provides a number of ways to control the mapping of jobs to the respective nodes of a cluster.

Collocal Blocks of Processes

CRE supports the collocation of blocks of processes—that is, all processes within a block are mapped to the same node.

Assume you are performing an LU decomposition on a 4x8 process grid. If minimization of communication within each block of four consecutive MPI ranks is most important, then these 32 processes could be launched in blocks of 4 collocated MPI processes by using the `-Z` or `-Zt` option, respectively:

```
% mprun -np 32 -Zt 4 a.out
% mprun -np 32 -Z  4 a.out
```

In either case, MPI ranks 0 through 3 will be mapped to a single node. Likewise, ranks 4 through 7 will be mapped to a single node. Each block of four consecutive MPI ranks is mapped to a node as a block. Using the `-Zt` option, no two blocks will be mapped to the same node—eight nodes will be used. Using the `-Z` option, multiple blocks might be mapped to the same node. For example, with the `-Zt` option, the entire job might be mapped to a single node if it has at least 32 CPUs.

Multithreaded Job

Consider a multithreaded MPI job in which there is one MPI process per node, with each process multithreaded to make use of all the CPUs on the node. You could specify 16 such processes on 16 different nodes by using:

```
% mprun -Ns -np 16 a.out
```

Round-Robin Distribution of Processes

Imagine that you have an application that depends on bandwidth for uniform, all-to-all communication. If the code requires more CPUs than can be found on any node within the cluster, it should be run over all the nodes in the cluster to maximize bisection bandwidth. For example, for 32 processes, this can be effected with the command:

```
% mprun -Ns -W -np 32 a.out
```

That is, CRE tries to map processes to distinct nodes (because of the `-Ns` switch, as in the preceding multithreaded case), but it will resort to "wrapping" multiple processes (`-W` switch) onto a node as necessary.

Detailed Mapping

For more complex mapping requirements, use the `mprun` switch `-m` or `-l` to specify a rankmap as a file or a string, respectively. For example, if the file `nodelist` contains:

```
node0
node0 2
node0
node1 4
node2 8
```

then the command:

```
% mprun -np 16 -m nodelist a.out
```

maps the first 4 processes to node0, the next 4 to node1, and the next 8 to node2. Refer to the *Sun HPC CluaterTools User's Guide* for more information about process mappings.

CHAPTER 8

Profiling

An important component of performance tuning is profiling, through which you develop a picture of how well your code is running and what sorts of bottlenecks it might have. Profiling can be a difficult task in the simplest of cases, and the complexities multiply with MPI programs because of their parallelism. Without profiling information, however, code optimization can be wasted effort.

This chapter describes:

This chapter includes a few case studies that examine some of the NAS Parallel Benchmarks 2.3. These are available from the NASA Ames Research Center at

```
http://www.nas.nasa.gov/Software/NPB/index.html
```

Note – The runs shown in this chapter were not optimized for the platforms on which they executed.

General Profiling Methodology

It is likely that only a few parts of a program account for most of its runtime. Profiling enables you to identify these "hot spots" and characterize their behavior. You can then focus your optimization efforts on the spots where they will have the most effect.

Profiling can be an experimental, exploratory procedure. So you might find yourself rerunning an experiment frequently. It is a challenge to design such runs so that they complete quickly, while still capturing the performance characteristics you are trying to study. There are several ways you can strip down your runs to achieve this balance, including reducing the data set and performing fewer loop iterations. However, regardless of which streamlining method you employ, keep the following caveats in mind:

- Try to maintain the same problem size, since changing the size of your data set can change the performance characteristics of your code. Similarly, reducing the number of processors used can mask scalability problems or produce ungeneralizable behavior.
- If the problem size must be reduced because only a few processors are available, try to determine how the data set should be scaled to maintain comparable performance behavior. For many algorithms, it makes most sense to maintain a fixed *subgrid* size. For example, if a full dataset of 8 Gbytes is expected to run on 64 processors, then maintain the fixed subgrid size of 128 Mbyte per processor by profiling a 512-Mbyte data set on 4 processors.
- Try to shorten experiments by running fewer iterations. One difficulty with this approach is that the long-term, steady-state performance behavior of your code might become dwarfed by otherwise inconsequential factors. In particular, code might behave differently the first few times it is executed than when buffers, caches, and other resources have been warmed up.

Basic Approaches

There are various approaches to profiling Sun HPC ClusterTools programs:

- *Use MPProf, a message-passing profiler introduced with the Sun HPC ClusterTools 5 release.* This extremely easy-to-use profiler offers both basic information about MPI performance of your program and Sun MPI–specific recommendations for environment variable tuning.
- *Use the Performance Analyzer.* (The programs Collector and Performance Analyzer are supplied in conjunction with the Sun Studio Compiler Collections.) This method is probably the most basic approach to profiling an HPC application on Sun systems, both for Sun MPI and non-MPI programs. No recompiling or relinking is required. Sampling data shows which routines are consuming the most time. User computation and MPI message passing are profiled, and caller-callee relationships are shown. Recompilation and relinking with the `-g` switch enable attribution to individual source-code lines with almost no loss in optimizations. On UltraSPARC microprocessors, hardware-based profiling can identify where floating-point operations, cache misses, and so forth, occur. A timeline view indicates synchronization and load imbalance problems. For information about using Performance Analyzer profiling, see "Performance Analyzer Profiling of Sun MPI Programs" on page 109.

- *Modify your source code to include timer calls.* This modification is most appropriate if you have reasonable familiarity with the program. You can place timers at a high level to understand gross aspects of the code, or at a fine level to study particular details. For information about the inserting timer calls using Sun MPI, see "Inserting MPI Timer Calls" on page 124.
- *Use the MPI profiling interface (PMPI) to diagnose other aspects of message-passing performance.* The MPI standard supports an interface for instrumentation of MPI calls, which enables you to apply custom or third-party instrumentation of MPI usage without modifying your application's source code. For more information about using the MPI profiling interface, see "Using the MPI Profiling Interface" on page 123.

Use the DTrace dynamic tracing utility. DTrace is a component of the Solaris 10 OS. DTrace is a comprehensive dynamic tracing utility that you can use to monitor the behavior of applications programs as well as the operating system itself. For more information about DTrace, refer to the *Solaris Dynamic Tracing Guide* (Part Number 817-6223). This guide is part of the Solaris 10 OS Software Developer Collection.

Solaris 10 OS documentation can be found on the web at the following location:

`http://www.sun.com/documentation`

TABLE 8-1 outlines the advantages and disadvantages associated with each of these methods of profiling.

TABLE 8-1 Profiling Alternatives

Method	Advantages	Disadvantages
MPProf	• Very simple to use. • Generates self-explanatory performance report. • Gives feedback in simple terms (such as which environment variables to set for better performance).	• Simplicity. • Has no knowledge of user code. • Performance suggestions are fallible.

TABLE 8-1 Profiling Alternatives *(Continued)*

Method	Advantages	Disadvantages
DTrace	• Included with the Solaris 10 OS • Can be used on live production systems to monitor behavior and track errors. • No need to recompile application code. • Can be attached and detached to and from an MPI program multiple times without disturbing the run. • Use of a scripting language allows you to generate complex output. • The D scripting language has many built-in functions, so scripts are less complicated to write.	• Need to learn another scripting language. • Very simple output processing (no GUI). • Very low level tool.
Performance Analyzer Profiling	• No recompilation or relinking is required. • Profiles whole programs: user computation and MPI message passing. • Identifies time-consuming routines. • With `-g` recompilation and relinking, gives attribution on a per-source-line basis with negligible loss in optimization level. • Shows caller–callee relationships. • Uses a style familiar to `gprof` users. • On UltraSPARC microprocessors, profiles based on hardware counters (floating-point operations, cache misses, and so forth). • Timeline functionality.	• Has very limited knowledge of MPI or message passing. • Improper timeline support for multinode runs.
Timers	• Very versatile.	• Requires manual instrumentation. • Requires that you understand the code.

TABLE 8-1 Profiling Alternatives *(Continued)*

Method	Advantages	Disadvantages
`gprof`	• Familiar tool. • Provides an overview of user code.	• Ignores time spent in MPI.
PMPI	• You can instrument or modify MPI without modifying source. • Allows use of other profiling tools.	• Profiles MPI usage only. • Requires integration effort.

The following are sample scenarios:

- *I don't know whether my program is spending time in serial computation or in MPI calls. The performance guide says I'm supposed to set Sun MPI environment variables to trade off memory for performance, but it seems like a lot of work to figure how what values to use.*

  Using MPProf, you can easily see if your program is spending a lot of time in MPI. It will suggest environment variable values for you.

- *I am rather unfamiliar with the code I'm running. I do not know whether my optimization efforts should focus on serial computation or message passing—or, for that matter, in which routines.*

  Using the Performance Analyzer, you can see which routines consume the most time.

- *I know that a few innermost loops are bottlenecks and I need more detailed information.*

  Adding timers and other instrumentation around innermost loops might help you if you already have some idea about your code's performance.

- *I have used certain MPI profiling tools in other environments and am used to them.*

  Depending on how those tools were constructed, the MPI profiling interface might allow you to continue using them with Sun HPC ClusterTools programs.

MPProf Profiling Tool

Sun HPC ClusterTools software includes the `mpprof` profiler, a tool that supports programming with the Sun MPI library.

To collect profiling data on a Sun MPI program, set the `MPI_PROFILE` environment variable before running the program. To generate a report using this data, invoke the `mpprof` command-line utility. For example:

```
% setenv MPI_PROFILE 1
% mprun -np 16 a.out
% mpprof mpprof.index.rm.jid
```

If profiling has been enabled, Sun MPI creates a binary-format profiling data file for each MPI process. The MPProf utility digests the data from these numerous, intermediate, data files—generating statistical summaries in ASCII text. Wherever possible, MPProf adds direct tuning advice to the statistical summaries of MPI process behavior.

MPProf does not provide detailed information about the ordering of message-passing events or interprocess synchronization. That is, MPProf is not a trace-history viewer.

Note – To get information on user code as well as MPI message-passing activity, use Performance Analyzer.

Sample MPProf Output

Output from the MPProf report generator is self explanatory and should require no further documentation. Nevertheless, sample output is included here for readers who do not have immediate access to this tool.

Overview

The first section (sample output shown in box below) is an overview that includes

- The name of your program
- When you ran your program
- What time period is covered in this report
- What fraction of time was spent in MPI

- How many MPI ranks are covered in the report

```
OVERVIEW
========
The program being reported on is
"/home/myusername/mycode/a.out,"
which ran as job name "cre.6975" on Fri Mar 17 18:45:04 2006.
Profiled Time Range:
Start at elapsed time 0.000029 secs
End at elapsed time   9.475821 secs
Total duration is     9.475792 secs
Fraction spent in MPI 21.5%
Elapsed time is measured from the end of MPI_Init. Data is being
reported for 16 processes of a 16-process job.
```

Load Balance

The Load section (sample output shown in box below) shows the distribution of how much time was spent in MPI per process. When a process spends much time in MPI, this may be because it's waiting for another process. Thus, performance bottlenecks may be in processes that spend little time in MPI. Those processes, as well as the nodes they ran on, are identified in this section. Other tools, such as the Performance Analyzer, are needed to pinpoint bottlenecks further if load imbalances are apparent.

```
LOAD BALANCE
============
Data is being reported on 16 MPI processes. The following histogram
shows how these processes were distributed as a function of the
fraction of the time the processes spent  in MPI routines:

Number of MPI Processes

10-|
 9-|
 8-|
 7-|                                                 #
 6-|                                                 #
 5-|                                                ##
 4-|                                                ##
 3-|                                               ###
 2-|                                               ###
 1-|  #                                           ####
 0-+----+----+----+----+----+----+----+----+----+----+
 15.0 20.0 25.0 30.0 35.0 40.0 45.0 50.0 55.0 60.0 65.0

  Percentage time in MPI

Rank   Hostname   MPI Time
0       node0     17.39%
15      node0     61.04%
4       node0     62.03%
13      node0     62.08%
11      node0     62.12%

[...]
```

Sun MPI Environment Variables

As discussed in Chapter 7, Sun MPI environment variables are set by default to values that will be appropriate in many cases, but runtime performance can in cases be enhanced by further tuning.

In particular, Sun MPI implements two-sided message passing between processes that cannot directly access one another's address spaces with internal shared buffers. If these buffers become congested, performance can suffer. MPProf monitors these buffers, watches for congestion, and suggests environment variable values for enhanced performance at the cost of higher memory usage.

The report discusses its recommendations and then provides a summary (sample output shown in box below) that may be "copy-and-pasted" into a session window or run script. The code may then be run again to determine whether better performance results from the recommendations.

```
MPI ENVIRONMENT VARIABLES
=========================
[...]

SUGGESTION SUMMARY
==================
[...]

In the C shell, these environment variables may be set by the
following commands:

setenv MPI_POLLALL 0
setenv MPI_PROCBIND 1
setenv MPI_SHM_SBPOOLSIZE 368640
setenv MPI_SPIN 1

In the Bourne or Korn shell, these environment variables may be
set by the following commands:

export MPI_POLLALL=0
export MPI_PROCBIND=1
export MPI_SHM_SBPOOLSIZE=368640
export MPI_SPIN=1
```

Breakdown by MPI Routine

The Breakdown by MPI Routine section of the report (sample output shown in box below) gives a breakdown by MPI routine. Attributing bytes sent and received to MPI routines can be tricky due to nonblocking and collective operations. Note that:

- For nonblocking sends, MPProf attributes bytes sent to the initiating call (such as `MPI_Isend`), since this is often the call that actually moves the data.
- For nonblocking receives, MPProf attributes bytes received to the completing call (such as `MPI_Wait`), since the actual number of bytes received is not known before this call.

- For collectives, MPProf attributes bytes sent and received as follows:

TABLE 8-2 MPProf Attributes Bytes Sent and Received

	Sent	Received
MPI_Barrier	0	0
MPI_Bcast	n (root)	n (nonroot)
MPI_Gather	n	np * n (root)
MPI_Scatter	np * n (root)	n
MPI_Allgather	n	np * n
MPI_Alltoall	np * n	np * n
MPI_Reduce	n	n (root)
MPI_Allreduce	n	n
MPI_Scan	n	n
MPI_Reduce_scatter	np * n	n

Assumptions:

- *n* is the number of elements passed to the MPI collective routine multiplied by the size of the data type used.
- *np* is the number of processes participating in the collective operation.
- Some byte counts are attributed only at the root process or the nonroot processes, as noted.

```
BREAKDOWN BY MPI ROUTINE
========================

Here, averages over all MPI processes profiled are reported. The
numbers in parentheses roughly indicate the variations there are
among all of the MPI processes. These variations are computed as
(1-min/max)/2 where "min" and "max" are the minimum and maximum
values, respectively, for each statistic reported. A total of 9
different MPI APIs were called.

MPI Routine      Time    Calls Made    Sent      Received
MPI_Allreduce  0.15 (39%)  11 (0%)  45276 (0%)  45276  (0%)
MPI_Alltoall   0.00  (8%)  11 (0%)    704 (0%)    704  (0%)
MPI_Alltoallv  0.55  (3%)  11 (0%)  23672 (0%)  23672  (2%)
MPI_Comm_rank  0.00 (16%)   1 (0%)      0 (0%)      0  (0%)
MPI_Comm_size  0.00  (8%)   1 (0%)      0 (0%)      0  (0%)
MPI_Irecv      0.00 (23%)   0 (0%)      0 (0%)      0  (0%)
MPI_Reduce     0.00 (49%)   2 (0%)     12 (0%)      0 (50%)
MPI_Send       0.00 (27%)   0 (0%)      3 (0%)      0  (0%)
MPI_Wait       0.00 (50%)   0 (0%)      0 (0%)      3  (0%)

Where "Time" is in seconds and "Sent" and "Received" are in bytes.
```

Time Dependence

Performance analysis tools are sometimes classified as *profiling* tools, which give overall performance characterization of a program, or *tracing* tools, which give detailed time histories of events that occur during program execution.

While MPProf is a profiling tool, it includes some time-dependent information (a sample of the Time Dependence section of the MPProf report is shown below). This information is coarse but can prove useful in different ways:

- If your program does not terminate normally (due to program error or your decision to abort the run), some profiling information is still available.
- You can monitor your program's performance while it is still running. Note that some collected data might not have been flushed to disk yet. This means that if you generate an MPProf report while your program is still running, the report might not represent the latest state of your program.
- You can sometimes identify different phases of computation (startup, steady-state computation, post processing, and so forth) and focus on any one of them for better analysis using the `mpprof` switches `-s` and `-e`.

In the following example, there is considerable broadcast and barrier activity before the program settles down to steady-state behavior, the performance of which is critical for long-running programs.

```
TIME DEPENDENCE
===============
Time periods may be missing if no MPI calls were made during the
period. Times for MPI calls that persist over multiple reporting
intervals will only be reported in a single interval; these
reported times may be greater than 100%.

period  MPI_Barrier  MPI_Bcast  MPI_Send  MPI_Waitall

 1         1.2%       6.9%      0.0%        0.0%
 5         0.0%       0.0%      0.0%      123.0%
 6         0.0%       0.0%      0.0%        4.2%
 7         0.0%     803.3%      0.1%        3.1%
16         0.0%     101.0%      3.9%        0.0%
17         0.0%       7.6%      0.0%        0.0%
18        32.5%       0.0%      0.5%       10.3%
19        18.9%       5.0%     52.4%        0.0%
20       191.9%       0.0%     22.6%       34.5%
21         0.0%       0.0%      0.0%        0.0%
22        10.9%       0.0%      0.0%        0.0%
23        16.7%       2.1%      0.2%        4.3%
24        21.7%       2.1%      0.3%        4.6%
25        21.0%       2.1%      0.3%        5.4%
26        21.1%       3.6%      0.3%        5.2%
27        20.1%       1.9%      0.4%        5.9%
28        20.6%       2.1%      0.3%        5.3%
29        20.4%       2.1%      0.3%        5.5%
30        17.9%       3.6%      0.2%        5.0%
31        22.6%       2.1%      0.3%        5.4%
```

Connections

A *connection* is a sender/receiver pair. MPProf shows how much point-to-point message traffic there is per connection, with one matrix showing numbers of messages and another showing numbers of bytes. Values are scaled.

A real application is likely to have various communication patterns in it, but the overall matrix may resemble some easy-to-recognize case. The following examples illustrate some simple patterns.

```
CONNECTIONS
===========

                    sender
                    0   1   2   3   4   5   6   7   8   9
receiver

                0   _  99  99  99  99  99  99  99  99  99
                1   _   _   _   _   _   _   _   _   _   _
                2   _   _   _   _   _   _   _   _   _   _
                3   _   _   _   _   _   _   _   _   _   _
                4   _   _   _   _   _   _   _   _   _   _
                5   _   _   _   _   _   _   _   _   _   _
                6   _   _   _   _   _   _   _   _   _   _
                7   _   _   _   _   _   _   _   _   _   _
                8   _   _   _   _   _   _   _   _   _   _
                9   _   _   _   _   _   _   _   _   _   _
```

```
CONNECTIONS
===========

                    _   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
                   99   _   _   _   _   _   _   _   _   _
```

```
CONNECTIONS
===========

                _ 99  _  _  _  _  _  _  _  _
             99  _ 99  _  _  _  _  _  _  _
              _ 99  _ 99  _  _  _  _  _  _
              _  _ 99  _ 99  _  _  _  _  _
              _  _  _ 99  _ 99  _  _  _  _
              _  _  _  _ 99  _ 99  _  _  _
              _  _  _  _  _ 99  _ 99  _  _
              _  _  _  _  _  _ 99  _ 99  _
              _  _  _  _  _  _  _ 99  _ 99
              _  _  _  _  _  _  _  _ 99  _
```

```
CONNECTIONS
===========

              _ 99  _  _  _  _  _  _  _ 99
             99  _ 99  _  _  _  _  _  _  _
              _ 99  _ 99  _  _  _  _  _  _
              _  _ 99  _ 99  _  _  _  _  _
              _  _  _ 99  _ 99  _  _  _  _
              _  _  _  _ 99  _ 99  _  _  _
              _  _  _  _  _ 99  _ 99  _  _
              _  _  _  _  _  _ 99  _ 99  _
              _  _  _  _  _  _  _ 99  _ 99
             99  _  _  _  _  _  _  _ 99  _
```

```
CONNECTIONS
===========

              _9____99____________________9______
              9_9_____9____________________9_____
              _9_9_____9____________________9____
              __9_9_____9____________________9___
              ___9_9_____9____________________9__
              ____9_9_____9____________________9_
              9____9_______9____________________9
              9_______9____99____________________
              _9_____9_9_____9___________________
              __9_____9_9_____9__________________
              ___9_____9_9_____9_________________
              ____9_____9_9_____9________________
              _____9_____9_9_____9_______________
              ______99____9_______9______________
              _______9_______9____99_____________
              ________9_____9_9_____9____________
              _________9_____9_9_____9___________
              __________9_____9_9_____9__________
              ___________9_____9_9_____9_________
              ____________9_____9_9_____9________
              _____________99____9_______9_______
              ______________9_______9____99______
              _______________9_____9_9_____9_____
              ________________9_____9_9_____9____
              _________________9_____9_9_____9___
              __________________9_____9_9_____9__
              ___________________9_____9_9_____9_
              ____________________99____9_______9
              9____________________9_______9____9
              _9____________________9_____9_9____
              __9____________________9_____9_9___
              ___9____________________9_____9_9__
              ____9____________________9_____9_9_
              _____9____________________9_____9_9
              ______9____________________99____9_
```

Once the connectivity matrices have been displayed, MPProf prints the average message length:

```
The average length of point-to-point messages was 13916 bytes per
message.
```

This message length can be compared to the product of some characteristic latency in your system and some characteristic bandwidth. For example, if “ping-pong” tests indicate an MPI latency of about 2 microseconds and bandwidth of about 500 Mbyte/sec, then the product is 1000 bytes. An average length of 13916 bytes suggests bandwidth is more important than latency to your application’s performance.

Multithreaded Programs

MPProf profiling might be used with multi-threaded MPI programs. If many threads per process are engaged in heavy MPI activity, however, MPProf data collection can slow down measurably. MPProf aggregates data for all threads on a process.

The `mpdump` Utility

It is sometimes possible to extract more information from MPProf data files than the report generator prints out. You can use the `mpdump` utility to convert MPProf binary data files to an ASCII format and then process the data directly, perhaps with customized scripts. The format of the ASCII files is undocumented, but it is easy to interpret.

Managing Disk Files

MPProf writes relatively little data to disk during profiling runs. This means that it is relatively easy to use compared to tracing tools, the data volumes of which must be managed carefully.

Nevertheless, one file per process is generated and it is worth managing these files. Specifically, MPProf profiling generates one data file per process per run, as well as one index file per run that points to the data files. The data files, by default, are stored in `/usr/tmp` of the respective nodes where the processes ran.

Use

```
% mpprof -r -g directory mpprof.index.rm.jid
```

to collect data files from their respective nodes to an archival location if you want to save the files.

Use

```
% mpprof -r -mpprof.index.rm.jid
```

to remove data files to clean up after a run.

Incorporating Environment Variable Suggestions

MPProf generates suggestions for tuning Sun MPI environment variables.

These suggestions are often helpful, but they do not guarantee performance improvement. Some experimentation may be necessary. If you incorporate MPProf's suggestions, the next profiling run may indicate further suggestions. Multiple reruns may be required before MPProf no longer has further recommendations, but most of the speedup, if any, is likely to result from the first round or two of experimentation.

Rerunning a program incorporating MPProf's tuning suggestions may be automated. Here is an example.

CODE EXAMPLE 8-1 Sample MPProf Session, Rerunning a Program To Tune Performance

```
% cat my_mpprof_script.csh
#!/bin/csh

set LOGFILE = mpprofrun.logfile

# always run with this on
setenv MPI_PROFILE 1
echo
echo trial-0
echo setenv MPI_PROFILE 1

# iterate at most 10 times
@ iteration = 0
while ( $iteration < 10 )

  # run the job
  $* >& $LOGFILE

  # archive this run

mkdir trial-$iteration
mpprof -r -g trial-$iteration mpprof.index.* < /dev/null
mv $LOGFILE trial-$iteration/log

# go to the archive
cd trial-$iteration

# report the basics
mpprof mpprof.index.* < /dev/null > mpprof.output
grep "Total duration is" mpprof.output
grep "Fraction spent in" mpprof.output

# increment interation count
```

CODE EXAMPLE 8-1 Sample MPProf Session, Rerunning a Program To Tune Performance

```
@ iteration = $iteration + 1

# look at the environment variable suggestions
grep "setenv" mpprof.output | grep -v Suggestion > mpprof.envvars
set wclist = ( `wc -l mpprof.envvars` )
echo

# start reporting on next round

if ( $wclist[1] != 0 ) then
  echo trial-$iteration
  cat mpprof.envvars
  source mpprof.envvars
endif

# clean up
rm -f mpprof.envvars
cd ..

# convergence?
if ( $wclist[1] == 0 ) then
  break
endif

end

% my_mpprof_script.csh mprun -np 16 a.out

trial-0
setenv MPI_PROFILE 1
  Total duration is    23.702518 secs
  Fraction spent in MPI 21.9

%trial-1
setenv MPI_POLLALL 0
setenv MPI_PROCBIND 1
unsetenv MPI_SHM_CPOOLSIZE
setenv MPI_SHM_SBPOOLSIZE 533504
setenv MPI_SPIN 1
  Total duration is    21.697837 secs
  Fraction spent in MPI 14.5%

trial-2
setenv MPI_SHM_NUMPOSTBOX 66
setenv MPI_SHM_SBPOOLSIZE 1064960
  Total duration is    19.901631 secs
  Fraction spent in MPI  5.7%
```

CODE EXAMPLE 8-1 Sample MPProf Session, Rerunning a Program To Tune Performance

```
trial-3
setenv MPI_SHM_SBPOOLSIZE 1598464
  Total duration is    19.993672 secs
  Fraction spent in MPI  5.9%

trial-4
setenv MPI_SHM_SBPOOLSIZE 2129920
  Total duration is    19.946838 secs
  Fraction spent in MPI  5.8%

trial-5
setenv MPI_SHM_SBPOOLSIZE 2661376
  Total duration is    19.906594 secs
  Fraction spent in MPI  5.7%


% ls -d trial-*

trial-0  trial-1  trial-2  trial-3  trial-4  trial-5
```

Performance Analyzer Profiling of Sun MPI Programs

The Performance Analyzer offers a good first step to understanding the performance characteristics of a program on Sun systems. It combines ease of use with powerful functionality.

As with most profiling tools, there are two basic steps to using Sun's performance analysis tools. The first step is to use the Collector to collect performance data. The second step is to use the Performance Analyzer to examine results. For example, this procedure can be as simple as replacing this:

```
% mprun -np 16 a.out 3 5 341
```

with this:

```
% mprun -np 16 collect a.out 3 5 341
% analyzer test.*.er
```

Sun's compilers and tools are usually located in the directory `/opt/SUNWspro/bin`. Check with your system administrator for details for your site.

The following sections show the use of the Performance Analyzer with Sun MPI programs, often revisiting variations of the preceding example.

Data Collection

There are several ways of using the Collector with MPI codes. For example, the simplest usage would be to add `collect` to the `mprun` command line:

```
% mprun -np 16 collect a.out 3 5 341
```

In this section we discuss some of the issues that can arise when using the Collector, and how to handle them.

Data Volume

The volume of collected data can grow large, especially for long-running or parallel programs. Though the Collector mitigates this problem, the scaling to large volumes remains an issue.

There are a number of useful strategies for managing these data volumes:

- *Increase the profiling interval.* The interval might be specified in milliseconds with the `-p` switch to the `collect` command. The default value is 10. The actual interval used depends on the resolution of the profiling system. Of course, while increasing the interval reduces the data volume, it can reduce the quality of the sampling data.

  For example, to reduce the number of profiled events roughly by a factor of two, use:

  ```
  % mprun -np 16 collect -p 20 a.out 3 5 341
  ```

- *Collect data on only a subset of the MPI processes.* In many cases, activity on one MPI process reflects performance behavior on all processes fairly closely. Or, if there are load imbalances among the processes, a larger subset might be used. Of course, limiting data collection to a subset of the processes might bias the profiling data. In particular, a master process might behave unlike any of the other processes. In the following example, we collect on only the first four MPI ranks using a script:

```
% cat csh-script
#!/bin/csh
if ( $MP_RANK < 4 ) then
  collect $*
else
  $*
endif
% mprun -np 16 csh-script a.out 3 5 341
```

- *Collect data to a local file system.* This method does not reduce the volume of data collected, but it helps mitigate the impact. Meanwhile, local file systems are often not mounted on all nodes, so gathering experiment data to a common location should also be part of this approach. In the following example, we collect local data and then gather it centrally:

```
% cat csh-script
#!/bin/csh
collect -d /tmp $*
er_mv /tmp/test.$MP_RANK.er myrun
% mkdir myrun
% mprun -np 16 csh-script a.out 3 5 341
```

  This strategy should be a routine part of data collection.

A large parallel job might run out of a central NFS-mounted file system. While this might be adequate for jobs that are not I/O intensive, it might cause a critical bottleneck for profiling sessions. If multiple MPI processes are trying to write large volumes of profiling data simultaneously over NFS to a single file system, that file system, along with network congestion, could lead to tremendous slowdowns and perturbations of the program’s performance. It is preferable to collect profiling data to local file systems and, perhaps, gather them to a central directory after program execution.

To identify local file systems, use:

```
% /usr/bin/df -lk
```

on each node of the cluster you will use, or ask your system administrator about large-volume, high-performance disk space.

One possible choice of a local file system is /tmp. Note that the /tmp file system on different nodes of a cluster refer to different, respectively local file systems. Also, the /tmp file system might not be very large, and if it becomes filled there might be a great impact on general system operability.

Data Organization

The Collector generates one "experiment" per MPI process. If there are multiple runs of a multiprocess job, therefore, the number of experiments can grow quickly.

To organize these experiments, it often makes sense to gather experiments from a run into a distinctive subdirectory. Use the commands er_mv, er_rm, and er_cp (again, typically under the directory /opt/SUNWspro/bin) to move, remove, or copy experiments, respectively. For more information on these utilities, refer to the corresponding man pages or the Sun Studio Compiler Collection documentation.

If you collect an experiment directly into a directory, make sure that the directory has already been created and, ideally, that no other experiments already exist in it.

Example

```
% mkdir run1
% er_rm -f run1/*.er
% mprun -np 16 collect -d run1 a.out 3 5 341
% mkdir run2
% er_rm -f run2/*.er
% mprun -np 16 collect -d run2 a.out 3 5 341
% mkdir run3
% er_rm -f run3/*.er
% mprun -np 16 collect -d run3 a.out 3 5 341
%
```

The er_rm steps are not required because (in this instance) we are using freshly created directories. Nevertheless, these steps serve as reminders to avoid the confusion that can result when too many experiments are gathered in the same directory.

Other Data Collection Issues

The Collector collects sampling information by default. Thus, Sun MPI calls will be profiled if they consume a sufficiently high fraction of runtime. Sometimes it is interesting to capture MPI calls more methodically. In particular, for time-line analysis, it is desirable to mark the beginnings and endings of MPI calls with higher resolution than sampling provides. MPI call tracing may be activated with the `-m on` switch to `collect`.

Sometimes it is desirable to collect data not only on a process, but its descendants, as well. The `collect` tag will follow descendants with the `-F on` switch.

For example, the Solaris 10 Operating System supports multiple page sizes with the `/usr/bin/ppgsz` command-line utility. To collect performance data on a code run with 4-Mbyte memory pages, our sample data collection becomes:

```
% mprun -np 16 collect -F on ppg sz -o heap=4M,stack=512K a.out 3 5 341
```

In the particular case of the `ppgsz` utility, a cleaner alternative may be to preload the `mpss.so.1` shared object, so that the `ppgsz` utility need not appear on the command line. For more information on multiple page size support, see the `ppgsz` and `mpss.so.1` man pages in the Solaris 10 Operating System.

Analyzing Profiling Data

Once data has been gathered with the Collector, it can be viewed with the Performance Analyzer. For example:

```
% analyzer myrun/test.*.er
```

Use of the Performance Analyzer is illustrated in the case study.

There are also command-line utilities to aid in data analysis. For more information, see the `er_print` and `er_src` man pages. Here are examples of their use:

- To view a summary of how time was spent in functions, use:

  ```
  % er_print -functions test.*.er
  ```

- To view a summary of how time was spent in MPI functions, use:

  ```
  % er_print -functions test.*.er | grep PMPI_
  ```

- To view caller-callee relationships (as with `gprof`), use:

  ```
  % er_print -callers-callees test.*.er
  ```

- To view where time is spent on a per-source-line basis, along with compiler commentary, compile and link with the `-g` switch and use:

  ```
  % er_print -source myfunction_ 1 test.*.er
  ```

  The `1` is used to distinguish in case of multiple instances of the named function.

- To view the compiler commentary associated with an object file without even running your program, compile with the `-g` switch and use:

  ```
  % er_src my_file.o
  ```

Case Study

In this case study, we examine the NPB 2.3 BT benchmark. We run the program using the following environment variable settings:

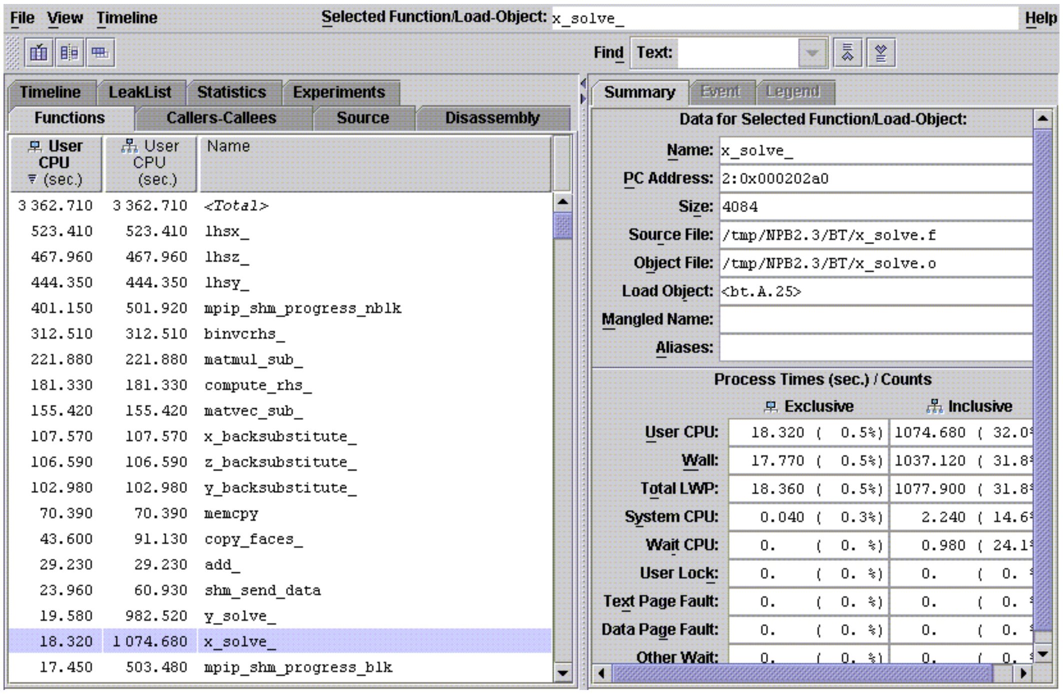
```
% setenv MPI_SPIN 1
% setenv MPI_PROCBIND 1
% setenv MPI_POLLALL 0
```

These settings are not required for Performance Analyzer profiling. We simply use them to profile our code as it would run in production. See Appendix C for more information about using Sun MPI environment variables for high performance. Also refer to the *Sun MPI Programming and Reference Guide*.

The job is run on a single, shared-memory node using 25 processes.

FIGURE 8-1 shows the Performance Analyzer's first view of the resulting profiling data. This default view shows how time is spent in different functions. Both exclusive and inclusive user CPU times are shown for each function, excluding and including, respectively, time spent in any functions it calls. The top line shows that a total of 3362.710 seconds, over all 25 processes, are profiled.

We see that the functions `LHSX()`, `LHSY()`, and `LHSZ()` account for 523.410+467.960+444.350=1435.72 seconds of that time.


File View Timeline
Help
Find Text:
Timeline LeakList Statistics Experiments
Functions Callers-Callees Source Disassembly
Summary Event Legend

Selected Function/Load-Object: x_solve_

User CPU (sec.)	User CPU (sec.)	Name
3 362.710	3 362.710	*<Total>*
523.410	523.410	lhsx_
467.960	467.960	lhsz_
444.350	444.350	lhsy_
401.150	501.920	mpip_shm_progress_nblk
312.510	312.510	binvcrhs_
221.880	221.880	matmul_sub_
181.330	181.330	compute_rhs_
155.420	155.420	matvec_sub_
107.570	107.570	x_backsubstitute_
106.590	106.590	z_backsubstitute_
102.980	102.980	y_backsubstitute_
70.390	70.390	memcpy
43.600	91.130	copy_faces_
29.230	29.230	add_
23.960	60.930	shm_send_data
19.580	982.520	y_solve_
18.320	1 074.680	x_solve_
17.450	503.480	mpip_shm_progress_blk

Data for Selected Function/Load-Object:

Name:	x_solve_
PC Address:	2:0x000202a0
Size:	4084
Source File:	/tmp/NPB2.3/BT/x_solve.f
Object File:	/tmp/NPB2.3/BT/x_solve.o
Load Object:	<bt.A.25>
Mangled Name:	
Aliases:	

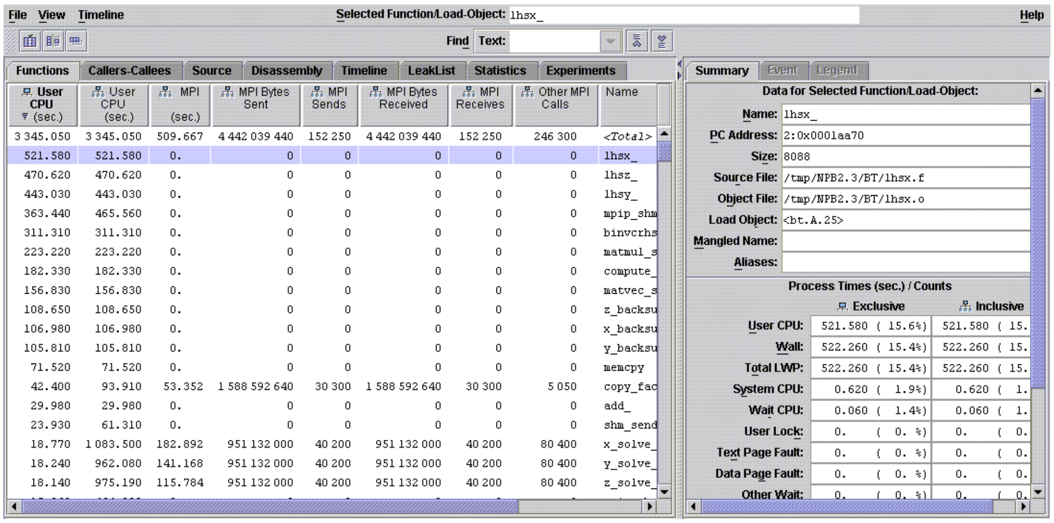
Process Times (sec.) / Counts

	Exclusive	Inclusive
User CPU:	18.320 (0.5%)	1074.680 (32.0
Wall:	17.770 (0.5%)	1037.120 (31.8
Total LWP:	18.360 (0.5%)	1077.900 (31.8
System CPU:	0.040 (0.3%)	2.240 (14.6
Wait CPU:	0. (0. %)	0.980 (24.1
User Lock:	0. (0. %)	0. (0.
Text Page Fault:	0. (0. %)	0. (0.
Data Page Fault:	0. (0. %)	0. (0.
Other Wait:	0. (0. %)	0. (0.

FIGURE 8-1 Performance Analyzer—Main View

Fortran programmers will note that the term *function* is used in the C sense to include all subprograms, whether they pass return values or not. Further, function names are those generated by the Fortran compiler. That is, by default they are converted to lower case and have an underscore appended.

In FIGURE 8-2, we see how this overview appears when MPI tracing is turned on with the Collector's `-m on` switch. Note that accounting for bytes sent and received is slightly different from the way accounting is handled by MPProf. In particular, for bytes received, the size of the receive buffer is used rather than the actual number of bytes received. For more information on how bytes are counted for MPI calls, refer to the Performance Analyzer documentation volume *Program Performance Analysis Tools*.



User CPU (sec.)	User CPU (sec.)	MPI (sec.)	MPI Bytes Sent	MPI Sends	MPI Bytes Received	MPI Receives	Other MPI Calls	Name
3 345.050	3 345.050	509.667	4 442 039 440	152 250	4 442 039 440	152 250	246 300	<Total>
521.580	521.580	0.	0	0	0	0	0	lhsx_
470.620	470.620	0.	0	0	0	0	0	lhsz_
443.030	443.030	0.	0	0	0	0	0	lhsy_
363.440	465.560	0.	0	0	0	0	0	mpip_shm
311.310	311.310	0.	0	0	0	0	0	binvcrhs
223.220	223.220	0.	0	0	0	0	0	matmul_s
182.330	182.330	0.	0	0	0	0	0	compute_
156.830	156.830	0.	0	0	0	0	0	matvec_s
108.650	108.650	0.	0	0	0	0	0	z_backsu
106.980	106.980	0.	0	0	0	0	0	x_backsu
105.810	105.810	0.	0	0	0	0	0	y_backsu
71.520	71.520	0.	0	0	0	0	0	memcpy
42.400	93.910	53.352	1 588 592 640	30 300	1 588 592 640	30 300	5 050	copy_fac
29.980	29.980	0.	0	0	0	0	0	add_
23.930	61.310	0.	0	0	0	0	0	shm_send
18.770	1 083.500	182.892	951 132 000	40 200	951 132 000	40 200	80 400	x_solve_
18.240	962.080	141.168	951 132 000	40 200	951 132 000	40 200	80 400	y_solve_
18.140	975.190	115.784	951 132 000	40 200	951 132 000	40 200	80 400	z_solve_

Data for Selected Function/Load-Object:

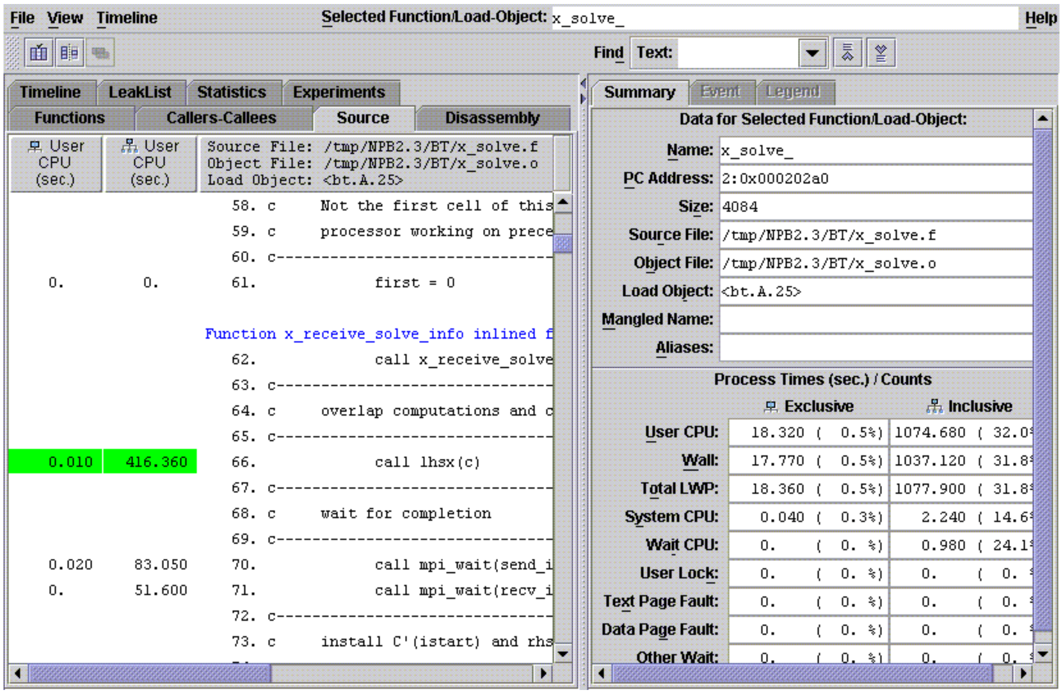
Name:	lhsx_
PC Address:	2:0x0001aa70
Size:	8088
Source File:	/tmp/NPB2.3/BT/lhsx.f
Object File:	/tmp/NPB2.3/BT/lhsx.o
Load Object:	<bt.A.25>
Mangled Name:	
Aliases:	

Process Times (sec.) / Counts

	Exclusive	Inclusive
User CPU:	521.580 (15.6%)	521.580 (15.
Wall:	522.260 (15.4%)	522.260 (15.
Total LWP:	522.260 (15.4%)	522.260 (15.
System CPU:	0.620 (1.9%)	0.620 (1.
Wait CPU:	0.060 (1.4%)	0.060 (1.
User Lock:	0. (0. %)	0. (0.
Text Page Fault:	0. (0. %)	0. (0.
Data Page Fault:	0. (0. %)	0. (0.
Other Wait:	0. (0. %)	0. (0.

FIGURE 8-2 Performance Analyzer—Main View With Tracing Enabled

We can see how time is spent within a subroutine if the code was compiled and linked with the -g switch. This switch introduces minimal impact on optimization and parallelization, and it can be employed rather freely by performance-oriented users. When we click on the Source tab, the Performance Analyzer brings up a text editor for the highlighted function. The choice of text editor can be changed under the Options menu with the Text Editor Options selection. The displayed, annotated source code includes the selected metrics, the user source code, and compiler commentary. A small fragment is shown in FIGURE 8-3. In particular, notice that the Performance Analyzer highlights hot (time-consuming) lines of code. Only a small fragment is shown since, in practice, the annotated source can become rather wide.



Selected Function/Load-Object: x_solve_

```
 User      User     Source File:  /tmp/NPB2.3/BT/x_solve.f
 CPU       CPU      Object File:  /tmp/NPB2.3/BT/x_solve.o
 (sec.)    (sec.)   Load Object:  <bt.A.25>
                       58.  c      Not the first cell of this
                       59.  c      processor working on prece
                       60.  c---------------------------------
 0.        0.          61.               first = 0

                     Function x_receive_solve_info inlined f
                       62.               call x_receive_solve
                       63.  c---------------------------------
                       64.  c      overlap computations and c
                       65.  c---------------------------------
 0.010     416.360     66.               call lhsx(c)
                       67.  c---------------------------------
                       68.  c      wait for completion
                       69.  c---------------------------------
 0.020     83.050      70.               call mpi_wait(send_i
 0.        51.600      71.               call mpi_wait(recv_i
                       72.  c---------------------------------
                       73.  c      install C'(istart) and rhs
```

Data for Selected Function/Load-Object:

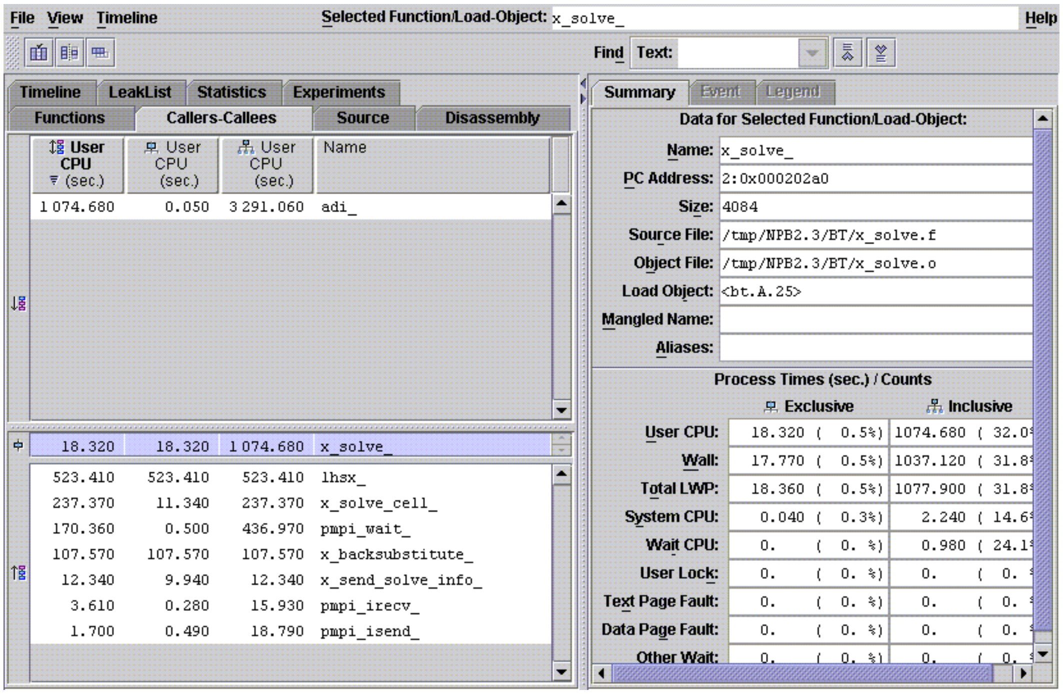
Name:	x_solve_
PC Address:	2:0x000202a0
Size:	4084
Source File:	/tmp/NPB2.3/BT/x_solve.f
Object File:	/tmp/NPB2.3/BT/x_solve.o
Load Object:	<bt.A.25>
Mangled Name:	
Aliases:	

Process Times (sec.) / Counts

	Exclusive	Inclusive
User CPU:	18.320 (0.5%)	1074.680 (32.0
Wall:	17.770 (0.5%)	1037.120 (31.8
Total LWP:	18.360 (0.5%)	1077.900 (31.8
System CPU:	0.040 (0.3%)	2.240 (14.6
Wait CPU:	0. (0. %)	0.980 (24.1
User Lock:	0. (0. %)	0. (0.
Text Page Fault:	0. (0. %)	0. (0.
Data Page Fault:	0. (0. %)	0. (0.
Other Wait:	0. (0. %)	0. (0.

FIGURE 8-3 Performance Analyzer—Source View

To get a better idea of where time is spent at a high level in the code, you can also click on the Callers–Callees tab. FIGURE 8-4 shows a possible Callers–Callee view. The selected function appears in the middle section, its callers appear above it, and its callees below. Selected metrics are shown for all displayed functions. By clicking on callers, you can find where time incurred in the particular function occurs in the source code at a high level. By clicking on callees, you can find more detail on expensive calls a particular function might be making. This sort of analysis is probably familiar to gprof users, but the Performance Analyzer has features that go beyond some of gprof's limitations. For more information about gprof, see "Using the gprof Utility" on page 125.



Selected Function/Load-Object: x_solve_

User CPU (sec.)	User CPU (sec.)	User CPU (sec.)	Name
1 074.680	0.050	3 291.060	adi_
18.320	18.320	1 074.680	x_solve_
523.410	523.410	523.410	lhsx_
237.370	11.340	237.370	x_solve_cell_
170.360	0.500	436.970	pmpi_wait_
107.570	107.570	107.570	x_backsubstitute_
12.340	9.940	12.340	x_send_solve_info_
3.610	0.280	15.930	pmpi_irecv_
1.700	0.490	18.790	pmpi_isend_

Data for Selected Function/Load-Object:

Name:	x_solve_
PC Address:	2:0x000202a0
Size:	4084
Source File:	/tmp/NPB2.3/BT/x_solve.f
Object File:	/tmp/NPB2.3/BT/x_solve.o
Load Object:	<bt.A.25>
Mangled Name:	
Aliases:	

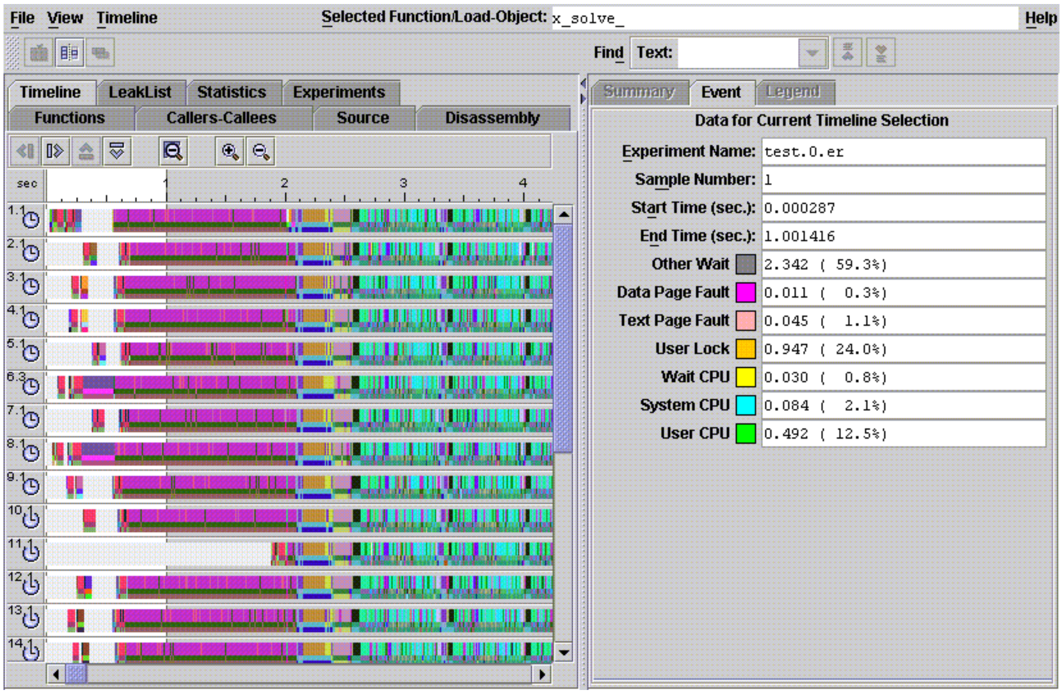
Process Times (sec.) / Counts

	Exclusive	Inclusive
User CPU:	18.320 (0.5%)	1074.680 (32.0
Wall:	17.770 (0.5%)	1037.120 (31.8
Total LWP:	18.360 (0.5%)	1077.900 (31.8
System CPU:	0.040 (0.3%)	2.240 (14.6
Wait CPU:	0. (0. %)	0.980 (24.1
User Lock:	0. (0. %)	0. (0.
Text Page Fault:	0. (0. %)	0. (0.
Data Page Fault:	0. (0. %)	0. (0.
Other Wait:	0. (0. %)	0. (0.

FIGURE 8-4 Performance Analyzer—Callers–Callees View

Different metrics can be selected for the various displays by clicking on the Metrics button, as seen at the bottom of FIGURE 8-1. You can choose which metrics are shown, the order in which they are shown, and which metric should be used for sorting.

Click on the Timeline tab to see a view similar to the one shown in FIGURE 8-5. Time is shown across the horizontal axis, while experiments appear on the vertical axis. Experiments are numbered sequentially, starting from 1. For example, MPI rank 7 might appear as experiment 8. The timeline view helps you see synchronization points, load imbalances, and other performance characteristics of your parallel MPI program.



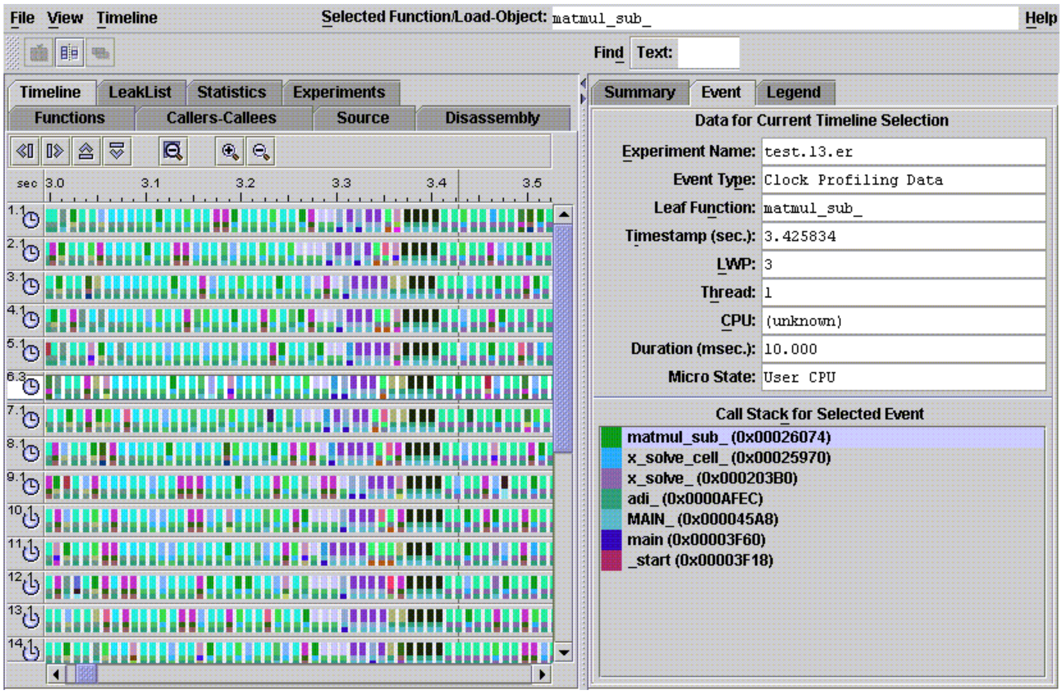
Selected Function/Load-Object: x_solve_

Data for Current Timeline Selection

Experiment Name:		test.0.er
Sample Number:		1
Start Time (sec.):		0.000287
End Time (sec.):		1.001416
Other Wait	⬛	2.342 (59.3%)
Data Page Fault	🟪	0.011 (0.3%)
Text Page Fault	🟥	0.045 (1.1%)
User Lock	🟧	0.947 (24.0%)
Wait CPU	🟨	0.030 (0.8%)
System CPU	🟦	0.084 (2.1%)
User CPU	🟩	0.492 (12.5%)

FIGURE 8-5 Performance Analyzer—Timeline View

In FIGURE 8-6, another timeline view is shown. Click on an event to see its callstack in the lower right corner. When you zoom in to the time scale of the clock sampling (10 milliseconds by default), clock-sampled events appear discretely on the view. Using MPI tracing helps maintain resolution even to the highest levels of zoom.



Selected Function/Load-Object: matmul_sub_

Text:

Data for Current Timeline Selection

Experiment Name:	test.13.er
Event Type:	Clock Profiling Data
Leaf Function:	matmul_sub_
Timestamp (sec.):	3.425834
LWP:	3
Thread:	1
CPU:	(unknown)
Duration (msec.):	10.000
Micro State:	User CPU

Call Stack for Selected Event

- matmul_sub_ (0x00026074)
- x_solve_cell_ (0x00025970)
- x_solve_ (0x000203B0)
- adi_ (0x0000AFEC)
- MAIN_ (0x000045A8)
- main (0x00003F60)
- _start (0x00003F18)

FIGURE 8-6 Performance Analyzer—Timeline View With Callstack

Because clocks on different nodes of a cluster are not guaranteed to be synchronized, and because the Performance Analyzer makes no attempt to synchronize time stamps from different nodes, timeline views of Sun MPI runs that were distributed over multiple nodes in a cluster are not guaranteed to be displayed properly.

Overview of Functions

In a profile, you will typically find many unfamiliar functions that do not appear explicitly in your code. Further, it can happen that none of the familiar MPI calls you do use will appear, such as `MPI_Isend()`, `MPI_Irecv()`, `MPI_Wait()`, or `MPI_Waitall()`.

FIGURE 8-7 shows examples of functions you might find in your profiles, along with explanations of what you are seeing. The functions are organized by load object, such as an executable (here, a.out) or a dynamic library.

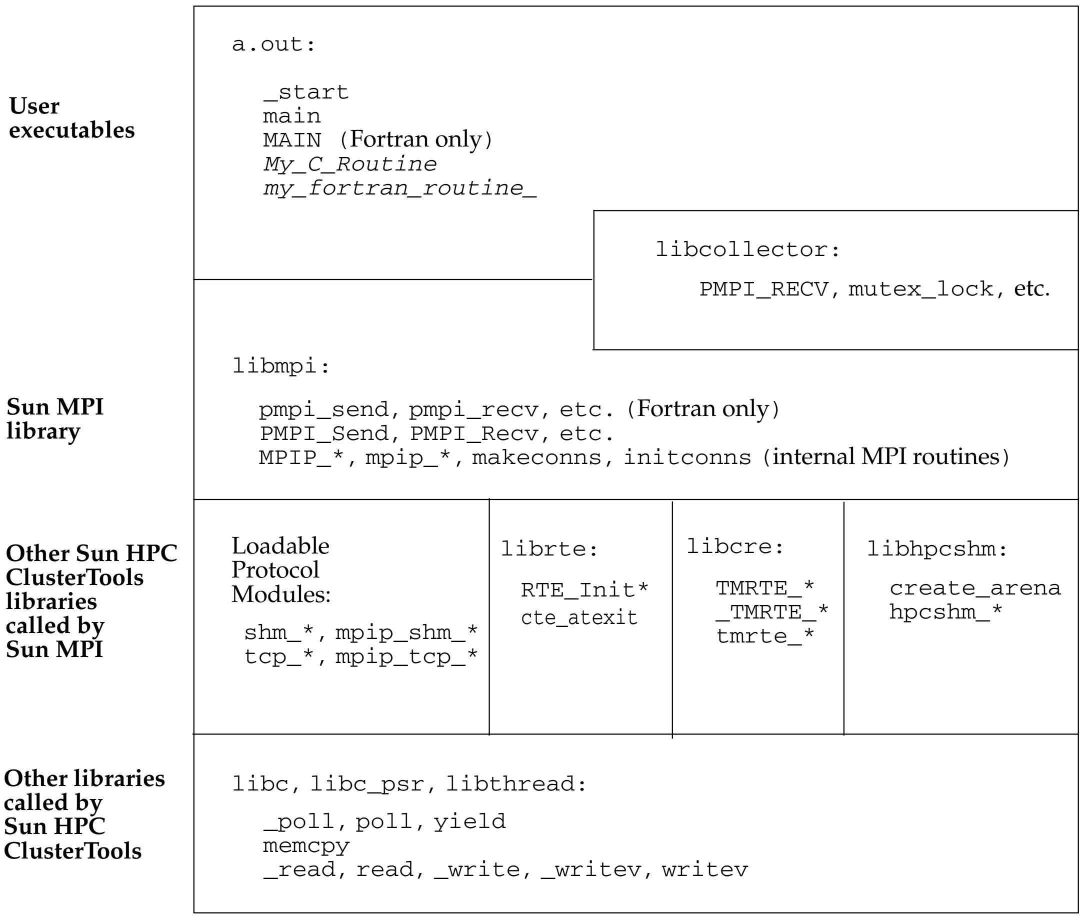

User executables
a.out:
_start
main
MAIN (Fortran only)
My_C_Routine
my_fortran_routine_
libcollector:
PMPI_RECV, mutex_lock, etc.
Sun MPI library
libmpi:
pmpi_send, pmpi_recv, etc. (Fortran only)
PMPI_Send, PMPI_Recv, etc.
MPIP_*, mpip_*, makeconns, initconns (internal MPI routines)
Other Sun HPC ClusterTools libraries called by Sun MPI
Loadable Protocol Modules:
shm_*, mpip_shm_*
tcp_*, mpip_tcp_*
librte:
RTE_Init*
cte_atexit
libcre:
TMRTE_*
_TMRTE_*
tmrte_*
libhpcshm:
create_arena
hpcshm_*
Other libraries called by Sun HPC ClusterTools
libc, libc_psr, libthread:
_poll, poll, yield
memcpy
_read, read, _write, _writev, writev

FIGURE 8-7 Examples of Functions That Might Appear in Profiles

Note that:

- The top function is `_start`, which calls the `main` function. In Fortran programs, the `main` function calls the `MAIN_` function.
- By default, the Fortran compiler converts subprogram names to lower case and appends underscores. For example, the Fortran routine `MY_FORTRAN_ROUTINE()` would be converted to `my_fortran_routine_()`.

- The MPI standard defines a *profiling interface*, which provides that MPI_* functions should also be accessible by using the shifted names PMPI_*. In the Sun MPI implementation, this means that all user-callable functions are named with their PMPI_* forms, with a pmpi_* wrapper for Fortran use.

  For example, a C call to MPI_Send() will enter the function PMPI_Send.

  Sun MPI uses a number of internal routines, which will appear in profiles.
- A Fortran call to MPI_SEND() enters the function pmpi_send_, which in turn calls PMPI_Send.
- libcollector intercepts particular calls to the Sun MPI or threads libraries to support synchronization tracing. Thus, functions such as PMPI_Recv and mutex_lock might appear twice in profiles — once belonging to a user-callable library and once belonging to libcollector.
- Sun MPI calls routines in other Sun HPC ClusterTools libraries.
- Various Sun HPC ClusterTools libraries call other standard libraries. Notably:
  - The _poll, poll, and yield calls might be called by a Sun MPI process for waiting.
  - The memcpy call is often called when an MPI process is copying data locally, such as for on-node message passing.
  - The _read, read, _write, _writev, and writev calls are used in off-node message passing over TCP.

One way to get an overview of which MPI calls are being made, and which are most important, is to look for the PMPI entry points into the MPI library. For our case study example:

```
% er_print -function proc-0.er | grep PMPI_
0.050       16.980      PMPI_Wait
0.030        0.810      PMPI_Isend
0.030       16.930      PMPI_Wait
0.020        0.490      PMPI_Irecv
0.           0.         PMPI_Finalize
0.           0.150      PMPI_Init
0.           1.630      PMPI_Waitall
0.           1.630      PMPI_Waitall
%
```

In this example, roughly 20 seconds out of 146.93 seconds profiled are due to MPI calls. The exclusive times are typically small and meaningless. Synchronizing calls, such as PMPI_Wait and PMPI_Waitall, appear twice, once due to libmpi and once to libcollector. Such duplication can be ignored.

If ever there is a question about what role an unfamiliar (but possibly time-consuming) function is playing, within the Performance Analyzer you can:

- Choose Callers–Callees to see which function is calling it.
- Choose Show Summary Metrics from the View menu to see what is displayed under Load Object.
- Choose Select Load Objects Included from the View menu to restrict viewing to functions that belong to specific load objects (such as your executable or `libmpi`).

MPI Wait Times

Time might be spent in MPI calls for a wide variety of reasons. Specifically, much of that time might be spent waiting for some condition (a message to arrive, internal buffering to become available, and so on), rather than in moving data.

While an MPI process waits, it might or might not tie up a CPU. Nevertheless, such wait time probably has a cost in terms of program performance.

Your options for ensuring that wait time is profiled include:

- Have Sun MPI spin a CPU aggressively during wait situations. This option requires turning off coscheduling and turning on spin behavior. Both are off by default. To do this, use the following environment variable settings:

```
% setenv MPI_COSCHED 0
% setenv MPI_SPIN    1
```

- Select wall-clock time, rather than CPU time, as the profiling metric. Profiling metrics are described earlier in this chapter in the case study.

Other Profiling Approaches

Both Sun MPI and the Solaris OS environment offer useful profiling facilities. Using the MPI profiling interface, you can investigate MPI calls. Using your own timer calls, you can profile specific behaviors. Using the Solaris `gprof` utility, you can profile diverse multiprocess codes, including those using MPI.

Using the MPI Profiling Interface

The MPI standard supports a profiling interface, which allows any user to profile either individual MPI calls or the entire library. This interface supports two equivalent APIs for each MPI routine. One has the prefix `MPI_`, while the other has

PMPI_. User codes typically call the MPI_ routines. A profiling routine or library will typically provide wrappers for the MPI_ routines that simply call the PMPI_ ones, with timer calls around the PMPI_ call.

You can use this interface to change the behavior of MPI routines without modifying your source code. For example, suppose you believe that most of the time spent in some collective call, such as MPI_Allreduce, is due to the synchronization of the processes that is implicit to such a call. Then, you can compile a wrapper such as the one that follows, and link it into your code before the –lmpi switch. The effect will be that time profiled by MPI_Allreduce calls will be due exclusively to the MPI_Allreduce operation, with synchronization costs attributed to barrier operations.

```
subroutine MPI_Allreduce(x,y,n,type,op,comm,ier)
integer x(*), y(*), n, type, op, comm, ier
call PMPI_Barrier(comm,ier)
call PMPI_Allreduce(x,y,n,type,op,comm,ier)
end
```

Profiling wrappers or libraries can be used even with application binaries that have already been linked. Refer to the Solaris man page for ld for more information about the environment variable LD_PRELOAD.

You can get profiling libraries from independent sources for use with Sun MPI. An example of a profiling library is included in the multiprocessing environment (MPE) from Argonne National Laboratory. Several external profiling tools can be made to work with Sun MPI using this mechanism. For more information on this library and on the MPI profiling interface, refer to the *Sun MPI Programming and Reference Guide*.

Inserting MPI Timer Calls

Sun HPC ClusterTools implements the MPI timer call MPI_Wtime (demonstrated in the example that follows) with the high-resolution timer gethrtime. If you use MPI_Wtime calls, you should use them to measure sections that last more than several microseconds. Times on different processes are not guaranteed to be synchronized. For information about the gethrtime timer, see the gethrtime(3C) man page.

When profiling multiprocess codes, ensure that the timings are not distorted by the asynchrony of the various processes. For this purpose, you usually need to synchronize the processes before starting and before stopping the timer.

In the following example, most processes might accumulate time in the interesting, timed portion, waiting for process 0 (zero) to emerge from uninteresting initialization. This condition would skew your program's timings. For example:

```
CALL MPI_COMM_RANK(MPI_COMM_WORLD,ME,IER)
IF ( ME .EQ. 0 ) THEN
  initialization
END IF
! place barrier here
! CALL MPI_BARRIER(MPI_COMM_WORLD, IER)
T_START = MPI_WTIME()
  timed portion
T_END = MPI_WTIME()
```

When stopping a timer, remember that measurements of elapsed time will differ on different processes. So, execute another barrier before the "stop" timer. Alternatively, use "maximum" elapsed time for all processes.

Avoid timing very small fragments of code. This is good advice when debugging uniprocessor codes, and the consequences are greater with many processors. Code fragments perform differently when timed in isolation. The introduction of barrier calls for timing purposes can be disruptive for short intervals.

Using the `gprof` Utility

The Solaris utility `gprof` might be used for multiprocess codes, such as those that use MPI. Several points should be noted:

- Compile and link your programs with the `-pg` (Fortran) or `-xpg` (C) switch.
- Use the environment variable `PROFDIR` to profile multiprocess jobs, such as those that use MPI.
- Use the `gprof` command after program execution to gather summary statistics either on individual processes or for multiprocess aggregates.

Note, however, that the `gprof` utility has several limitations.

- The `gprof` utility requires recompilation and relinking.
- Many libraries do not have `gprof` versions. For example, activity spent within Sun MPI calls do not appear in `gprof` profiles.
- The `gprof` utility apportions time equally among all callers. For example, assume a matrix-multiply routine is called from one caller for small matrices and an equal number of times from another caller for large matrices. The `gprof` utility attributes time spent in the matrix multiplication equally to both callers, even if the large-matrix operations are substantially more time consuming.

- It does not count time spent in sleeps and yields, which can skew results.
- It loses the relationships between process ids (used to tag profile files) and MPI process ranks.
- Its profiles from different processes might overwrite one another if a multiprocess job spans multiple nodes.

Note that the Performance Analyzer is simple to use, provides the profiling information that `gprof` does, offers additional functionality, and avoids the pitfalls. Thus, `gprof` users are highly encouraged to migrate to the Performance Analyzer for both MPI and non-MPI codes.

For more information about the `gprof` utility, refer to the `gprof` man page. For more information about the Performance Analyzer, refer to the documentation supplied with the Sun Studio Compiler Collections.

APPENDIX **A**

Sun MPI Implementation

This appendix discusses various aspects of the Sun MPI implementation that affect program performance:

Many of these characteristics of the Sun MPI implementation can be tuned at runtime with environment variables, as discussed in Appendix B.

Yielding and Descheduling

In many programs, too much time in MPI routines is spent waiting for particular conditions, such as the arrival of incoming data or the availability of system buffers. This busy waiting costs computational resources, which could be better spent servicing other users' jobs or necessary system daemons.

Sun MPI has a variety of provisions for mitigating the effects of busy waiting. This feature allows MPI jobs to run more efficiently, even when the load of a cluster node exceeds the number of processors it contains. Two methods of avoiding busy waiting are yielding and descheduling:

- *Yielding* – A Sun MPI process can yield its processor with a Solaris system call if it waits busily too long.

- *Descheduling* – Alternatively, a Sun MPI process can deschedule itself. In descheduling, a process registers itself with the "spin daemon" (`spind`), which will poll for the gating condition on behalf of the process. Descheduling consumes fewer resource than having the process poll, because the `spind` daemon can poll on behalf of multiple processes. The process will once again be scheduled either if the `spind` daemon wakes the process in response to a triggering event, or if the process restarts spontaneously, according to a preset timeout condition.

Yielding is less disruptive to a process than descheduling, but descheduling helps free resources for other processes more effectively. As a result of these policies, processes that are tightly coupled can become coscheduled. Yielding and coscheduling can be tuned with Sun MPI environment variables, as described in Appendix B.

Progress Engine

When a process enters an MPI call, Sun MPI might act on a variety of messages. Some of the actions and messages might not pertain to the call at all, but might relate to past or future MPI calls.

To illustrate, consider the following code sequence:

```
computation
CALL MPI_SEND()
computation
CALL MPI_SEND()
computation
CALL MPI_SEND()
computation
```

Sun MPI behaves as one would expect. That is, the computational portion of the program is interrupted to perform MPI blocking send operations, as illustrated in FIGURE A-1.

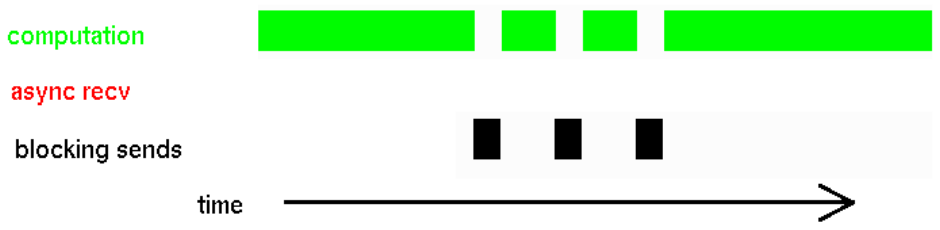


FIGURE A-1 Blocking Sends Interrupt Computation

Now, consider the following code sequence:

```
computation
CALL MPI_IRECV(REQ)
computation
CALL MPI_WAIT(REQ)
computation
```

In this case, the nonblocking receive operation conceptually overlaps with the intervening computation, as in FIGURE A-2.

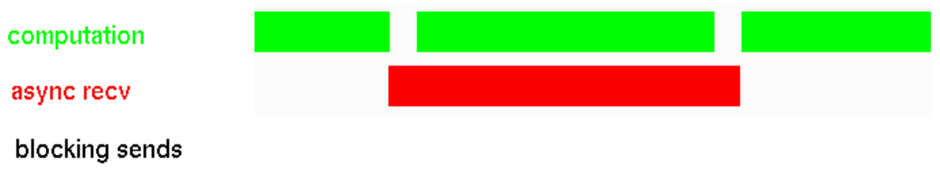


FIGURE A-2 Nonblocking Operations Overlap With Computation

In fact, however, progress on the nonblocking receive is suspended from the time the `MPI_Irecv()` routine returns until the instant Sun MPI is once again invoked, with the the `MPI_Wait()` routine. No actual overlap of computation and communication occurs, and the situation is as depicted in FIGURE A-3.

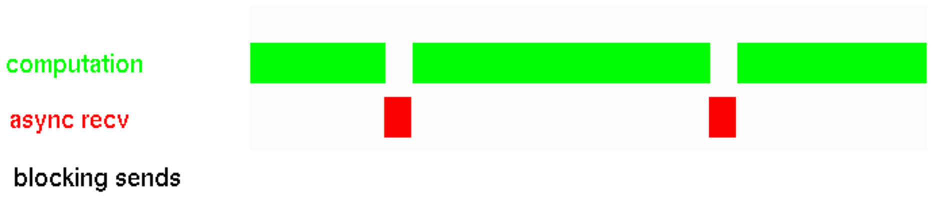


FIGURE A-3 Computational Resources Devoted Either to Computation or to MPI Operations

Nevertheless, reasonably good overlap between computation and nonblocking communication can be realized, given that the Sun MPI library is able to progress a number of message transfers within one MPI call. Consider the following code sequence, which combines the previous examples:

```
computation
CALL MPI_IRECV(REQ)
computation
CALL MPI_SEND()
computation
CALL MPI_SEND()
computation
CALL MPI_SEND()
computation
CALL MPI_WAIT(REQ)
computation
```

Now, there is effective overlap of computation and communication, because the intervening, blocking sends also progress the nonblocking receive, as depicted in FIGURE A-4. The performance payoff is not due to computation and communication happening at the same time. Indeed, a CPU still only computes or else moves data—never both at the same time. Rather, the speed-up results because scheduling of computation with the communication of multiple messages is better interwoven.

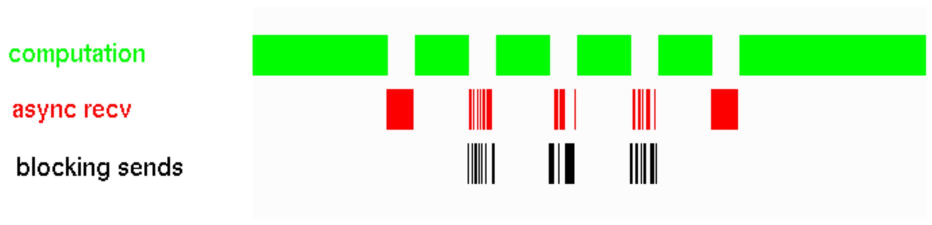


FIGURE A-4 Progress Made on Multiple Messages by a Single MPI Call That Does Not Explicitly Reference the Other Messages

In general, when Sun MPI is used to perform a communication call, a variety of other activities might also take place during that call, as we have just discussed. Specifically:

1. A process might progress any outstanding, nonblocking sends, depending on the availability of system buffers.

2. A process might progress any outstanding, nonblocking receives, depending on the availability of incoming data.

3. A process might generally poll for any messages whatsoever, to drain system buffers.

4. A process must periodically watch for message cancellations from other processes, in case another process issues an `MPI_Cancel()` call for a send.

5. A process might choose to yield its computational resources to other processes, if no useful progress is being made.

6. A process might choose to deschedule itself, if no useful progress is being made.

A nonblocking MPI communication call will return whenever there is no progress to be made. For example, system buffers might be too congested for a send to proceed, or there might not yet be any more incoming data for a receive.

In contrast, a blocking MPI communication call might not return until its operation has completed, even when there is no progress to be made. Such a call will repeatedly try to make progress on its operation, also checking all outstanding nonblocking messages for opportunities to perform constructive work (items 1–4). If these attempts prove fruitless, the process will periodically yield its CPU to other processes (item 5). After multiple yields, the process will attempt to deschedule itself by using the `spind` daemon (item 6).

Shared-Memory Point-to-Point Message Passing

Sun MPI uses a variety of algorithms for passing messages from one process to another over shared memory. The characteristics of the algorithms, as well as the ways in which algorithms are chosen at runtime, can largely be controlled by Sun MPI environment variables, which are described in Appendix B. This section describes the background concepts.

Postboxes and Buffers

For on-node, point-to-point message passing, the sender writes to shared memory and the receiver reads from there. Specifically, the sender writes a message into shared-memory buffers, depositing pointers to those buffers in shared-memory postboxes. As soon as the sender finishes writing any postbox, that postbox, along with any buffers it points to, might be read by the receiver. Thus, message passing is pipelined—a receiver might start reading a long message, even while the sender is still writing it.

FIGURE A-5 depicts this behavior. The sender moves from left to right, using the postboxes consecutively. The receiver follows. The buffers F, G, H, I, J, K, L, and M are still "in flight" between sender and receiver, and they appear out of order. Pointers from the postboxes are required to keep track of the buffers. Each postbox can point to multiple buffers, and the case of two buffers per postbox is illustrated here. Message data is buffered in the labeled areas.

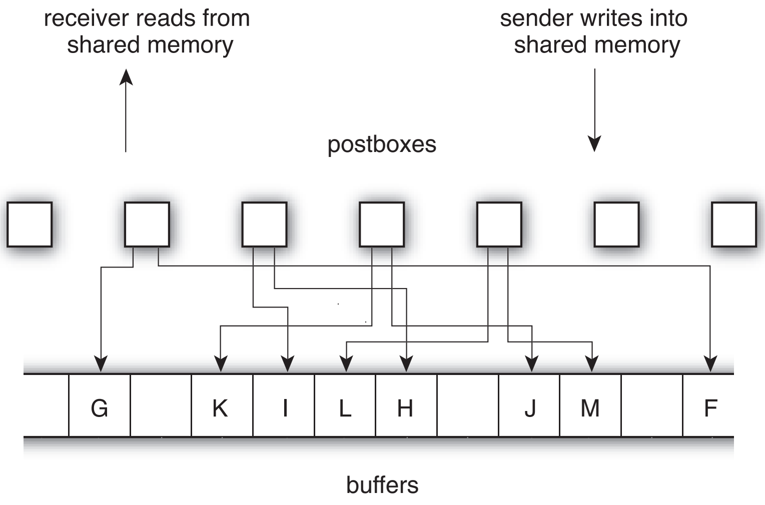

receiver reads from shared memory
sender writes into shared memory
postboxes
G
K
I
L
H
J
M
F
buffers

FIGURE A-5 Snapshot of a Pipelined Message

Pipelining is advantageous for long messages. For medium-size messages, only one postbox is used, and there is effectively no pipelining, as suggested in FIGURE A-6. Message data is buffered in the shaded areas.

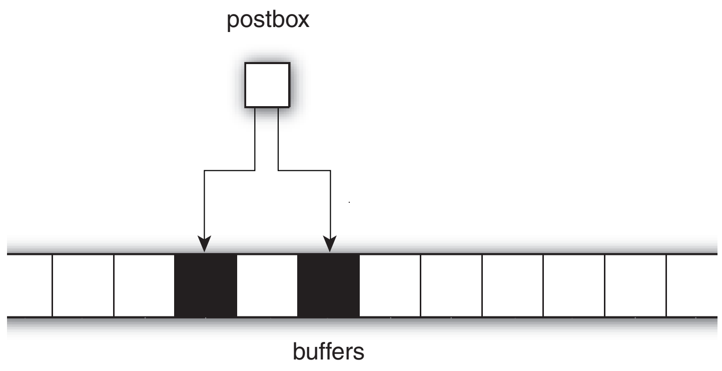


FIGURE A-6 A Medium-Size Message Using Only One Postbox

Further, for extremely short messages, data is squeezed into the postbox itself, in place of pointers to buffers that would otherwise hold the data, as illustrated in FIGURE A-7. Message data is buffered in the shaded area.

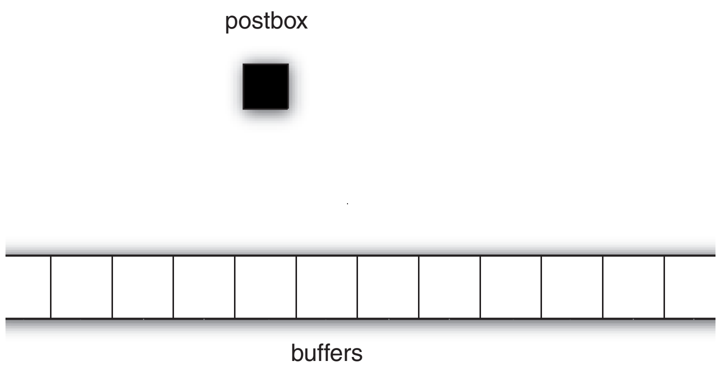


FIGURE A-7 A Short Message Squeezing Data Into the Postbox — No Buffers Used

For very long messages, it might be desirable to keep the message from overrunning the shared-memory area. In that limit, the sender is allowed to advance only one postbox ahead of the receiver. Thus, the footprint of the message in shared memory is limited to at most two postboxes at any one time, along with associated buffers. Indeed, the entire message is cycled through two fixed sets of buffers.

FIGURE A-8 and FIGURE A-9 show two consecutive snapshots of the same cyclic message. The two sets of buffers, through which all the message data is being cycled, are labeled X and Y. The sender remains only one postbox ahead of the receiver. Message data is buffered in the labeled areas.

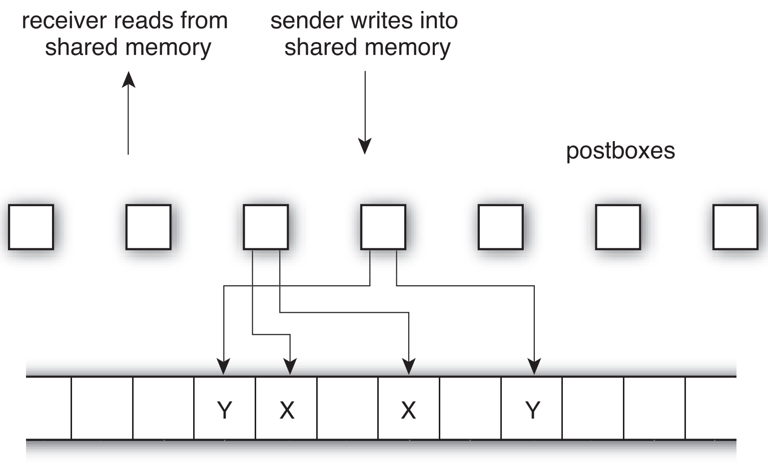

receiver reads from shared memory
sender writes into shared memory
postboxes
Y
X
X
Y

FIGURE A-8 First Snapshot of a Cyclic Message

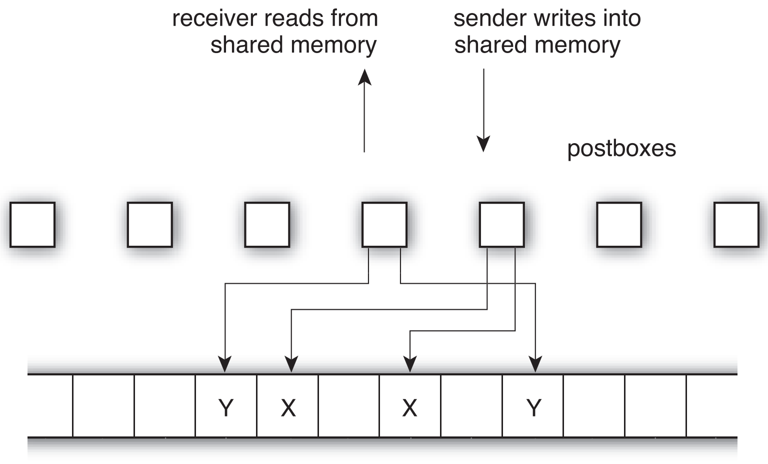

receiver reads from shared memory
sender writes into shared memory
postboxes
Y
X
X
Y

FIGURE A-9 Second Snapshot of a Cyclic Message

Connection Pools Versus Send-Buffer Pools

In the following example, we consider *n* processes that are collocal to a node.

A connection is a sender-receiver pair. Specifically, for *n* processes, there are *n* x (*n*–1) connections. That is, A sending to B uses a different connection than B sending to A, and any process sending to itself is handled separately.

Each connection has its own set of postboxes. For example, in FIGURE A-10, there are two unidirectional connections for each pair of processes. There are 5x4=20 connections in all for the 5 processes. Each connection has shared-memory resources, such as postboxes, dedicated to it. The shared-memory resources available to one sender are shown.

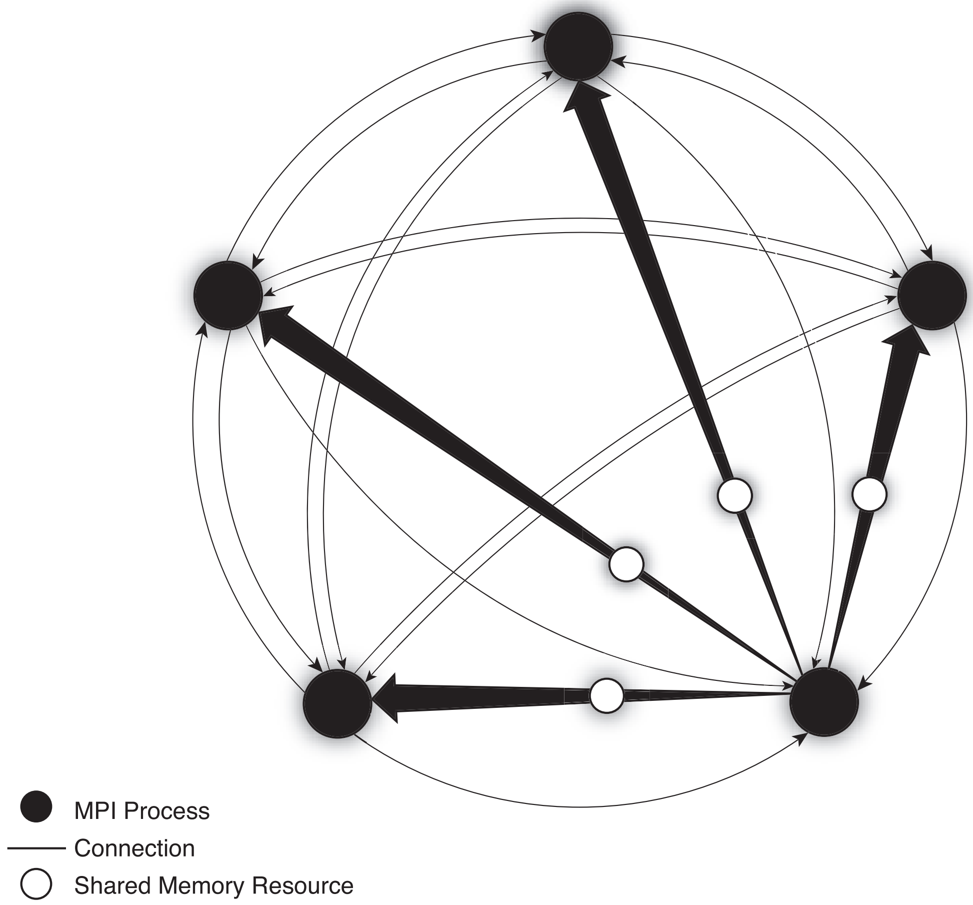


FIGURE A-10 Shared-Memory Resources Dedicated per Connection

By default, each connection also has its own pool of buffers. Users might override the default use of connection pools, however, and cause buffers to be collected into *n* pools, one per sender, with buffers shared among a sender's *n*–1 connections. An illustration of *n* send-buffer pools is shown in FIGURE A-11. The send-buffer pool available to one sender is shown.

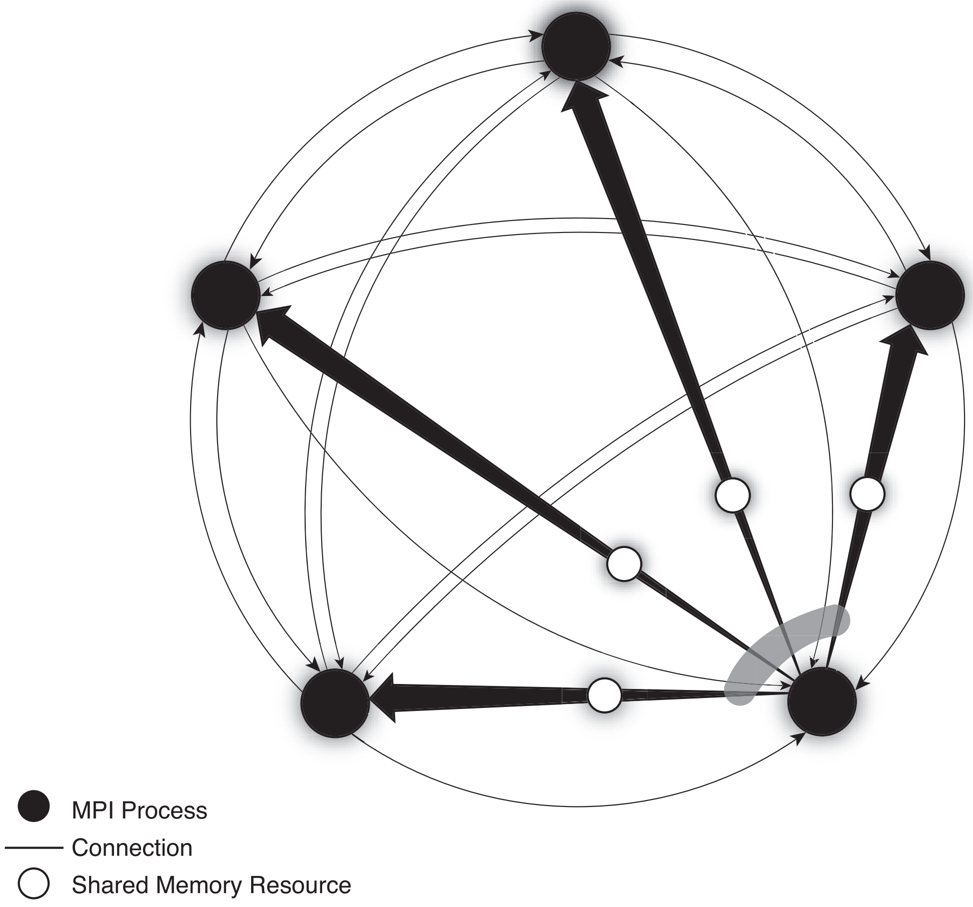


FIGURE A-11 Shared-Memory Resources per Sender — Example of Send-Buffer Pools

Eager Versus Rendezvous

Another issue in passing messages is the use of the rendezvous protocol. By default, a sender will be eager and try to write its message without explicitly coordinating with the receiver (FIGURE A-12). Under the control of environment variables, Sun MPI can employ rendezvous for long messages. Here, the receiver must explicitly indicate readiness to the sender before the message can be sent, as seen in FIGURE A-13.

To force all connections to be established during initialization, set the `MPI_FULLCONNINIT` environment variable:

```
% setenv MPI_FULLCONNINIT 1
```

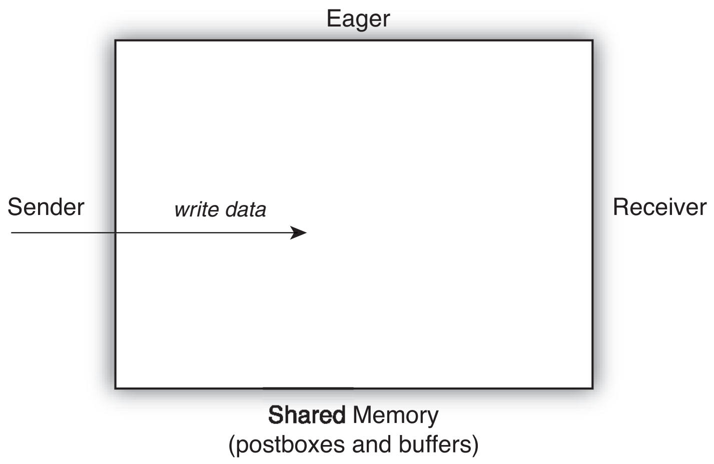


FIGURE A-12 Eager Message-Passing Protocol

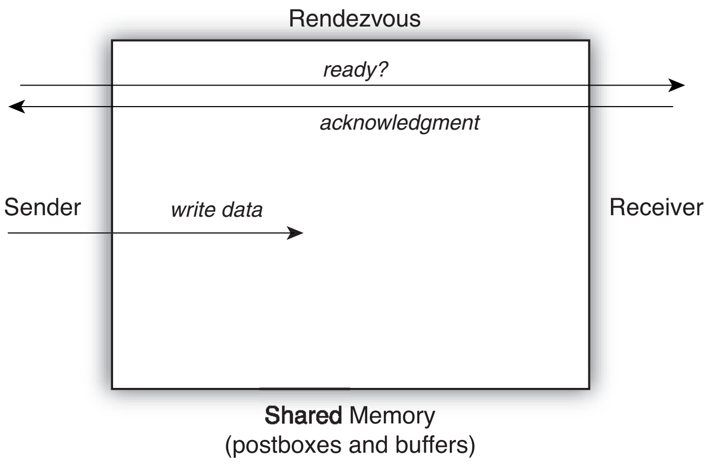


FIGURE A-13 Rendezvous Message-Passing Protocol

Performance Considerations

The principal performance consideration is that a sender should be able to deposit its message and complete its operation without coordination with any other process. A sender might be kept from immediately completing its operation if:

- Rendezvous is in force. (Rendezvous is suppressed by default.)
- The message is being sent cyclically (cyclic message passing is suppressed by default).
- The shared-memory resources (either buffers or postboxes) are temporarily congested. Shared-memory resources can be increased by setting Sun MPI environment variables at runtime to handle any burst of message-passing activity.

Using send-buffer pools, rather than connection pools, helps pool buffer resources among a sender's connections. For a fixed total amount of shared memory, this process can deliver effectively more buffer space to an application, improving performance. Multithreaded applications can suffer, however, because a sender's threads would contend for a single send-buffer pool instead of for (*n*–1) connection pools.

Rendezvous protocol tends to slow performance of short messages, not only because extra handshaking is involved, but especially because it makes a sender stall if a receiver is not ready. Long messages can benefit, however, if there is insufficient memory in the send-buffer pool, or if their receives are posted in a different order than they are sent.

Pipelining can roughly double the point-to-point bandwidth between two processes. It might have little or no effect on overall application performance, however, if processes tend to get considerably out of step with one another, or if the nodal backplane becomes saturated by multiple processes exercising it at once.

Full Versus Lazy Connections

Sun MPI, in default mode, starts up connections between processes on different nodes only as needed. For example, if a 32-process job is started across four nodes, eight processes per node, then each of the 32 processes has to establish 32–8=24 remote connections for full connectivity. If the job relied only on nearest-neighbor connectivity, however, many of these 32x24=768 remote connections would be unneeded.

On the other hand, when remote connections are established on an "as needed" basis, startup is less efficient than when they are established en masse at the time of the `MPI_Init()` call.

Timing runs typically exclude warmup iterations and, in fact, specifically run several untimed iterations to minimize performance artifacts in start-up times. Hence, both full and lazy connections perform equally well for most interesting cases.

Optimizations for Collective Operations

Many MPI implementations effect collective operations in terms of individual point-to-point messages. In contrast, Sun MPI exploits special, collective, algorithms to exploit the industry-leading size of Sun servers and their high-performance symmetric interconnects to shared memory. These optimizations are employed for one-to-many (broadcast) and many-to-one (reduction) operations, including barriers. To a great extent, users need not be aware of the implementation details, since the benefits are realized simply by utilizing MPI collective calls. Nevertheless, a flavor for the optimizations is given here through an example.

Consider a broadcast operation on 8 processes. The usual, distributed-memory broadcast uses a binary fan-out, as illustrated in FIGURE A-14.

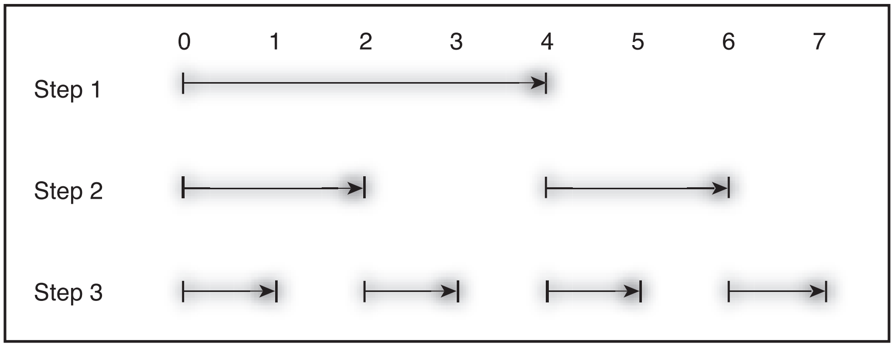


FIGURE A-14 Broadcast With Binary Fan-Out, First Example

In Step 1, the root process sends the data halfway across the cluster. In Step 2, each process with a copy of the data sends a distance one fourth of the cluster. For 8 processes, the broadcast is completed in Step 3. More generally, the algorithm runs somewhat as

$\log_2(NP)$ X time to send the data point-to-point

There are several problems with this algorithm. They are explored in the following sections, and the solutions used in Sun MPI are briefly mentioned. For more information, see the paper *Optimization of MPI Collectives on Clusters of Large-Scale SMPs*, by Steve Sistare, Rolf vandeVaart, and Eugene Loh of Sun Microsystems, Inc. This paper is available at:

`http://portal.acm.org/ft_gateway.cfm?id=331555&type=pdf`

Network Awareness

In a cluster of SMP nodes, the message-passing performance on a node is typically far better than that between nodes.

For a broadcast operation, message passing between nodes in a cluster can be minimized by having each participating node receive the broadcast exactly once. In our example, this optimal performance might be realized if, say, processes 0-3 are on one node of a cluster, while processes 4-7 are on another. This is illustrated in FIGURE A-15.

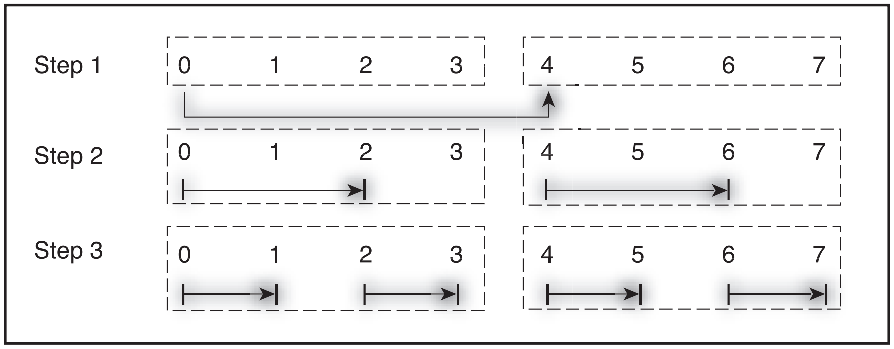


FIGURE A-15 Broadcast With Binary Fan-Out, Second Example

Unless the broadcast algorithm is network aware, however, nodes in the cluster might receive the same broadcast data repeatedly. For instance, if the 8 processes in our example were mapped to two nodes in a round-robin fashion, Step 3 would entail four identical copies of the broadcast data being sent from one node to the other at the same time, as in FIGURE A-16.

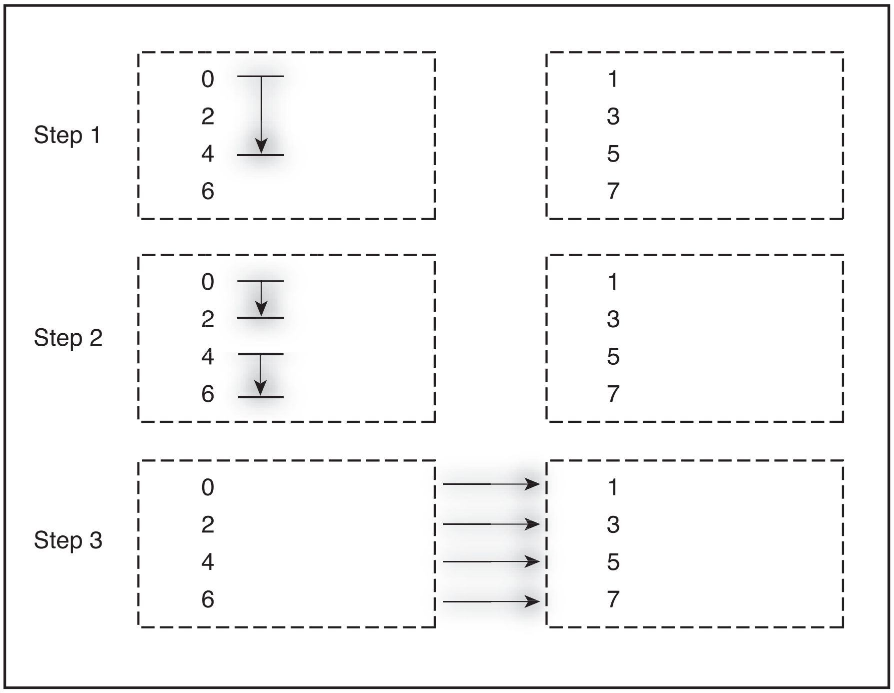


FIGURE A-16 Broadcast With Binary Fan-Out, Third Example

Or, even if the processes were mapped in a block fashion, processes 0-3 to one node and 4-7 to another, if the root process for the broadcast were, say, process 1, excessive internodal data transfers would occur, as in FIGURE A-17.

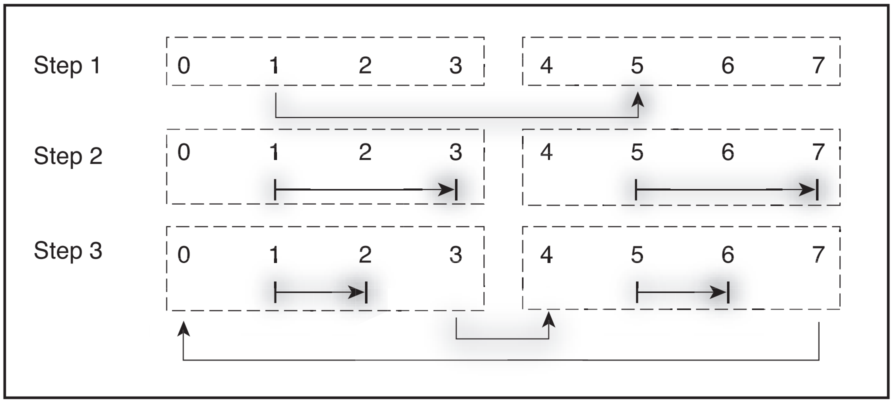


FIGURE A-17 Broadcast With Binary Fan-Out, Fourth Example

In Sun MPI, processes are aware of which other processes in their communication groups are collocal with them. This information is used so that collective operations on clusters do not send redundant information over the internodal network.

Shared-Memory Optimizations

Communication between two processes on the same node in a cluster is typically effected with high performance by having the sender write data into a shared-memory area and the receiver read the data from that area.

While this provides good performance for point-to-point operations, even better performance is possible for collective operations.

Consider, again, the 8-process broadcast example. The use of shared memory can be illustrated as in FIGURE A-18. The data is written to the shared-memory area 7 times and read 7 times.

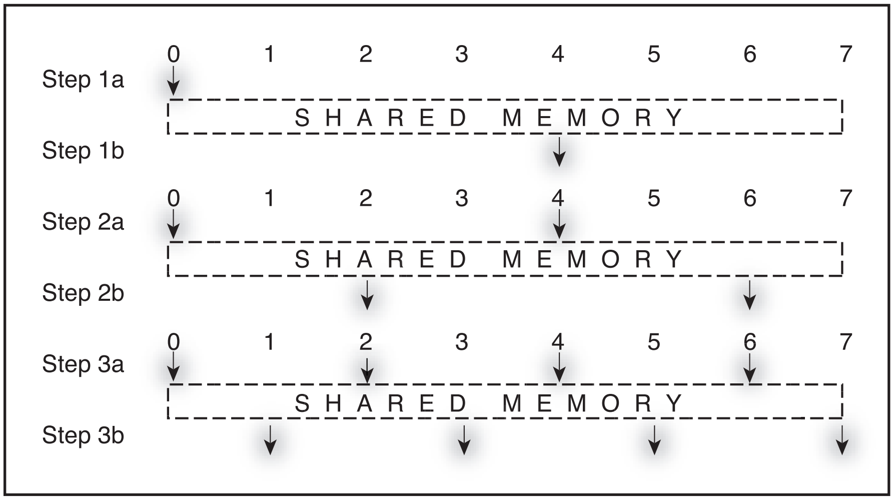


FIGURE A-18 Broadcast Over Shared Memory With Binary Fan-Out, First Case

In contrast, by using special collective shared-memory algorithms, the number of data transfers can be reduced and data made available much earlier to receiving processes, as illustrated in FIGURE A-19. With a collective implementation, data is written only once, and is made available much earlier to most of the processes.

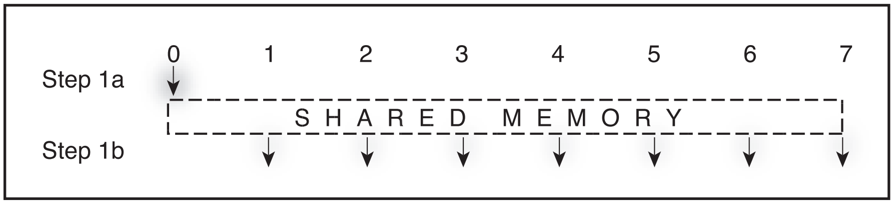


FIGURE A-19 Broadcast Over Shared Memory With Binary Fan-Out, Second Case

Sun MPI uses such special collective shared-memory algorithms. Sun MPI also takes into account possible hot spots in the physical memory of an SMP node. Such hot spots can sometimes occur if, for example, a large number of processes are trying to read simultaneously from the same physical memory, or if multiple processes are sharing the same cache line.

Pipelining

Even in the optimized algorithms discussed in the previous section, there is a delay between the time when the collective operation starts and the time when receiving processes can start receiving data. This delay is especially pronounced when the time to transfer the data is long compared with other overheads in the collective operation.

Sun MPI employs pipelining in its collective operations. This means that a large message might be split into components and different components processed simultaneously. For example, in a broadcast operation, receiving processes can start receiving data even before all the data has left the broadcast process.

For example, in FIGURE A-19, the root (sender) writes into the shared-memory area and then the receiving processes read. If the broadcast message is sufficiently large, the receiving processes might well sit idle for a long time, waiting for data to be written. Further, a lot of shared memory would have to be allocated for the large message. With pipelining, the root could write a small amount of data to the shared area. Then, the receivers could start reading as the root continued to write more. This enhances the concurrency of the writer with the readers and reduces the shared-memory footprint of the operation.

As another example, consider a broadcast among 8 different nodes in a cluster, so that shared-memory optimizations cannot be used. A tree broadcast, such as shown in FIGURE A-14, can be shown schematically as in FIGURE A-20, view a, for a large message. Again, the time to complete this operation grows roughly as

$\log_2(NP) \quad X \quad$ time to send the data point-to-point

In contrast, if the data were passed along a bucket brigade and pipelined, as illustrated in FIGURE A-20, view b, then the time to complete the operation goes roughly as the time to send the data point-to-point. The specifics depend on the internodal network, the time to fill the pipeline, and so on. The basic point remains, however, that pipelining can improve the performance of operations involving large data transfers.

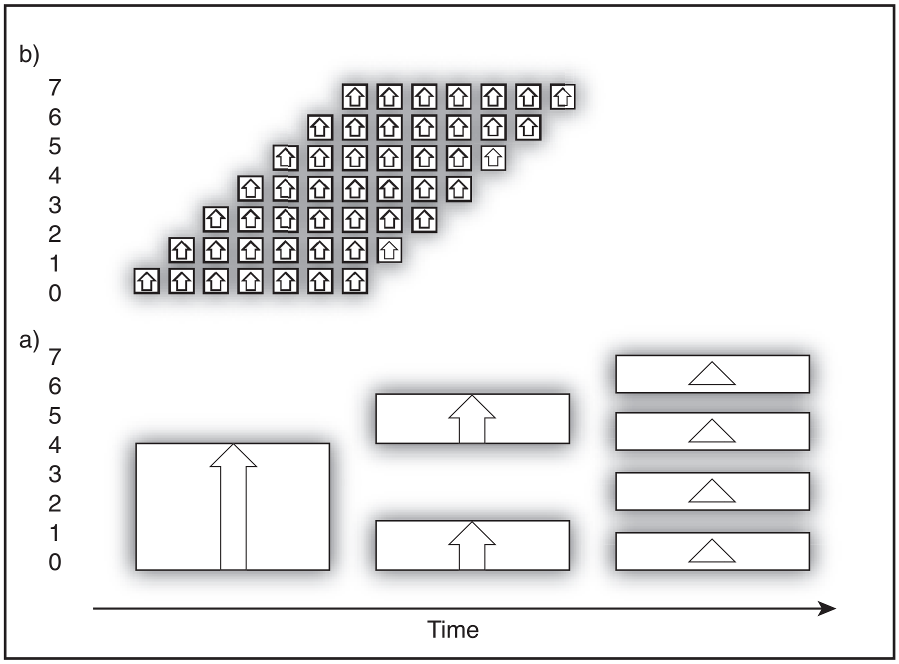


FIGURE A-20 Tree Broadcast versus Pipelined Broadcast of a Large Message

Multiple Algorithms

In practice, multiple algorithms are used to optimize any one particular collective operation. For example, network awareness is used to detect which processes are collocal on a node. Communications between a node might use a particular network algorithm, while collocal processes on a node would use a different shared-memory algorithm. Further, if the data volume is sufficiently large, pipelining might also be used.

Performance models for different algorithms are employed to make runtime choices among the algorithms, based on process group topology, message size, and so on.

One-Sided Message Passing Using Remote Process

An important performance advantage of MPI-2 one-sided communication is that the remote process does not need to be involved. Nevertheless, Sun MPI will invoke the remote process when it is required.

When an MPI rank cannot directly access another rank's address space, Sun MPI must invoke the remote process to effect one-sided transfers. This occurs, for example, when the window was not created with the `MPI_Alloc_mem` routine or when the connection between the ranks uses the TCP protocol module.

There are two ways that the remote process may provide services for one-sided operations:

- The process itself may provide the services. This happens when the process makes an MPI call and the Sun MPI progress engine, described earlier in this appendix, alerts the process that such services are required. The disadvantage of this approach is that the latency for one-sided operations may become high if the remote process makes MPI calls infrequently.
- The process may spawn an agent thread to act on its behalf. While this allows services to be provided even when the remote process is not making any MPI calls, this approach does create an extra thread that may compete with MPI processes for CPUs and other system resources and so oversubscribe the system.

The use of an agent thread may be tuned with `MPI_USE_AGENT_THREAD`.

APPENDIX **B**

Sun MPI Environment Variables

This appendix describes some Sun MPI environment variables and their effects on program performance. It covers the following topics:

Prescriptions for using MPI environment variables for performance tuning are provided in Chapter 7. Additional information on these and other environment variables can be found in the *Sun MPI Programming and Reference Guide*.

These environment variables are closely related to the details of the Sun MPI implementation, and their use requires an understanding of the implementation. More details on the Sun MPI implementation can be found in Appendix A.

Yielding and Descheduling

A blocking MPI communication call might not return until its operation has completed. If the operation has stalled, perhaps because there is insufficient buffer space to send or because there is no data ready to receive, Sun MPI will attempt to progress other outstanding, nonblocking messages. If no productive work can be performed, then in the most general case Sun MPI will yield the CPU to other processes, ultimately escalating to the point of descheduling the process by means of the `spind` daemon.

Setting `MPI_COSCHED=0` specifies that processes should not be descheduled. This is the default behavior.

Setting `MPI_SPIN`=1 suppresses yields. The default value, 0, allows yields.

Polling

By default, Sun MPI polls generally for incoming messages, regardless of whether receives have been posted. To suppress general polling, use `MPI_POLLALL`=0.

Shared-Memory Point-to-Point Message Passing

The size of each shared-memory buffer is fixed at 1 Kbyte. Most other quantities in shared-memory message passing are settable with MPI environment variables.

For any point-to-point message, Sun MPI will determine at runtime whether the message should be sent via shared memory, remote shared memory, or TCP. The flowchart in FIGURE B-1 illustrates what happens if a message of B bytes is to be sent over shared memory.

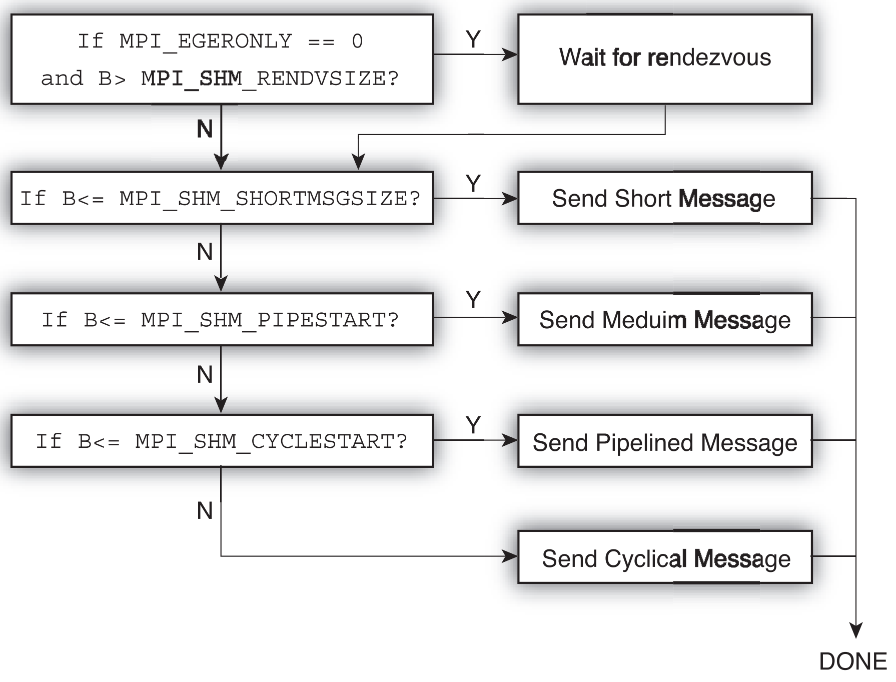

If MPI_EGERONLY == 0 and B> MPI_SHM_RENDVSIZE?
Y
Wait for rendezvous
N
If B<= MPI_SHM_SHORTMSGSIZE?
Y
Send Short Message
N
If B<= MPI_SHM_PIPESTART?
Y
Send Meduim Message
N
If B<= MPI_SHM_CYCLESTART?
Y
Send Pipelined Message
N
Send Cyclical Message
DONE

FIGURE B-1 Message of B Bytes Sent Over Shared Memory

For pipelined messages, `MPI_SHM_PIPESIZE` bytes are sent under the control of any one postbox. If the message is shorter than 2 x `MPI_SHM_PIPESIZE` bytes, the message is split roughly into halves.

For cyclic messages, `MPI_SHM_CYCLESIZE` bytes are sent under the control of any one postbox, so that the footprint of the message in shared memory buffers is 2 x `MPI_SHM_CYCLESIZE` bytes.

The postbox area consists of `MPI_SHM_NUMPOSTBOX` postboxes per connection.

By default, each connection has its own pool of buffers, each pool of size `MPI_SHM_CPOOLSIZE` bytes.

By setting `MPI_SHM_SBPOOLSIZE`, users can specify that each sender has a pool of buffers, each pool having `MPI_SHM_SBPOOLSIZE` bytes, to be shared among its various connections. If `MPI_SHM_CPOOLSIZE` is also set, then any one connection might consume only that many bytes from its send-buffer pool at any one time.

Memory Considerations

In all, the size of the shared-memory area devoted to point-to-point messages is

n x (*n* – 1) x (`MPI_SHM_NUMPOSTBOX` x (64 + `MPI_SHM_SHORTMSGSIZE`) + `MPI_SHM_CPOOLSIZE`)

bytes when per-connection pools are used (that is, when `MPI_SHM_SBPOOLSIZE` is not set), and

n x (*n* – 1) x `MPI_SHM_NUMPOSTBOX` x (64 + `MPI_SHM_SHORTMSGSIZE`) + *n* x `MPI_SHM_SBPOOLSIZE`

bytes when per-sender pools are used (that is, when `MPI_SHM_SBPOOLSIZE` is set).

Performance Considerations

A sender should be able to deposit its message and complete its operation without waiting for any other process. You should typically:

- Use the default setting of `MPI_EAGERONLY`, or set `MPI_SHM_RENDVSIZE` to be larger than the greatest number of bytes any on-node message will have.
- Use the default setting of `MPI_SHM_CYCLESTART`.
- Increase `MPI_SHM_CPOOLSIZE` to ensure sufficient buffering at all times.

In theory, rendezvous can improve performance for long messages if their receives are posted in a different order than their sends. In practice, the right set of conditions for overall performance improvement with rendezvous messages is rarely met.

Send-buffer pools can be used to provide reduced overall memory consumption for a particular value of `MPI_SHM_CPOOLSIZE`. If a process will only have outstanding messages to a few other processes at any one time, then set `MPI_SHM_SBPOOLSIZE` to the number of other processes times `MPI_SHM_CPOOLSIZE`. Multithreaded applications might suffer, however, since then a sender's threads would contend for a single send-buffer pool instead of for multiple, distinct connection pools.

Pipelining, including for cyclic messages, can roughly double the point-to-point bandwidth between two processes. This is a secondary performance effect, however, since processes tend to get considerably out of step with one another, and since the nodal backplane can become saturated with multiple processes exercising it at the same time.

Restrictions

- The short-message area of a postbox must be large enough to point to all the buffers it commands. In practice, this restriction is relatively weak since, if the buffer pool is not too fragmented, the postbox can point to a few, large, contiguous regions of buffer space. In the worst case, however, a postbox will have to point to many disjoint, 1-Kbyte buffers. Each pointer requires 8 bytes, and 8 bytes of the short-message area are reserved. Thus, to avoid runtime errors

  (`MPI_SHM_SHORTMSGIZE` – 8) x 1024 / 8

  should be at least as large as

  ```
  max(
   MPI_SHM_PIPESTART,
   MPI_SHM_PIPESIZE,
   MPI_SHM_CYCLESIZE)
  ```

- If a connection-pool buffer is used, it must be sufficiently large to accommodate the minimum footprint any message will ever require. This means that to avoid runtime errors, `MPI_SHM_CPOOLSIZE` should be at least as large as

  ```
  max(
   MPI_SHM_PIPESTART,
   MPI_SHM_PIPESIZE,
   2 x MPI_SHM_CYCLESIZE)
  ```

- If a send-buffer pool is used and all connections originating from this sender are moving cyclic messages, there must be at least enough room in the send buffer pool to advance one message:

  `MPI_SHM_SBPOOLSIZE` ≥ ((*np* – 1) + 1) x `MPI_SHM_CYCLESIZE`

- Other restrictions are noted in TABLE B-1.

Shared-Memory Collectives

Collective operations in Sun MPI are highly optimized and make use of a *general buffer pool* within shared memory. `MPI_SHM_GBPOOLSIZE` sets the amount of space available on a node for the "optimized" collectives in bytes. By default, it is set to 20971520 bytes. This space is used by `MPI_Bcast()`, `MPI_Reduce()`, `MPI_Allreduce()`, `MPI_Reduce_scatter()`, and `MPI_Barrier()`, provided that two or more of the MPI processes are on the node.

Memory is allocated from the general buffer pool in three different ways:

- When a communicator is created, space is reserved in the general buffer pool for performing barriers, short broadcasts, and a few other purposes.
- For larger broadcasts, shared memory is allocated out of the general buffer pool. The maximum buffer-memory footprint in bytes of a broadcast operation is set by an environment variable as

  (n / 4) x 2 x `MPI_SHM_BCASTSIZE`

  where n is the number of MPI processes on the node. If less memory is needed than this, then less memory is used. After the broadcast operation, the memory is returned to the general buffer pool.
- For reduce operations,

  n x n x `MPI_SHM_REDUCESIZE`

  bytes are borrowed from the general buffer pool and returned after the operation.

In essence, `MPI_SHM_BCASTSIZE` and `MPI_SHM_REDUCESIZE` set the pipeline sizes for broadcast and reduce operations on large messages. Larger values can improve the efficiency of these operations for very large messages, but the amount of time it takes to fill the pipeline can also increase. Typically, the default values are suitable, but if your application relies exclusively on broadcasts or reduces of very large messages, then you can try doubling or quadrupling the corresponding environment variable using one of the following:

```
% setenv MPI_SHM_BCASTSIZE 65536
% setenv MPI_SHM_BCASTSIZE 131072
% setenv MPI_SHM_REDUCESIZE 512
% setenv MPI_SHM_REDUCESIZE 1024
```

If `MPI_SHM_GBPOOLSIZE` proves to be too small and a collective operation happens to be unable to borrow memory from this pool, the operation will revert to slower algorithms. Hence, under certain circumstances, performance optimization could dictate increasing `MPI_SHM_GBPOOLSIZE`.

Running Over TCP

TCP ensures reliable dataflow, even over los-prone networks, by retransmitting data as necessary. When the underlying network loses a lot of data, the rate of retransmission can be very high, and delivered MPI performance will suffer accordingly. Increasing synchronization between senders and receivers by lowering the TCP rendezvous threshold with `MPI_TCP_RENDVSIZE` might help in certain cases. Generally, increased synchronization will hurt performance, but over a loss-prone network it might help mitigate performance degradation.

If the network is not *lossy*, then lowering the rendezvous threshold would be counterproductive and, indeed, a Sun MPI safeguard might be lifted. For reliable networks, use

```
% setenv MPI_TCP_SAFEGATHER 0
```

to speed `MPI_Gather()` and `MPI_Gatherv()` performance.

Summary Table Of Environment Variables

TABLE B-1 Sun MPI Environment Variables

Name	Units	Range	Default
Informational			
MPI_PRINTENV	(None)	0 or 1	0
MPI_QUIET	(None)	0 or 1	0
MPI_SHOW_ERRORS	(None)	0 or 1	0
MPI_SHOW_INTERFACES	(None)	0 – 3	0
Shared Memory Point-to-Point			
MPI_SHM_NUMPOSTBOX	Postboxes	1	16
MPI_SHM_SHORTMSGSIZE	Bytes	Multiple of 64	256
MPI_SHM_PIPESIZE	Bytes	Multiple of 1024	8192
MPI_SHM_PIPESTART	Bytes	Multiple of 1024	2048
MPI_SHM_CYCLESIZE	Bytes	Multiple of 1024	8192
MPI_SHM_CYCLESTART	Bytes	—	The default value is 0x7fffffff for 32-bit and 0x7fffffffffffffff for 64-bit Operating Systems. That is, by default there is no cyclic message passing.

TABLE B-1 Sun MPI Environment Variables *(Continued)*

Name	Units	Range	Default
MPI_SHM_CPOOLSIZE	Bytes	Multiple of 1024	24576 if MPI_SHM_SBPOOLSIZE is not set MPI_SHM_SBPOOLSIZE if it is set
MPI_SHM_SBPOOLSIZE	Bytes	Multiple of 1024	(Unset)
Shared Memory Collectives			
MPI_SHM_BCASTSIZE	Bytes	Multiple of 128	32768
MPI_SHM_REDUCESIZE	Bytes	Multiple of 64	256
MPI_SHM_GBPOOLSIZE	Bytes	>256	20971520
TCP			
MPI_TCP_CONNTIMEOUT	Seconds	≥0	600
MPI_TCP_CONNLOOP	Occurrences	≥0	0
MPI_TCP_SAFEGATHER	(None)	0 or 1	1
One-Sided Communication			
MPI_USE_AGENT_THREAD	(None)	0 or 1	0
Polling and Flow			
MPI_FLOWCONTROL	Messages	≥0	0
MPI_POLLALL	(None)	0 or 1	1
Dedicated Performance			
MPI_PROCBIND	(None)	0 or 1	0
MPI_SPIN	(None)	0 or 1	0
Full vs. Lazy Connections			
MPI_FULLCONNINIT	(None)	0 or 1	0
Eager vs. Rendezvous			
MPI_EAGERONLY	(None)	0 or 1	1

TABLE B-1 Sun MPI Environment Variables *(Continued)*

Name	Units	Range	Default
`MPI_SHM_RENDVSIZE`	Bytes	≥1	24576
`MPI_TCP_RENDVSIZE`	Bytes	≥1	49152
Collectives			
`MPI_CANONREDUCE`	(None)	0 or 1	0
`MPI_OPTCOLL`	(None)	0 or 1	1
Coscheduling			
`MPI_COSCHED`	(None)	0 or 1	(Unset, or “2”)
`MPI_SPINDTIMEOUT`	Milliseconds	≥0	1000
Handles			
`MPI_MAXFHANDLES`	Handles	≥1	1024
`MPI_MAXREQHANDLES`	Handles	≥1	1024

Index