Sun HPC ClusterTools™ 4 Performance Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 U.S.A.
650-960-1300

Part No. 816-0656-10
August 2001, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Contents

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 and S3L libraries
- Executing parallel applications with either the Sun Cluster Runtime Environment (CRE) or Platform Computing's Load-Sharing Facility Suite (LSF)
- Profiling parallel applications using the Prism™ development environment

How This Book Is Organized

This manual covers the following topics.

- *Quick Reference* - A summary of performance tips
- Chapter 1 - *Introduction: The Sun HPC ClusterTools Solution*
- Chapter 2 - *Choosing Your Programming Model and Hardware*
- Chapter 3 - *Performance Programming* with the Sun MPI (message-passing) library
- Chapter 4 - *Sun S3L Performance Guidelines,* for getting the most from this optimized library of scientific routines
- Chapter 5 - *Compilation and Linking* for top performance
- Chapter 6 - *Runtime Considerations and Tuning*

- Chapter 7 - *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 Solaris Commands

This document may not contain information on basic Solaris™ commands and procedures.

See one or both of the following for this information:

- AnswerBook2™ online documentation for the Solaris Operating Environment
- Other software documentation that you received with your system.

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	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

Shell	Prompt
C shell	`%`
C shell superuser	#
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	#

Related Documentation

Application	Title	Part Number
All	*Sun HPC ClusterTools 4 Product Notes*	816-0647-10
Sun S3L Programming	*Sun S3L 4.0 Programming Guide*	816-0652-10
Sun S3L Programming	*Sun S3L 4.0 Reference Manual*	816-0653-10
Sun MPI Programming	*Sun MPI 5.0 Programming and Reference Guide*	816-0651-10
Sun MPI Programming	*Sun HPC ClusterTools 4 User's Guide*	816-0650-10
Prism	*Prism 6.2 User's Guide*	816-0654-10
Prism	*Prism 6.2 Reference Manual*	816-0655-10

Accessing Sun Documentation Online

The `docs.sun.com`SM web site enables you to access a select group of Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

`http://docs.sun.com`

Ordering Sun Documentation

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at:

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

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number of your document in the subject line of your email.

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

Compilation and linking are discussed in Chapter 5.

- Use Forte™ Developer 6 update 2 for best performance.
- Use the `f90` compiler for all Fortran codes, including Fortran 77.

  See "Compiler Version" on page 73.

- Use the `mpf77`, `mpf90`, `mpcc`, and `mpCC` utilities where possible. Link with `-lmpi`. For programs that use S3L, link with `-ls3l`.

  ```
  % mpf90 -fast -g a.f -lmpi
  % mpcc  -fast -g a.c -ls3l -lmopt
  ```

  See "Using the `mp*` Utilities" on page 73.

- Compile with `-fast`.

  See "`-fast`" 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`

See "`-xarch`" on page 75.

- Compile and link with `-g`.

See "`-g`" on page 75.

- Link with `-lopt` for C programs.
- Compile with `-xdepend`.
- 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.

Sun MPI Environment Variables

The Sun MPI environment variables are discussed in Chapter 6 and Appendix B.

- Especially if you will leave at least one idle processor per node to service system daemons, consider using

```
% setenv MPI_SPIN 1
```

See "Are You Running on a Dedicated System?" on page 79.

- If there are no other MPI jobs running and your job is single-threaded,

```
% setenv MPI_PROCBIND 1
```

See "Are You Running on a Dedicated System?" on page 79.

- Suppress cyclic message passing with

  ```
  % setenv MPI_SHM_CYCLESTART 0x7fffffff
  ```

  or, in a 64-bit environment, with

  ```
  % setenv MPI_SHM_CYCLESTART 0x7fffffffffffffff
  ```

  See "Suppress Cyclic Messages" on page 79.

- If system buffers are used "safely" (that is, code does not rely on unlimited buffering to avoid deadlock)

  ```
  % setenv MPI_POLLALL 0
  ```

  If this setting causes your code to deadlock, try using larger buffers, as noted in the next bullet.

  See "Does the Code Use System Buffers Safely?" on page 80.

- If you are willing to trade memory for performance, increase buffering with

  ```
  % setenv MPI_SHM_SBPOOLSIZE 8000000
  ```

  ```
  % setenv MPI_SHM_NUMPOSTBOX 256
  ```

  See "Are You Willing to Trade Memory for Performance?" on page 80.

- Move certain "warm-up" effects to `MPI_Init()` with

  ```
  % setenv MPI_FULLCONNINIT 1
  ```

  This smooths performance profiles and speeds certain portions of code, but `MPI_Init()` can take up to minutes.

  See "Initializing Sun MPI Resources" on page 82.

- If more runtime diagnostic information is desired,

  ```
  % setenv MPI_PRINTENV 1
  ```

  ```
  % setenv MPI_SHOW_INTERFACES 3
  ```

  ```
  % setenv MPI_SHOW_ERRORS 1
  ```

  See "Is More Runtime Diagnostic Information Needed?" on page 83.

Job Launch on a Multinode Cluster

- Checking Load

	CRE	LSF	UNIX
How high is the load?	% **mpinfo -N**	% **lsload**	% **uptime**
What is causing the load?	% **mpps -e**	% **bjobs -u all**	% **ps -e**

See "Running on a Dedicated System" on page 77.

- 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 "Bisection Bandwidth" on page 84.
- Examples of Job Launch With CRE
  - To run jobs in the background, perhaps from a shell script, use `-n`:

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

    or

    ```
    % 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 85.

- 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 86.

- 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 87.

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

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

  See “Detailed Mapping” on page 87.

- Examples of Job Launch With LSF

  See “Multinode Job Launch Under LSF” on page 88.

  - To eliminate core dumps, use

    ```
    % bsub -C 0 -I -n 32 a.out
    ```

  - To run 32 processes, with each block of consecutive 4 processes run on a distinct node:

    ```
    % bsub –I –n 32 –R "span[ptile=4]" a.out
    ```

MPI Programming Tips

- Minimize number and volume of messages.

  See "Reducing Message Volume" on page 21.

- Reduce serialization and improve load balancing.

  See "Reducing Serialization" on page 22 and "Load Balancing" on page 22.

- 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 22.

- Pay attention to buffering:
  - Do not assume unlimited internal buffering by Sun MPI.
  - Tune Sun MPI environment variables at run time to increase system buffering (see Chapter 6 and Appendix B).
  - 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 23.

- 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 25.

- 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 25.

- Take advantage of MPI collective operations.

See “Sun MPI Collectives” on page 26.

- 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 27.
- 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 27.

Forte Developer Profiling

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

- Before you start:
  - Use the most recent Forte Developer version and update number.
  - Set your path to include Forte Developer software, usually `/opt/SUNWspro/bin`
- Basic usage to collect performance data and analyze results:

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

- To control data volumes:
  - Increase the profiling interval. For example:

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

  - Collect data on only a subset of the MPI processes.
  - Collect data to a local file system, such as `/tmp`, or a file system identified by your system administrator, or by

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

- Data organization:
  - Organize experiments (one per process) into subdirectories (one per multi-process run). Use the commands `er_mv`, `er_rm`, and `er_cp`.
  - If you collect an experiment directly into a directory with the `collect -d` switch, make sure that the directory has already been created and, ideally, that no other experiments already exist in it.
- Data collection with Forte Developer 6 update 1 or later:
  - Use `collect` as a standalone command (`dbx` is no longer required):

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

- Loading data:
  - The performance analyzer accepts experiment names on the command line, such as:

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

  - After the analyzer has been started, use the Experiment menu to Add and Drop individual experiments.
- 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.
  - 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, collect data in the first place with

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

Prism Profiling

The Prism environment offers ease of use with its MPI Performance Analysis. If you accept the default values, Prism's TNF profiling requires no special compilation or linking and no special invocation.

Prism Profiling is discussed in Chapter 7 and in the *Prism User's Guide*.

- Launch a basic profiling session with three commands or menu choices:

Prism Command	Menu Entry
tnfcollection on	Performance : Collection
run	Execute : Run
tnfview	Performance : Display TNF Data

  - Prism's TNF browser (`tnfview`) opens with a timeline view of your profiling data.
  - Note how the events in the window represent elapsed time.
  - Inspect the representation for any obvious structure indicating interprocess synchronization or program behavior.
  - Middle-drag the mouse to zoom the timeline view.

- To Display Profiling Statistics:
  - Click on the graph icon in the timeline window to open the graph window.

    MPI calls appear on the list of Interval Definitions.
  - Identify the MPI calls that consume the most time and what fraction of overall time they account for.

    1. Click on a routine under Interval Definitions.
    2. Click on Create a dataset from this interval definition.

3. Click on the Table tab.

4. Note the time spent under Latency Summation.

5. Repeat steps 1, 2, and 4 for other interesting MPI calls.

- Identify the largest message sizes and which message sizes are responsible for the most time.

  The TNF browser displays byte counts as bytes, sendbytes, or recvbytes. The TNF browser reports byte counts in `_start` or `_end` probes. The TNF browser's byte counts for `MPI_Wait` or `MPI_Test` calls are bytes received (zero bytes usually indicate completions of asynchronous sends).

  1. Click on the the Plot tab—Select values under X axis, Y axis, and Field:, then click on Refresh.

  2. Click on the Table tab—Select values under Group intervals by this data element: (note that the browser may display the last column in hexadecimal format).

  3. Click on the Histogram tab—Select values under Metric and Field:, then click on Refresh.

- To find hot spots:

  1. Click on the Plot tab.

  2. Click on a high-latency event to center the timeline view about a hotspot.

  3. Return to the timeline view.

  4. Navigate about the hotspot using the navigation buttons.

     1. Open the Navigate by list and select current vid.

     2. Click on the arrow icons to move forward and backward.

     3. Read data about selected events in the Event Table.

- To control the volume of profiling data, consider:
  - Scale down the run (reduce the number of iterations or the number of processes).
  - Use larger trace buffers.
  - Selectively enable probes.
  - Profile isolated sections of code by terminating jobs early, by modifying user source code, or by isolating sections at run time.

CHAPTER 1

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 based on the UltraSPARC™ processor, 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 UltraSPARC clusters, 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`

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 2.

Processors

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

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 MBytes.

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. Initial versions are at 600, 750, and 900 MHz, with a published roadmap to 1.5+ GHz. L2 cache sizes are up to 8 MBytes.

Nodes

Nodes (the units of a cluster) may be as small as workstations or workgroup servers. For example, the Sun Blade™ 1000 can have up to two UltraSPARC-III CPUs and 8 GBytes of main memory.

The largest server based on the UltraSPARC-II microprocessor is the Enterprise™ 10000. This popular server hosts up to 64 processors at 400 MHz. It utilizes a 12.5-GByte/s backplane and can support up to 64 GBytes of memory. Thus, a single such server may itself have the power of a reasonable-size cluster, while offering very high bandwidth, low latency, and a single, large, uniform address space to 64 CPUs at once.

At the time of this writing, the largest announced server based on the UltraSPARC-III processor is the Sun Fire™ 6800, which can host up to 24 processors and up to 192 GBytes of memory. The backplane can sustain 9.6 GByte/s of data traffic.

Clusters

SMPs may be clustered by means of any Sun-supported TCP/IP interconnect, such as 100BASE-T Ethernet or ATM.

Individual Sun HPC ClusterTools message-passing applications can have up to 2048 processes running on as many as 64 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 4 software runs under the Solaris 8 (32-bit or 64-bit) Operating Environment.

Another software suite, Sun Forte™ Developer software, can be used to develop shared-memory applications. These may be multithreaded or may be parallelized to some extent during compilation, but they are limited to running within a single SMP. Programmers can use tools from both the Forte Developer suite and the HPC ClusterTools suite to develop distributed-memory applications that also exhibit parallelism (multithreading) within nodes.

The differences between HPC ClusterTools software and Forte Developer software are explored in Chapter 2. The present chapter focuses on describing the capabilities of Sun HPC ClusterTools software.

The ClusterTools suite is layered on the Forte Developer suite, and uses its compilers for C, C++, and Fortran. However, the ClusterTools suite provides specialized versions of development tools for its message-passing programs:

- Sun MPI library of message-passing and I/O routines
- Sun S3L, an optimized scientific subroutine library
- Sun Parallel File System, for use with MPI I/O
- Prism™ graphical development environment for debugging and performance profiling of message-passing programs
- Sun CRE, a runtime environment that manages the resources of a server or cluster to execute message-passing programs

- Sun runtime environment plugins for use with Platform Computing's LSF resource management suite (an alternative to CRE)

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 implements all of the MPI 1.2 standard as well as a significant subset of the MPI 2.0 feature list. In addition, Sun MPI provides the following features:

- Seamless use of different network protocols; for example, code compiled on a Sun HPC system that has a Scalable Coherent Interface (SCI) network can be run without change on a cluster that has an ATM network.
- Multiprotocol, thread-safe support such that MPI picks the fastest available medium for each type of connection (such as shared memory, SCI, or ATM).
- 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 Forte Developer compilers. MPI provides full support for Fortran 77, C, and C++, and basic support for Fortran 90.

Chapter 3 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.

Sun S3L

The Sun Scalable Scientific Subroutine Library (Sun S3L) provides a set of parallel and scalable capabilities and tools that are used widely in scientific and engineering computing. Built on top of MPI, it provides highly optimized implementations of vector and dense matrix operations (level 1, 2, 3 Parallel BLAS), FFT, tridiagonal

solvers, sorts, matrix transpose, and many other operations. Sun S3L also provides optimized versions of a subset of the ScaLAPACK library, along with utility routines to convert between S3L and ScaLAPACK descriptors.

S3L is thread-safe and also supports the multiple instance paradigm, which enables an operation to be applied concurrently to multiple, disjoint data sets in a single call. Sun S3L routines can be called from applications written in F77, F90, C, and C++. This library is described in more detail in Chapter 4.

Sun Parallel File System

The Sun HPC ClusterTools Parallel File System (PFS) component provides high-performance file I/O for MPI applications running in a cluster-based, distributed-memory environment.

PFS files closely resemble UFS files, but they provide significantly higher file I/O performance by striping files across multiple nodes. This means that the time required to read or write a PFS file can be reduced by an amount roughly proportional to the number of file server nodes in the PFS file.

Sun PFS is optimized for the large files and complex data access patterns that are characteristic of HPC applications.

Prism Environment

The Prism environment is the Sun HPC ClusterTools graphical programming environment. It enables you to develop, execute, and debug multithreaded or unthreaded message-passing programs and to visualize data at any stage in the execution of a program.

The Prism environment also supports performance profiling of message-passing programs. The analysis provides an overview of what MPI calls, message sizes, or other characteristics account for the execution time. You can display information about the sort of message-passing activity in different phases of a run, identify time-consuming events, and, with simple mouse clicks, investigate any of them in detail.

The Prism profiling capabilities are described in more detail in Chapter 7. The environment can be used with applications written in Fortran 77, Fortran 90, C, and C++.

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 Operating Environment but enhanced to support multiprocess execution. It provides the tools for configuring and managing clusters, nodes, logical sets of processors (*partitions*), and PFS I/O servers.

Alternatively, Sun HPC message-passing programs can be executed by third-party resource-management software, such as the Load Sharing Facility suite from Platform Computing Corporation.

CHAPTER 2

Choosing Your Programming Model and Hardware

This chapter outlines some points to consider in planning how to go about developing or porting 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 Forte Developer software 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.

As detailed in Chapter 1, the basic Sun HPC ClusterTools programming model is distributed-memory message passing. Such a program executes as a collection of Solaris 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 forego ClusterTools software and utilize only multithreaded parallelism, running it 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 Forte 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 may be because they share a common address space, which is the case with multiple threads of control within a single process. Or, it may result from employing a software mechanism for sharing memory.

  Parallelism that is generated by Forte Developer compilers or programmed as multiple threads requires either a single processor or an SMP. SMP servers give their executing processes equal (*symmetric*) access to their shared memory.

- *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 SMPs. They can also, of course, run on a single SMP or even on a serial processor.

Table 2.1 summarizes these two product suites.

TABLE 2-1 Comparison of Sun Forte Developer and Sun HPC ClusterTools Suites

	Sun Forte Developer Suite	Sun HPC ClusterTools Suite
Target hardware	Any Sun workstation or SMP	Any Sun workstation, SMP, or cluster
Memory model	Shared memory	Distributed memory
Runtime resource manager	Solaris Operating Environment	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, therefore, 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 2.2.

TABLE 2-2 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	HPF (not part of ClusterTools 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 Forte 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? Since Sun SMP servers can be very large—up to 64 processors in the current generation—a single server (and thus shared memory) may be adequate for some purposes. 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 may 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 2-3.

TABLE 2-3 Speedup with Number of Processors

Processors (NP)	Run time (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 example above, 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 may 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 may 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 may increase with added resources.

Still, the news is not all bad. With the high-speed interconnects within and between nodes, as described in Chapter 1, and with the programming techniques described in this manual, your application may well achieve high, and perhaps near linear, speedups for some number of processors. And, in certain situations, you might even achieve superlinear scalability, since 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 run time and amortize the cost of interprocess communication. Table 2-4 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 2-4 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 may exceed the performance benefits of parallelization. In such cases, performance may 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 may not be related to point-to-point bandwidth since the performance of the system can degrade under load (many active processes).

To suggest orders of magnitude, Table 2.5 shows sample values of these parameters for two Sun HPC platforms: a large SMP and a 64-node cluster.

TABLE 2-5 Sample Performance Values for MPI Operations on Two Sun Platforms

Platform	Latency (microseconds)	Bandwidth (Mbyte/s)	Crossover size = lat x bw (bytes)	Platform Bisection bandwidth (Mbyte/s)
SMP E10000 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 64 CPUs in the current generation.

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. Since 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 250-MHz processor would have a peak floating-point performance of 500 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, since 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 since 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 a round fraction, say, one-third or one-fourth, of processor speed. It 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 a 83-MHz system clock. Also, performance speedup from 250-MHz processors to 333-MHz processors, both with the same system speed, 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 E10000 server.

Memory bandwidth is directly related to memory latency. For MPI point-to-point communications, it is useful to think of latency and bandwidth as distinct quantities. For memory access, however, transfers are always in units of whole cache lines, and so latency and bandwidth are coupled.

Memory size is required to support large applications efficiently. While the Solaris operating environment will run applications even when there is insufficient physical memory, such use of 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 30-CPU Sun E6000 server, with a 2.6-Gbyte/s backplane, this means that a large all-to-all operation can run at about 650 Mbyte/s aggregate bandwidth. On a 64-CPU Sun E10000 server, with a 12.5-Gbyte/s backplane, an aggregate 3.1 Gbyte/s bandwidth can be achieved. (Here, bandwidth is the rate at which bytes are either sent or received.)

For cluster performance, the *interconnect* between nodes is typically characterized by its latency and bandwidth. Choices include any network that supports TCP, such as HIPPI, ATM, or Gigabit Ethernet, and the Sun Fire™ high-performance cluster interconnect (when available).

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 (e.g., SBus or 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, typically the latter is 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, may 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 3

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 may work well on one hardware platform, those very tweaks may 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 may be speeded up simply by relinking to a higher-performing library. As examples,

Operations...	may 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
Certain ScaLAPACK routines	linking to Sun S3L

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 5).

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 run time 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:

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, since 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 may induce serialization. In multiprocess programming, serialization may 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, daytime 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 may be changed at run time with Sun MPI environment variables, which are discussed in Chapter 6 "Runtime Considerations and Tuning".

- 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, since the data would be available before it is 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 may also be buffered internally multiple times by an MPI implementation.

There are performance consequences to such buffering. Among them:

- Some degree of synchronization may 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 may 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 6.

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 6 “Runtime Considerations and Tuning”.

Sun MPI environment variables may not be a suitable solution in every case. For example, you may 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 may 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 run time 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 may 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 may 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 may be used to probe status, for example, `MPI_Test`.

Nonblocking operations may 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 pingpong 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 effected 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 may 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 may 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 run time 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 “Sun MPI Implementation”.

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 since 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 may retransmit a runaway volume of data, causing MPI performance to suffer.

For this reason, applications running over TCP may benefit from throttled communications. The following suggestions are likely to increase synchronization and degrade performance. Nonetheless, they may be needed when running over TCP 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, since these routines account for loss-prone networks.
- Set the Sun MPI environment variable `MPI_EAGERONLY` to 0 at run time 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.
- Synchronizing protocols (such as rendezvous or cyclic messaging) should be avoided. Rendezvous is suppressed by default. To suppress cyclic messages, you should specify:

  ```
  % setenv MPI_SHM_CYCLESTART 0x7fffffff
  ```

- 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 3-1 for four processes.

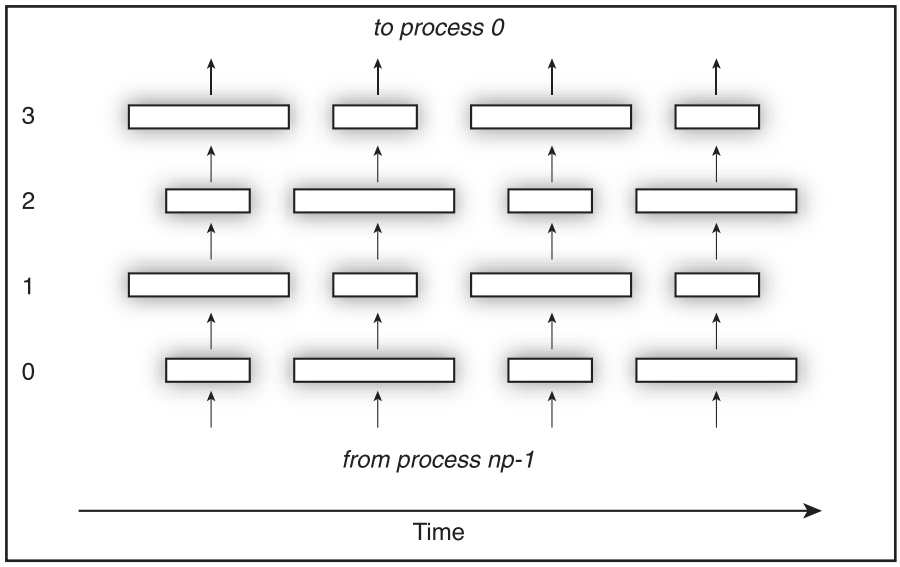


FIGURE 3-1 Basic Ring Sending Algorithm

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 3-2.

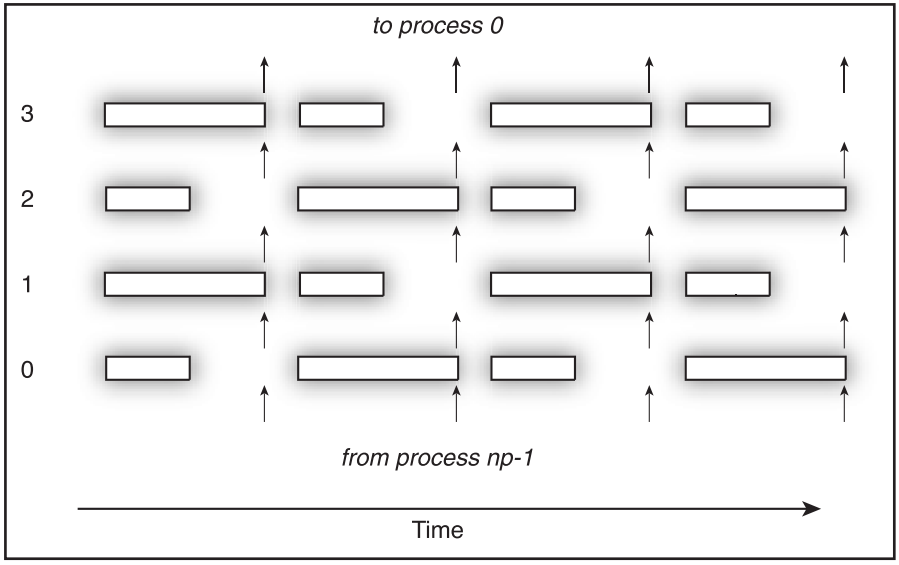


FIGURE 3-2 Basic Ring Sending Algorithm With Synchronization

Several variations on this basic algorithm are used in the timing experiments shown below. What follows is a brief description of each of the five algorithm variations used.

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. Thus, unless the `MPI_Send` call buffers the data, the code will deadlock: everyone will be sending and no one will be receiving. 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.)

TABLE 3-1 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.

TABLE 3-2 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.

TABLE 3-3 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.

TABLE 3-4 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 below in "Use of `MPI_Testall`" on page 48.)

TABLE 3-5 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 above subroutines should be combined with other source code and compiled using

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

TABLE 3-6 shows a sample Fortran 90 program that serves as the driver..

TABLE 3-6 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(6,1000)
niter = 10
n = 0
do while ( n .le. maxn )

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

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

enddo

1000 format("     bytes/msg          sec/iter         Mbyte/sec")

! shut down

call MPI_Finalize(ier)

end




subroutine time_me(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

! 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
```

```
! pump a lot of data through to warm up buffers
subroutine warm_up_buffers(n,x)
include 'mpif.h'
real(8) x(n,*)

! 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
niter = 100 * 1024 * 1024  ! large # bytes, bigger than all buffers
niter = niter / ( 8 * n )  ! convert to number of iterations

! iterate
do i = 1, niter
  call MPI_Sendrecv(x(:,1),n,MPI_REAL8,iup  ,1, &
                    x(:,2),n,MPI_REAL8,idown,1, &
                    MPI_COMM_WORLD,MPI_STATUS_IGNORE,ier)
end do

end
```

Note the following features of the driver program in TABLE 3-6:

- *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/s 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 above 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 E6000 server with a 250-MHz CPU and a 4-Mbyte L2 cache. The command line for program execution is:

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

Baseline Results

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

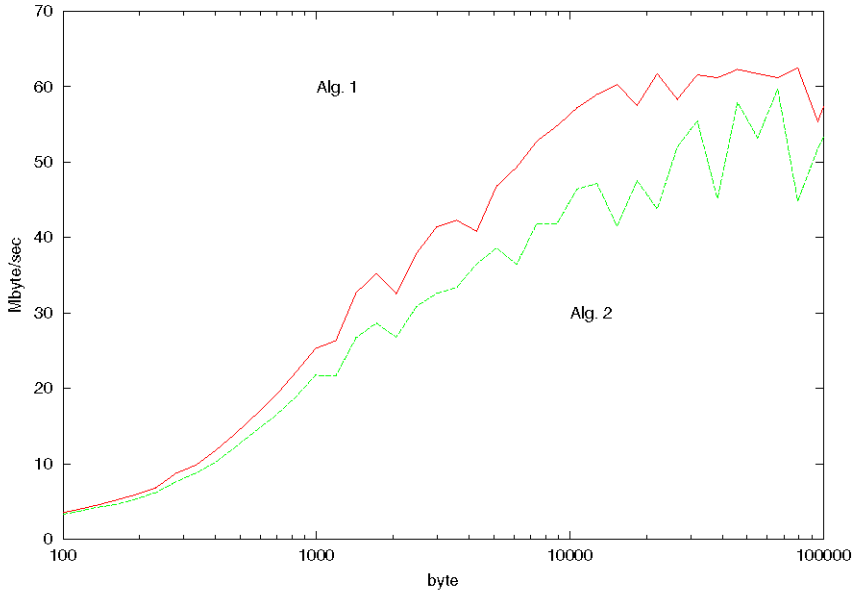


FIGURE 3-3 Bandwidth as a function of message size for Algorithms 1 and 2.

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 24 microseconds per iteration for short messages and about 60 Mbyte/s for long messages. Note that this is not a standard pingpong test, but a rough comparison can be made by noting that each iteration of the test example is roughly equivalent to a pingpong round trip.

Directed Polling

Now, let us rerun this experiment with directed polling. This is effected 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 3-4 shows the resulting bandwidth as a function of message size.

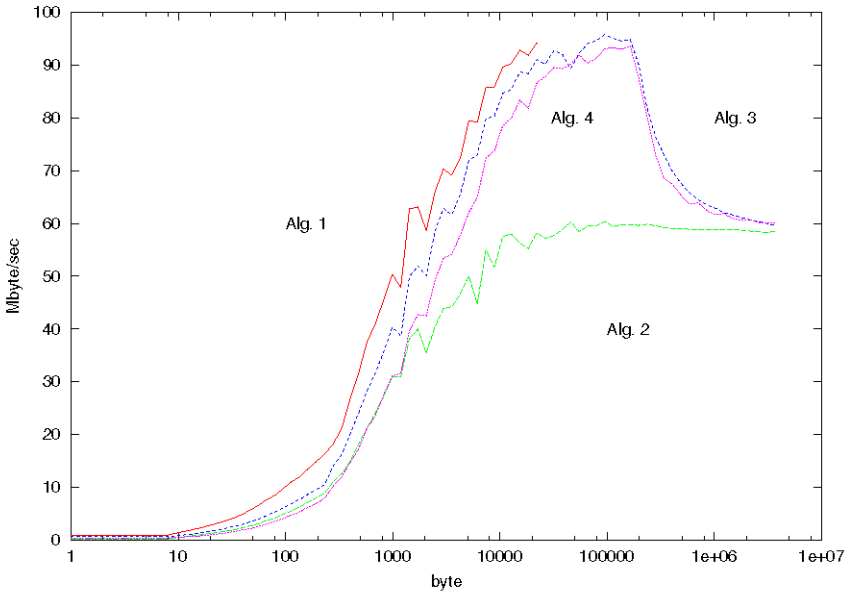


FIGURE 3-4 Bandwidth as a function of message size with directed polling.

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 41), 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 and is the fastest of the three algorithms that avoid deadlock.

Algorithm 4, which employs nonblocking operations, is slightly slower than Algorithm 3.

TABLE 3-7 is a summary of the short-message iteration times and long-message bandwidths for the case of directed polling. Again, these figures are only roughly comparable to standard pingpong tests.

TABLE 3-7 Directed Polling Performance Results

Algorithm	Short message iteration time	Long message bandwidth
1	7-8 usec	(deadlock)
2	13 usec	60 Mbyte/s
3	13 usec	95 Mbyte/s
4	20 usec	93 Mbyte/s

While Algorithm 1 exhibits some desirable performance characteristics, it is not MPI-compliant and depends on internal MPI buffering to avoid deadlock. Among the deadlock-free algorithms, it appears that Algorithm 3, employing `MPI_Sendrecv` calls, achieves the best performance.

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 (TABLE 3-6 on page 37) picks up the command-line arguments (**`1 200`**) to induce an artificial load imbalance among the MPI processes.

FIGURE 3-5 shows bandwidth as a function of message size when nonblocking operations are used.

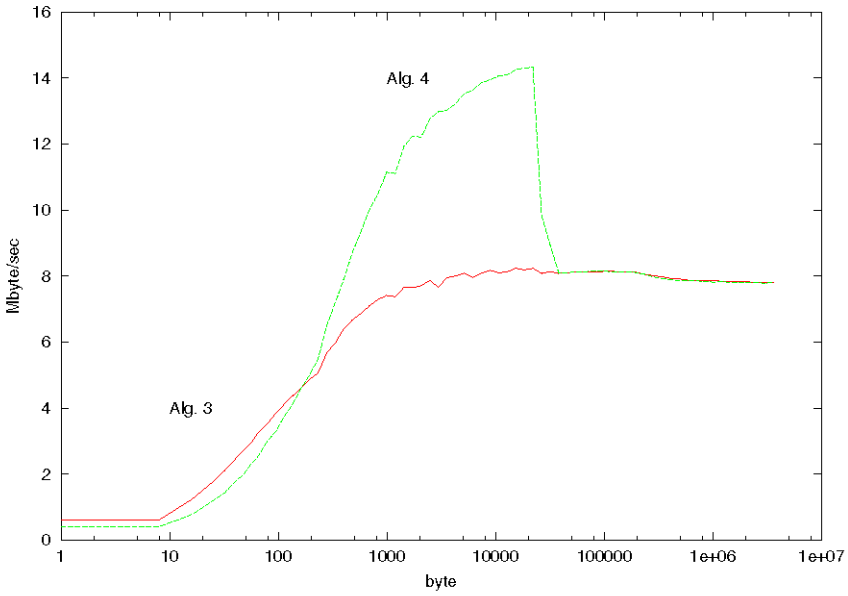


FIGURE 3-5 Bandwidth as a function of message size with nonblocking operations.

While Algorithm 3 (`MPI_Sendrecv`) is faster than 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. Comparison of bandwidths shows that the nonblocking operations of Algorithm 4 are slightly slower than the blocking operations of Algorithm 3 at the shortest message sizes, but the decoupling of processes at longer message sizes improves performance.

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.

This 24-Kbyte limit is actually caused by two things. One is the use of cyclic messages, which is a synchronization between sender and receive to limit the footprint of a message in buffer memory. The onset of cyclic message passing with the SHM PM can be controlled with the Sun MPI environment variable `MPI_SHM_CYCLESTART`, whose default value is 24576. The second cause of the 24-Kbyte limit is the size of the SHM PM buffers. Buffer sizes may 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 may be found in Appendix A. More information about the associated environment variables may be found in Appendix B.

Now, we rerun with:

```
% setenv MPI_POLLALL 0
% setenv MPI_SHM_SBPOOLSIZE 20000000
% setenv MPI_SHM_NUMPOSTBOX 2048
% setenv MPI_SHM_CYCLESTART 0x7fffffff
% mprun -np 4 a.out
% mprun -np 4 a.out 1 200
% unsetenv MPI_POLLALL
% unsetenv MPI_SHM_SBPOOLSIZE
% unsetenv MPI_SHM_NUMPOSTBOX
% unsetenv MPI_SHM_CYCLESTART
```

Here, we have set the environment variables not only to employ direct polling, but also to increase internal buffering and effectively suppress cyclic message passing.

FIGURE 3-6 shows bandwidth as a function of message size for the case of highly synchronized processes (the command line specifying a.out with additional arguments).

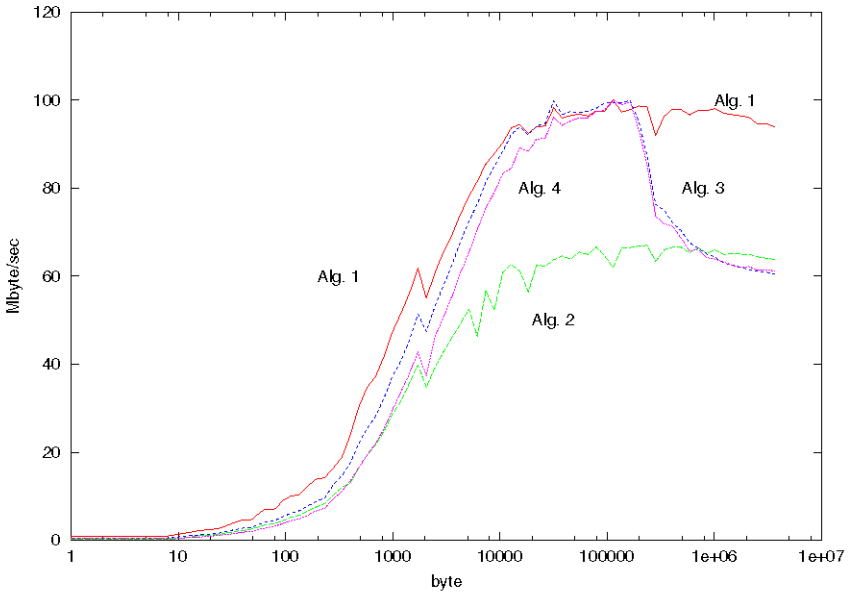


FIGURE 3-6 Bandwidth as a function of message size with highly synchronized processes.

Having increased the buffering and suppressed cyclic message passing, 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 3-7 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.

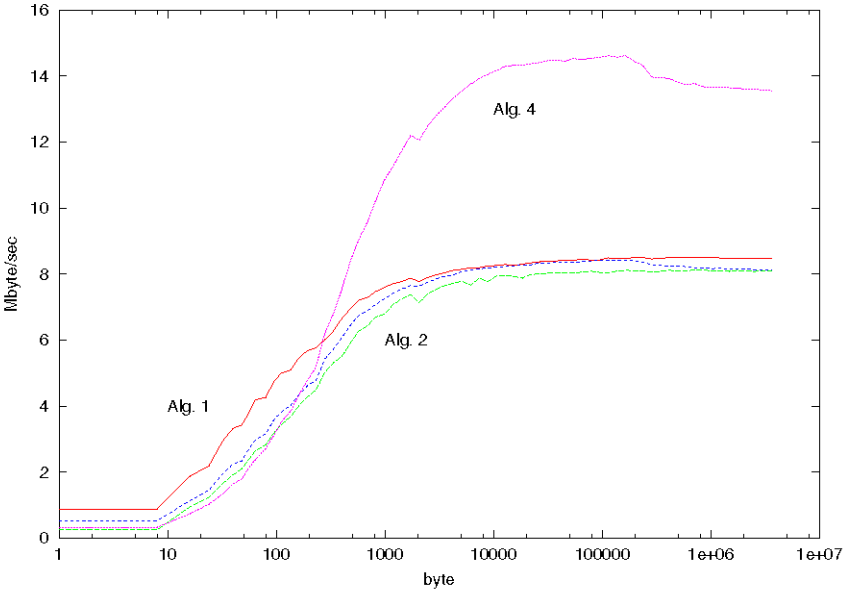


FIGURE 3-7 Bandwidth as a function of message size with load imbalance.

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
% setenv MPI_SHM_CYCLESTART 0x7fffffff
% 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
% unsetenv MPI_SHM_CYCLESTART
```

The third command-line argument to `a.out` specifies, in some way, the amount of computation to be performed between `MPI_Testall` calls. Note also that cyclic message passing is explicitly suppressed.

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 3-8 plots bandwidth against message size for the various frequencies of MPI_Testall calls.

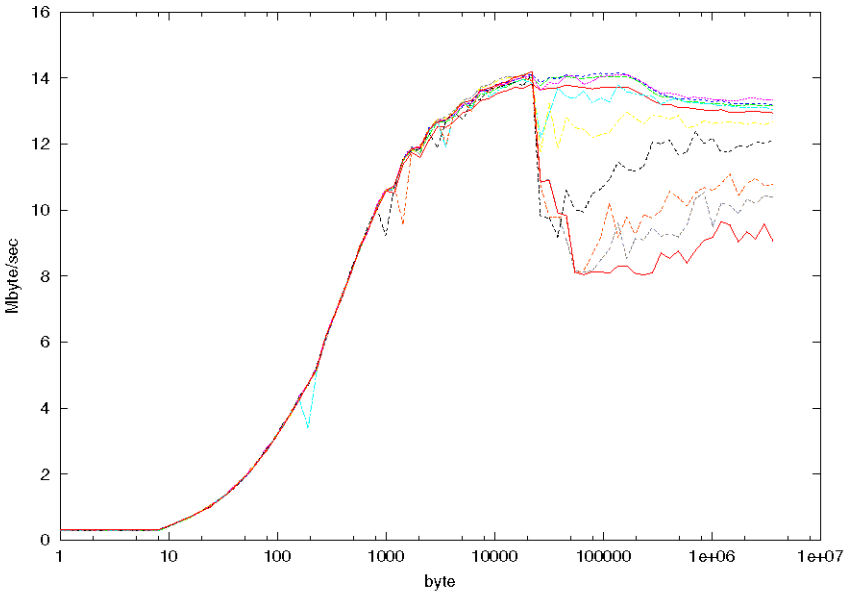


FIGURE 3-8 Bandwidth as a function of message size with `MPI_Testall` calls.

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 3-9. This figure plots bandwidth as a function of the amount of computation performed between `MPI_Testall` calls.

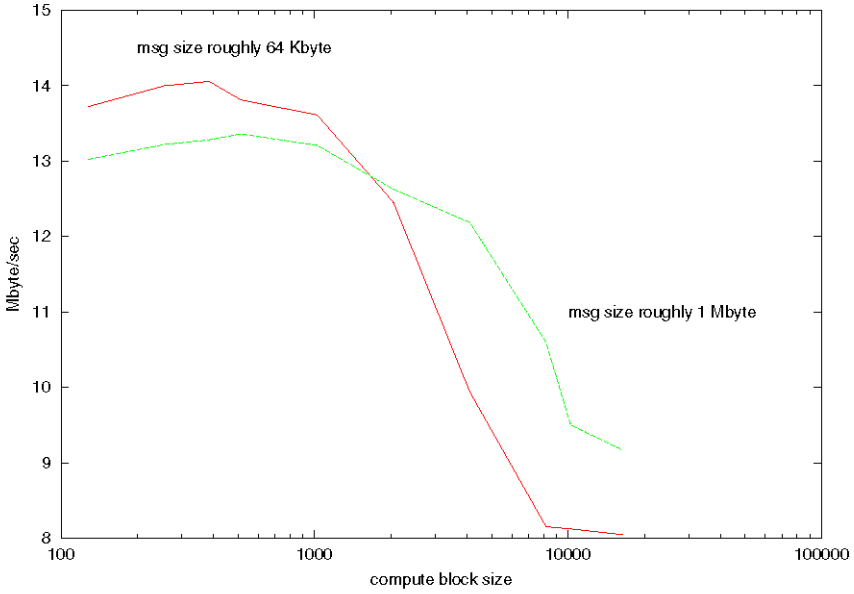


FIGURE 3-9 Bandwidth as a function of computation between `MPI_Testall` calls.

For clarity, FIGURE 3-9 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/s. 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 4

Sun S3L Performance Guidelines

Introduction

This chapter discusses a variety of performance issues as they relate to use of Sun S3L routines. The discussions are organized along the following lines:

- Linking in the Sun Performance Library™
- Using legacy code containing ScaLAPACK calls
- Array distribution
- Process grids
- Runtime mapping to a cluster
- Using shared memory to lower communication costs
- Using smaller data types
- Miscellaneous performance guidelines for individual Sun S3L routines

Link in the Appropriate Version of the Sun Performance Library

Sun S3L relies on functions in the Sun Performance Library (`libsunperf`) for numerous computations within each process. For best performance, make certain your executable uses the architecture-specific version of `libsunperf`. You can do this by linking your program with `-xarch=v8plusa` for 32-bit executables or `-xarch=v9a` for 64-bit executables.

At run time, the environment variable `LD_LIBRARY_PATH` can be used to override link-time library choices. Ordinarily, you should not use this environment variable as it might link a suboptimal library, such as the generic SPARC™ version, rather than one optimized for an UltraSPARC processor.

To unset the `LD_LIBRARY_PATH` environment variable, use

```
% unsetenv LD_LIBRARY_PATH
```

To confirm which libraries will be linked at run time, use

```
% ldd executable
```

If Sun S3L detects that a suboptimal version of `libsunperf` was linked in, it will print a warning message. For example:

```
S3L warning: Using libsunperf not optimized for UltraSPARC.
For better performance, link using –xarch=v8plusa
```

Note – For single-process jobs, most Sun S3L functions call the corresponding Sun Performance Library interface if such an interface exists. Thus, the performance of Sun S3L functions on a single process is usually similar to that of single-threaded Sun Performance Library functions.

Legacy Code Containing ScaLAPACK Calls

Many Sun S3L functions support ScaLAPACK application programming interfaces (APIs). This means you can increase the performance of many parallel programs that use ScaLAPACK calls simply by linking in Sun S3L instead of the public domain software.

Alternatively, you might convert ScaLAPACK array descriptors to S3L array handles and call S3L routines explicitly. By converting the ScaLAPACK array descriptors to the equivalent Sun S3L array handles, you can visualize distributed ScaLAPACK arrays with Prism and use the Sun S3L simplified array syntax for programming. You will also have full use of the Sun S3L toolkit functions.

Sun S3L provides the function `S3L_from_ScaLAPACK_desc` that performs this API conversion for you. See the `S3L_from_ScaLAPACK_desc` man page for details.

Array Distribution

One of the most significant performance-related factors in Sun S3L programming is the distribution of S3L arrays among MPI processes. S3L arrays are distributed, axis by axis, using mapping schemes that are familiar to users of ScaLAPACK or High Performance Fortran. That is, elements along an axis might have any one of the following mappings:

- local – All elements are owned by (that is, local to) the same MPI process.
- block – The elements are divided into blocks with, at most, one block per process.
- cyclic – The elements are divided into small blocks, which are allocated to processes in a round-robin fashion, cycling over processes repeatedly, as needed.

FIGURE 4-1 illustrates these mappings with examples of a one-dimensional array distributed over four processes.

For multidimensional arrays, mapping is specified separately for each axis, as shown in FIGURE 4-2. This diagram illustrates a two-dimensional array's row and column axes being distributed among four processes. Four examples are shown, using a different combination of the three mapping schemes in each. The value represented in each array element is the rank of the process on which that element resides.

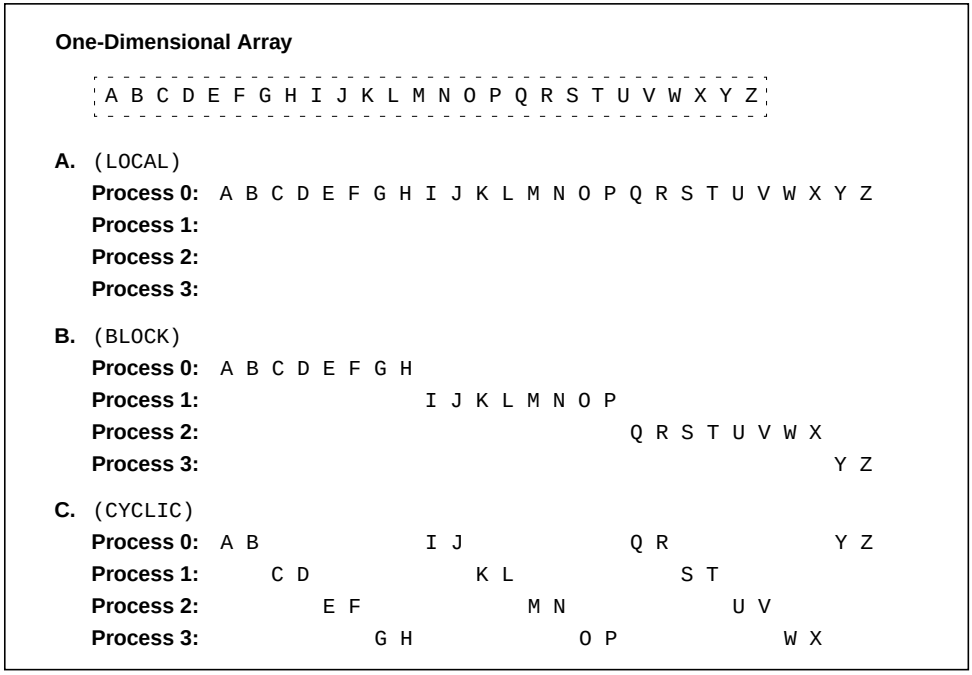


FIGURE 4-1 Array Distribution Examples for a One-Dimensional Array

A. `(LOCAL,BLOCK)`

```
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
```

B. `(LOCAL,CYCLIC)`

```
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
0 0 1 1 2 2 3 3 0 0 1 2 2 2 3 3
```

C. `(BLOCK,BLOCK)`

```
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
0 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3
```

D. `(CYCLIC,CYCLIC)`

```
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
0 0 2 2 0 0 2 2 0 0 2 2 0 0 2 2
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
1 1 3 3 1 1 3 3 1 1 3 3 1 1 3 3
```

NOTE: The value in each array element indicates the rank of the process on which that element resides.

FIGURE 4-2 Array Distribution Examples for a Two-Dimensional Array

In certain respects, local distribution is simply a special case of block distribution, which is just a special case of cyclic distribution. Although related, the three distribution methods can have very different effects on both interprocess communication and load balancing among processes. TABLE 4-1 summarizes the relative effects of the three distribution schemes on these performance components.

TABLE 4-1 Amount of Communication and of Load Balancing with Local, Block, and Cyclic Distribution

	Local	Block	Cyclic
Communication (such as near-neighbor communication)	none (optimal)	some	most (worst)
Load balancing (such as operations on left-half of data set)	none (worst)	some	most (optimal)

The next two sections provide guidelines for when you should use local and cyclic mapping. When none of the conditions describe below apply, use block mapping.

When To Use Local Distribution

The chief reason to use local mapping is that it eliminates certain communication.

The following are two general classes of situations in which local distribution should be used:

- Along a single axis – The detailed versions of the Sun S3L FFT, sort, and grade routines manipulate data only along a single, specified axis. When using the following routines, performance is best when the target axis is local.
  - `S3L_fft_detailed`
  - `S3L_sort_detailed_up`
  - `S3L_sort_detailed_down`
  - `S3L_grade_detailed_up`
  - `S3L_grade_detailed_down`
- Operations that use the multiple-instance paradigm – When operating on a full array using a multiple-instance Sun S3L routine, make data axes local and distribute instance axes. See the chapter on multiple instance in the *Sun S3L Programming Guide*.

When To Use Cyclic Distribution

Some algorithms in linear algebra operate on portions of an array that diminish as the computation progresses. Examples within Sun S3L include LU decomposition (`S3L_lu_factor` and `S3L_lu_solve`), singular value decomposition (`S3L_gen_svd`), and the least-squares solver (`S3L_gen_lsq`). For these Sun S3L routines, cyclic distribution of the data axes improves load balancing.

Choosing an Optimal Block Size

When declaring an array, you must specify the size of the block to be used in distributing the array axes. Your choice of block size not only affects load balancing, it also trades off between concurrency and cache-use efficiency.

Note – Concurrency is the measure of how many different subtasks can be performed at a time. Load balancing is the measure of how evenly the work is divided among the processes. Cache-use efficiency is a measure of how much work can be done without updating cache.

Specifying large block sizes will block multiple computations together. This leads to various optimizations, such as improved cache reuse and lower MPI latency costs. However, blocking computations reduces concurrency, which in turn inhibits parallelization.

A block size of 1 maximizes concurrency and provides the best load balancing. However, small block sizes degrade cache-use efficiency.

Since the goals of maximizing concurrency and cache-use efficiency conflict, you must choose a block size that will produce an optimal balance between them. The following guidelines are intended to help you avoid extreme performance penalties:

- Use the same block size in all dimensions.
- Limit the block size so that data does not overflow the L2 (external) cache. Cache sizes vary, but block sizes should typically not go over 100.
- Use a block size of at least 20 to 24 to allow cache reuse.
- Scale the block size to the size of the matrix. Keep the block size small relative to the size of the matrix to allow ample concurrency.

There is no simple formula for determining an optimal block size that covers all combinations of matrices, algorithms, numbers of processes, and other such variables. The best guide is experimentation, while keeping the points just outlined in mind.

Illustration of Load Balancing

This section demonstrates the load-balancing benefits of cyclic distribution for an algorithm that sums the lower triangle of an array.

The section begins by showing how block distribution results in load imbalance for this algorithm (see FIGURE 4-3). In this example, the array's column axis is block-distributed across processes 0–3. Since process 0 must operate on many more elements than the other processes, total computational time will be bounded by the time it takes process 0 complete. The other processes, particularly process 3, will be idle for much of that time.

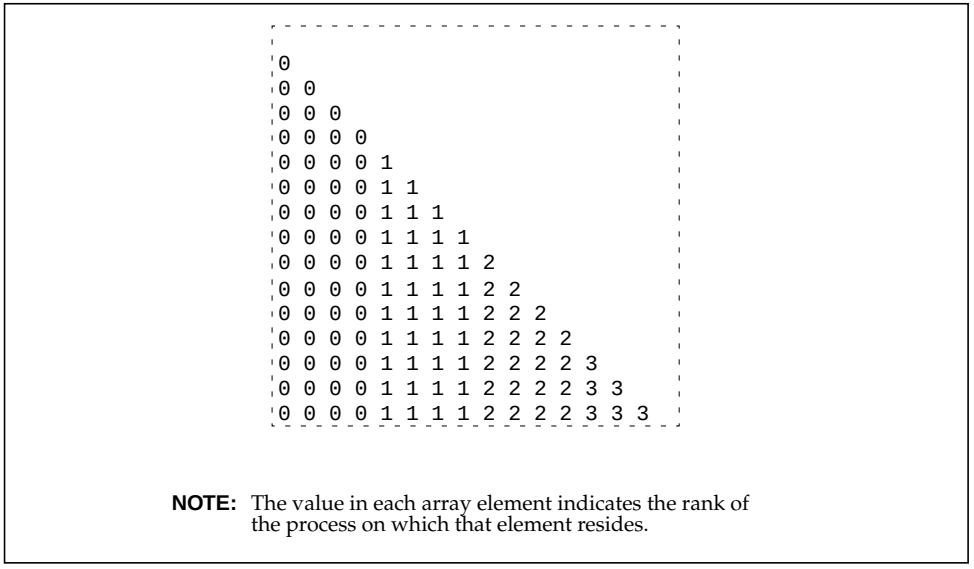
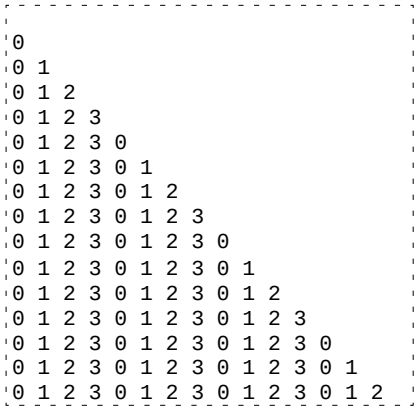


FIGURE 4-3 `LOCAL, BLOCK` Distribution of a 16x16 Array Across Four Processes

FIGURE 4-4 shows how cyclic distribution of the column axis delivers better load balancing. In this case, the axis is distributed cyclically, using a block size of 1. Although process 0 still has more elements to operate on than the other processes, cyclic distribution significantly reduces its share of the array elements.



NOTE: The value in each array element indicates the rank of the process on which that element resides.

FIGURE 4-4 `LOCAL,CYCLIC` Distribution of a 16x16 Array Across Four Processes

The improvement in load balancing is summarized in TABLE 4-2. In particular, note the decrease in the number of elements allocated to process 0, from 54 to 36. Since process 0 still determines the overall computational time, this drop in element count can be seen as a computational speed-up of 150 percent.

TABLE 4-2 Numbers of Elements the Processes Operate on in FIGURE 4-3 and FIGURE 4-4

	FIGURE 4-3 (`BLOCK`)	FIGURE 4-4 (`CYCLIC`)
Process 0	54	36
Process 1	38	32
Process 2	22	28
Process 3	6	24

Process Grid Shape

Ordinarily, Sun S3L will map an S3L array onto a process grid whose logical organization is optimal for the operation to be performed. You can assume that, with few exceptions, performance will be best on the default process grid.

However, if you have a clear understanding of how a Sun S3L routine will make use of an array and you want to try to improve the routine's performance beyond that provided by the default process grid, you can explicitly create process grids using `S3L_set_process_grid`. This toolkit function allows you to control the following process grid characteristics.

- The grid's rank (number of dimensions)
- The number of processes along each dimension
- The order in which processes are organized – column order (the default) or row order
- The rank sequence to be followed in ordering the processes

For some Sun S3L routines, a process grid's layout can affect both load balancing and the amount of interprocess communication that a given application experiences. For example,

- A `1 x 1 x 1 x ... x NP` process grid (where `NP` = number of processes) makes all but the last array axis local to their respective processes. The last axis is distributed across multiple processes. Interprocess communication is eliminated from every axis but the last. This process grid layout provides a good balance between interprocess communication and optimal load balancing for many algorithms. Except for the axis with the greatest stride, this layout also leaves data in the form expected by a serial Fortran program.
- Use a square process grid for algorithms that benefit from cyclic distributions. This will promote better load balancing, which is usually the primary reason for choosing cyclic distribution.

Note that these generalizations can, in some situations, be nullified by various other parameters that also affect performance. If you choose to create a nondefault process grid, you are most likely to arrive at an optimal block size through experimentation, using the guidelines described here as a starting point.

Runtime Mapping to a Cluster

The runtime mapping of a process grid to nodes in a cluster can also influence the performance of Sun S3L routines. Communication within a multidimensional process grid generally occurs along a column axis or along a row axis. Thus, you should map all the processes in a process grid column (or row) onto the same node so that the majority of the communication takes place within the node.

Runtime mapping of process grids is effected in two parts:

- The multidimensional process grid is mapped to one-dimensional MPI ranks within the `MPI_COMM_WORLD` communicator. By default, Sun S3L uses *column-major* ordering. See FIGURE 4-5 for an example of column-major ordering of a 4x3 process grid. FIGURE 4-5 also shows row major ordering of the same process grid.

- MPI ranks are mapped to the nodes within the cluster by CRE (or other resource manager)
  . This topic is discussed in greater detail in Chapter 6.

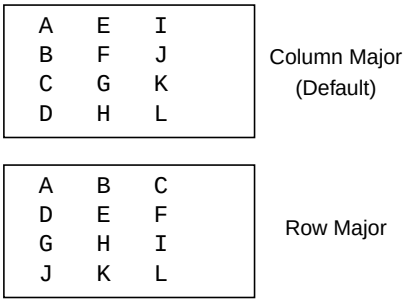


FIGURE 4-5 Examples of Column- and Row-Major Ordering for a 4x3 Process Grid

The two mapping stages are illustrated in FIGURE 4-6.

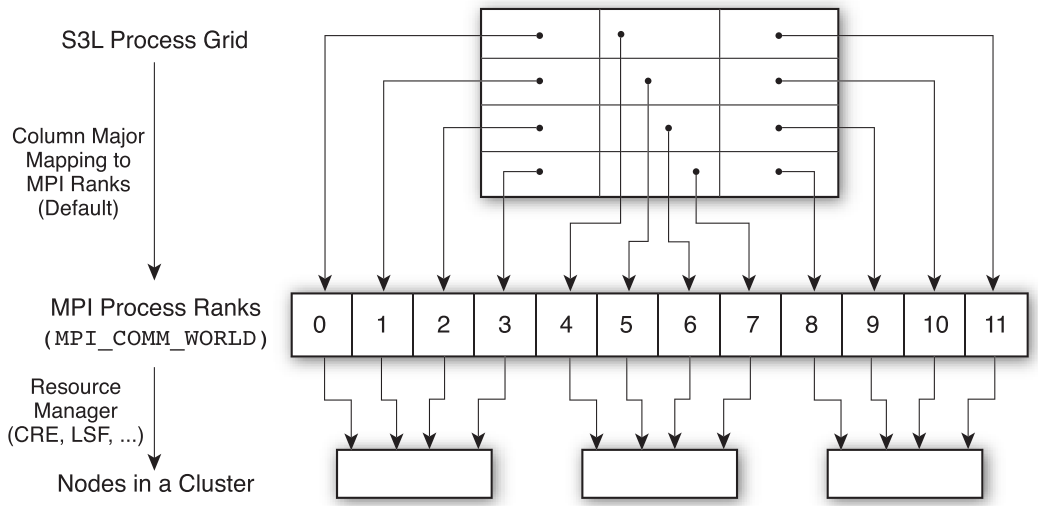


FIGURE 4-6 Process Grid and Runtime Mapping Phases (Column-Major Process Grid)

Neither stage of the mapping, by itself, controls performance. Rather, it is the combination of the two that determines the extent to which communication within the process grid will stay on a node or will be carried out over a network connection, which is an inherently slower path.

Although the ability to control process grid layout and the mapping of process grids to nodes give the programmer considerable flexibility, it is generally sufficient for good performance to:

- Group consecutive processes so that communication between processes remains within a node as much as possible.
- Use column-major ordering, which Sun S3L uses by default.

Note – If you do decide to use `S3L_set_process_grid`—for example, to specify a nondefault process-grid shape—use `S3L_MAJOR_COLUMN` for the `majorness` argument. This will give the process grid column-major ordering. Also, specify `0` for the `plist_length` argument. This will ensure that the default rank sequence is used. That is, the process rank sequence will be 0, 1, 2, ..., rather than some other sequence. See the `S3L_set_process_grid` man page for a description of the routine.

For example, assume that 12 MPI processes are organized as a 4x3, column-major process grid. To ensure that communication between processes in the same column remain *on node*, the first four processes must be mapped to one node, the next four processes to one node (possibly the same node as the first four processes), and so forth.

If your runtime manager is CRE, use

```
% mprun –np 12 –Z 4 a.out
```

For LSF, use

```
% bsub –I –n 12 –R “span[ptile=4]” a.out
```

Note that the semantics of CRE and LSF examples differ slightly. Although both sets of command-line arguments result in all communication within a column being on node, they differ in the following ways:

- The CRE command allows multiple columns to be mapped to the same node.
- The LSF command allows no more than one column per node.

Chapter 6 contains a fuller discussion of runtime mapping.

Use Shared Memory to Lower Communication Costs

Yet another way of reducing communication costs is to run on a single SMP node and allocate S3L data arrays in shared memory. This allows some Sun S3L routines to operate on data *in place*. Such memory allocation must be performed with `S3L_declare` or `S3L_declare_detailed`.

When declaring an array that will reside in shared memory, you need to specify how the array will be allocated. This is done with the `atype` argument. TABLE 4-3 lists the two `atype` values that are valid for declaring an array for shared memory and the underlying mechanism that is used for each.

TABLE 4-3 Using `S3L_declare` or `S3L_declare_detailed` to Allocate Arrays in Shared Memory

atype	Underlying Mechanism	Notes
`S3L_USE_MMAP`	`mmap`(2)	Specify this value when memory resources are shared with other processes.
`S3L_USE_SHMGET`	System V `shmget`(2)	Specify this value *only* when there will be little risk of depriving other processes of physical memory.

Smaller Data Types Imply Less Memory Traffic

Smaller data types have higher ratios of floating-point operations to memory traffic, and so generally provide better performance. For example, 4-byte floating-point elements are likely to perform better than double-precision 8-byte elements. Similarly, single-precision complex will generally perform better than double-precision complex operations.

Performance Notes for Specific Routines

This section contains performance-related information about individual Sun S3L routines. TABLE 4-4 summarizes some recommendations. Symbols used in the table include

N	linear extent of an array
N_{elem}	number of elements in an array
$N_{nonzero}$	number of nonzero elements in a sparse array
N_{rhs}	number of right-hand-side vectors
NB	block size for a block or block-cyclic axis distribution
NP	number of MPI processes
NPR	number of processes along the row axis
NPC	number of processes along the column axis
N/A	does not apply

TABLE 4-4 Summary of Performance Guidelines for Specific Routines

Operation	Operation Count	Optimal Distribution	Optimal Process Grid	shmem Optimi-zations?
`S3L_mat_mult`	2 N^3 (real) 8 N^3 (complex)	same block size for both axes	square	no
`S3L_matvec_sparse`	2 N $N_{nonzero}$ (real) 8 N $N_{nonzero}$ (complex)	N/A	N/A	yes
`S3L_lu_factor`	2 $N^3/3$ (real) 8 $N^3/3$ (complex)	block cyclic; same NB for both axes; NB = 24 or 48	1*NP (small N); square (big N)	no
`S3L_fft`, `S3L_ifft`	5 N_{elem} $\log_2(N_{elem})$	block; (also see `S3L_trans`)	1*1*1* ... *NP	yes
`S3L_rc_fft`, `S3L_cr_fft`	5 $(N_{elem}/2)\log_2(N_{elem}/2)$	block; (also see `S3L_trans`)	1*1*1* ... *NP	yes
`S3L_fft_detailed`	5 N_{elem} $\log_2(N)$	target axis local	N/A	N/A
`S3L_gen_band_factor`, `S3L_gen_trid_factor`	(iterative)	block	1*NP	no
`S3L_sym_eigen`	(iterative)	block; same NB for both axes	NPR*NPC, where NPR < NPC	no
`S3L_rand_fib`	N/A	N/A	N/A	no
`S3L_rand_lcg`	N/A	block	1*1*1* ... *NP	no

TABLE 4-4 Summary of Performance Guidelines for Specific Routines

Operation	Operation Count	Optimal Distribution	Optimal Process Grid	shmem Optimi-zations?
`S3L_gen_lsq`	$4\ N^3/3 + 2\ N^2N_{rhs}$	block-cyclic; same NB for both axes	square	no
`S3L_gen_svd`	$O(N^3)$ (iterative)	block-cyclic; same NB for both axes	square	no
`S3L_sort`, `S3L_sort_up`, `S3L_sort_down`, `S3L_grade_up`, `S3L_grade_down`	N/A	block	1*1*1* ... *NP	no
`S3L_sort_detailed_up`, `S3L_sort_detailed_down`, `S3L_grade_detailed_up`, `S3L_grade_detailed_down`	N/A	target axis local	N/A	no
`S3L_trans`	N/A	block	1*1*1* ... *NP NP=power of two	yes

The operation count expressions shown in TABLE 4-4 provide a yardstick by which a given routine's performance can be evaluated. They can also be used to predict how run times are likely to scale with problem size.

For example, assume a matrix multiply yields 350 Mflops per second on a 250-MHz UltraSPARC processor, which has a peak performance of 500 Mflops per second. The floating-point efficiency is then 70 percent, which can be evaluated for acceptability.

Floating-point efficiency is only an approximate guideline for determining an operation's level of performance. It cannot exceed 100 percent, but it might legitimately be much lower under various conditions, such as when operations require extensive memory references or when there is an imbalance between floating-point multiplies and adds. Often, bandwidth to local memory is the limiting factor. For iterative algorithms, the operation count is not fixed.

S3L_mat_mult

`S3L_mat_mult` computes the product of two matrices. It is most efficient when:

- The array is distributed to a large number of processes organized in a square process grid.
- The same block size is used for both axes.

If it is not possible to provide these conditions for a matrix multiply, ensure that the corresponding axes of the two factors are distributed consistently. For example, for a matrix multiply of size (m,n) = (m,k) x (k,n), use the same block size for the second axis of the first factor and the first axis of the second factor (represented by `k` in each case).

S3L_matvec_sparse

Sun S3L employs its own heuristics for distributing sparse matrices over MPI processes. Consequently, you do not need to consider array distribution or process grid layout for `S3L_matvec_sparse`.

Shared-memory optimizations are performed only when the sparse matrix is in `S3L_SPARSE_CSR` format and the input and output vectors are both allocated in shared memory.

S3L_lu_factor

The `S3L_lu_factor` routine uses a parallel, block-partitioned algorithm derived from the ScaLAPACK implementation. It provides best performance for arrays with cyclic distribution.

The following are useful guidelines to keep in mind when choosing block sizes for the `S3L_lu_factor` routine:

- Use the same block size in both axes.
- Use a block size in the 24-100 range to promote good cache reuse but to prevent cache overflows.
- Use a smaller block size for smaller matrices or for larger numbers of processes to promote better concurrency.

The `S3L_lu_factor` routine has special optimizations for double-precision, floating-point matrices. Based on knowledge of the external cache size and other process parameters, it uses a specialized matrix multiply routine to increase overall performance, particularly on large matrices.

These optimizations are available to arrays that meet the following conditions:

- The array is two-dimensional.
- The array is allocated with `S3L_declare_detailed`, using `S3L_USE_MEMALIGN64` for the `atype` argument .
- The array's data type is double-precision, floating-point.
- Both axes have the same block size, which should be 24 or 48.

When deciding on a process grid layout for LU factorization, your choices will involve making a trade-off between load balancing and minimizing communication costs. Pivoting will usually be responsible for most communication. The extreme ends of the trade-off spectrum are summarized below:

- To minimize the communication cost of pivoting, choose a 1 x NP process grid, where NP is the number of MPI processes.
- To optimize computational load balancing, choose a nearly square process grid.

Some experimentation will be necessary to arrive at the optimal trade-off for your particular requirements.

S3L_fft, S3L_ifft, S3L_rc_fft, S3L_cr_fft, S3L_fft_detailed

Performance is best when the extents of the array can be factored into small, prime factors no larger than 13.

The operation count expressions given in TABLE 4-4 for the FFT family of routines provide a good approximation. However, the actual count will depend to some extent on the radix (factors) used. In particular, for a given problem size, the real-to-complex and complex-to-real FFTs have half the operation count and half the memory requirement of their complex-to-complex counterparts.

The transformed axis should be local. If a multidimensional transform is desired, make all but the last axis local.

It is likely that the resulting transpose will dominate the computation, at least in a multinode cluster. See the discussion of `S3L_trans`.

S3L_gen_band_factor, S3L_gen_trid_factor, S3L_gen_band_solve, S3L_gen_trid_solve

These routines tend to have relatively low communication costs, and so tend to scale well.

For best performance of the factorization routines, make the all the axes of the array to be factored local, except for the last axis, which should be block distributed.

Conversely, the corresponding solver routines perform best when the first axis of the right-hand side array is block distributed and all other axes are local.

S3L_sym_eigen

The performance of `S3L_sym_eigen` is sensitive to interprocess latency.

If both eigenvectors and eigenvalues are computed, execution time might be as much as an order of magnitude longer than if only eigenvalues are computed.

S3L_rand_fib, S3L_rand_lcg

`S3L_rand_fib` and `S3L_rand_lcg` initialize parallel arrays using a Lagged-Fibonacci and a Linear Congruential random number generator, respectively. An array initialized by the Lagged-Fibonacci routine will vary depending on the array distribution. In contrast, array initialization by the Linear Congruential method will produce the same result regardless of the array's distribution.

Because the Linear Congruential random number generator must ensure that the resulting random numbers do not depend on how the array is distributed, it has the additional task of keeping account of the global indices of the array elements. This extra overhead is minimized when local or block distribution is used and greatly increased by distributing the array cyclically. `S3L_rand_lcg` can be two to three times slower with cyclic distributions than with local or block distributions.

Since `S3L_rand_fib` fills array elements with random numbers regardless of the elements' global indices, it is significantly faster than `S3L_rand_lcg`.

The `S3L_rand_lcg` routine is based on 64-bit strings. This means it performs better on `S3L_long_integer` data types than on `S3L_integer` elements.

`S3L_rand_fib`, on the other hand, is based on 32-bit integers. It generates `S3L_integer` elements twice as fast as for `S3L_long_integer` output.

Both algorithms generate floating-point output more slowly than integers, since they must convert random bit strings into floating-point output. Complex numbers are generated at half the rate of real numbers, since twice as many must be generated.

S3L_gen_lsq

`S3L_gen_lsq` finds the least-squares solution of an overdetermined system. It is implemented with a QR algorithm. The operation count shown in TABLE 4-4 applies to real, square matrices. For a real, rectangular (M,N) matrix, the operation count scales as

$2\ N\ N_{rhs}(2M–N) + 2\ N^2\ (M–N/3)$ for $M >= N$

$2\ N\ N_{rhs}(2M–N) + 2\ M^2\ (N–M/3)$ for $M < N$

For complex elements, the operation count is four times as great.

S3L_gen_svd

For `S3L_gen_svd`, the convergence of the iterative algorithm depends on the matrix data. Consequently, the count is not well-defined for this routine. However, `S3L_gen_svd` does tend to scale as N^3.

If the singular vectors are computed, the run time can be roughly an order of magnitude longer than if only singular values are extracted.

The `A`, `U`, and `V` arrays should all be on the same process grid for best performance.

S3L_gen_iter_solve

Most of the time spent in this routine is in `S3L_mat_vec_sparse`.

Overall performance depends on more than just the floating-point rate of that subroutine. It is also significantly influenced by the matrix data and by the choice of solver, preconditioner, initial guess, and convergence criteria.

S3L_acorr, S3L_conv, S3L_deconv

The performance of these functions depends on the performance of S3L FFTs and, consequently, on the performance of the S3L transposes.

S3L_sort, S3L_sort_up, S3L_sort_down, S3L_sort_detailed_up, S3L_sort_detailed_down, S3L_grade_up, S3L_grade_down, S3L_grade_detailed_up, S3L_grade_detailed_down

These routines do not involve floating-point operations. The operation count can vary greatly, depending on the distribution of keys, but it will typically scale from O(N) to O(N log(N)).

Sorts of 64-bit integers can be slower than sorts of 64-bit floating-point numbers.

S3L_trans

`S3L_trans` provides communication support to the Sun FFTs as well as to many other Sun S3L algorithms. Best performance is achieved when axis extents are all multiples of the number of processes.

S3L Toolkit Functions

The S3L Toolkit functions are primarily intended for convenience rather than performance. However, some significant performance variations do occur. For example:

- `S3L_copy_array` can be very fast or extremely slow depending on how well the two arrays are aligned.
- `S3L_forall` performance entails relatively significant overhead for each element operated on for function types `S3L_ELEM_FN1` and `S3L_INDEX_FN`. In contrast, the function type `S3L_ELEM_FNN` amortizes such overhead over many elemental operations.
- `S3L_set_array_element`, `S3L_set_array_element_on_proc`, `S3L_get_array_element`, and `S3L_get_array_element_on_proc` perform very small operations. Consequently, overhead costs are a significant component for these routines (as with the `S3L_forall` function types `S3L_ELEM_FN1` and `S3L_INDEX_FN`).

CHAPTER 5

Compilation and Linking

This chapter describes the Forte Developer compiler switches that typically give the best performance for Sun MPI programs.

For more detailed information on compilation, see 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 latest release supported by Sun HPC ClusterTools 4 is Forte Developer 6 update 2.

For Fortran programmers, another way to enhance performance is to use the `f90` (equivalent to the `f95`) compiler, instead of `f77`. The `f90` compiler supports the Fortran 77 language features, but also has optimizations not present in the `f77` compiler.

Using 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 `mpf77` and `mpcc`, but it applies equally to the various scripts and aliases just mentioned. For example, you can use:

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

to compile a Fortran 77 program that uses Sun MPI, or

```
% mpcc -fast -g a.c -ls3l -lmopt
```

to compile a C program that uses Sun S3L. Note that these utilities automatically link in MPI if S3L use is specified.

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

–fast

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

Part of `-fast` 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 `-xtarget`. 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 `-xarch`, as discussed below.

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

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

-xarch

The second most important compiler switch for maximizing performance is `-xarch`. While `-fast` picks many performance-oriented settings by default, you should specify a value for `-xarch` 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 `-xarch` 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.

Note the following requirements when using `-xarch`:

- To compile or build 64-bit object binaries, you must use the Solaris 8 Operating Environment.
- To execute 64-bit binaries, you must use the Solaris 8 Operating Environment with the 64-bit kernel.
- Object files in 64-bit format can be linked only with other object files in the same format.

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

-g

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

With the Forte Developer 6 compilers, however, there is virtually no loss of performance with this switch. Further, `-g` compilation enables source-code annotation by the Forte Developer performance analyzer, which provides important performance tuning information. Thus, `-g` might be considered to be one of the basic switches in to use in performance-tuning work.

Other Useful Switches

Performance benefits from linking in the optimized math library. For Fortran, `-fast` invokes `-xlibmopt` automatically. For C, be sure to add `-lmopt` to your link line:

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

Include `-xdepend` to perform analysis of data dependencies in loops, which could lead to loop restructuring and performance enhancements. The `-fast` switch might already include `-xdepend` for Fortran compilation, but not for C.

Include `-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 `-xvector[=yes]`, 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 `-xprefetch` to enable compiler generation of prefetch instructions. The `-fast` switch might already include `-xprefetch` for Fortran compilation, but not for C. Sometimes, `-xprefetch` can slow performance, so it might best be used selectively. For example, you can compile some files with `-xprefetch[=yes]` 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 `-xrestrict`, which causes the compiler to treat pointer-valued function parameters as restricted pointers. Other information about pointer aliasing can be provided to the compiler via `-xalias_level`. See the *C User's Guide* for more details.

C programmers should also consider `-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 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 `-xsfpconst` 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 `-stackvar`.

CHAPTER 6

Runtime Considerations and Tuning

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

This chapter discusses the performance implications of:

For most users, the most important section of the chapter will be the discussion of tuning Sun MPI environment variables at run time. While the default values are generally effective, some tuning may help improve performance, depending on your particular circumstances.

Running on a Dedicated System

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

	CRE	LSF	UNIX
How high is the load?	% **mpinfo -N**	% **lsload**	% **uptime**
What is causing the load?	% **mpps -e**	% **bjobs -u all**	% **ps -e**

To find out what the load is on a cluster, use the appropriate command (`mpinfo` or `lsload`) depending on the resource manager (CRE or LSF) in use at your site. Standard UNIX commands such as `uptime` give the same information, but only for one node.

To find out what processes are contributing to a load, again use the appropriate command, depending on resource manager. The information will be provided for all nodes and organized according to parallel job. However, this will show 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, `fsflush` flushes memory periodically to disk. On a server with a lot of memory, the default behavior of this daemon may cause a background load of only about 0.2, representing a small fraction of one 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, since MPI processes are often very tightly coupled. (For the particular case of `fsflush`, 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.

User tuning of MPI environment variables can be restricted by the system administrator through a configuration file. You can use `MPI_PRINTENV`, described below, to verify settings.

The suggestions in this section are listed roughly in order of decreasing importance. That is, leading items are perhaps most common or most drastic. In some cases, diagnosis of whether environment variables would be helpful is aided by Prism

profiling, as described in Chapter 7. More information on Sun MPI environment variables can be found in Appendix B "Sun MPI Environment Variables" and in the *Sun MPI Programming and Reference Guide*.

Are You Running on a Dedicated System?

If your system's capacity is sufficient for running your 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 `uptime`, `lsload`, or `mpinfo`, as discussed at the beginning of this chapter.

To run more aggressively:

- **`% 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`**

  This setting causes Sun MPI to bind each MPI process to a different processor by using a particular mapping. You may not see a great performance benefit for jobs that use few processes on a node. Don't use this setting with multiple MPI jobs on a node or with multithreaded jobs: If multiple MPI jobs on a node use this setting, they will compete for the same processors. Also, if your job is multithreaded, multiple threads will compete for a processor.

Suppress Cyclic Messages

Sun MPI supports cyclic message passing for long messages between processes on the same node. Cyclic message passing induces added synchronization between sender and receiver, which in some cases may hurt performance. Suppress cyclic message passing with:

```
% setenv MPI_SHM_CYCLESTART 0x7fffffff
```

Or, if you are operating in a 64-bit Solaris environment, use:

```
% setenv MPI_SHM_CYCLESTART 0x7fffffffffffffff
```

For a description of cyclic messages, see Appendix A “Sun MPI Implementation”.

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 this setting:

```
% setenv MPI_POLLALL 0
```

If deadlock results from this setting, you may 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?

It is common for senders to stall while waiting for other processes to free shared-memory resources.

One simple solution to this is to increase Sun MPI’s consumption of shared memory. For example, you might try:

```
% setenv MPI_SHM_SBPOOLSIZE 8000000
```

```
% setenv MPI_SHM_NUMPOSTBOX 256
```

for ample buffering in a variety of situations.

Unfortunately, there is no one-size-fits-all solution to the trade-off between memory and performance. These sample settings target better performance. The Sun MPI default settings target low memory consumption. This section discusses considerations that will enable you to make a more discriminating trade-off.

It is helpful to think of data traffic per connection, the logical "path" from a particular sender to a particular receiver, since many Sun MPI buffering resources are allocated on a per-connection basis. A sender may emit a burst of messages on a connection, during which time the corresponding receiver may not be depleting the buffers.

The following discussion refers exclusively to messages that are exchanged between processes on the same node—for example, messages in an MPI program that executes wholly on a single SMP server.

Profiling may be needed to diagnose stalled senders. For more information on profiling, see Chapter 7. In particular, analyzing time in relation to message size for MPI send calls can be helpful. For example,

- If performance of send calls, such as `MPI_Send()` or `MPI_Isend()`, appears to reflect reasonable on-node bandwidths (on the order of 100 Mbyte/s), ample shared memory resources are probably available to accommodate senders.
- If blocking sends (such as `MPI_Send()`) are taking much more time than the message sizes warrant, stalling may be the cause.
- If nonblocking sends (such as `MPI_Isend()`) are taking much less time than the message sizes warrant, there may be a hidden problem. The sender may find insufficient shared-memory resources and exit the call immediately, leaving message data unsent. The TNF probe `MPI_Isend_end` should always return a "done" argument equal to 1.
- If calls such as `MPI_Wait()` or `MPI_Testany()`, which complete or could complete nonblocking send operations (like `MPI_Isend()`), spend too much time completing sends, it is likely that buffering is insufficient. See FIGURE 7-9 on page 115 for an example.

If you know or can assume that senders will stall only on occasional long messages, but never on bursts of many short messages, you can take another approach to profiling. In this case, use profiling to determine the length of the longest message ever sent.

To eliminate sender stalls by increasing shared-memory resources, you must set Sun MPI environment variables. Arbitrary adjustments to these environment variables can lead to unforeseen consequences. As a rule, do not decrease the following environment variables below their default values. For complete information on Sun MPI environment variables, including default values, ranges of legal values, and memory implications, see Appendix B "Sun MPI Environment Variables" or the *Sun MPI Programming and Reference Guide*.

One approach is simply to use fixed settings, as shown in the example at the beginning of this section. For more detailed tuning, note that you have to allocate:

- *buffers* for message data, in one of these ways:
  - on a per-connection basis (that is, for each sender-receiver pair) with `MPI_SHM_CPOOLSIZE`
  - on a per-sender basis (that is, for each sender) with `MPI_SHM_SBPOOLSIZE`
- *postboxes* for buffer pointers, ensuring at least one postbox for each 8192 bytes of data per connection

Consider the following examples.

- *Example 1* – An MPI process will post 20 short sends to another process before "listening" for any receives. Use:

  ```
  % setenv MPI_SHM_NUMPOSTBOX 20
  ```

- *Example 2* – Interprocess messages may be as long as 200000 bytes. Since such a message may require as many as 200000 / 8192 ≈ 24.4 postboxes, use:

  ```
  % setenv MPI_SHM_CPOOLSIZE 300000
  % setenv MPI_SHM_NUMPOSTBOX 30
  ```

  (Values have been rounded up to ensure ample buffering.) For `np=64`, the above allocation can take about 64 * 63 * 300000 bytes, or about 1200 Mbytes.

- *Example 3* – Although interprocess messages may be as long as 200000 bytes, an MPI process communicates with only four other processes at a time in this way. Use:

  ```
  % setenv MPI_SHM_SBPOOLSIZE 1200000
  % setenv MPI_SHM_NUMPOSTBOX 30
  ```

  This case use the same number of postboxes as in Example 2. Each "send-buffer pool" is four times as large as a "connection pool" in Example 2, but there are fewer pools. For `np=64`, the new buffer allocation can take about 64 * 1200000 bytes, or about 75 Mbytes.

Initializing Sun MPI Resources

Use of certain Sun MPI Resources may be relatively expensive when they are first used. This 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 may be set to move some degree of resource initialization to the `MPI_Init()` call. Use:

```
% setenv MPI_FULLCONNINIT 1
```

Note that this does *not* tend to improve overall performance. However, it may 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?

You can set some Sun MPI environment variables to print out extra diagnostic information at run time:

```
% setenv MPI_PRINTENV 1
% setenv MPI_SHOW_INTERFACES 3
% setenv MPI_SHOW_ERRORS 1
```

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, including minimizing communication costs, load balancing, bisection bandwidth, and the role of I/O servers.

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 may 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 may 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 may 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 may also be desirable to leave at least one CPU per node idle (see "Running on a Dedicated System" on page 77). Additionally, if bandwidth to

memory is more important than interprocess communication, you may 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 since the resource manager (CRE or LSF) may not know at job launch that processes will spawn other LWPs.

Bisection Bandwidth

Many cluster configurations provide relatively little internodal bandwidth per node. Meanwhile, bisection bandwidth may be the limiting factor for performance on a wide range of applications. In this case, if you must run on multiple nodes, you may prefer to run on more nodes rather than on fewer.

This point is illustrated qualitatively in FIGURE 6-1. The high-bandwidth backplanes of large Sun servers provide excellent bisection bandwidth for a single node. Once you have multiple nodes, however, the interface between each node and the network will become the bottleneck. Bisection bandwidth starts to recover again when the number of nodes—actually, the number of network interfaces—increases.

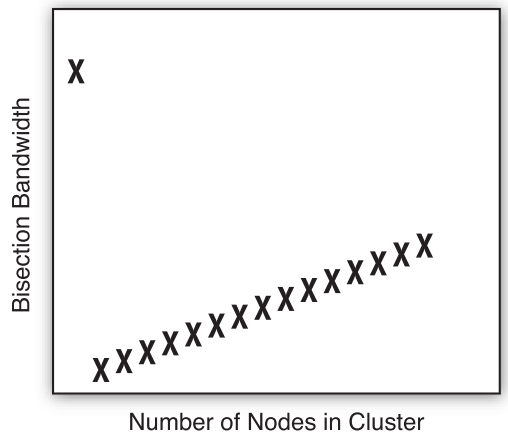


FIGURE 6-1 Bisection bandwidth increases with the number of nodes, but a single node is even better.

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.

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 may improve performance. If the program makes scant use of I/O, you may prefer to avoid I/O nodes, since they may consume nodal resources. If multiple I/O servers are used, you may 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 may 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. Otherwise, the job could block. For example,

```
% 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 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 `limit coredumpsize 0`. If the parent shell is `sh`, use `ulimit -c 0`.

To limit core dumps for jobs submitted with the LSF `bsub` command, use the `-C 0` switch to `bsub`.

Examples of Job Launch on a Cluster

This section presents examples of efficient parallel job launches on a multinode cluster using the CRE and LSF runtime environments, respectively.

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 using Sun S3L. 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 `-Z` or `-Zt`:

```
% 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 may be mapped to the same node. For example, with `-Zt`, the entire job may 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 multithreaded case above), 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. See the *Sun HPC CluaterTools User's Guide* for more information about process mappings.

Multinode Job Launch Under LSF

LSF controls most aspects of resource allocation on a cluster, leaving the user limited control of the mapping of jobs to noces.

Collocal Blocks of Processes

LSF supports the collocation of blocks of processes—that is, all processes within a block are mapped to the same node. With LSF, different blocks will be mapped to different nodes. For example, consider a multithreaded MPI job with one MPI process per node, and 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:

```
% bsub -I -n 16 -R "span[ptile=1]" a.out
```

Or, assume that you are performing an LU decomposition on a 4x8 process grid using Sun S3L. If minimization of communication within each block of four consecutive MPI ranks is most important, then these 32 processes would be launched on 8 different nodes using:

```
% bsub -I -n 32 -R "span[ptile=4]" a.out
```

On the other hand, this approach will distribute the blocks of processes over 8 nodes, regardless of whether a node could accommodate more processes. So consider again the 4x8 process grid, but this time assume that each node in your cluster has 14 CPUs. You could further aggregate processes and so further minimize communication by assigning blocks of 12 consecutive MPI ranks (3 blocks of 4 each), using:

```
% bsub -I -n 32 -R "span[ptile=12]" a.out
```

Finally, consider a cluster with four nodes, each hosting an I/O server. If you run an 8-process application that is I/O throughput bound, the processes should be spread over all the nodes to maximize aggregate throughput to disk. You can launch the 8 processes on 4 different nodes using:

```
% bsub -I -n 8 -R "span[ptile=2]" a.out
```

Multithreaded Job

By default, LSF launches one process per reserved CPU. To reserve more CPUs per process, for example for multithreaded jobs, use the `-sunhpc` switch.

For example, consider a job in which there are 8 MPI processes and each process is multithreaded 2 ways. You can reserve 8x2=16 CPUs with LSF and yet launch only 8 processes by using:

```
% bsub –I –n 16 -sunhpc -n 8 a.out
```

CHAPTER 7

Profiling

An important component of performance tuning is profiling, in which you develop a picture of how well your code is running and what sorts of bottlenecks it may 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
```

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 run time. 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, and 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, from reducing the data set to performing fewer loop iterations, but keep these 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 may become dwarfed by otherwise inconsequential factors. In particular, code may behave differently the first few times it is executed than when buffers, caches, and other resources have been warmed up.

Basic Approaches

There are a variety of approaches to profiling Sun HPC ClusterTools programs:

- Use the Forte Developer performance analyzer (the programs: Sampling Collector and Sampling Analyzer) — This 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 `-g` enable attribution to individual source-code lines with almost no loss in optimizations. On UltraSPARC-III microprocessors, hardware-based profiling can identify where floating-point operations, cache misses, and so forth, occur. For information about using Forte Developer profiling, see “Forte Developer Profiling of Sun MPI Programs” on page 95.
- Run your program under the Prism environment to understand the MPI message-passing activities — Prism tracing gives you a picture of the message-passing behavior of your program and how its characteristics—synchronization patterns, message sizes, and so on—may be impairing performance. For information about Prism tracing, see “Using the Prism Environment to Trace Sun MPI Programs” on page 108.

- Modify your source code to include timer calls — This 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 129.
- 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. That 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 128.

TABLE 7-1 Profiling Alternatives

Method	Advantages	Disadvantages
Forte Developer 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-III microprocessors, profiles based on hardware counters (floating-point operations, cache misses, and so forth)	• Has no special knowledge of MPI or message passing (such as bytes sent, senders, receivers, and so forth) • No time-line functionality
Prism Environment	• Uses (by default) the pre-instrumented Sun MPI library (manual instrumentation optional) • Provides lots of data on MPI usage • Integrated with other Prism tools	• Generates large data files • Requires manual instrumentation to generate data on user code
Timers	• Very versatile	• Requires manual instrumentation • Requires that you understand the code
`gprof`	• Familiar tool • Provides an overview of user code	• Ignores time spent in MPI
PMPI Interface	• 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 am running a code with which I am rather unfamiliar. 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 Forte Developer Sampling Analyzer, you can see which routines consume the most time.
- *I would like to visualize the load imbalances, serial bottlenecks, global synchronization, or other interprocess effects that may be impairing my program's scalability* — Prism tracing can help you judge whether and how message-passing calls are taking up significant time.
- *I know that a few innermost loops are bottlenecks and I need more detailed information* — Adding timers and other instrumentation around innermost loops may 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 may allow you to continue using them with Sun HPC ClusterTools programs.

The remainder of this chapter discusses Forte Developer profiling and Prism tracing in detail, and then returns to a brief discussion of the alternative approaches.

Forte Developer Profiling of Sun MPI Programs

The Forte Developer Sampling Analyzer offers a good first step to understanding the performance characteristics of a program on UltraSPARC systems. It combines ease of use with powerful functionality. In particular, it can indicate whether more detailed analysis of message-passing activity, such as with Prism tracing, is warranted.

As with most profiling tools, there are two basic steps to using Forte Developer performance analysis tools. The first step is to use the Forte Developer Sampling Collector to collect performance data. The second step is to use the Sampling Analyzer (hereafter referred to as the Analyzer) to examine results. For example, this procedure can be as simple as replacing

```
% mprun –np 16 a.out 3 5 341
```

with

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

Forte Developer compilers and tools are usually located in `/opt/SUNWspro/bin`. Check with your system administrator for details for your site.

The following pages show the use of the Analyzer with Sun MPI programs, often revisiting variations of the above example.

Software Releases

With each release of Forte Developer and Sun HPC ClusterTools software, the interoperability of the Forte Developer performance analyzer with Sun MPI has improved. Therefore, for both software suites, it is recommended that you use the latest available release.

Recent releases of Forte Developer have supplied

- Compilation and linking with `-g` provides complete source-line information and compiler commentary to the Analyzer with virtually no reduction in optimization levels or parallelization.
- Vastly improved performance analysis tool, the Analyzer.
- Sampling collection that can be used apart from `dbx`.
- Profiling based on UltraSPARC III hardware counters.
- Specification of multiple experiments on the Analyzer's command line.
- Default experiment names that include MPI process rank.
- Standalone source browser `er_src` for viewing annotated source and disassembly code, including compiler commentary, without having to load an experiment.

Recent releases of Sun HPC ClusterTools software have added:

- Ability to launch `dbx` from CRE. For example,

  % **mprun -np 16 dbx a.out**

- Run-time environment variables `MP_RANK` and `MP_NPROCS`, which tell each process the MPI rank and total number of processes in the job
- Support for launching shell scripts from mprun. For example.,

  % **mprun -np 16** *shell-script*

Data Collection

There are several ways of using the Forte Developer Sampling Collector with MPI codes. For example, with Forte Developer 6 update 1, the simplest usage could look like:

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

On the other hand, there are many benefits to launching the data collection using a shell script. Among them:

- Collect data for only a subset of MPI processes
- Number experiment files according to MPI rank
- Collect data to a local file system and then gather them to a central location

Typically, the extra effort of using a shell script is worthwhile.

Data Volume

The volume of collected data can grow large, especially for long-running or parallel programs. Though the Sampling 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 may 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 on 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 may be used. Of course, limiting data collection to a subset of the processes may bias the profiling data. In particular, a master process may behave unlike any of the other processes.
- Collect data to a local file system. This does not reduce the volume of data collected, but it helps mitigate the impact.

  This strategy should be a routine part of data collection.

A large parallel job may run out of a central NFS-mounted file system. While this may be adequate for jobs that are not I/O intensive, it may 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 `/tmp` on different nodes of a cluster refer to different, respectively local file systems. Also, `/tmp` may not be very large, and if it becomes filled there may be a great impact on general system operability.

Data Organization

The Sampling 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 `/opt/SUNWspro/bin`) to move, remove, or copy experiments. For more information on these utilities, see the corresponding man pages or the Forte Developer 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 since (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.

Data Collection with Forte Developer 6

In Forte Developer 6, the Sampling Collector could originally be invoked only through the `dbx` debugger.

The `dbx` debugger can generate a large volume of informational messages, especially for multiprocess jobs, that are not useful for profiling data collection. Two ways of reducing informational messages are:

- Add the following line to your `.dbxrc` file.

  ```
  dbxenv suppress_startup_message 5.0
  ```

- Use the `-q` (quiet) switch to `dbx`.

Example 1. If you know you want to collect data for each MPI process, you may use

```
% cat dbx-script
collector enable
collector store filename proc-$MP_RANK.er
run 3 5 341
quit
% mprun -np 16 dbx -q -c "source dbx-script" a.out
%
```

Example 2. If you know you want to collect data for only the first 4 MPI processes, collect to the `/tmp` file system, and then gather the results after execution, you may use

```
% cat dbx-script
collector enable
collector store directory /tmp
collector store filename proc-$MP_RANK.er
run 3 5 341
quit
% cat csh-script
#!/bin/csh
if ( $MP_RANK < 4 ) then
dbx –q –c "source dbx-script" a.out
er_mv /tmp/proc-$MP_RANK.er .
else
a.out 3 5 341
endif
% mprun -np 16 csh-script
%
```

Data Collection with Forte Developer 6 update 1

With update 1, the need to invoke the Sampling Collector through `dbx` was lifted, simplifying data collection.

Example 1. If you know you want to collect data for each MPI process, and it is not important to be able to identify the output data by MPI rank, you may use

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

Example 2. If you know you want to collect data for only the first 4 MPI processes, collect to the `/tmp` file system, and then gather the results after execution, you may use

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

Analyzing Profiling Data

Once data has been gathered with the Sampling Collector, it may be viewed with the Sampling Analyzer.

Loading Data

To start the Analyzer, use the command

```
% analyzer
```

Then, using the Experiment menu, you may Add and Drop individual experiments. You may also bring up a particular experiment by specifying it on the command line:

```
% analyzer proc-0.er
```

With Forte Developer 6 update 1 or later, you may specify multiple experiments on the command line directly. For example:

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

Case Study

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

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

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

A script is used to collect profiling data for MPI rank 0, only. The job is run on a single, shared-memory node using 25 processes.

Basic Features of the Sampling Analyzer

The first view that the Analyzer shows of the resulting profiling data is shown in FIGURE 7-1. 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 146.93 seconds are profiled. We see that the functions `LHSX()`, `LHSY()`, and `LHSZ()` account for 23.69+21.89+19.87=65.45 seconds of that time.

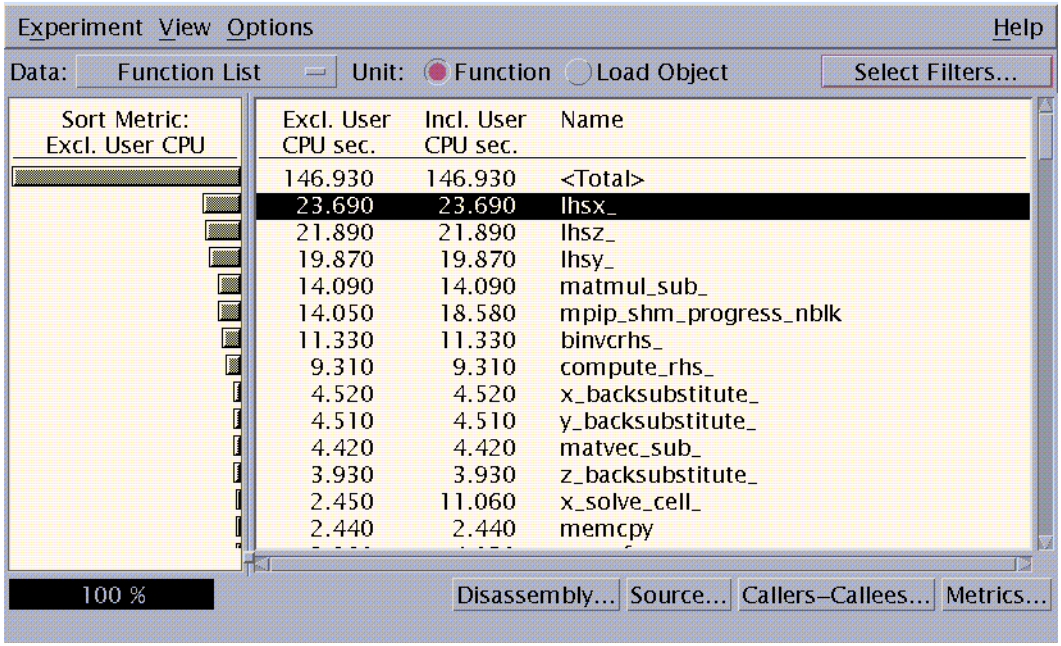


FIGURE 7-1 Analyzer—Main View

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.

We can see how time is spent within a subroutine if the code was compiled and linked with the `-g` switch. Again, starting with Forte Developer 6 compilers, 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 button at the bottom of the window, the Analyzer brings up a text editor for the highlighted function. The choice of text editor may 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 7-2. In particular, notice that the Analyzer highlights hot (time-consuming) lines of code with color and with the ## markers in the first two columns. Only a small fragment is shown since, in practice, the annotated source can become rather wide.

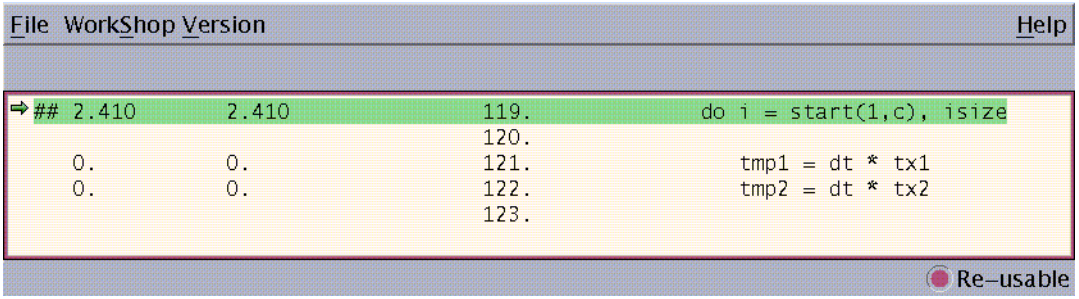


FIGURE 7-2 Analyzer—Source View

To get a better idea where time is spent at a high level in the code, you can also click on the Callers-Callees button, shown at the bottom of FIGURE 7-1. For example, one might get a view as in FIGURE 7-3. 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 may be making. This sort of analysis is probably familiar to `gprof` users, but the Analyzer has features that go beyond some of `gprof`'s limitations. For more information about `gprof`, see "Using `gprof`" on page 130.

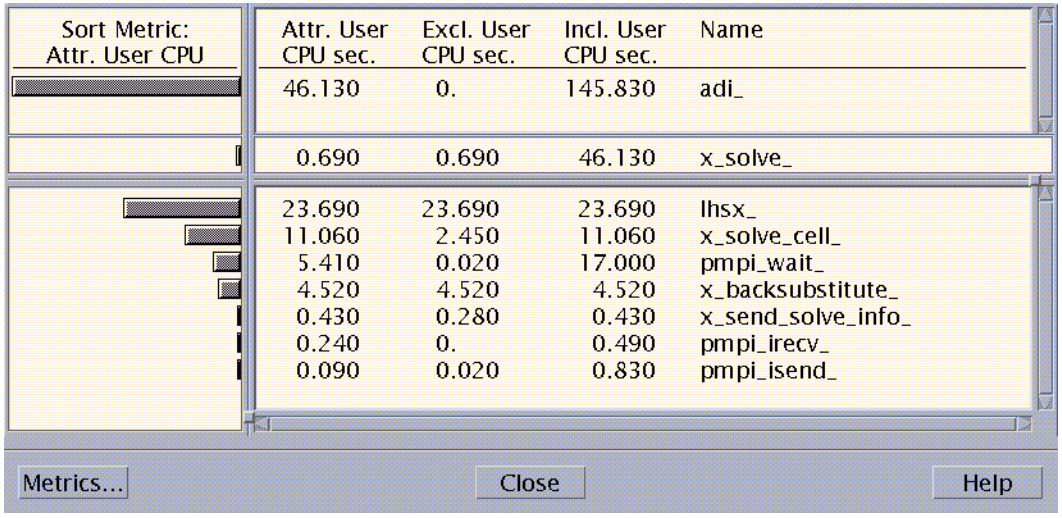


FIGURE 7-3 Analyzer—Callers-Callees View

Different metrics can be selected for the various displays by clicking on the Metrics button, as seen at the bottom of FIGURE 7-1. You can choose which metrics are shown, the order in which they are shown, and which metric should be used for sorting.

Sometimes it is useful to produce performance analysis information without invoking the Analyzer's graphical interface. This can be achieved by using the `er_print` utility. For example:

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

The first command produces the function list of FIGURE 7-1. The second gives a complete caller-callee list, similar to that shown in FIGURE 7-2. The first two commands together provide all of the information that can be expected from `gprof`. The third command produces an annotated listing of subroutine `LHSX()`, as shown in FIGURE 7-3. More information on the `er_print` utility can be found on the `er_print` man page.

Overview of Functions

Typically in a profile, you will find many unfamiliar functions that do not appear explicitly in your code. Further, there may be none of the familiar MPI calls—`MPI_Isend()`, `MPI_Irecv()`, `MPI_Wait()`, or `MPI_Waitall()`—you do use.

In FIGURE 7-4 are 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 (`a.out`) or a dynamic library.

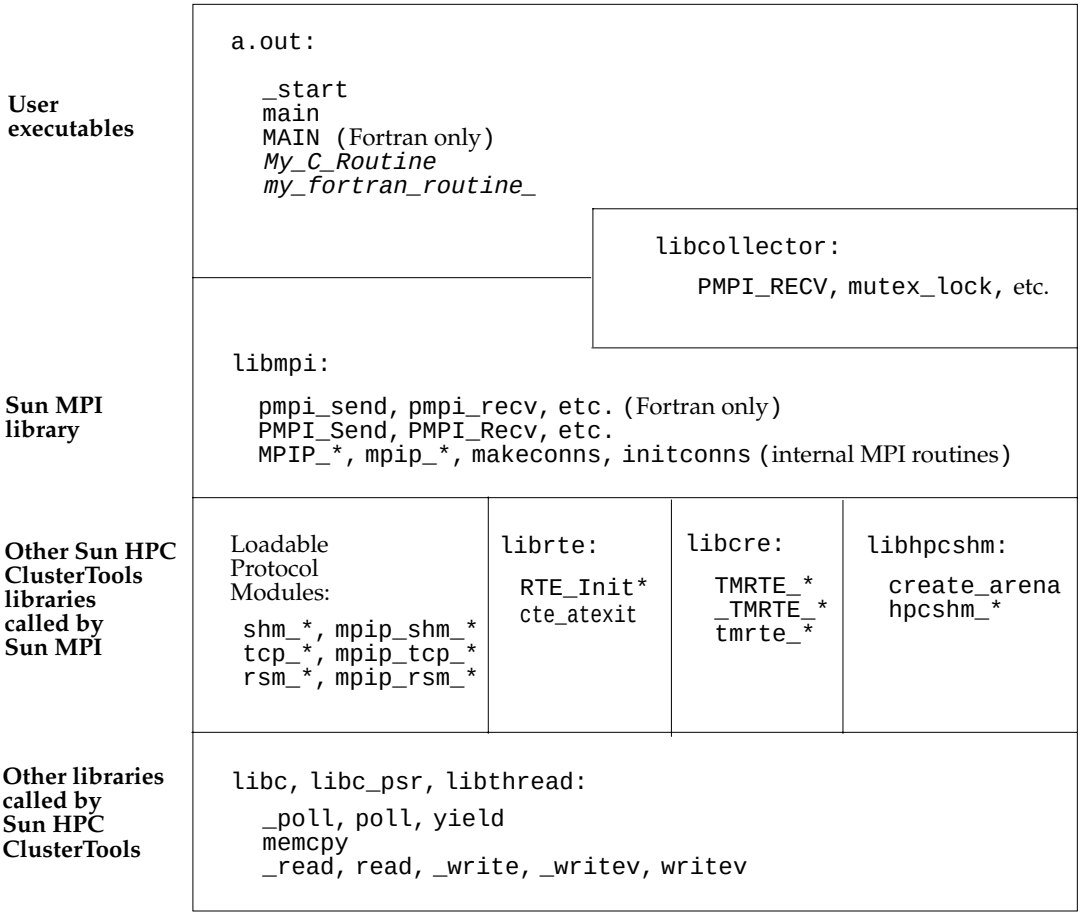


FIGURE 7-4 Examples of Functions That May Appear in Profiles

Note that:

- The top function is `_start`, which calls `main`. In Fortran programs, `main` calls `MAIN_`.
- The Fortran compiler by default converts subprogram names to lower case and appends underscores. For example, 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 via 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` may appear twice in profiles — once belonging to a user-callable library and once belonging to `libcollector`.
- Sun MPI calls routines in other ClusterTools libraries.
- Various ClusterTools libraries call other standard libraries. Notably:
  - `_poll`, `poll`, and `yield` may be called by an MPI process for waiting
  - `memcpy` is often called when an MPI process is copying data locally — such as for on-node message passing
  - `_read`, `read`, `_write`, `_writev`, `writev` 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 Analyzer you may:

- 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 may be spent in MPI calls for a wide variety of reasons. Specifically, much of that time may 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 may or may not tie up a CPU. Nevertheless, such wait time probably costs program performance.

There are several things you can do to ensure that wait time is profiled. One is to have Sun MPI spin a CPU aggressively during wait situations. This requires turning off coscheduling and turning on spin behavior. Both are off by default. You may use the following environment variable settings:

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

Another strategy is to select wall-clock time, instead of CPU time, as the profiling metric. Selection of metrics was discussed above under "Basic Analyzer Features".

Using the Prism Environment to Trace Sun MPI Programs

The Prism environment supports profiling program performance using the Solaris trace normal form (TNF) facilities. The Sun MPI library is preinstrumented with TNF probes, facilitating the use of the Prism environment to profile Sun MPI programs.

Prism tracing requires no special compilation or linking. Its simple graphical interface allows you to review MPI usage within an application and find the parts that need tuning. You may visually inspect patterns of MPI calls for general activity, excessive synchronization, or other large-scale behaviors. Other views summarize which MPI calls are made, which ones account for the most time, which message sizes or other characteristics cause slowdowns, and so on.

Statistical analyses allow you to see which MPI routines, message sizes, or other characteristics account for an appreciable fraction of the run time. You can click on hot spots in a statistical display to examine the detailed sequence of events that led up to that hot spot.

No instrumentation is required for Prism tracing since the Sun MPI library is preinstrumented. You may optionally add your own probes to extract more information from performance experiments.

This chapter illustrates Prism tracing with two case studies. If you are new to the Prism programming environment, you may still find it useful to walk through these examples. For a detailed treatment of Prism tracing functionality and usage, however, please refer to the *Prism User's Guide*.

The first case study is a popular HPC benchmark in computational fluid dynamics (CFD). It relies heavily on point-to-point communications. The second case study is based on sorting and collective communications.

Note – TNF terminology in the following discussions is not specific to MPI, as TNF applies to a far broader context. Hence, you should understand a few TNF terms in their general meaning. For example, in MPI, *latency* means the time required to send a null-length message; whereas, in the TNF context, latency is the elapsed time for an interval (the period bracketed by a pair of TNF events). In TNF, a *thread* may refer to any thread of control, such as an MPI process.

First Prism Case Study – Point-to-Point Message Passing

The benchmark code considered in this case study is a popular HPC benchmark in computational fluid dynamics (CFD), the NPB 2.3 BT code.

Data Collection

In our NPB 2.3 BT example, we first run the benchmark using these environment variable settings

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

These settings are not required for Prism tracing. We use them to profile our code as it would run in production. See Appendix B and the *Sun MPI Programming and Reference Guide* for more information about using Sun MPI environment variables for high performance.

We run the benchmark under the Prism programming environment on a single, shared-memory node using 25 processes with the command:

You must specify the `-n` argument to the Prism environment (with any message-passing program) even if you use only one process. The run took 135 seconds to complete.

```
% prism -n 25 a.out
```

To use Prism tracing on a 32-bit binary within a Solaris environment, start the Prism environment using the `-32` command line option. For example,

```
% prism -32 -n 25 a.out
```

▼ To Collect Performance Data

1. **Click on Collection (from the Performance menu).**
2. **Click on Run (from the Execute menu).**
3. **Click on Display TNF Data (from the Performance menu).**

   The timeline window will appear. The horizontal axis shows time, in milliseconds (ms). The vertical axis shows MPI process rank, which is labeled as the virtual thread ID.

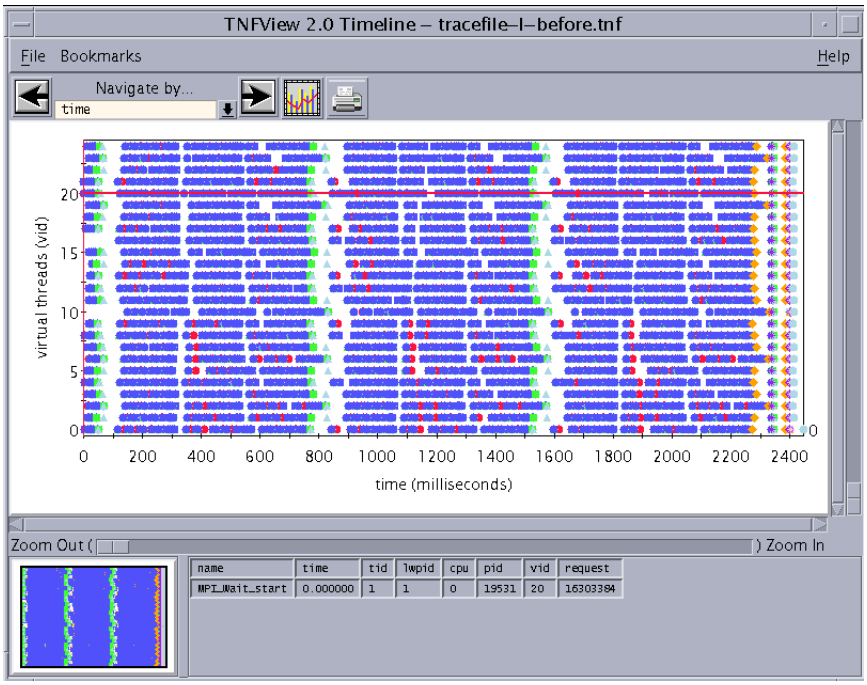


FIGURE 7-5 The Timeline View of TNF Probes

The Prism programming environment creates TNF trace data files by merging data from buffers that belong to each process in your program. The trace buffers have limited capacity (by default, each buffer contains 128 Kbytes). If the TNF probes generate more data than the buffers can hold, the buffers *wrap*, overwriting earlier trace data with later data. For example, FIGURE 7-5 shows 3 iterations of the CFD benchmark program, spanning roughly 2 seconds. However, the benchmark program executes 200 iterations and spans a total elapsed time of approximately 135 seconds. The trace file displayed in the window contains only 1/70 of the events generated during the run of the full benchmark. Since the last iterations of the benchmark are representative of the whole run *in this example*, this 2-second subset of the benchmark program's run is appropriate. You must determine whether buffer wraparound affects your program's profiling data. For information about controlling buffer wraparound, see "Coping With Buffer Wraparound" on page 121.

Message-Passing Activity At a Glance

In FIGURE 7-5, we see three iterations, each taking roughly 700 ms. By holding down the middle mouse button while dragging over one such iteration, you can produce the expanded view shown in FIGURE 7-6. More detail becomes evident. There are three important phases in each iteration, which correspond to the x, y, and z axes in

this three-dimensional computation. Some degree of synchronization among MPI processes is evident. Consecutive blocks of 5 processes each are synchronized at the end of phase 1 (at about 1050 ms), while every fifth process is synchronized at the end of phase 2 (at about 1300 ms). This is indicative of the benchmark running on an underlying 5x5 process grid.

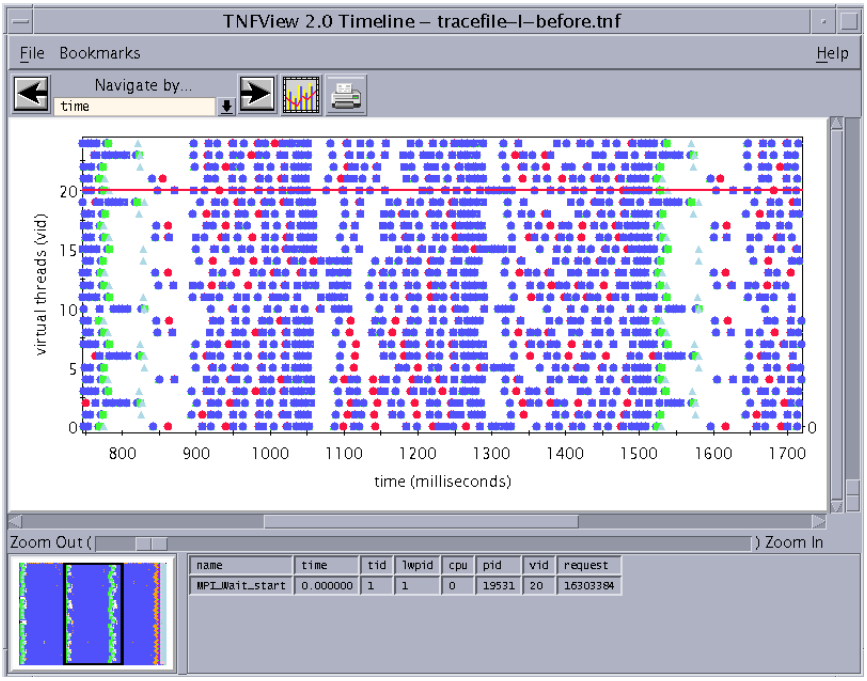


FIGURE 7-6 Expanded View of One Iteration

Summary Statistics of MPI Usage

We now change views by clicking on the graph button at the top of `tnfview`'s main window. A new window pops up and in the Interval Definitions panel you can see which MPI APIs were called by the benchmark, as in FIGURE 7-7.

To study usage of a particular MPI routine, click on the routine's name in the list under Interval Definitions and then click on Create a dataset from this interval definition. The window will resemble FIGURE 7-7.

While each point in FIGURE 7-5 or FIGURE 7-6 represented an *event*, such as the entry to or exit from an MPI routine, each point in FIGURE 7-7 is an *interval* — the period of time between two events that is spent inside the MPI routine. The scatter plot graph

shows three 700-ms iterations with three distinct phases per iteration. The vertical axis shows that `MPI_Wait` calls are taking as long as 60-70 ms, but generally much less.

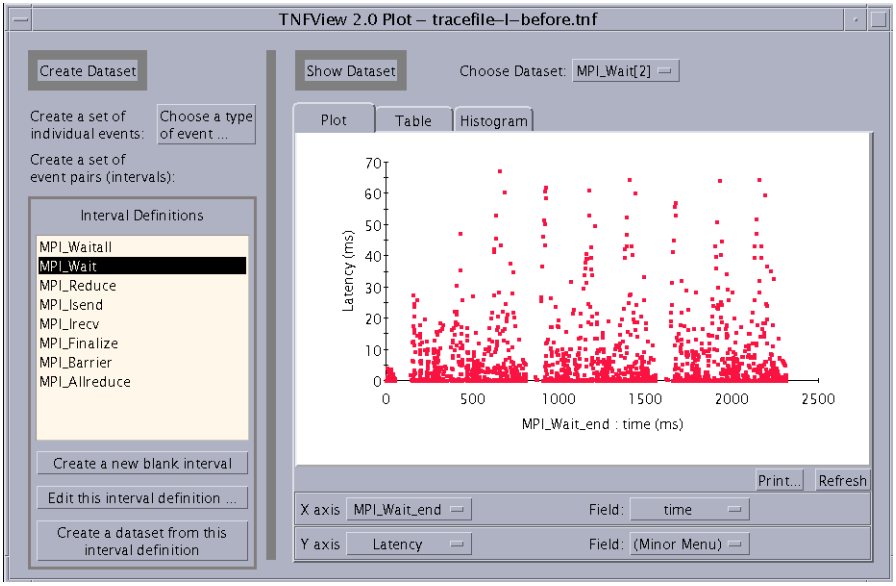


FIGURE 7-7 Graph Window Showing a Scatter Plot of Interval Data

Next, click on the Table tab to produce a summary similar to that depicted in FIGURE 7-8. The first column (Interval Count) indicates how many occurrences of the interval are reported, the second column (Latency Summation) reports the total time spent in the interval, the third column gives the average time per interval, and the fourth column lists the data element used to group the intervals. (In the current release, `tnfview` usually reports the fourth column in hexadecimal format.) In the case of FIGURE 7-8, some *threads* (MPI processes) spent as long as 527 ms in `MPI_Wait` calls. Since only about 2.6 seconds of profiling data is represented, this represents roughly 20 percent of the run time. By repeatedly clicking on other intervals (MPI calls) in the list under Interval Definitions and then on Create a dataset from the selected interval definition, you can examine times spent in other MPI calls and verify that `MPI_Wait` is, in fact, the predominant MPI call for this benchmark.

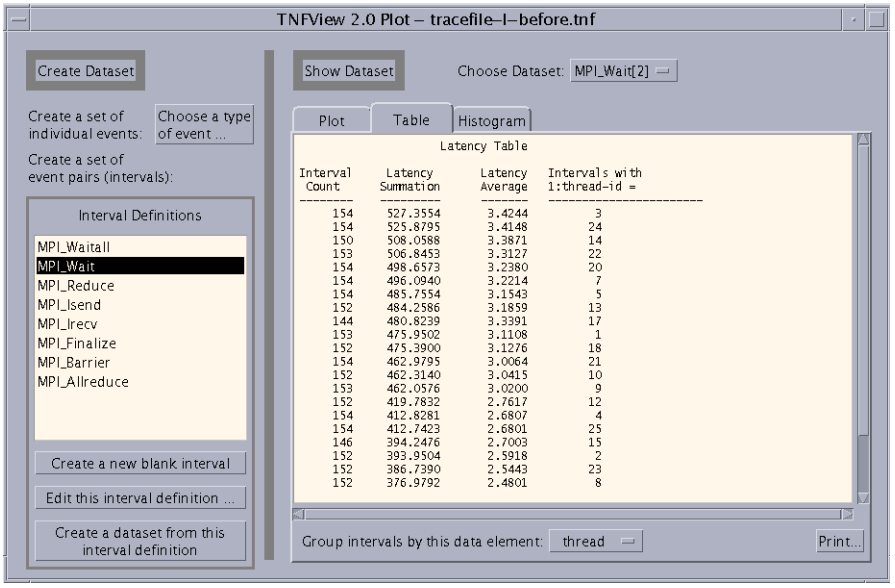


FIGURE 7-8 Graph Window Showing a Summary Table of Interval Data

To understand this further we can analyze the dependence of `MPI_Wait` time on message size using the Plot, Table, or Histogram views. For example, click on the Plot tab to return to the view of FIGURE 7-7. The X axis is being used to plot fields in the `MPI_Wait_end` probe. Click on the X-axis Field button and choose bytes. Then, click on Refresh and you should see a view like the one in FIGURE 7-9.

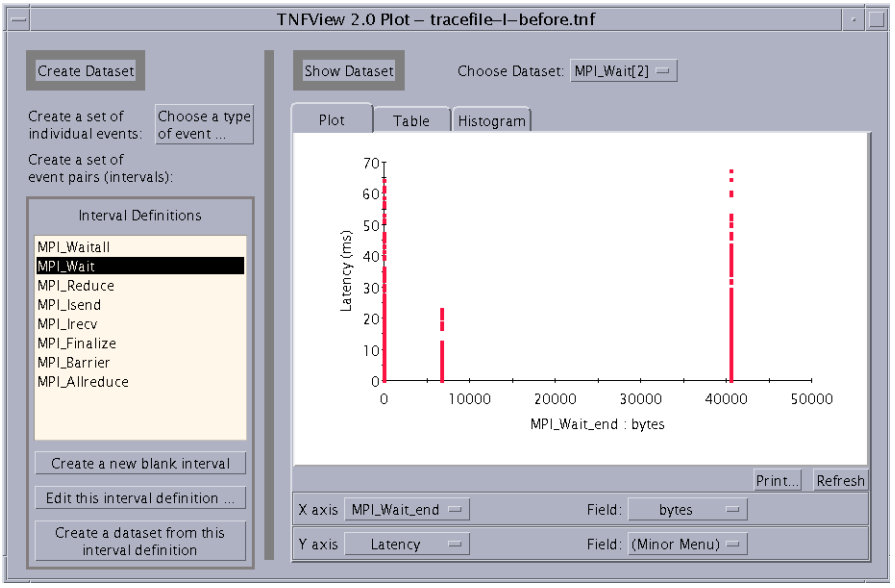


FIGURE 7-9 Scatter Plot of Time Spent in `MPI_Wait` as a Function of the Number of Bytes Received. Zero Bytes Indicate The Completion of a Send, Rather Than a Receive.

The byte counts on the X axis in FIGURE 7-9 are bytes received. In particular, `MPI_Wait` calls that report zero bytes are completing nonblocking send operations. The rest are completing nonblocking receive operations.

Note – In MPI, the number of bytes in a message is known on the sender's side when the send is posted and on the receiver's side when the receive is completed (rather than posted). Sun MPI's TNF probes report the number of bytes in a message in the same way. For example, a nonblocking send reports TNF byte information with the `MPI_Isend` call while a nonblocking receive reports TNF byte information with the `MPI_Wait` call. For more information on what information TNF arguments report, see the *Sun MPI Programming and Reference Guide*.

We see from the figure that an appreciable amount of time is being spent waiting for sends to complete. This is indicative of buffer congestion, which prevents senders from writing their messages immediately into shared-memory buffers. We can remedy this situation by rerunning the code with Sun MPI environment variables set for large buffers.

FIGURE 7-9 shows that there are messages that are just over 40 Kbytes. To increase buffering substantially past this size, we add:

```
% setenv MPI_SHM_CPOOLSIZE 102400
% setenv MPI_SHM_CYCLESTART 0x7fffffff
```

to our list of runtime environment variables. For further information about Sun MPI environment variables, see Appendix B of this volume.

Finding Hot Spots

Timings indicate that adding these new environment variables speeds the benchmark up by 5 percent. This speedup is encouraging since our code spends only 20 percent of the time in MPI in the first place. In particular, the time spent on `MPI_Wait` calls that terminate `MPI_Isend` calls has practically vanished.

Nevertheless, `MPI_Wait` continues to consume the most time of any MPI calls for this job. Having rerun the job with larger MPI buffers, we may once again generate a scatter plot of time spent in `MPI_Wait` calls (`MPI_Wait` latency) as a function of elapsed time. Compare the new distribution shown in FIGURE 7-10 and with the one shown in FIGURE 7-7. The new distribution shows that `MPI_Wait` times have decreased dramatically. However, tall fingers of high-latency calls shoot up roughly every 200 ms. These fingers are a symptom of message-passing traffic functioning as global synchronizations. Indeed, these synchronizations occur as computation in this three-dimensional application goes from x to y to z axis. This MPI time has little to do with how fast MPI is moving data.

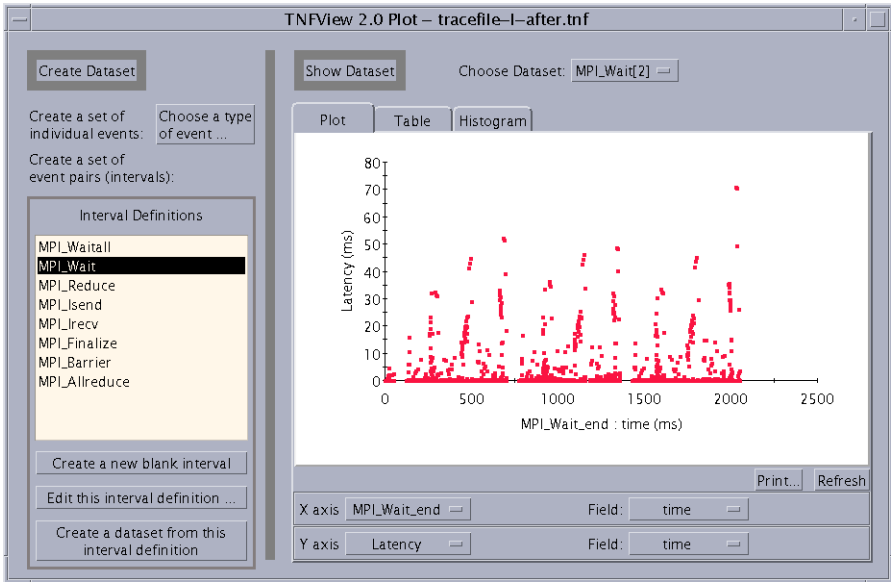


FIGURE 7-10 Scatter Plot of Time Spent in `MPI_Wait` as a Function of Elapsed Time, Running with Sun MPI Environment Variables That Relieve Buffer Congestion

To study such a slowdown in detail, click on a high-latency point in the scatter plot. This centers the cross-hairs back in the timeline view on the selected interval. Within the timeline plot, you can navigate through a sequence of events using the `tnfview` Navigate by... pull-down list, and left and right (*previous* and *next*) arrows. For example, by pulling down Navigate by... and selecting current vid, you can restrict navigation forward and backward on one particular MPI process. By using these detailed navigation controls, you can confirm which sequence of MPI calls characterize hot spots, or which interprocess dependencies are causing long synchronization delays.

Second Prism Case Study – Collective Operations

Our first case study centered about point-to-point communications. Now, let us turn our attention to one based on collective operations. We examine the NPB 2.3 IS code, which sorts sets of keys. The benchmark was run under the Prism programming environment on a single, shared-memory node using 16 processes. Once again, we begin by setting Sun MPI environment variables:

```
% setenv MPI_SPIN 1
% setenv MPI_PROCBIND 1
```

since we are interested in the performance of this benchmark as a *dedicated* job.

Synchronizing Skewed Processes: Timeline View

The message-passing part of the code involves a bucket sort, implemented with an `MPI_Allreduce`, an `MPI_Alltoall`, and an `MPI_Alltoallv`, though no such knowledge is required for effective profiling with the Prism programming environment. Instead, running the code under the Prism environment, we quickly see that the most time-consuming MPI calls are `MPI_Alltoallv` and `MPI_Allreduce`. (See "Summary Statistics of MPI Usage" on page 112.) Navigating a small section of the timeline window, we see a tight succession of `MPI_Allreduce`, `MPI_Alltoall`, and `MPI_Alltoallv` calls. One such iteration is shown in FIGURE 7-11; we have shaded and labeled time-consuming sections.

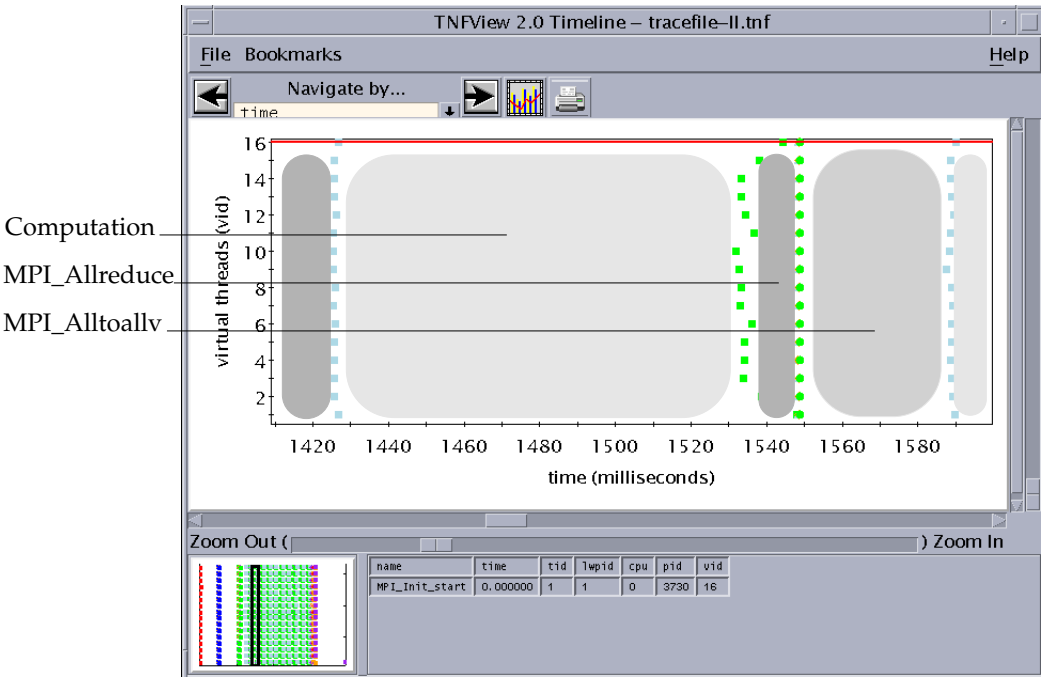


FIGURE 7-11 One Iteration of the Sort Benchmark

Synchronizing Skewed Processes: Scatter Plot View

The reason `MPI_Allreduce` costs so much time may already be apparent from this timeline view. The start edge of the `MPI_Allreduce` region is ragged, while the end edge is flat.

We can see even more data in one glance by going to a scatter plot. In FIGURE 7-12, time spent in `MPI_Allreduce` (its *latency*) is plotted against the finishing time for each call to this MPI routine. There is one warm-up iteration, followed by a brief gap, and then ten more iterations, evenly spaced. In each iteration, an MPI process might spend as long as 10 to 30 ms in the `MPI_Allreduce` call, but other processes might spend vanishingly little time in the reduce. The issue is not that the operation is all that time consuming, but simply that it is a synchronizing operation, and so early-arriving processes have to spend some time waiting for latecomers.

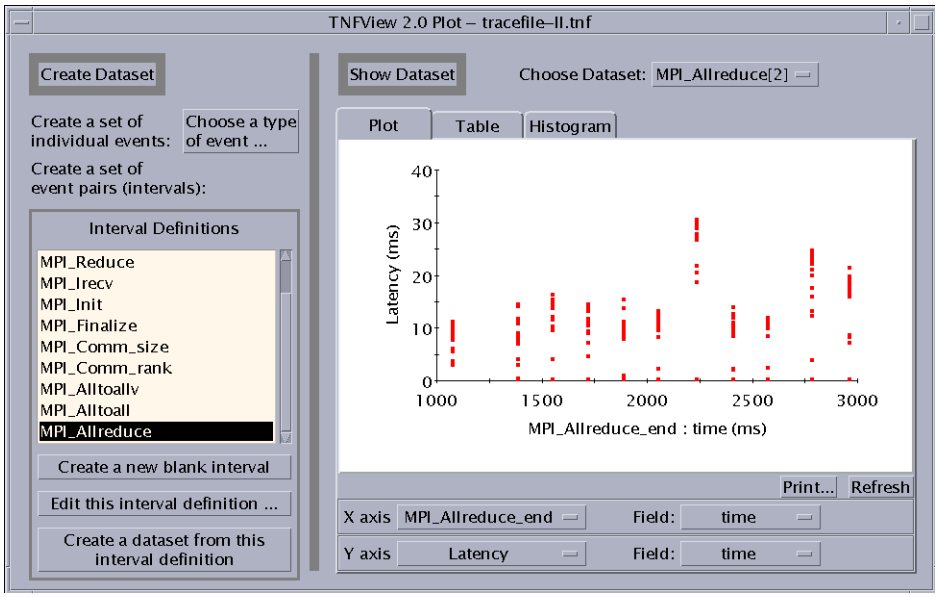


FIGURE 7-12 Scatter Plot of MPI_Allreduce Latencies (x axis: MPI_Allreduce_end)

As in FIGURE 7-10, brief, well-defined instants of very high-latency message-passing calls typically signal moments in code execution of considerable interprocess synchronization.

The next MPI call is to `MPI_Alltoall`, but from our Prism profile we discover that it occurs among well-synchronized processes (thanks to the preceding `MPI_Allreduce` operation) and uses very small messages (64 bytes). It consumes very little time.

Interpreting Performance Using Histograms

The chief MPI call is this case study is the `MPI_Alltoallv` operation. The processes are still well synchronized, as we saw in FIGURE 7-11, but we learn from the Table display that there are on average 2 Mbytes of data being sent or received per process. Clicking on the Histogram tab, we get the view seen in FIGURE 7-13. There are a few, high-latency outliers, which a scatter plot would indicate take place during the first warm-up iteration. Most of the calls, however, take roughly 40 ms. The effective bandwidth for this operation is therefore:

```
(2 Mbyte / process) * 16 processes / 40 ms = 800 Mbyte/second
```

Basically, each datum undergoes two copies (one to shared memory and one from shared memory) and each copy entails two memory operations (a load and a store), so this figure represents a memory bandwidth of 4 * 800 Mbyte/s = 3.2 Gbyte/s. This benchmark was run on an HPC 6000 server, whose backplane is rated at 2.6 Gbyte/s. Our calculation is approximate, but it nevertheless indicates that we are seeing saturation of the SMP backplane and we cannot expect to do much better with our `MPI_Alltoallv` operation.

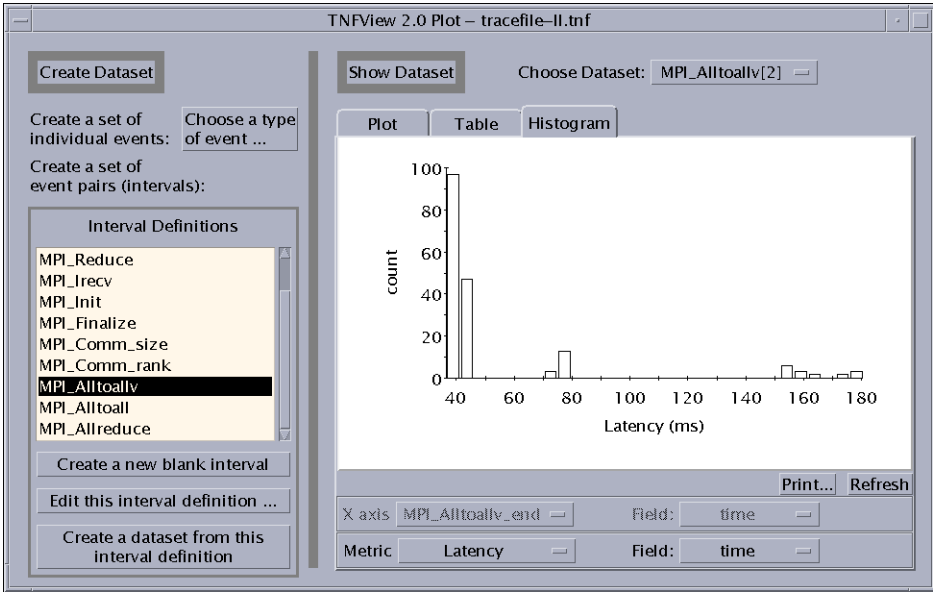


FIGURE 7-13 Histogram of `MPI_Alltoallv` Latencies

Performance Analysis Tips

While Prism tracing might involve only a few mouse clicks, more advanced techniques offer more sophisticated results.

Coping With Buffer Wraparound

Event-based profiling can collect a lot of data. TNF probe data collection employs buffer wraparound, so that once a buffer file is filled the newer events will overwrite older ones. Thus, final traces do not necessarily report events starting at the beginning of a program and, indeed, the time at which events start to be reported

may vary slightly from one MPI process to another, depending on the amount of probed activity on each process. Nevertheless, trace files will generally show representative profiles of an application since newer, surviving events tend to represent execution during steady state.

If buffer wraparound is an issue, then solutions include:

- Scaling down the run (number of iterations or number of processes).
- Using larger trace buffers.
- Selective enabling of probes.
- Profiling isolated sections of code by terminating jobs early.

  Profiling isolated sections of code by modifying user source code.

  Profiling isolated sections of code by isolating sections at run time.

Scaling Down the Run

You should usually perform code profiling and tuning on "stripped down" runs so that many profiling experiments may be run. We describe these precautions in detail at the beginning of this chapter.

Using Larger Trace Buffers.

To increase the size of trace buffers beyond the default value, use the Prism command:

`(prism all)` **`tnffile`** *filename size*

where *size* is the size in Kbytes of the trace buffer for each process. The default value is 128 Kbytes. Larger values, such as 256, 512, or 1024, can sometimes prove more useful.

By default, the Prism programming environment places trace buffers in `/usr/tmp` before they are merged into the user's trace file. If this file partition is too small for very large traces, you can redirect buffers to other directories using the `PRISM_TNFDIR` environment variable. In order to minimize profile disruption caused by writing very large trace files to disk, you should use local file systems such as `/usr/tmp` and `/tmp` whenever possible instead of file systems that are mounted over a network.

Note – While the Prism programming environment usually cleans up trace buffers after the final merge, abnormal conditions could cause the Prism environment to leave large files behind. Users who abort profiling sessions with large traces should check `/usr/tmp` periodically for large, unwanted files.

Selectively Enabling Probes

You can focus data collection on the events that are most relevant to performance in order either to reduce sizes of buffer files or to make profiling less intrusive. Prism performance analysis can disturb an application's performance characteristics, so you should consider focusing data collection even if larger trace buffers are an option.

TNF probes are organized in probe groups. The probe groups are structured as follows:

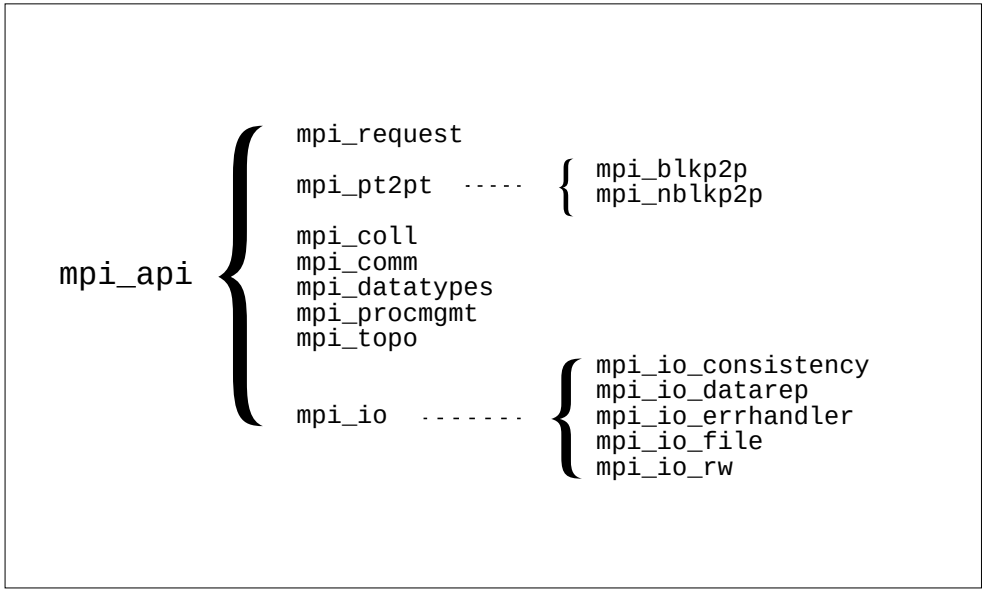


FIGURE 7-14 Sun MPI TNF Probe Groups

There are several probes that belong to both the `mpi_request` group and the `mpi_pt2pt` group. For further information about probe groups, see the *Sun MPI Programming and Reference Guide*.

For message-passing performance, typically the most important groups are:

- `mpi_pt2pt` – point-to-point message passing
- `mpi_request` – other probes for nonblocking point-to-point calls
- `mpi_coll` – collectives
- `mpi_io_rw` – file I/O

Tracing Isolated Sections of Code — Terminating Data Collection Mid-Course

If you are especially interested in the steady-state performance characteristics of the code, you might experiment with terminating a run early. If you choose to terminate the run early, you will not spend time waiting for job completion when you already have the profiling data you want. Further, by letting the job finish you risk the possibility of uninteresting, post-processing steps overwriting the interesting steady-state trace data.

To interrupt program execution, click on Interrupt, set a breakpoint, or use a Prism command such as:

```
sh sleep 200
```

to wait a prescribed length of time.

Then, turn off data collection and view the performance data. Once you have viewed performance data, you cannot resume collecting data in the same run. However, you can run your program to completion.

Tracing Isolated Sections of Code — From Within Source Code

You can turn TNF data collection on and off within user source code, using the routines `tnf_process_disable`, `tnf_process_enable`, `tnf_thread_disable`, and `tnf_thread_enable`. Since these are C functions, Fortran usage would require added hints for the compiler, as follows:

```
call tnf_process_disable()   !$pragma c(tnf_process_disable)
call tnf_process_enable()    !$pragma c(tnf_process_enable)
call tnf_thread_disable()    !$pragma c(tnf_thread_disable)
call tnf_thread_enable()     !$pragma c(tnf_thread_enable)
```

Whether you call these functions from C or Fortran, you must then link with `–ltnfprobe`. For more information, see the Solaris man pages on these functions.

Tracing Isolated Sections of Code — At Run Time

The Prism programming environment allows users to turn collection on and off during program execution whenever execution is stopped: for example, with a break point or by using the `interrupt` command.

If the profiled section will be entered and exited many times, data collection may be turned on and off automatically using tracepoints. Note that the term "trace" is used now in a different context. For TNF use, a *trace* is a probe. For the Prism programming environment and other debuggers, a *tracepoint* is a point where execution stops and possibly an action takes place but, unlike a *breakpoint*, program execution resumes after the action.

For example, if data collection should be turned on at line 128 but then off again at line 223, you can specify:

```
(prism all) trace at 128 {tnfcollection on}
(prism all) trace at 223 {tnfcollection off}
```

If you compiled and linked the application with high degrees of optimization, then specification of line numbers may be meaningless. If you compiled and linked the application without `-g`, then specifying numbers will not work. In such cases, you can turn data collection on and off at entry points to routines using `trace in` *routine* syntax, providing that those routines have not been inlined. For example:

```
(prism all) trace in routine1 {tnfcollection on}
(prism all) trace in routine2 {tnfcollection off}
```

Prism tracepoints have detectable effects on the behavior of the code being profiled. The effects of the tracepoints can originate from:

- Displaying a message when a tracepoint is encountered (modifying the event by using the Prism Event Table can suppress such a message)
- Making operating system calls
- Synchronizing MPI processes
- Responding to a breakpoint
- Polling after a breakpoint

For this reason, you should not use `trace` commands inside inner loops, where they would execute repeatedly, distorting your program's performance. Use Prism tracepoints to turn data collection on and off only around large amounts of code execution.

Inserting TNF Probes Into User Code

While Sun HPC ClusterTools libraries have TNF probes for performance profiling, user code probably will not. You can add probes manually, but since they are C macros you can add them only to C and C++ code. To use TNF probes from Fortran code, you must make calls to C code, such as in this C file, `probes.c`:

```
#include <tnf/probe.h>
void my_probe_start_(char *label_val) {
  TNF_PROBE_1(my_probe_start,"user_probes","",
    tnf_string,label,label_val);
}
void my_probe_end_  (double *ops_val) {
  TNF_PROBE_1(my_probe_end  ,"user_probes","",
    tnf_double,ops,*ops_val);
}
```

The `start` routine accepts a descriptive string, while the `end` routine takes a double-precision operation count.

Then, using Fortran, you might write in `main.f`:

```
DOUBLE PRECISION OPERATION_COUNT
OPERATION_COUNT = 2.D0 * N
CALL MY_PROBE_START("DOTPRODUCT")
XSUM = 0.D0
DO I = 1, N
   XSUM = XSUM + X(I) * Y(I)
END DO
CALL MY_PROBE_END(OPERATION_COUNT)
```

Note – The Fortran compiler converts routine names to lowercase and appends an underscore character.

To compile and link, use:

```
% tmcc -c probes.c
% tmf77 main.f probes.o -lmpi -ltnfprobe
```

By default, the Prism command `tnfcollection on` enables all probes. Alternatively, these sample probes could be controlled through their probe group `user_probes`. Profile analysis can use the interval `my_probe`.

For more information on TNF probes, consult the man page for `TNF_PROBE(3X)`.

Collecting Data Batch Style

For more involved data collection experiments, you can collect TNF profiling information in *batch* mode, for viewing and analysis in a later, interactive session. Such collection may be performed using the commands-only mode of the Prism environment, invoked with `prism -C`. For example, the simplest data collection experiment would be

```
% prism -C -n 8 a.out << EOF
tnfcollection on
tnfenable mpi_pt2pt
tnfenable mpi_request
tnfenable mpi_coll
tnfenable mpi_io_rw
run
wait
quit
EOF
```

The wait command is needed to keep file merge from happening until after the program has completed running. See the *Prism User's Guide* for more information on commands-only mode.

Accounting for MPI Time

Sometimes you will find it difficult to account for MPI activity. For example, if you issue a nonblocking `send` or `receive` (`MPI_Isend` or `MPI_Irecv`), the data movement may occur during that call, during the corresponding `MPI_Wait` or `MPI_Test` call, or during any other MPI call in between.

Similarly, general polling (such as with the environment variable `MPI_POLLALL`) may skew accounting. For example, an incoming message may be read during a send call because polling causes arrivals to be polled aggressively.

TNF Tracing Using the `tnfdump` Utility

You can implement custom post-processing of TNF data using the `tnfdump` utility, which converts TNF trace files, such as the one produced by the Prism programming environment, into an ASCII listing of timestamps, time differentials, events, and probe arguments.

To use this command, specify

```
% tnfdump filename
```

where *filename* is the name of the TNF trace data file produced by the Prism programming environment.

The resulting ASCII listing, produced on the standard output, can be several times larger than the tracefile and may require a wide window for viewing. Nevertheless, it is full of valuable information.

For more information about the `tnfdump` command, see the `tnfdump`(`1`) man page.

Other Profiling Approaches

Both MPI and the Solaris 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_` APIs that simply call the `PMPI_` ones, with timer calls around the `PMPI_` call.

You may 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 might compile a wrapper such as the one shown below, and link it into your code before

–lmpi. 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 may be used even with application binaries that have already been linked. See the Solaris man page for ld for more information on the environment variable LD_PRELOAD.

You can get profiling libraries from independent sources for use with Sun MPI. Typically, their functionality is rather limited compared to that of the Prism environment with TNF, but for certain applications their use may be more convenient or they may represent useful springboards for particular, customized profiling activities. 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, see the *Sun MPI Programming and Reference Guide*.

Inserting MPI Timer Calls

Sun HPC ClusterTools implements the Sun MPI timer call MPI_Wtime (demonstrated in the example below) 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 gethrtime, see the gethrtime(3C) man page.

When profiling multiprocess codes, you need to 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 may accumulate time in the interesting, timed portion, waiting for process 0 (zero) to emerge from uninteresting initialization. This 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 `gprof`

The Solaris utility `gprof` may be used for multiprocess codes, such as those that use MPI. It can be helpful for profiling user routines, which are not automatically instrumented with TNF probes by Sun HPC ClusterTools software. Several points should be noted:

- Compile and link your programs with `–pg` (Fortran) or `–xpg` (C).
- 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 `gprof` has several limitations.

- `gprof` 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.

- `gprof` 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. `gprof` attributes time spent in the matrix multiplication equally to both caller, even if the large-matrix operations are substantially more time consuming.
- `gprof` does not count time spent in sleeps and yields, which can skew results.
- `gprof` loses the relationships between process ids (used to tag profile files) and MPI process ranks.
- `gprof` profiles from different processes may overwrite one another if a multiprocess job spans multiple nodes.
- `gprof` does not interoperate with Prism tracing.

Note that the 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 Analyzer for both MPI and non-MPI codes.

For more information about `gprof`, see the `gprof` man page. For more information about the Forte Developer performance analyzer, see the Analyzer documentation.

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 run time 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 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. This is less resource consuming than having the process poll, since 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 may act on a variety of messages. Some of the actions and messages may not pertain to the call at all, but may relate to past or future MPI calls.

To illustrate, consider the 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.

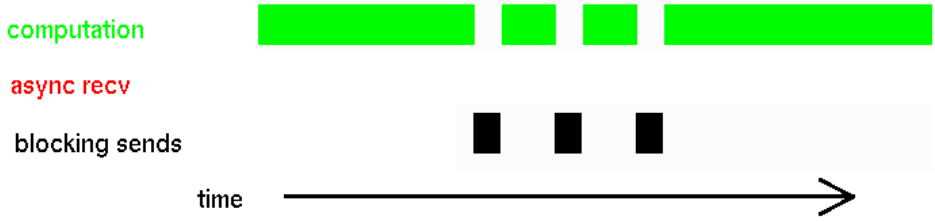


FIGURE A-1 Blocking Sends Interrupt Computation

Now, consider the 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 Conceptually, nonblocking operations overlap with computation.

In fact, however, progress on the nonblocking receive is suspended from the time the `MPI_Irecv()` returns until the instant Sun MPI is once again invoked, with the `MPI_Wait()`. There is no actual overlap of computation and communication, and the situation is as depicted in FIGURE A-3.

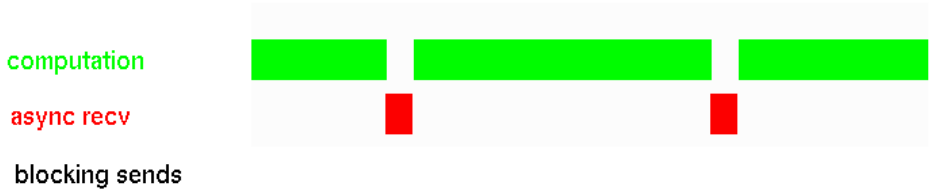


FIGURE A-3 Computational resources are devoted either to user computation or to MPI operations, but not both at once.

Nevertheless, reasonably good overlap between computation and nonblocking communication can be realized, since the Sun MPI library is able to progress a number of message transfers within one MPI call. Consider the code sequence

```
computation
CALL MPI_IRECV(REQ)
computation
CALL MPI_SEND()
computation
CALL MPI_SEND()
computation
CALL MPI_SEND()
computation
CALL MPI_WAIT(REQ)
computation
```

which combines the previous examples. 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 speedup results because scheduling of computation with the communication of multiple messages is better interwoven.

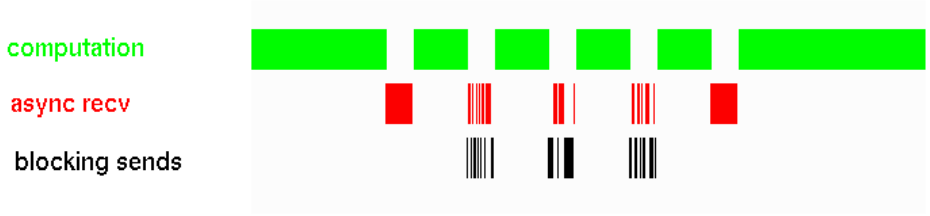


FIGURE A-4 Progress may be made on multiple messages by a single MPI call, even if that call does not explicitly reference the other messages.

In general, when Sun MPI is used to perform a communication call, a variety of other activities may also take place during that call, as we have just discussed. Specifically,

1. A process may progress any outstanding, nonblocking sends, depending on the availability of system buffers.
2. A process may progress any outstanding, nonblocking receives, depending on the availability of incoming data.
3. A process may 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 may choose to yield its computational resources to other processes if no useful progress is being made.
6. A process may 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 may be too congested for a send to proceed, or there may not yet be any more incoming data for a receive.

In contrast, a blocking MPI communication call may 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 above). 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 via 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 run time 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, may be read by the receiver. Thus, message passing is pipelined — a receiver may 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.

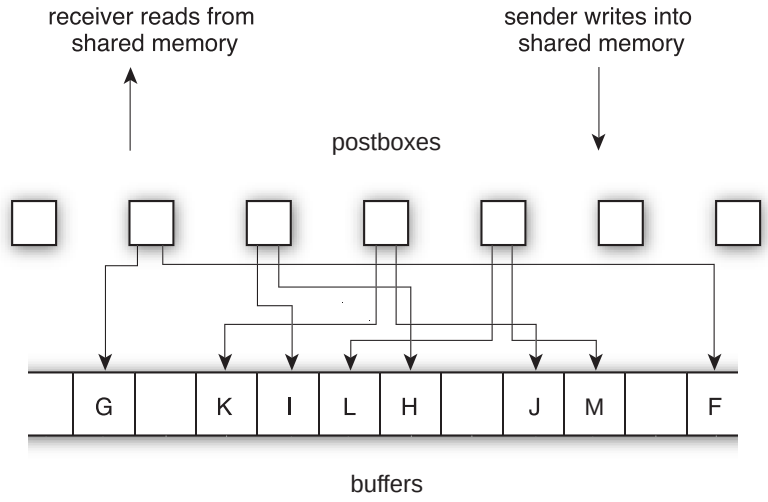


FIGURE A-5 Snapshot of a pipelined message. Message data is buffered in the labeled areas.

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.

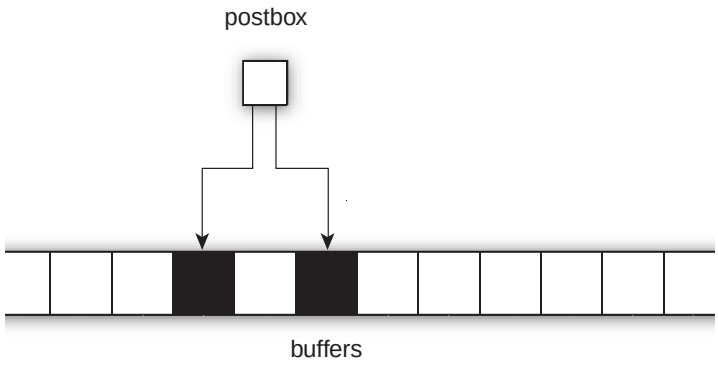


FIGURE A-6 A medium-size message uses only one postbox. Message data is buffered in the shaded areas.

Further, for extremely short messages, data is squeezed into the postbox itself, in place of pointers to buffers that would otherwise hold the data, illustrated in FIGURE A-7.

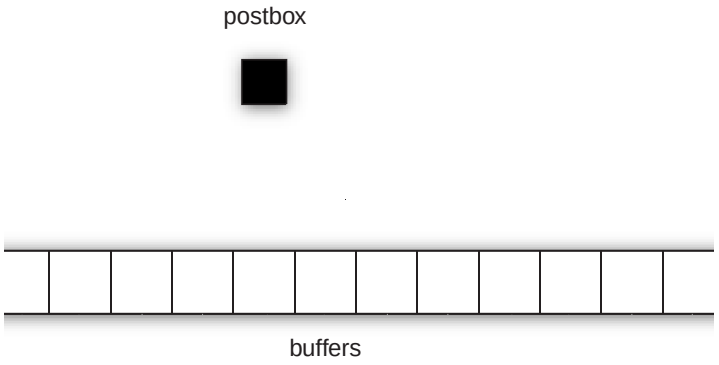


FIGURE A-7 A short message squeezes data into the postbox and does not use any buffers. Message data is buffered in the shaded area.

For very long messages, it may 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.

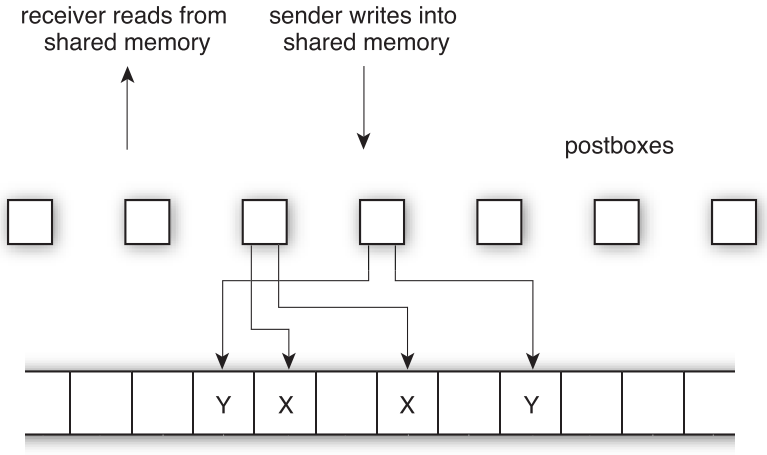


FIGURE A-8 First snapshot of a cyclic message. Message data is buffered in the labeled areas.

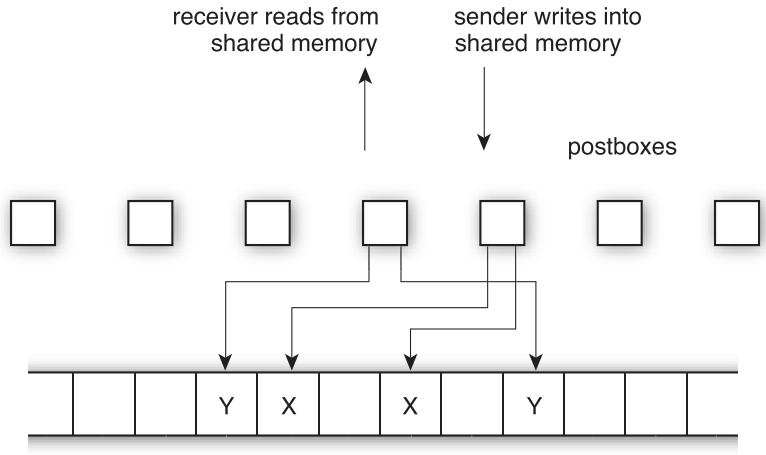


FIGURE A-9 Second snapshot of a cyclic message. Message data is buffered in the labeled areas.

Connection Pools Vs. 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.

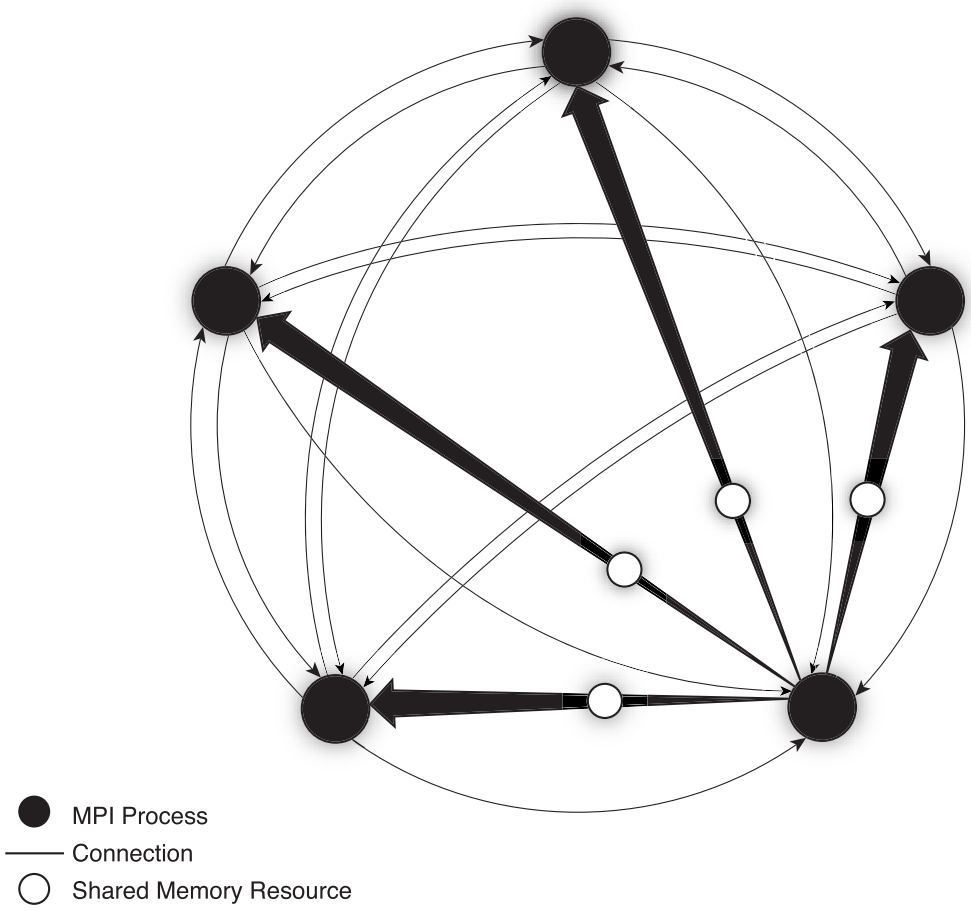


FIGURE A-10 Shared-memory resources that are dedicated per connection include postboxes and, optionally, buffer pools. The shared-memory resources available to one sender are shown.

By default, each connection also has its own pool of buffers. Users may 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.

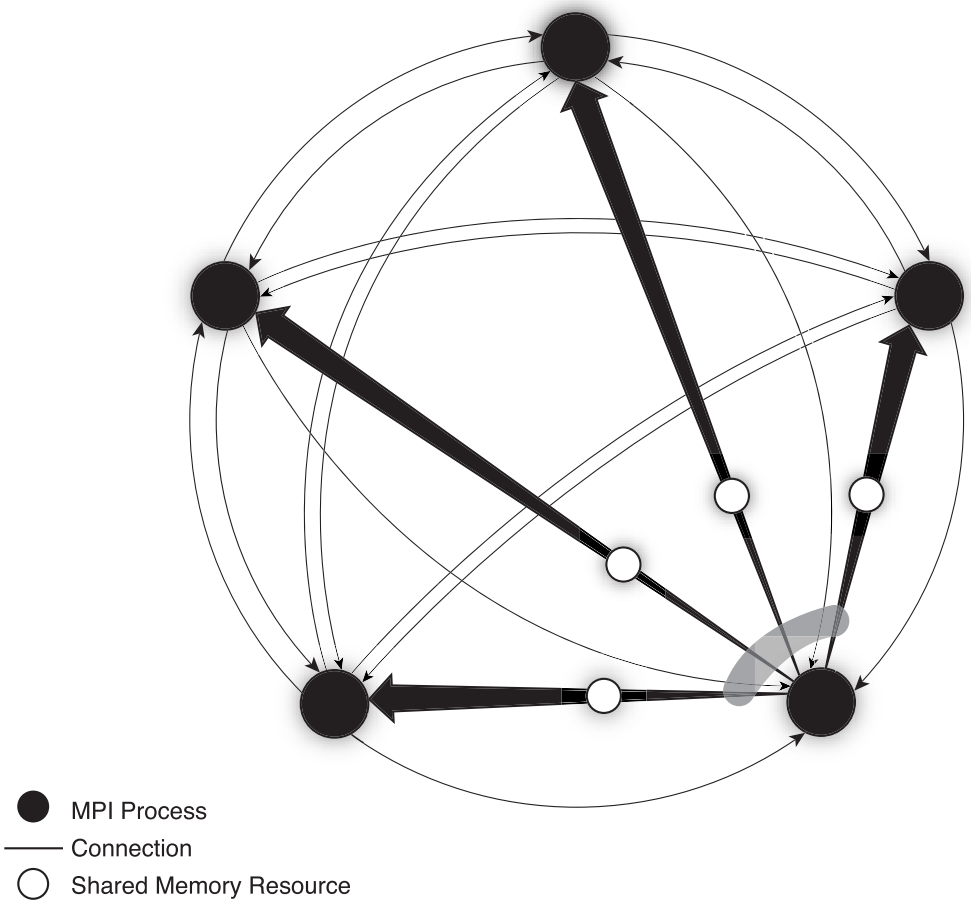


FIGURE A-11 Shared-memory resources per sender — for example, send-buffer pools. The send-buffer pool available to one sender is shown.

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
```

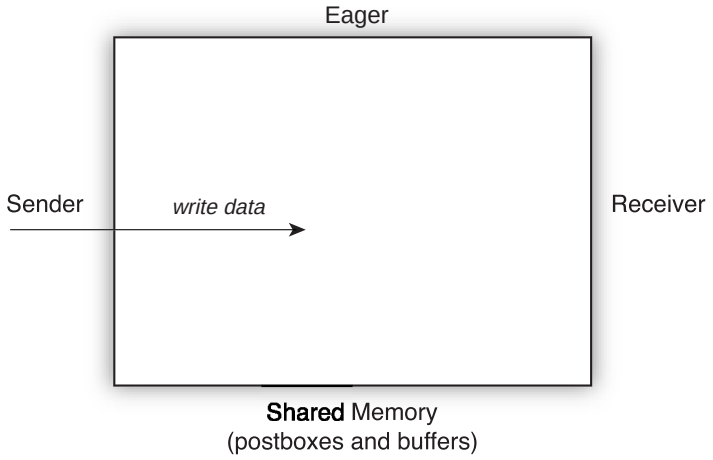


FIGURE A-12 Eager Message-Passing Protocol

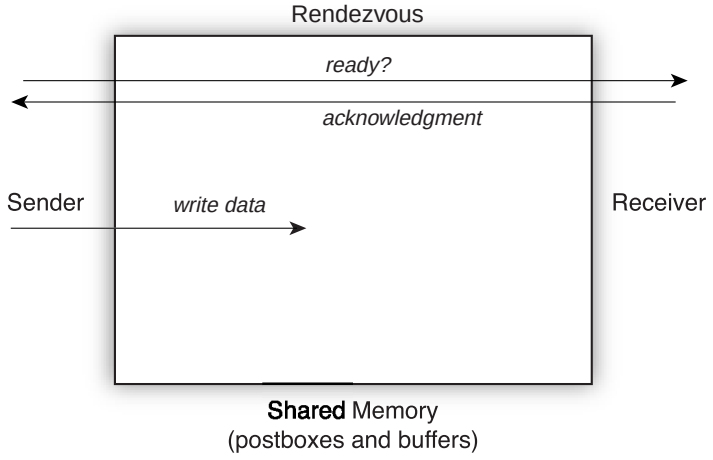


FIGURE A-13 Rendezvous Message-Passing Protocol

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 may be kept from immediately completing its operation if:

- Rendezvous is in force. (Rendezvous is suppressed by default.)
- The message is being sent cyclically. This behavior can be suppressed by setting `MPI_SHM_CYCLESTART` very high — for example,

  ```
  % setenv MPI_SHM_CYCLESTART 0x7fffffff
  ```

- The shared-memory resources (either buffers or postboxes) are temporarily congested. Shared-memory resources can be increased by setting Sun MPI environment variables at run time 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 can deliver effectively more buffer space to an application, improving performance. Multithreaded applications can suffer, however, since 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 may 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 `MPI_Init()`.

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.

RSM Point-to-Point Message Passing

Sun MPI supports high-performance message passing by means of the remote shared memory (RSM) protocol, running on the Sun Fire series high-performance cluster interconnect (when available). Sun MPI over RSM attains:

- Low latency from bypassing the operating system
- High bandwidth from striping messages over multiple channels

The RSM protocol has some similarities with the shared-memory protocol, but it also differs substantially, and environment variables are used differently.

Messages are sent over RSM in one of two fashions:

- Short messages are fit into multiple postboxes and no buffers are used.
- Pipelined messages are sent in 1024-byte buffers under the control of multiple postboxes.

Short-message transfer is illustrated in FIGURE A-14. The first 23 bytes of a short message are sent in one postbox, and 63 bytes are sent in each of the subsequent postboxes. No buffers are used. For example, a 401-byte message travels as 23+63+63+63+63+63+63=401 bytes and requires 7 postboxes.

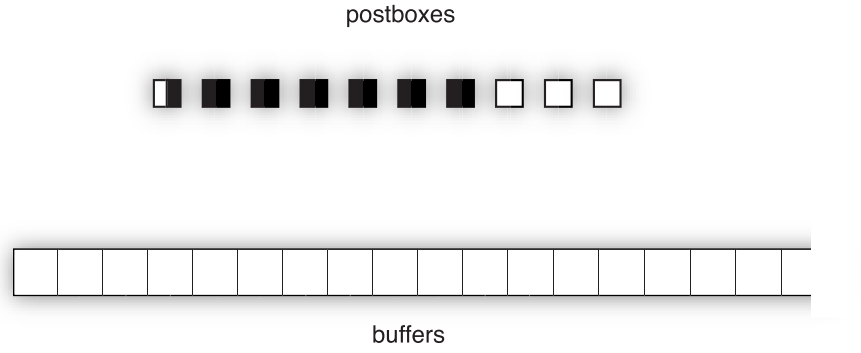


FIGURE A-14 A short RSM message. Message data is buffered in the shaded areas.

Pipelining is illustrated in FIGURE A-15. Postboxes are used in order, and each postbox points to multiple buffers.

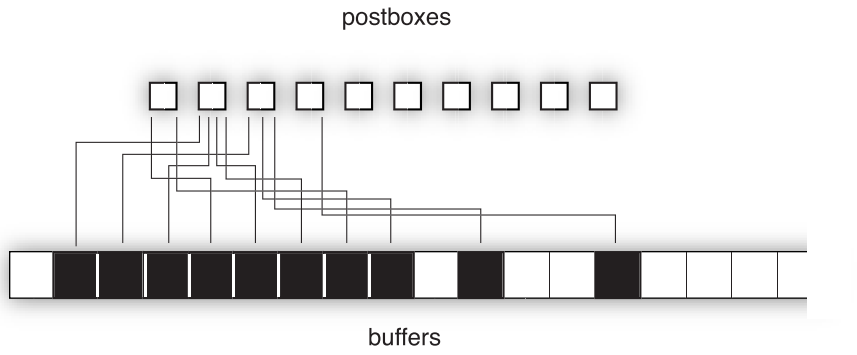


FIGURE A-15 A pipelined RSM message. Message data is buffered in the shaded areas.

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-16.

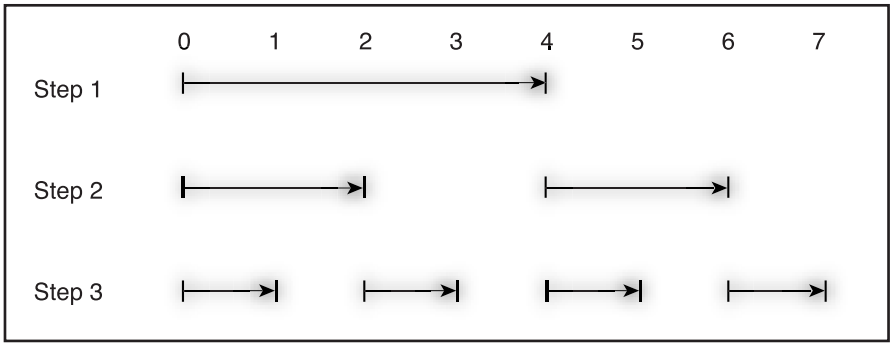


FIGURE A-16 Broadcast With Binary Fan-Out, First Example

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://www.supercomp.org/sc99/proceedings/papers/vandevaa.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-17.

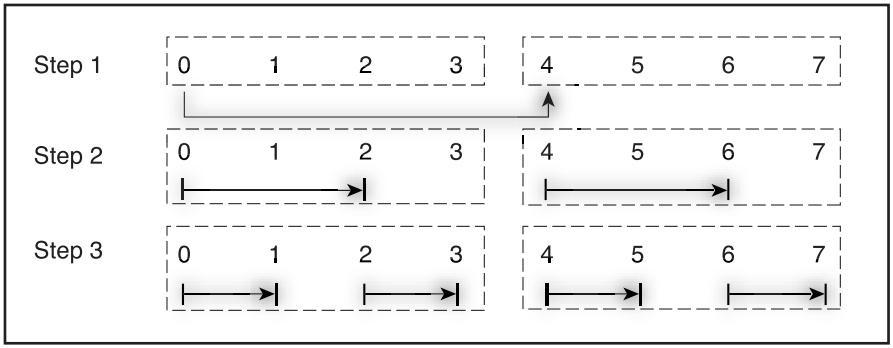


FIGURE A-17 Broadcast With Binary Fan-Out, Second Example

Unless the broadcast algorithm is network aware, however, nodes in the cluster may 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-18.

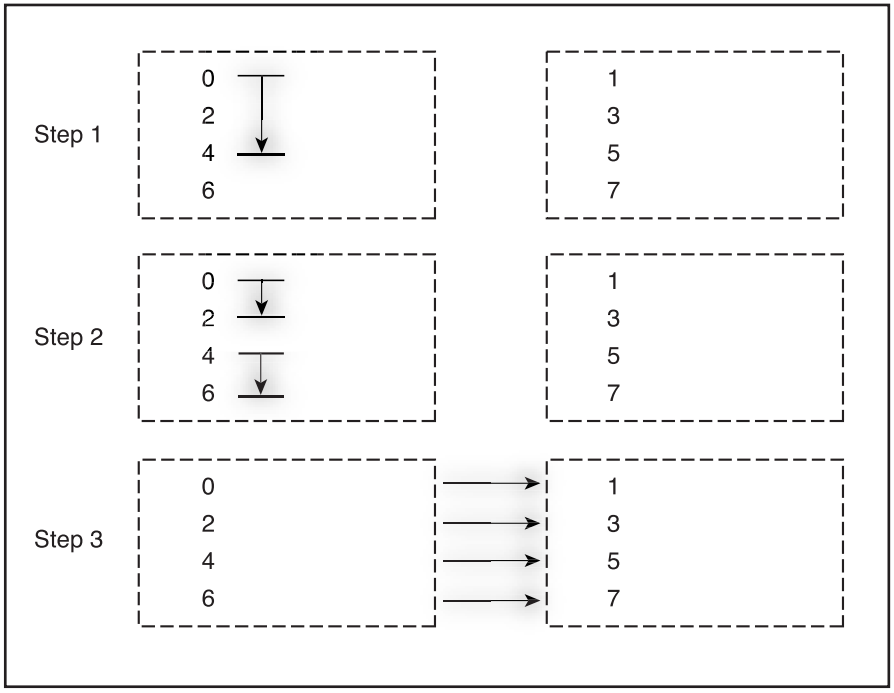


FIGURE A-18 Broadcast With Binary Fan-Out, Third Example

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-19.

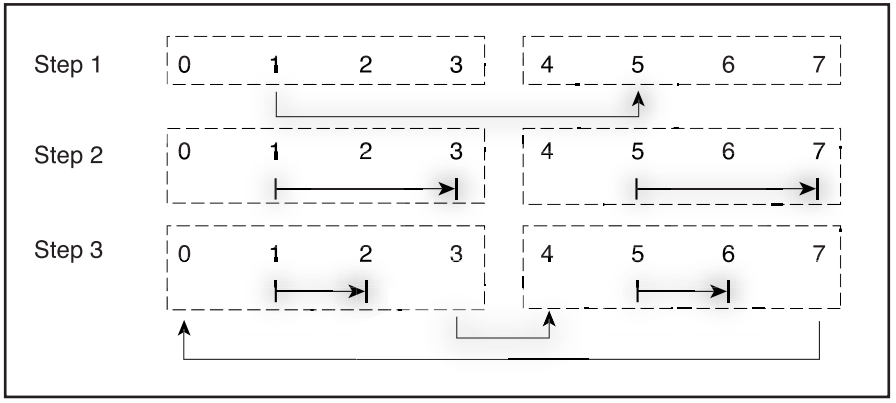


FIGURE A-19 Broadcast With Binary Fan-Out, Fourth Example

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

Communications between two processes on the same node in a cluster are 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-20. The data is written to the shared-memory area 7 times and read 7 times.

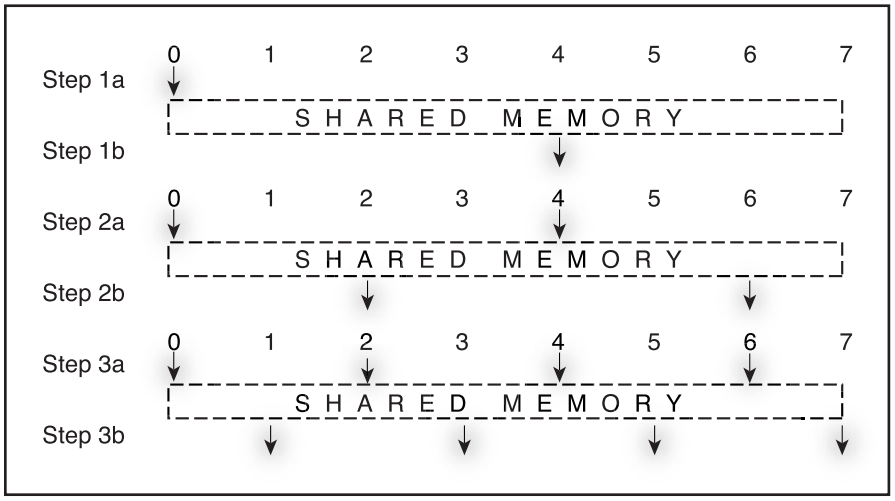


FIGURE A-20 Broadcast Over Shared Memory With Binary Fan-Out, First Case

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-21. With a collective implementation, data is written only once, and is made available much earlier to most of the processes.

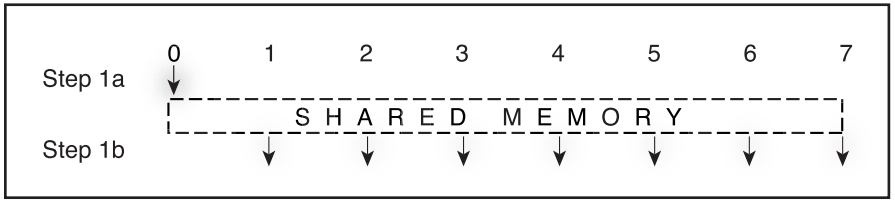


FIGURE A-21 Broadcast Over Shared Memory With Binary Fan-Out, Second Case

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-21, 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 may 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-16, can be shown schematically as in FIGURE A-22, view a, for a large message. Again, the time to complete this operation grows roughly as

$\log_2(NP)$ X 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-22, 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.

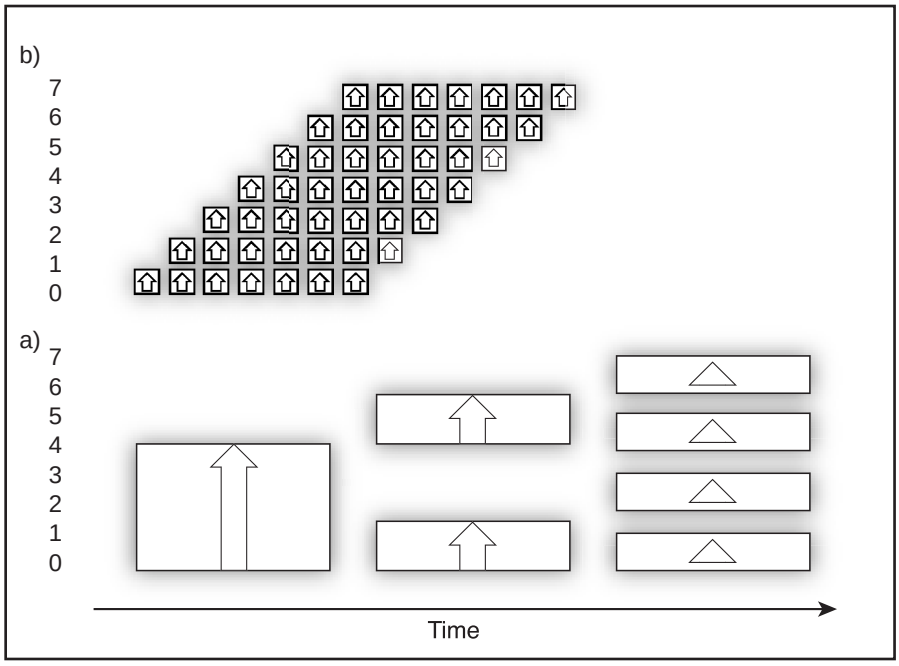


FIGURE A-22 Tree Broadcast versus Pipelined Broadcast of a Large Message

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 may 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 may also be used.

Performance models for different algorithms are employed to make run-time choices among the algorithms, based on process group topology, message size, and so on.

APPENDIX **B**

Sun MPI Environment Variables

This appendix describes some Sun MPI environment variablesand their effects on program performance. It covers the following topics:

Prescriptions for using MPI environment variables for performance tuning are provided in Chapter 6. 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 run time 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.

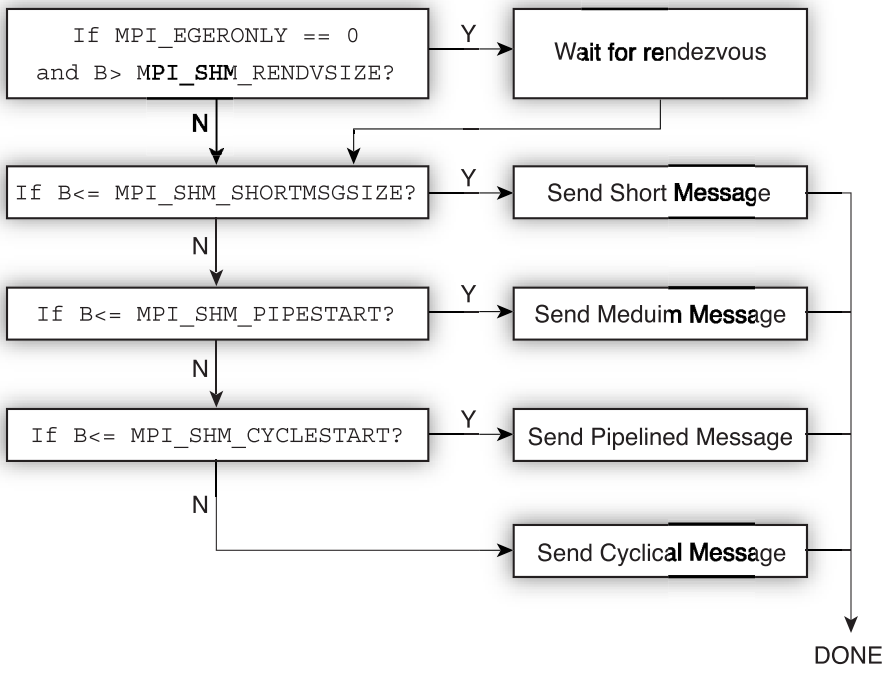


FIGURE B-1 Message of B Bytes Sent Over Shared Memory

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 mightconsume 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).

Cyclic message passing limits the size of shared memory that is needed to transfer even arbitrarily large messages.

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.
- Increase MPI_SHM_CYCLESTART so that no messages will be sent cyclically.
- 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 on page 166.

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 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.

RSM Point-to-Point Message Passing

The RSM protocol has some similarities with the shared-memory protocol, but it also differs substantially, and environment variables are used differently.

The maximum size of a short message is `MPI_RSM_SHORTMSGSIZE` bytes, with a default value of 401 bytes. Short RSM messages can span multiple postboxes, but they still do not use any buffers.

The most data that will be sent under any one postbox using buffers for pipelined messages is `MPI_RSM_PIPESIZE` bytes.

There are `MPI_RSM_NUMPOSTBOX` postboxes for each RSM connection.

If `MPI_RSM_SBPOOLSIZE` is unset, then each RSM connection has a buffer pool of `MPI_RSM_CPOOLSIZE` bytes. If `MPI_RSM_SBPOOLSIZE` is set, then each process has a pool of buffers that is `MPI_RSM_SBPOOLSIZE` bytes per remote node for sending messages to processes on the remote node.

Unlike the case of the shared-memory protocol, values of the `MPI_RSM_PIPESIZE`, `MPI_RSM_CPOOLSIZE`, and `MPI_RSM_SBPOOLSIZE` environment variables are merely requests. Values set with the `setenv` command or printed when `MPI_PRINTENV` is used might not reflect effective values. In particular, only when connections are actually established are the RSM parameters truly set. Indeed, the effective values could change over the course of program execution if lazy connections are employed.

Striping refers to passing a message over multiple hardware links to get the speedup of their aggregate bandwidth. The number of hardware links used for a single message is limited to the smallest of these values:

- `MPI_RSM_MAXSTRIPE`
- `rsm_maxstripe` (if specified by the system administrator in the `hpc.conf` file)
- the number of available links

When a connection is established between an MPI process and a remote destination process, the links that will be used for that connection are chosen. A job can use different links for different connections. Thus, even if `MPI_RSM_MAXSTRIPE` or `rsm_maxstripe` is set to 1, the overall job could conceivably still benefit from multiple hardware links.

Use of rendezvous for RSM messages is controlled with `MPI_RSM_RENDVSIZE`.

Memory Considerations

Memory is allocated on a node for each remote MPI process that sends messages to it over RSM. If *np_local* is the number of processes on a particular node, then the memory requirement on the node for RSM message passing from any one remote process is

np_local x (`MPI_RSM_NUMPOSTBOX` x 128 + `MPI_RSM_CPOOLSIZE`)

bytes when `MPI_RSM_SBPOOLSIZE` is unset, and

np_local x `MPI_RSM_NUMPOSTBOX` x 128 + `MPI_RSM_SBPOOLSIZE`

bytes when `MPI_RSM_SBPOOLSIZE` is set.

The amount of memory actually allocated mightbe higher or lower than this requirement.

- The memory requirement is rounded up to some multiple of 8192 bytes with a minimum of 32768 bytes.
- This memory is allocated from a 256-Kbyte (262,144-byte) segment.
  - If the memory requirement is greater than 256 Kbytes, then insufficient memory will be allocated.
  - If the memory requirement is less than 256 Kbytes, some allocated memory will go unused. (There is some, but only limited, sharing of segments.)

If less memory is allocated than is required, then requested values of `MPI_RSM_CPOOLSIZE` or `MPI_RSM_SBPOOLSIZE` (specified with a `setenv` command and echoed if `MPI_PRINTENV` is set) can be reduced at run time. This can cause the requested value of `MPI_RSM_PIPESIZE` to be overridden as well.

Each remote MPI process requires its own allocation on the node as described above.

If multiway stripes are employed, the memory requirement increases correspondingly.

Performance Considerations

The pipe size should be at most half as big as the connection pool:

2 x MPI_RSM_PIPESIZE ≤ MPI_RSM_CPOOLSIZE

Otherwise, pipelined transfers will proceed slowly. The library adjusts MPI_RSM_PIPESIZE appropriately.

For pipelined messages, a sender must synchronize with its receiver to ensure that remote writes to buffers have completed before postboxes are written. Long pipelined messages can absorb this synchronization cost, but performance for short pipelined messages will suffer. In some cases, increasing the value of MPI_RSM_SHORTMSGSIZE can mitigate this effect.

Restriction

If the short message size is increased, there must be enough postboxes to accommodate the largest size. The first postbox can hold 23 bytes of payload, while subsequent postboxes in a short messages can each take 63 bytes of payload. Thus, 23 + (MPI_RSM_NUMPOSTBOX – 1) x 63 ≤ MPI_RSM_SHORTMSGSIZE.

Summary Table

TABLE B-1 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	—	24576
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

TABLE B-1 MPI Environment Variables *(Continued)*

name	units	range	default
MPI_TCP_CONNLOOP	occurrences	>=0	0
MPI_TCP_SAFEGATHER	(none)	0 or 1	1
RSM			
MPI_RSM_NUMPOSTBOX	postboxes	1 – 15	128
MPI_RSM_SHORTMSGSIZE	bytes	23 – 905	3918 bytes
MPI_RSM_PIPESIZE	bytes	multiple of 1024 up to 15360	64 Kb
MPI_RSM_CPOOLSIZE	bytes	multiple of 1024	256 Kb
MPI_RSM_SBPOOLSIZE	bytes	multiple of 1024	(unset)
MPI_RSM_MAXSTRIPE	bytes	≥1	• *rsm_maxstripe*, if set by system administrator in `hpc.conf` file • otherwise 2
MPI_RSM_DISABLED	(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
MPI_SHM_RENDVSIZE	bytes	>=1	24576
MPI_TCP_RENDVSIZE	bytes	>=1	49152
MPI_RSM_RENDVSIZE	bytes	>=1	256 Kb

TABLE B-1 MPI Environment Variables *(Continued)*

name	units	range	default
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

T

U

Y