SiCortex® System Programming Guide

For Software Version 3.0 Field Test

Trademarks

Cray the a registered trademark of Cray, Inc.

Intel is the registered trademark of Intel Corporation.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. The registered trademark Linux is used pursuant to a sublicense from the Linux Mark Institute, the exclusive licensee of Linus Torvalds, owner of the mark in the U.S. and other countries.

Lustre is the registered trademark of Cluster File Systems, Inc.

MIPS and MIPS64 are registered trademarks of MIPS Technologies, Inc.

NIST is a registered trademark of the National Institute of Standards and Technology, U.S. Department of Commerce.

OpenMP is a trademark of Silicon Graphics, Inc.

PCI, PCI Express, and PCIe are registered trademarks, and EXPRESSMODULE is a trademark of PCI-SIG.

Perl is the registered trademark of The Perl Foundation.

SiCortex is a registered trademark, and the SiCortex logo, SC5832, SC648, and PathScale are trademarks of SiCortex, Incorporated.

TAU Performance System is a trademark of the joint developers: University of Oregon Performance Research Lab; Los Alamos National Laboratory Advanced Computing Laboratory; and The Research Centre Jülich, ZAM, Germany.

TotalView is a registered trademark of TotalView Technologies, LLC.

Vampir is a registered trademark of Wolfgang E. Nagel.

All other brand and product names are trademarks or service marks of their respective owners.

Copyrights

Disclaimer

The content of this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by SiCortex, Inc.

Document Number 2906-03 Rev. 01
Published August 26, 2008

Contacting SiCortex and Getting Support

SiCortex is on-line at http://www.sicortex.com. Our Web pages provide information on the company and products, including access to technical information and documentation, product overviews, and product announcements.

You can search the SiCortex Knowledge Base or participate in forum discussions online at http://www.sicortex.com/support after you register.

You can reach SiCortex Technical Support by e-mailing questions to support@sicortex.com or by calling:

- 978.897.0214 main number
- +1 877 SICORTE x289 (+1 877.742.6783 x289) toll free number

What's this Book About and Who's it for?

This manual targets C, C++, and Fortran application developers, who have experience coding programs that run on Linux systems. With few exceptions, the Linux environment on SiCortex systems mirrors that on any other Linux system. This manual describes the exceptions and how to work with them. Perl and Python programmers will notice no difference in the SiCortex Linux environment.

Conventions of Notation

Bold Denotes a selection to make in a GUI program.

For example, **File>Process>Startup** directs the user to select **File** located on the application's toolbar, then **Process**, and then **Startup**.

`monospaced font` Denotes code examples wherever they occur and command sequences and their arguments, which are entered at the system prompt.

Italics Denotes a term or a cross reference in general text.

⚠ Denotes a caution or warning, such as a dependency that must be satisfied before continuing a process.

☼ Denotes a tip, hint, or reminder.

SiCortex

Table of Contents

Chapter 1 Introducing the SiCortex System

In this section:

Built to support the dominant High Performance Technical Computing (HPTC) software model, the SiCortex System with its MPI/Linux software suite empowers users to quickly develop applications that can tackle the most complex and computationally intensive problems that face the scientific, engineering, and financial communities today.

Overview of the SiCortex System Architecture

A SiCortex System (hereafter, in this document, called *System*) consists of a number of six-way, symmetric multiprocessing (SMP) compute nodes connected by an Interconnect Fabric. An SC5832 has 972 nodes, and an SC648 has 108.

Node Components

Each node consists of one SiCortex node chip (Figure 1 on page 8) and two industry-standard DDR2 memory modules. A node chip contains six 64-bit processors, their L1 and L2 caches, two memory controllers (one for each memory module), the Interconnect Fabric interface components (the Fabric Links, the Fabric Switch, and the DMA engine), and a PCI Express® (PCIe®) interface. The PCIe controller provides control for external I/O devices only, not for the Interconnect Fabric.

For architectural details and programming considerations related to the node components, see *Chapter 9, Processor and Memory System Functional Features on* page 131 .

On the node chip, the DMA engine, Fabric Switch, and Fabric Links provide the interface to the Interconnect Fabric. The DMA engine connects the memory system to the Fabric Switch, which forwards traffic between incoming and outgoing links, and to and from the DMA engine.

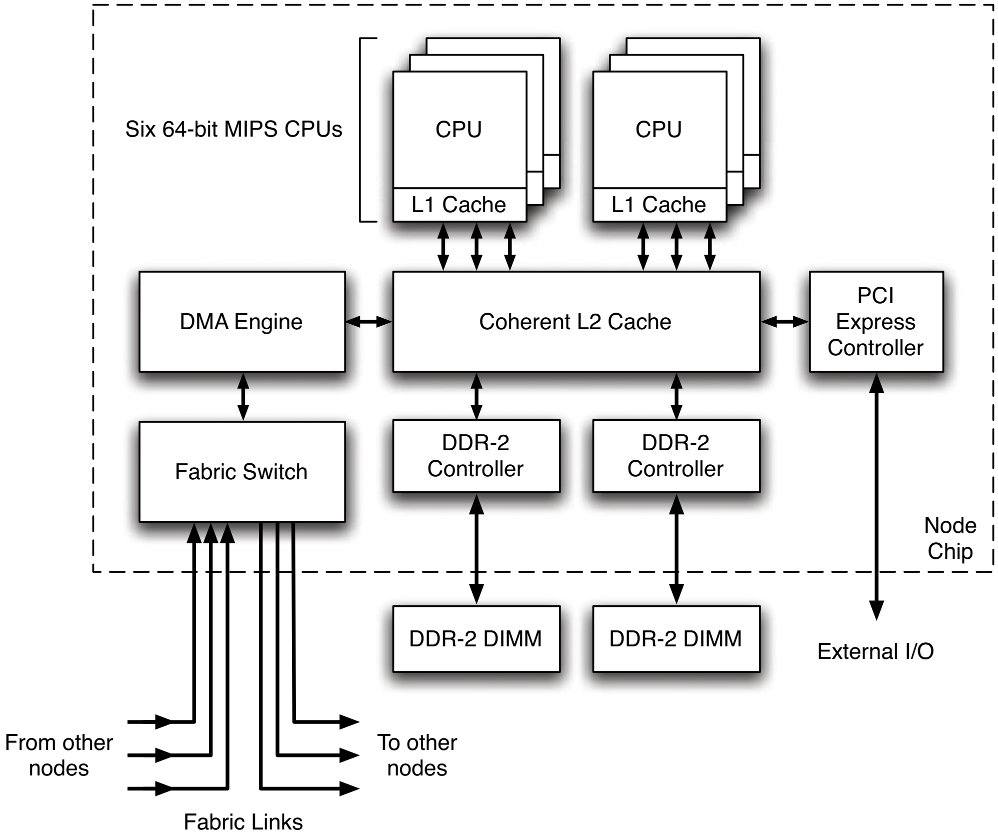

Figure 1. Overview of SiCortex node

All nodes in a System are connected through a degree-3 directed Kautz network. Twenty-seven nodes populate a module, and all modules plug into the System's backplane. Of the twenty-seven nodes on a module, three have their PCIe busses connected to EXPRESSMODULE™ slots, and a fourth is attached to an on-module PCIe dual gigabit-Ethernet controller. The PCIe interface on all other nodes is disabled.

The Interconnect Fabric

The nodes communicate with one another through DMA over the Interconnect Fabric, a fast network used for internode IP networking and direct user-mode communications. The network, based on a degree 3 Kautz graph (Figure 2 on page 9), enables internode messages to arrive at their destination within a maximum number of hops. For 108 node Systems, the maximum is four, and for 972 node Systems, the maximum is six.

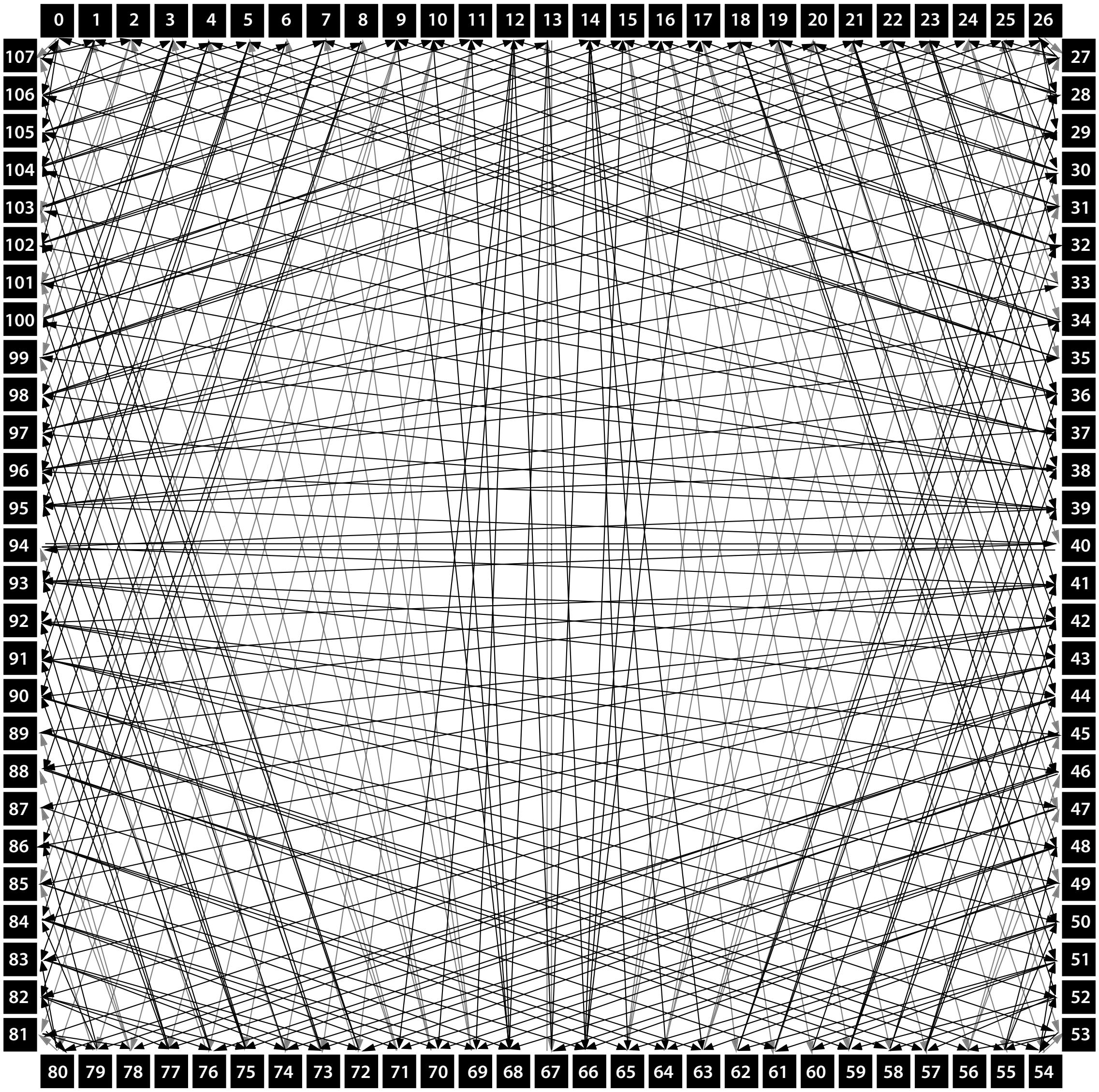

Figure 2. 3-degree directed Kautz network for an SC648 System

Each node transmits to three other nodes and receives from a different three nodes. Not only does this design reduce message latency and network congestion, it also ensures that the failure of one node increases the hop count of a message by no more than one, and that all other nodes remain reachable.(For detailed information on the Kautz graph, see the white paper *A New Generation of Cluster Interconnect* posted on the SiCortex web site at http://www.sicortex.com.)

System I/O

The SC5832 System has 108 PCIe I/O ports and seventy-two gigabit Ethernet I/O ports. The SC648 System has 12 PCIe I/O ports and eight gigabit Ethernet I/O ports. These ports provide direct I/O and Network-Attached Storage (NAS) connection to both NFS and Lustre external file systems. Internally, Network Block Device or NFS (depending on the System model) provides the root file system on each node. The interface between the internal and external file systems is transparent, so users need not know low-level details to perform I/O operations. Figure 3 shows some possible connection schemes for peripherals.

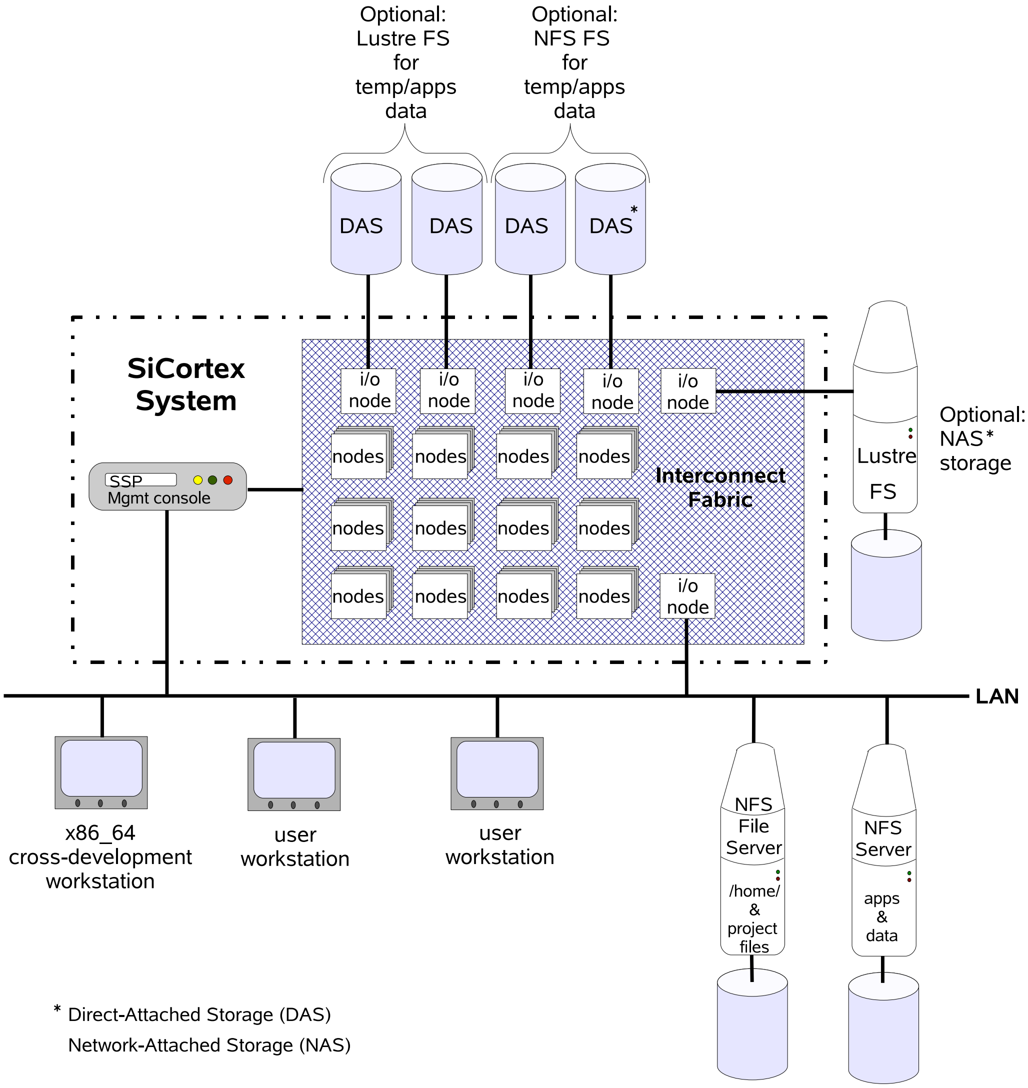

Figure 3. Typical I/O connections for a SiCortex System

With FabriCache™ enabled, a reserved portion of main memory functions as a parallel file system. Managed by Lustre logic, an integral part of each node's kernel, FabriCache provides memory-speed I/O for I/O intensive applications. The FabriCache is configurable, so all of main memory on a subset of nodes can be configured to function as a parallel file system, accessible by all processors in the System. Lustre logic ensures data coherency and integrity, and the Interconnect Fabric ensures reliability of data transmission. Like disk file systems, the mechanics of accessing files in the FabriCache is transparent to users and requires no special programming. Unlike disk file systems, FabriCache is not persistent, so files are not preserved when the System is powered off or rebooted.

☼ Ask your System Administrator whether Fabricache is enabled. For details on accessing and using it, see *Using a FabriCache File System* on page 22.

For more information on the FabriCache, see *The SiCortex FabriCache™: Measure Its Abilities in Genomes/sec* newsletter. For details on the SiCortex implementation of the Lustre files system, see *The Luster High Performance File System* white paper. Both documents are available on the SiCortex web site at http://www.sicortex.com.

The Application Development Environment

SiCortex provides users a rich set of tools for quickly developing and running the complex, computationally intense applications that are required to solve today's problems in science, engineering and finance. These tools include compilers, libraries, debuggers, and performance tools that are optimized for SiCortex systems.

Software Development Suites

To enable users to develop applications on the System or on any x86_64 Linux workstation, SiCortex provides two software development suites: the native software suite and the cross-development software suite.

In this guide, the term *native* refers to the nodes' MIPS64 processors, so the native software suite runs on and produces object code that runs on the System's nodes.

Also in this guide, the term *cross-development* refers to any x86_64 Linux workstation that has the SiCortex cross-development toolkit installed on it. Though it runs on an x86_64 Linux system, the cross-development software suite produces object code that runs on the System's MIPS64 processors.

Both development software suites are provisioned with the same standard GNU tools and utilities[*], the same software performance tools, and the same libraries (MPI, data formatting, math, and science). Both software suites include precompiled library binaries for both n32 and n64 ABIs.

The terms n32 and n64 refer to the two Application Binary Interfaces (ABI) that the System supports. These ABIs enable applications to run in either n32 or n64 mode. With its 64-bit pointer representation, n64 mode provides virtual memory sizes for processes that are larger than 2 GB. With its smaller pointers, n32 mode provides more efficient use of cache and memory. Both ABI modes have equal access to all processor features, including the 64-bit data registers, so that in n32 mode, when code declares 64-bit integers, the processor performs 64-bit integer operations. (See *Chapter 10, Understanding the Application Binary Interfaces* for more details.)

Compiler Suites

SiCortex software provides the same suite of compilers for both development environments:

- PathScale compiler suite for Fortran 77/90/95, C, and C++.

  ☼ Only the PathScale compilers support OpenMP.

- GNU compilers for C and C++ (gcc v.4.1).
- Both compiler suites can produce either n32 or n64 ABI object code.
- Binaries produced by the GNU and PathScale compilers are interoperable (as long as they are generated using the same ABI), so users can link together applications and libraries that are compiled separately, using any of the supplied compilers.

GNU Tools and Utilities

The standard GNU compilation tools and utilities included are:

Native GNU	**Cross-Development GNU**[*]
gcc/g++	scgcc/scg++
ld, ldd	scld
as	
addr2line	scaddr2line
ar	scar
gprof	
gdb	scgdb
nlmconv	
nm	scnm

[*] Except ldd (list dynamic dependencies), which is not included in the cross-development software suite.

Native GNU	**Cross-Development GNU***
objcopy	scobjcopy
objdump	scobjdump
ranlib	scranlib
readelf	screadelf
size	
strings	
strip	scstrip

* On the cross-development system, you use the *sc* prefix to call the compilers and utilities.

The `scman` command is provided in the cross-development toolkit. On the cross-development workstation, it enables you to access the man pages for the mips64 version of the tools and utilities that are installed on the System's nodes, without logging onto a node.

When you use the `man` command on the cross-development workstation, you access the man pages for the x86_64 tools and utilities that are installed on the workstation.

When you use the `man` or `scman` command on the nodes, you access the man pages for the mips64 tools and utilities that are installed on the nodes.

Libraries

For a full description of the math and science libraries, see *Chapter 6, Using the Optimized Math and Science Libraries*.

All libraries are supplied in static and dynamic versions for both the n32 and n64 ABIs.

Both development software suites include the same libraries.

Math & Science Libraries:

AtlasBLAS
GotoBLAS
FFTW
LAPACK
libscm
libscs
libscstr
PETSc
ScaLAPACK
SPRNG

Data Formatting & Communications Libraries:

HDF5 (Hierachical Data Format)
NetCDF (Network Common Data Format)
libscmpi (SiCortex MPI library)

Debugging Tools

Both development software suites include the GNU debugger, `gdb`. To use `gdb` to debug from the cross-development environment, you need to use `gdbserver` or other remote debugging utility.

For debugging details, see *Chapter 7, Developing MPI Applications*.

Performance Tools

The SiCortex System provides performance-monitoring hardware and software tools that enable users to develop applications optimized for high performance.

- Performance-monitoring hardware

  The nodes include hardware performance counters that provide data to performance-analysis tools. On the System, *perfmon2* provides the standard software interface to the nodes' performance-monitoring hardware.

- Software Tools

  The development software suites include a wide range of performance tools:

Aggregate Performance	**Statistical Sampling**	**Trace Analysis & Visualization**
Papiex/PAPI	HPCToolkit hpcex hpcstruct hpcproftt hpcprof-flat hpcviewer	TAU/tauex
mpipex/mpiP		Vampir*
ioex		GPTL/gptlex
gprof		

* Optimized for the SiCortex System, but users must purchase from ParaTools

For details, see *Chapter 5, Optimizing Application Performance*.

Chapter 2 Running Applications

In this section:

Application executables typically reside either locally on a user's workstation or on an externally mounted, shared Lustre[*] or other file system that is connected to the System, either directly to System I/O nodes or remotely over a network connection (see Figure 3, on page 10).

☼ If your application's executable files and data are on your local workstation, copy them to a shared file system that is mounted and visible on the System.

[*] For a brief description, see page 8.

Logging on to the System

Before you can run an application on the System, you have to connect to a head node and get a shell.

☼ Each System ships with internal node names defined for it. Internal node names consist of the System id (`scx` for the SC5832, `sci` for the SC1458, `sc1` for the SC648, and `sca` for the SC072) with the module id (m#) and node id (n#) appended to it: *sc#-m#n#*. (The terms *node name* and *host name* are often used synonymously.) A module has twenty-seven nodes. On module 0, internal node names range from sc#-m0n0 to sc#-m0n26[*]; on module 1, they range from sc#-m1n0 to sc#-m1n26, and so on. The SC5832 has thirty-six modules, the SC1458 has nine, the SC648 has four, and the SC072 has one.

Most likely, your System Administrator has configured a head node[†] to function as the point-of-entry to different partitions, and assigned it a unique site name. Using the assigned site name, log on to the head node.

In all cases, when you log on to the head node, the shell prompt displays the node's internal name; for example, `sc1-m0n6`. Hereafter in this chapter, `sc1-m0n6` identifies the head node in all code examples.

Connecting to a Head Node

From your workstation `ssh` to a head node, for example:

```
gs113:~$ ssh <my_system_headnode>
Last login: Tue May 1 10:12:55 2007 from gs113.companyb
sc1-m0n6:~$
```

Specifying a Partition

To run an application you must specify the partition to run it on, using `srun`'s `-p partition` option.

SiCortex systems ship with some preconfigured sample partitions:

- *System id* (`scx` or `sci` or `sc1` or `sca`)
  Includes all nodes on the System.
- `scx-comp` Includes all nodes on the SC5832, except the built-in dual GigE nodes m[0,2,4,6,32,34]n6.

[*] The SC072 has only one module with twelve nodes, so node names range from sca-m0n0 to sca-m0n11.

[†] The SC072 has a preconfigured head node, `head` (internal name: `sca-m0n8`).

- `sci-comp` Includes all nodes on the SC1458, except two built-in dual GigE nodes, m[0,1]n6, and m8n6.
- `sc1-comp` Includes all nodes on the SC648, except m0n6, a built-in dual GigE node, and m3n6.
- `[sc1]-comp1` Includes all nodes on the SC648, except the I/O nodes m*n1 and node m0n6.
- `[sc1]-comp3` Includes all nodes on the SC648, except the I/O nodes m*n3 and node m0n6.
- sc2-comp Includes all nodes on the SC072, but avoids using m0n8 (head) unless a job requires it.

☼ These sample partitions are intended to serve as examples only. Ask the System Administrator which partition to use since he or she may have configured other partitions for specific user groups or applications.

Hereafter in this chapter, `-p sc1-comp1` is used in all example `srun` commands that demonstrate how to run multinode applications.

Running and Managing Multinode Applications

SLURM (Simple Linux Utility for Resource Management) implements resource management and job scheduling on the System for applications, such as MPI programs, that typically run multiple processes (in SLURM terminology, *tasks*) on multiple nodes. Before you can run a job, you need the appropriate permissions and sufficient resources available to run your application. To start and manage multinode jobs, you use SLURM commands.

For more information, see the SLURM man pages: `slurm(1)`, `srun(1)`, `salloc(1)`, `scancel(1)`, `sinfo(1)`, `squeue(1)`, and `scontrol(1)`.

⚠ Before you can run your job, your application directories must be on a mounted, shared Lustre or other network file system that is visible on the node to which you are connected.

Starting a Multinode Job

To start a multinode job, use `srun`.

```
$ srun -p <partition> [args] <appname|jobscript> [args]
```

The srun command submits the job to the local SLURM job controller, initiates all processes on an appropriate set of nodes, and, if necessary, blocks until the needed resources are free to run the job.

The srun command runs a program just like a shell does, but unlike a shell, it can start multiple tasks on multiple nodes. Each of the tasks is a separate process that executes the same program. By default, SLURM allocates one processor per task, but starts tasks on multiple processors as necessary. The argument `-n` specifies the number of tasks, and the argument `-N` specifies the number of nodes.

```
$ srun -p sc1-comp1 -n 2 myprogram
        # runs 2 tasks, each on a different processor

$ srun -p sc1-comp1 -n 7 -N 4 myprogram
        # runs 7 tasks distributed across 4 nodes

$ srun -p sc1-comp1 -N 9 myprogram
        # runs 9 tasks on 9 different nodes

$ srun -p sc1-comp1 -n 3 -c 2 myprogram
        # starts 3 tasks, and allocates 2 processors per
        # task

$ srun -p development -N 6 myprogram
        # runs 6 tasks on six nodes in the partition
        # named development
```

If you specify more tasks than the number of requested nodes can handle, SLURM automatically allocates additional nodes and distributes the tasks across them. However, if you specify more nodes than tasks, SLURM issues a warning, reallocates resources, then proceeds to process the job:

```
sc1-m0n6:~$ srun -p sc1-comp1 -n 2 -N 4 hostname
srun: Warning: can’t run 2 processes on 4 nodes, setting
nnodes to 2
sc1-m0n0
sc1-m0n2
sc1-m0n0:~$
```

By default, SLURM broadcasts `stdin` from the attached terminal to all of the processes and returns each process’ `stdout` and `stderr` to the terminal.

⚠ However, SLURM buffers `stdout`. This behavior can cause unexpected results. For example, if a job crashes before completing, there is no indication of it because SLURM continues to hold off output while it waits for the job to finish. In this scenario, you would cancel the job using `scancel` (see *Canceling a Job: scancel and ^C on* page 20).

☼ You can control the buffering of `stdout`. For details, see *Appendix A, SLURM I/O Buffering on* page 147.

Batch Jobs

You run a batch job by submitting a script program to SLURM to run. The script contains all of the commands and arguments to run the job, typically other programs, such as MPI applications or simple `srun` commands. Upon submitting a batch job for execution, `srun` exits immediately, and the job runs when SLURM determines that adequate resources are available.

To submit scripts for SLURM to run when needed resources become available on the System and no higher priority jobs are pending, use the `sbatch command`:

```
$ sbatch -p sc1-comp1 /home/work/myscript.sh
```

SLURM runs the script on the first node allocated to the job, with `STDIN` redirected from `/dev/null` and `STDOUT` and `STDERR` redirected to the file `*jobname*.out` in the current working directory, unless you specify another file name.

Allocating Resources

SLURM schedules jobs subject to resource availability. You can use the `salloc` command to acquire and hold resources for your use:

```
sc1-m0n6:~$ salloc -p sc1-comp1 -N 4
sc1-m0n6:~$
```

This option blocks until the requested resources are available, then spawns a subshell. From this subshell, you can run interactively on the allocated resources multiple parallel jobs or a job script. Once space on a partition is allocated, you do not have to specify the `-p <partition>` on subsequent invocations of `srun`:

```
sc1-m0n6:~$ srun -N 4 hostname
sc1-m0n0
sc1-m0n4
sc1-m0n3
sc1-m0n2
```

Because the subshell has already acquired the requested resources, jobs started within the subshell run immediately.

⚠ After you are done, you must `exit` the subshell to release the resources.

Managing Multinode Jobs

Once a job is running, you can use SLURM commands to track its progress and to stop/restart it. To do so, you need to know its job id.

Monitoring a Running Job: squeue and scontrol

The squeue command displays the job id and job name, with the status and resource information for every job currently managed by the SLURM control daemon. With no options specified, the report displays this information: job id, partition, job name, user name, job status, time used thus far (hours:minutes:seconds), total nodes, and node list.

```
sc1-m0n6:~$ squeue
JOBID  PARTITION  NAME   USER  ST  TIME  NODES  NODELIST
21                myprog  bhill  R  0:25      1  sc1-m0n0
```

The scontrol command provides more detailed information about individual jobs, even if the job in question has already finished.

```
sc1-m0n6:~$ scontrol show job 9
  JobId=9 UserId=bhill (1198) GroupId=users (110)
  Name=myprog
  Priorit=42948796 Partition=test BatchFlag=0
  AllocNode:Sid=sc1-m0n0:8 TimeLimit=UNLIMITED Exit
   Code=0:0
  JobState=COMPLETED StartTime=01/05-16:05:32 EndTime=
  NodeList=sc1-mon[0,2-4] NodeListIndices=
  AllocCPUs=6*4
  ReqProcs=24 ReqNodes=4 ReqS:C:T=0
  Shared=0Contiguous=0 CPUS/task=0
  MinProcs=0 MinSockets=0 MinCores=0 MinThreads=0
  MinMemory=0 MinTmpDisk=0 Features=(null)
  Depenency=0 Account=(null) Reason=None Network=(null)
  ReqNodeList=(null) ReqNodeListIndices=
  ExcNodeList=(null) ExcNodeListIndices=
  SubmitTime=05/01-16:05:35 SuspendTime=None PreSusTime=0
```

Canceling a Job: scancel and ^C

The scancel command cancels a running or pending job using the job's id (only job owners and administrators can cancel jobs).

```
sc1-m0n6:~$ srun -p sc1-comp1 -b /home/work/myscript.sh
srun: jobid 21 submitted

sc1-m0n6:~$ scancel 21
sc1-m0n6:~$ squeue
JOBID  PARTITION  NAME  USER  ST  TIME  NODES  NODELIST
$
```

Alternatively, you can issue `^C` (`SIGINIT`) signals to cancel a running job. After `srun` starts a job, it blocks until all of the job's tasks terminate. Signals sent to `srun` during this time are broadcast to all of the tasks. SLURM handles `^C` signals a special way:

- One `^C` signal generates a status report for all of the associated tasks:

```
sc1-m0n6:~$ srun -p sc1-comp1 -N 2 sleep 10
srun: interrupt (one more within 1 sec to abort)
srun: task[0-1]: running
```

- Two `^C` signals within one second typically terminates all of the associated tasks:

```
sc1-m0n6:~$ srun -p sc1-comp1 -N 2 sleep 10
srun: interrupt (one more within 1 sec to abort)
srun: task[0-1]: initializing
srun: sending Ctrl-C to job
srun: canceling job
```

- Three `^C` signals within one second immediately terminates the job and its remote tasks.

⚠ Do not `kill/skill` `srun` to cancel a SLURM job! Doing so only terminates `srun`. The tasks continue to run, but not under SLURM management. If you mistakenly `kill/skill` an `srun` job, you can use `squeue` to get the job id and then either:

- `scancel` the job, or
- `sattach -p sc1-comp1 <jobid> -j`, to reattach `srun` to the job, and then use the `^C` sequence to cancel it.

☼ If you cannot clear your job using any of these methods, report it to your System Administrator. He or she can.clear it using `scontrol`.

Monitoring Node or Partition Status: sinfo

The sinfo command reports the current status information on partitions and individual nodes. With no options specified, the report displays, for all nodes and partitions on the System, this information: partition, availability, time limit, node count, node state, node list. For example:

```
sc1-m0n6:~$ sinfo
PARTITION   AVAIL   TIMELIMIT   NODES   STATE   NODELIST
test           up   infinite        4    idle   sc1-m0n[0-3]
```

Running and Managing Single-Node Applications

We encourage users to run all applications under SLURM control using the `srun` command. For single-node applications, include `-N 1` on the command line to instruct SLURM to run the application on only one node. For a single process, include `-n 1` on the command line.

However, if you elect to bypass SLURM and instead launch the program executable from the shell command line, you can eliminate conflicts over resource allocations by making the job visible to SLURM, the utility that manages resources for multinode jobs (see *Running and Managing Multinode Applications on* page 17). Otherwise, your applications may interfere with SLURM jobs that run on the same nodes as your applications.

To make a non-SLURM job visible to SLURM:

- Use the SLURM `salloc` command to allocate resources to a shell from which you will run your single-node application, for example:

  ```
  sc1-m0n6$ salloc -p sc1-comp1 -w sc1-m0n2 bash
  salloc: granted job allocation 56980
  sc1-m0n6$
  ```

- At this point the node, specified by `-w sc1-m0n2`, is associated with your bash shell. You can run applications on the node from the shell as long as the shell persists; the allocation ends only when you terminate the shell. Because SLURM knows that this node is allocated to your shell, it will not attempt to run other jobs on the node until you release it.

  For details, see the SLURM man page `salloc(1)`.

To monitor and control running applications and processes launched bypassing SLURM, use any of the common Linux shell commands and utilities: `ps`, `kill`, `nice`, `renice`, `jobs`, `bg`, `fg`, and so on.

Running n32 Applications

You run n32 applications just like you run n64 applications:

```
$ srun -p <partition> [args] <myapp> [args]
```

You do have to build them differently. For details, see page 33 and page 35.

Using a FabriCache File System

For a brief description of the FabriCache feature, see page 10. For a full description, see *The SiCortex FabriCache®: Measure its Abilities in Genomes/sec.* newsletter at www.sicortex.com/5832_newsletter.

Because FabriCache is RAM-based, it is non persistent, but accessing data stored in it is very fast. This makes FabriCache ideal for storing intermediate data while an application is using it.

Your System Administrator may have set up a FabriCache file system on the nodes in a special SLURM partition named `<partition>_clients`. This partition includes only those nodes that are configured to use the FabriCache file system. Applications can access the FabriCache file system on all nodes in `<partition>_clients` at `/tmp/fabclient/<partition>`.

☼ Use the sinfo command without any arguments to list all of the partitions available on the System and the nodes included in each. Look for a `<partition>_clients` entry and note the nodes included in it.

Getting data in and out of FabriCache

You can use the `scp` command or the `rsync` command to copy your data in and out of the FabriCache file system.

Copy your data (and application executable) to any one of the nodes included in the `<partition>_clients` partition. Because FabriCache is a shared file system, all nodes included in `<partition>_clients` can access the data.

Your application reads and writes data to the FabriCache file system as it would any other shared file system.

Running a FabriCache job

To submit a job that uses the FabriCache file system:

1. Make the `/tmp/fabclient/<partition>` your working SLURM directory. On the head node:

```
sc1-m0n6:~$ export \
             SLURM_WORKING_DIR=/tmp/fabclient/<partition>
```

2. Submit your job to SLURM the usual way, using the FabriCache partition. For example:

```
sc1-m0n6:~$ srun -p <partition>_clients -N 20 \
                              ./simulation/my_sim
```

☼ Make sure you don't request more resources than `<partition>_clients` has.

Controlling FabriCache jobs

You monitor and control FabriCache jobs just like you do any SLURM job. For details, see *Running and Managing Multinode Applications on* page 17.

Troubleshooting SLURM Jobs

If you encounter problems while running a job that you submitted through SLURM, you can usually determine the cause then work around it to successfully run the job.

Node or link failures

Typically, when a node or link fails, the affected node seems to disappear such that you cannot ssh to it. Both the squeue and the sinfo commands report a node is not responding by appending an asterisk to the reported state of the node:

```
sc1-m0n6:~$ sinfo -p test
PARTITION   AVAIL   TIMELIMIT  NODES  STATE   NODELIST
test          up     infinite      1    idle   sc1-m0n0
test          up     infinite      1    alloc  sc1-m0n2
test          up     infinite      2    down*  sc1-m0n[1,3]
```

When this occurs, scancel the job, then rerun it using the same job parameters you used previously. By default, SLURM does not allocate downed nodes.

Alternatively, you can rerun your job using srun's -x <node_list> (for example, -x sc1-m0n[1,3]) option to explicitly exclude the downed nodes.

Be sure to inform your System Administrator when you encounter downed nodes.

Disabled nodes and links

For various reasons, your System Administrator may have designated particular nodes or links as disabled in a system configuration file, which the boot software uses to configure routing throughout the System. Both squeue and sinfo will report disabled nodes as STATE/Drain.

When jobs request nodes that are in the Drain state, SLURM will queue those requests and wait for resources to become available. Such jobs queued to run on disabled nodes will remain queued until the nodes are rebooted or otherwise returned to service. This may take some time.

If your job gets stuck in the job queue, scancel it. Rerun your job avoiding nodes that are in the Drain state.

Chapter 3 Compiling and Linking Applications

In this section:

SiCortex software development suites support two development environments—native and cross-development—and provide two compiler suites—GNU and PathScale—that work in both development environments.

Installing the Cross-Development Toolkit

The Cross-Development Toolkit is included on the SiCortex software installation DVD, and the instructions for installing the toolkit are included in *The SiCortex® System Administration Guide*. See your System Administrator if you want to install the cross-development toolkit on your workstation.

Choosing a Compiler

Of the two compiler suites, SiCortex maintains the GNU compilers as robust and compatible tools for the System, but is continuously improving the PathScale compilers, so they generate the most efficient code for applications in which processor performance is critical.

Both compiler suites support C and C++, but only the PathScale compiler suite supports Fortran. (Neither the `gfortran` nor the `g77` compiler is supported.) And only the PathScale compiler suite supports OpenMP.

Otherwise, you can use any of the compilers, and because all of them generate object files that are interoperable (if all use the same ABI), you can compile your application using different compilers for different program components.

If you are compiling a large application composed of both Fortran and C or C++ code, and the main entry point into the application is C or C++ code, you can use the GNU or PathScale C/C++ compilers to compile and link the application. If you do so, then you must explicitly add the Fortran runtime libraries to the linker command line. For example, your command line might look like this:

```
$ gcc -o my_big_app file1.o file2.o -lpathfortran
```

If you use the PathScale C/C++ compiler and your code calls `libm` and `libscm` functions (for a description, see *Libscm Tuned Math Library on* page 104), you must add `-lscm` and `-lm` to the link line, as the second pass of feedback compilation may require an explicit `-lscm -lm`.

With some exceptions, the PathScale C/C++ compiler supports the same C/C++ language extensions as the GNU compiler. The PathScale compiler does not support these C/C++ extensions:

- C extensions

  SSE3 intrinsics, many of the `__builtin_` functions, a `goto` outside of the block (i.e. locally declared labels, labels as values), nested functions, complex integers (`__Complex int`), structures generated on the fly, and thread-local storage (`__thread`)

- C++ extensions

  Java exceptions, `java_interface` and `init_priority` attributes

Using Compiler Options

The compilers default to the n64 ABI and generate n64 executables. You can change this default behavior and generate n32 executables (see *Overriding the Default ABI on* page 139). (For details on the supported ABIs, see *Chapter 10, Understanding the Application Binary Interfaces on* page 135.)

Table 1 shows the defaults for the compilers' debug options and how they affect compiler optimization. For details on the debug levels, see the compiler's man pages or www.pathscale.com.

Table 1. Debug defaults

-g [1/2/3]	GNU default	PathScale default
-g w/no specified level	-g2	-g2
-g# w/no specified -O	-O0	-O0
-g1	–	passes `--export-dynamic` to the linker

With debugging disabled, the GNU compilers default to `-O0` optimization, but the PathScale compilers default to `-O2`. *Build Optimizations on* page 53 describes the levels of optimization as defined by the GNU and PathScale compilers.

If you intend to use the performance tools, use the compiler's `-g` debug option.

PathScale Compiler Options

The `pathf95` compiler follows the name-mangling rules described in *Compiler Name Mangling* on page 140. Because the libraries supplied with the SiCortex software comply with these rules, we strongly recommend that you avoid using compiler flags that alter the compiler's default method of applying underscores to symbol names. If you encounter link problems that you cannot resolve any other way, try the compiler's `-fno-second-underscore` option.

The PathScale compiler defaults to `mips5kf` (but also accepts `5kf`) for the `-march`, `-mtune`, and `-mcpu` flags. If you use any of these flags, make sure you set them to `mips5kf` or `5kf`, otherwise the compiler will demand a license, then abort the compile.

For Fortran applications that pass array sections, compile using `pathf95`'s `-LANG:copyinout=OFF` option (the default is `ON`). Otherwise, the compiler generates code that expensively copies in and out of the routines that receive these arguments. In general, it's always a good idea to set this flag to `OFF`. For example, the POP model runs six times faster on a few hundred processors when this flag is set to `OFF`.

When linking against Fortran libraries, missing symbols are often a result of passing the `-fno-second-underscore` flag to the compiler. If this happens, recompile your application without the flag. For more information, see *Compiler Name Mangling* on page 140.

We suggest that you follow this general procedure for finding the optimal compile setup for your application:

- Progressively compile and compare results starting with `-O2` (default with debugging disabled), `-O3`, `-O3 -ipa`, `-O3 -OPT:Ofast`, and `-Ofast`.

  ☼ `Ofast` incorporates `-ffast-math`, which automatically links the tuned math library, libscm. For C/C++ applications that specify `-lm` on the link line, `-ffast-math` causes the compiler to also autolink `-lmpath` and `-lm`. To disable linking libscm when using `-Ofast`, add `-fno-fast-math` to the command line. For more details, see *Accessing the libscm Library* on page 106.

  The `pathopt2` tool automates this process and, by default, uses this sequence. It compiles and runs applications as specified in a target file, and generates a sorted list of execution times for each run. You can create target files that direct `pathopt2` to test other compile options. You can also use `pathopt2` to troubleshoot problems that occur at specific optimization levels (for details, see the PathScale documentation).

  For `-O3 -OPT:Ofast` and `-Ofast`, check to see if the results are accurate. These options target maximizing performance, but because they rearrange computations, they can adversely affect floating-point accuracy. If you do encounter numerical problems, try `-O3 -Opt:Ofast:ro=1` or `-O3 -Opt:Ofast:div_split=OFF`.

- Try `-OPT:alias=disjoint` to avoid aliasing issues; `-LNO:prefetch=0` to avoid doing prefetches that don't work; `-OPT:unroll_times=0` to prevent loop unrolling; the `-ipa` (interprocedural analysis) option, which operates on the whole application and is most beneficial at `-O3` (`-ipa` is automatically enabled by `-Ofast`).

- By default, the PathScale C/C++ compilers use the `libm` `sqrt()` function rather than the MIPS `sqrt.d` instruction. This behavior can adversely affect the performance of some applications. You can use the `-fno-math-errno` compiler flag to force the PathScale compiler to generate sqrt.d, but doing so causes the compiler to ignore the `ERRNO` setting in math functions. (`-Ofast` enables `-fno-math-errno`.)

- Feedback Directed Compilation (FDO)

  FDO requires compiling a program twice—once to generate the program's profile information, which the compiler then uses to tune the program during the second compilation.

- Options for diagnosing coding errors that usually surface only at higher optimization levels:
  - `-OPT:alias=no_parm` instructs the pathf95 Fortran compiler to assume that program parameters do alias other parameters.
  - `-LANG:rw_const=ON` instructs the compiler to allocate constant values in read/write memory, enabling a callee to modify a constant argument.

    This option reduces the compiler's ability to propagate constant values, which may impact performance of the generated code.
  - The following options enable you to find and deal with uninitialized variables in program code. They affect local scalar and array variables and memory returned by `alloc()`; they do not affect globals, memory allocated by `malloc()`, or Fortran common data.

    `-trapuv` Initializes variables to floating-point NaNs and enables the processor to detect floating-point calculations (not assignments) involving NaNs. Instead of generating incorrect results, uninitialized variables in the program cause the program to abort.

    `-zerouv` At program run time, initializes variables to `0`. This option covers for applications that assume incorrectly that memory is always initialized to `0`, and impacts performance only slightly.

For details on compiler options and usage, see the appropriate compiler man pages and user documentation.

GNU Compiler Options

We suggest that you follow this general procedure for finding the optimal gcc/scgcc setup for your application:

- Progressively compile and compare results starting with the default flags:

  ```
  -O2 -mips64 -march=5kf -mtune=5kf
  ```

  In most cases, the default flags improve performance, but you can always experiment with additional flags to coax even better results.
- Next try adding one or more of these flags:

  ```
  -fno-schedule-insns -fno-schedule-insns2 -fno-peephole
  -fno-peephole2
  ```

(By default, `-fschedule-insns`, `-fschedule-insns2`, `-fpeephole`, and `-fpeephole2` are enabled at `-O2`, `-O3`, and `-Os`.)

The `-fno-schedule-insns` flag is particularly useful because it prevents the compiler from moving all loads to the start of the code block in an attempt to have the data available when it is needed later in the code. Without this check, the compiler's behavior can cause stalls because of the System's *hits under misses* policy (for details, see page 133).

The `-fno-schedule-*` and `-fno-peephole*` flags may best be suited for the low-level, assembly-like code. But because instruction scheduling by the compiler can also decrease performance of low-level code, before you use these options for other purposes, read their man pages.

- Consider using the `-ffast-math` optimization flag, but do not invoke it with the `-O` option because doing so can result in incorrect output for applications that require strict adherence to IEEE or ISO specifications for math functions. `-ffast-math` sets many other math optimization flags and causes the preprocessor macro `__FAST_MATH__` to be defined.

  Also consider trying the `-mno-check-zero-division` flag to prevent trapping on integer division by zero.

  ⚠ The `-ffast-math` and `-mno-check-zero-division` flags may not be suitable for all programs because each makes assumptions about the program code.

- If you intend to profile your application, use the `-g` option.

Summary of Simple Build Methods

For a single-file application, Table 2 on page 31 shows what to type on the command line to build it natively on the System, and Table 3 shows what to type on the command line to build it on the cross-development workstation.

Table 2. Command line syntax for building single-file applications natively

Environment	Single File	Makefile	Autotools/Configure
Native compile (on the System)	pathcc file.c	make	./configure make
	pathCC file.cpp	make	./configure make
	pathf95 file.f|f90|F|F90	make	./configure make
	[*]mpif77 file.f|F mpif90 file.f90|F90 mpicc file.c mpicxx file.cpp	make	./configure make
	gcc file.c	make	./configure make

[*] The mpi** wrapper scripts automatically link using the appropriate PathScale compiler with the optimized MPI library. To use the debug version of the MPI library, add `--mpilib=-lscmpi_debug` to the link line. When not using the wrapper scripts for MPI applications, you must add `-lscmpi` (or `-lscmpi_debug`) to the link line.

The sc* wrapper scripts call the corresponding underlying compiler and linker. On the cross-development workstation, the wrapper scripts tell the compiler and linker where to find the correct directories for the cross-compile headers and libraries. Using the sc* wrapper scripts natively on the nodes works the same as calling the corresponding tool directly, except that the overhead of the extra step slows the process.

Table 3. Command line syntax for building single-file applications on the cross-development

Environment	Single File	Makefile	Autotools/Configure

Cross-compile (on the cross development workstation)	scpathcc file.c	CC="scpathcc" make	CC="scpathcc" ./configure --host=mips64el-gentoo-linux-gnu
			--build=x86_64-pc-linux-gnu
			CC="scpathcc" make

	scpathCC file.cpp	CXX="scpathCC" make	CXX="scpathCC" ./configure --host=mips64el-gentoo-linux-gnu
			--build=x86_64-pc-linux-gnu
			CXX="scpathCC" make

	scpathf95 file.f|f90|F|F90	F77="scpathf95" make
		F90="scpathf95" make
		FC="scpathf95" make
			F77="scpathf95" ./configure --host=mips64el-gentoo-linux-gnu
			--build=x86_64-pc-linux-gnu
			F90="scpathf95" ./configure --host=mips64el-gentoo-linux-gnu
			--build=x86_64-pc-linux-gnu
			FC="scpathf95" ./configure --host=mips64el-gentoo-linux-gnu
			--build=x86_64-pc-linux-gnu
			F77|F90|FC="scpathf95" make

Table 3. Command line syntax for building single-file applications on the cross-development (Cont’d)

Environment	Single File	Makefile	Autotools/Configure

	scmpif77 file.f|F
	scmpif90 file.f90|F90
	scmpicc file.c
	scmpicxx file.cpp
		F77="scmpif77" make
		F90="scmpif90" make
		CC="scmpicc" make
		CXX="scmpicxx" make
			F77="scmpif77" ./configure --host=mips64el-gentoo-linux-gnu --build=x86_64-pc-linux-gnu
			F90="scmpif905" ./configure --host=mips64el-gentoo-linux-gnu --build=x86_64-pc-linux-gnu
			CC="scmpicc" ./configure --host=mips64el-gentoo-linux-gnu --build=x86_64-pc-linux-gnu
			CXX="scmpicxx" ./configure --host=mips64el-gentoo-linux-gnu --build=x86_64-pc-linux-gnu
			[F77|F90]="scmpif[77|90]" make
			[CC|CXX]="scmpi[cc|cxx]" make

	scgcc file.c	CC="scgcc" make	CC="scgcc" ./configure --host=mips64el-gentoo-linux-gnu --build=x86_64-pc-linux-gnu
			CC="scgcc" make

Porting or Building an Application Natively on the System

Whether you are porting an existing application or building a new application, there are only a few things you need to do before you can run it on the System:

1. Install the source files and any libraries not supplied with the SiCortex software suite on a shared file system that is visible on the System’s nodes.

2. Log on to the System. (For details, see *Logging on to the System* on page 16.)

3. Allocate another node in the partition using `srun`’s `-A` option.

   ```
   $ srun -p sc1-comp1 -A -N1
   ```

   For more information on using `srun`’s `-A` option, see *Allocating Resources* on page 19.

4. Find the node that SLURM allocated.

   ```
   $ squeue | grep <username>
   ```

5. Connect to the allocated node.

   ```
   $ ssh <allocated_node_returned_by_squeue>
   ```

6. Set compiler and linker options.

⚠ If you use autotools and you need to change the application's build system or add other parameters to the configure command, edit the application's `configure.in` or `configure.ac` and `Makefile.am` files, not the configure or makefile files generated by autotools. Then use `automake` and `autoconf` to regenerate the makefiles.

- By default, the compiler builds n64 ABI executables (for details, see *Chapter 10, Understanding the Application Binary Interfaces on* page 135). So, if yours is an n32 application, you need to specify that to the compiler.

  Append `32` to the end of the compiler name; for example, `pathcc32`, `pathf9532`, `pathCC32`, and so on.

☼ Make sure your System Administrator has mounted the n32 buildroot on the nodes where you intend to build your n32 application.

- If your application uses user-supplied libraries or header files, you need to specify their location to the compiler:

  For **user-supplied libraries**, supply the `-L <pathname>` flag to the compiler command line, or add `LDFLAGS="-L <pathname>"` to your Makefile or to the `./configure` command line if you are using autotools.

☼ Determine any interlibrary dependencies and list the libraries accordingly. (Libraries that use functions or symbols contained in other libraries must appear on the link line before the libraries whose functions or symbols they use.) This rule also applies to object files.

  For **user-supplied header files**, supply the `-I <pathname>` flag to the compiler command line, or add `CFLAGS|FFLAGS="-I <pathname>"` to your Makefile or to the `./configure` command line if you are using autotools.

- If you want to compile for debugging, use the compiler's `-g` option. If you want to include information for the `gprof` profiler, use the compiler's `-gp` option.

7. Select a compiler. See *Summary of Simple Build Methods on* page 30.

We recommend using `-lscmpi_debug` until you have finished debugging your MPI application, and then switch over to `-lscmpi`.

For more details, see *MPI Library* on page 118.

8. Launch the compiler/linker at the command line, or type `make` to compile and link.

   To run your application, see *Chapter 2, Running Applications* on page 15. To debug your application, see *Chapter 4, Debugging Applications on* page 39. To optimize your application, see *Chapter 5, Optimizing Application Performance on* page 51.

Building an Application on the Cross-Development Workstation

On the cross-development workstation, you call the compilers and associated utilities by adding the `sc` prefix to the name of the tool. For example, the `gcc` cross-compiler is `scgcc` and the linker is `scld`. The `sc` prefix instructs the cross-compiler and linker to look in the correct directories for the cross-compiled headers and libraries supplied with the SiCortex cross-development software suite.

Except for the `sc` prefix, the basic procedure for compiling and linking applications on the cross-development workstation mirrors that for compiling and linking on the System, unless you use autotools.

On the workstation that has the SiCortex Cross-Development Toolkit installed:

1. Create or edit your makefiles and header files as needed.

   ⚠ If you use autotools and you need to change the application's build system or add other parameters to the configure command, edit the application's `configure.in` or `configure.ac` and `Makefile.am` files, not the configure or Makefile files generated by autotools. Then use `automake` and `autoconf` to regenerate the makefiles.

2. Review the options for setting the behavior of the compiler and linker starting at Step 6 on page 32, and then consider these additional options:

- If building an n32 application, specify that to the compiler by appending `32` to the end of the compiler name; for example, `scpathcc32`, `scpathf9532`, `scpathCC32`, and so on.

- If using autotools, you need to specify the *host* and *build* architectures (the environment on which the executable will run and the environment on which the executable is generated, respectively) when you run `configure`, so `configure` can set up correctly for cross compiling. To do so, on the `./configure` command line, type:

```
--host=mips64el-gentoo-linux-gnu \
                              --build=x86_64-pc-linux-gnu
```

3. Specify a compiler. See *Summary of Simple Build Methods on* page 30.

4. Launch the compiler/linker at the command line, or type `make` to compile and link.

   ☼ If you compiled your application on nonshared storage in the cross-development workstation, first copy your executable file and its data to a shared file system that is mounted and visible on the System.

   To run your application, see *Chapter 2, Running Applications* on page 15. To debug your application, see *Chapter 4, Debugging Applications on* page 39. To optimize your application, see *Chapter 5, Optimizing Application Performance on* page 51.

Troubleshooting Autotools-Based Cross-Compile Errors

If you use Autotools to build your application, you may encounter some of the problems listed here.

Build system incorrectly detects the target type

Only the compiler/linker knows how to detect the target system’ s type. Some Makefiles use other methods to detect its type, such as calls to `uname`, that are incompatible for cross-compiling. Replace any such incompatible code in the Makefile with code that uses the toolchain program environment variables (`CC`, `LD`, `AR`, `RANLIB`). Then set the variables to their correct values when you invoke the Makefile; for example, `CC=scgcc make`.

./configure uses incorrect default values

```
error: too few arguments to function 'gettimeofday'
error: incompatible types in assignment
error: conflicting types for 'malloc'
memcmp.c:11 error: conflicting types for 'memcmp'
```

Remedy: Determine what configuration variable is causing the problem (compare native and cross-compile runs of `./configure`) and reassign it the correct value as part of running `./configure`.

Example: `ac_cv_type_signal=void ./configure…`

./configure ignores the AR environment variable

Most likely, the Makefile contains hard-coded calls to the `ar` or `ranlib` programs. Modify the Makefile to use the environment variables `AR` and `RANLIB`, but defaulted to `ar` and `ranlib`. This scheme enables users to modify the `ar` and `ranlib` programs to use the correct cross-compile versions.

Build system uses wrong version of AR or RANLIB

```
Error: In function '__start'
: undefined reference to 'main' collect2
: ld returned 1 exit status
```

Remedy:

```
AR=scar RANLIB=scranlib ./configure \
     --host=mips64el-gentoo-linux-gnu --build=x86_64-pc-linux-gnu
     make
     make DESTDIR=/cross-root install
```

If `configure` ignores the `AR` variable, edit the `configure.in` file and add `AC_CHECK_PROG(AR,ar)` directly following the `AC_PROG_RANLIB` line. Run `autoconf`, and then rerun `configure`.

Link error: archive with no index

Build scripts target different architectures for the compile and the link processes.

```
...ld: SOMEFILE.o: Relocations in generic ELF (EM: 62)
SOMEFILE.o: could not read symbols: File in wrong format
collect2: ld returned 1 exit status
```

Remedy: Check to see if the wrong compiler is hardcoded in the makefile, and if so, change the code to the correct compiler. Else, the makefile may use a `CC` environment variable that the configure script has failed to override. You can manually override the makefile `CC` (`CXX`, `FC`, etc) variable by setting `CC` to the correct value before running `make`.

Including/linking against previously compiled header or library files

Header file or Libraries not found.

Remedy:

```
CFLAGS=-I/dir/include LDFLAGS=-L/dir/lib \
./configure --host=mips64el-gentoo-linux-gnu \
    --build=x86_64-pc-linux-gnu
make
```

Configure tries to run an executable during the build process

This often occurs when a build system tries to compile and execute a build tool that it uses in the build process. It typically generates an error such as: `./foo: cannot execute binary file`

Remedy: If the build tool is meant to run where the application does, and the only purpose is to generate output for the build system to use (e.g. to test for system characteristics), try running the tool on the System. Record the results, then modify the build system to use those results, instead of running the tool.

Application uses a configure script to build

Set the environment variables:

`CC=scgcc, LD=scld, AR=scar, RANLIB=scranlib` (maybe compilers too, e.g. `F77=scpathf95`) and `--host=mips64el-gentoo-linux-gnu` \
`--build=x86_64-pc-linux-gnu`

Compiling Reference Information

- Reference to online man pages:
  - http://pathscale.com PathScale User Documents
  - http://www.gnu.org/manual/manual.html GNU Manuals Online
  - http://www.linux.org/docs/ Linux Online Documentation
- References to books:
  - Stallman, Richard M., et. al. *GNU Make: A Program for Directing Recompilation*. The GNU Press, 2004.
  - Stallman, Richard M., et. al. *Using GCC: The GNU Compiler Collection Reference Manual*. The GNU Press., 2000.
  - Stallman, Richard M., et. al. *Volume 2: GNU Reference: Using and Porting the GNU Compiler Collection (GCC)*. Iuniverse, Inc, 2003.
  - Vaughan, Gary V., et al. *GNU AUTOCONF, AUTOMAKE, and LIBTOOL*. New Riders Publishing, 2000.

Chapter 4 Debugging Applications

In this section:

Compiling Tips for Debugging

- Compile your program using the compiler's `-g` debugging option.

  All of the supplied compilers use the `-g` option to enable user-friendly debugging. For details, see *Chapter 3, Compiling and Linking Applications* on page 25.

  You can debug your executables without compiling with the `-g` option, but doing so, you can view only the assembler code in the debuggers.

- Use optimization flags judiciously. Increased optimization can prevent the debugger from setting breakpoints and expanding variables.

Debugging with gdb

The SiCortex software toolkit includes the GNU debugger, `gdb`, which you can run natively on the System or remotely from the cross-development workstation to debug an application running on the System. Regardless of where you run it from, `gdb` works with the standard tools exactly as it does on any other platform.

⚠ The gdb debugger does not work with Fortran derived data types.

Debugging Natively with gdb

Run gdb on the System just as you would run it on any other platform:

1. Log on to the System. For details, see *Chapter 2, Running Applications* on page 15.
2. Compile the application using the -g option. For details, see *Porting or Building an Application Natively on the System* on page 32.
3. Start up gdb on the System, specifying the name of the application to debug, and then run the program using gdb commands.

Handling Core Dumps

⚠ The System does not automatically generate core dump files. To enable this feature, you must issue `ulimit -c unlimited` (bash) or `limit coredumpsize unlimited` (tcsh) before you run your application.

You can examine the core file generated when an application crashes. To do so, use any of the following commands within gdb.

- To start up gdb specifying the program you want to debug, use either command sequence:

  ```
  $ gdb <program>  or
  $ gdb -e program
  ```

- To start up gdb specifying the program and core file you want to debug, use either command sequence:

  ```
  $ gdb <program> <core file> or
  $ gdb -e program -c corefile
  ```

You can create a core file of a program that is running in gdb to save a snapshot of its state at any given time. To do so, while the program is running in gdb:

```
(gdb) gcore [filename]
```

Use this command to generate a core dump of the inferior process (the process that gdb spawns to run your program). The optional `filename` specifies the name of the file to which the output is written. If no file name is specified, gdb writes the output to the file `core.pid`, where `pid` is the id of the inferior process.

Using Stack Traces

You can examine information about your program's call stack routines and the local variables, registers, and function parameters your program's routines use. To do so, use these commands from within `gdb`:

- Use the `frame` command to move from one stack frame to another and print the stack frame you specify:

  ```
  (gdb) frame [stackframe]
  ```

  To specify a stack frame, supply its address or the stack frame number that `gdb` assigned to it. If you do not specify a stack frame, this command prints the information for the current stack frame (frame number `0`).

- Use `select-frame` command to move from one stack frame to another, without printing the frame information:

  ```
  (gdb) select-frame
  ```

- Use the `backtrace` command to print a summary of existing stack frames to trace the antecedents of the current stack frame:

  ```
  $ (gdb) backtrace [args]  or  $ (gdb) bt [args]
      n            /* Print only the innermost n frames
      -n           /* Print only the outermost n frames
      full[n, -n]  /* Print the local variables too
  ```

  The output displays one line per frame, starting at stack frame number 0 followed by its caller, stack frame number 1, and so on.

  If you do not supply an argument, this command prints a backtrace of the entire stack frame. To stop the backtrace, enter the system interrupt character, typically `Ctrl-c`.

  By default, `gdb` displays the backtrace only for the current thread in a multithreaded program. To display the backtrace of all threads, use the `thread apply` command this way:

  ```
  (gdb) thread apply all backtrace
  ```

See the `gdb` man page for more details.

Debugging Remotely with gdb

You can use the `gdbserver` program, included in the `gdb` software, to debug an application remotely from the cross-development workstation:

1. Log on to the System. For details, see *Chapter 2, Running Applications* on page 15.

2. Cross-compile the application on the cross-development workstation.(For details on cross-compiling, see *Building an Application on the Cross-Development Workstation* on page 34.)

   For example: `xdev $ scgcc -g prog.c -o prog`

3. Start up `gdbserver` on the System. Specify the link over which `gdb` and `gdbserver` will communicate and the name of the program to debug (in this case `./prog`).

   For example: `HOST $ gdbserver :7654 ./prog`

   This example specifies the TCP port `:7654`.

4. Start up sc`gdb` on the cross-development workstation and again specify the program to debug.

   In this example: `xdev` $ `scgdb ./prog`

5. When `scgdb` returns the `(gdb)` prompt, first tell `gdb` where to find the shared libraries, then establish the connection to the System—in this case, using the same TCP port, `:7654`, specified in Step 3.

   ```
   xdev $ (gdb) set sysroot <PATH_TO_ROOTFS>*
   xdev $ (gdb) target remote HOST:7654
   ```

   `scgdb` then connects over the network to the `gdbserver` program running on the System.

6. Run the debugger as you would normally. You can use all of the usual commands to examine and change data and to step through or continue the remote program.

See the `gdbserver` man page for more details.

[*] The System ships with the rootfs in `/opt/sicortex/rootfs/default`.

Debugging with TotalView

With TotalView, you debug single- and multinode applications remotely from the cross-development workstation.

- Ask your System Administrator whether TotalView is installed on the System and on the cross-development workstation. One license for each instance (process) is required.
- TotalView works with code generated from all of the supplied compilers. Make sure you compile code you want to debug using the compiler's `-g` option.

  ⚠ Compiling with optimization enabled can interfere with the values examined within the debugger. Generally, reliability of these values decreases with increasing optimization. At level `-O3`, these values are unreliable.
- SiCortex TotalView does not include these features: memory debugging, support for SHMEM or PVM, o32 executables, watchpoints, or compiled EVAL points.

For complete documentation on the TotalView debugger, visit www.totalviewtech.com.

Environment Setup

The TotalView front end provides both graphical and command line interfaces for debugging parallel applications. It launches applications on the System via the network.

The TotalView front end runs only on an x86_64 workstation that has the SiCortex cross-development toolkit installed, and it must have access to the MPI libraries.

Before you start up the TotalView debugger:

- The TotalView software resides in `<install_path>/toolworks/totalview.<version>/` on the cross-development workstation. It's easier to put this in your path than to type it each time you invoke the debugger.
- By default, TotalView uses `ssh -X` to create a remote shell. It's much easier if you set up `ssh -X` to execute a command without requiring a password.
- Make sure your application executables are in a directory that is visible through the same path from both the cross-development workstation and the head node where `srun` launches the job.

- Because TotalView is an `X` application, if you `access` the cross-development workstation from your user workstation, you need to supply the `-X` option to `ssh` to open an `X` display.

Starting a Job

You can start a job either from the command line or from within the TotalView GUI. To invoke TotalView on SiCortex systems, you must use the `sc` prefix: `sctotalview` and `sctv8` are equivalent and access the GUI interface; `sctotalviewcli` and `sctv8cli` are equivalent and access the command line interface.

Command synopsis:

```
# To debug an application in the CLI version
$ sctv8cli -r <nodename> <pathtoexecutable> -a <args>

# To debug an application in the GUI version
$ sctv8 -r <nodename> srun -a <args> pathtoexecutable>
```

As an example, to launch the TotalView GUI and debug the MPI program, `a.out`, located in `~/helloworld/mpi`:

1. On the cross-development workstation, start up TotalView and submit the job to SLURM:

   ```
   $ sctv8 -r <head_node> srun -a -p sc1 -n2
                               ~/helloworld/mpi/a.out
   ```

   Where:

   `-r <head_node>` Specifies the head node on which to launch the `srun` command.

   `-a` Pass all following arguments to `srun`.

   `-n 2` Specifies two processes.

   `~/helloworld/mpi/a.out`
   Is the executable to debug.

   TotalView opens with the Process and Root windows displaying a clean slate.

⚠ To quit TotalView, use the **File>Exit** method from the menu bar, not the ☒ icon in the upper-right corner of the GUI. Clicking the **X** icon leaves an `srun.mips.<processID>` file for every process your program started in the working directory, which you must remove manually. The **File>Exit** method removes all such `srun.mips.<processID>` files automatically.

⚠ Because TotalView runs your job using `srun`, when you quit TotalView, unless you ran the program to completion, you must also `scancel` the job. For details, see *Canceling a Job: scancel and ^C* on page 20.

For detailed instructions on using the TotalView Debugger, see the user documentation at www.totalviewtech.com.

Memory Debugging with DUMA

Detect Unintentional Memory Access (DUMA) uses the virtual memory hardware to place protected pages around dynamic allocations and to track memory accesses. It works with C and C++ applications.

You can link the DUMA library (`libduma`) with the application, or you can preload the library using the `duma` script.

To debug your C/C++ application using DUMA, you need only:

- Compile your application using the compiler's `-g` option, with or without linking with the DUMA library;
- Run the executable (with the `duma` script if you did not link with the DUMA library);
- Run `gdb` on any resulting core dump.

By default DUMA detects memory overruns. So, for example, to debug memory overruns for the simple `foo.c` application:

```
test@sc1-m0n6: ~$ cat foo.c
main()
{
  char *a = malloc(10);
  a[10] = 0;
}
```

Compile the `foo.c` application:

```
test@sc1-m0n6: ~$ gcc -g foo.c
foo.c: In function 'main':
foo.c:3: warning: incompatible implicit declaration of
built-in function 'malloc'
```

Run the `duma` script on the resulting executable:

```
test@sc1-m0n6: ~$ duma ./a.out
DUMA 2.5.12 (shared library, NO_LEAKDETECTION)
Copyright (C) 2006 Michael Eddington <meddington@gmail.com>
Copyright (C) 2002-2008 Hayati Ayguen <h_ayguen@web.de>, Procitec GmbH
Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

/home/test/SVN/build/tools/usr/bin/duma: line 17: 4895 Segmentation fault (core dumped)
(export LD_PRELOAD=libduma.so.0.0.0; export DYLD_INSERT_LIBRARIES=libduma.dylib; export
DYLD_FORCE_FLAT_NAMESPACE=1; exec $* )
```

Then run gdb on the resulting core dump:

```
test@sc1-m0n6: ~$ gdb ./a.out core.sc1-m0n6.scsystem.4895
GNU gdb 6.7.1
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "mips64el-gentoo-linux-gnu"...
Using host libthread_db library "/lib64/libthread_db.so.1".
Reading symbols from /net/home/test/SVN/perf/usr/lib/libduma.so.0.0.0...done.
Loaded symbols for /home/test/SVN/build/tools/usr/lib/libduma.so.0.0.0
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/libpthread.so.0...done.
Loaded symbols for /lib64/libpthread.so.0
Reading symbols from /usr/lib64/libstdc++.so.6...done.
Loaded symbols for /usr/lib64/libstdc++.so.6
Reading symbols from /lib64/libm.so.6...done.
Loaded symbols for /lib64/libm.so.6
Reading symbols from /usr/lib64/libgcc_s.so.1...done.
Loaded symbols for /usr/lib64/libgcc_s.so.1
Reading symbols from /lib64/ld.so.1...done.
Loaded symbols for /lib64/ld.so.1
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
#0  0x000000012000097c in main () at foo.c:4
4         a[11] = 0;
```

To detect memory underruns, use the DUMA_PROTECT_BELOW=<int> environment variable:

```
test@sc1-m0n6: ~$ DUMA_PROTECT_BELOW=1 duma ./a.out
DUMA 2.5.12 (shared library, NO_LEAKDETECTION)
Copyright (C) 2006 Michael Eddington <meddington@gmail.com>
Copyright (C) 2002-2008 Hayati Ayguen <h_ayguen@web.de>, Procitec GmbH
Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
```

To see a list of all memory allocations your application uses, use the DUMA_SHOW_ALLOC environment variable:

```
test@sc1-m0n6: ~/SLURM$ DUMA_SHOW_ALLOC=1 srun -p sf2 -n1 duma ./pi | head -30
DUMA 2.5.12 (shared library, NO_LEAKDETECTION)
Copyright (C) 2006 Michael Eddington <meddington@gmail.com>
Copyright (C) 2002-2008 Hayati Ayguen <h_ayguen@web.de>, Procitec GmbH
Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

DUMA: Allocating 123 bytes.
DUMA: Freeing 123 bytes.
DUMA: Allocating 8 bytes.
.
.
.
```

Memory Debugging with Mudflap

GCC's powerful built-in memory debugging tool, `mudflap`, combines source code instrumentation with additional runtime support.

Use Mudflap this way:

```
test@sc1-m0n6: ~/tests$ gcc -fmudflap unwind.c -lmudflap
```

`-fmudflap` Instructs the compiler to instrument all risky pointer/array dereferencing operations, some standard library string/heap functions, and some other associated constructs having range/validity tests.

For multithreaded applications, use `-fmudflapth` instead.

`-lmudflap` The libmudflap runtime library. To link with the libmudflap library, you must supply both `-fmudflap` and `-lmudflap` on the link line.

For multithreaded applications, use `-lmudflapth` instead.

You can control the runtime behavior of your instrumented code by using the `MUDFLAP_OPTIONS` environment variable to set various mudflap parameters. For details, see http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging.

Here is a simple application, `hello_world`, that writes past the end of an array:

```
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  char *a =(char *)malloc(1000*sizeof(char));
  memset(a,0,1004);
  printf("Hello world\n");
  fclose(stdout);
  exit(0);
}
```

Compile it using `mudflap`:

```
test@sc1-m0n6: ~$ gcc -g -Wall -fmudflap hello_world.c
-lmudflap
```

Run the executable:

```
test@sc1-m0n6: ~$ ./a.out
*******
mudflap violation 1 (check/write): time=1179835120.249191 ptr=0x120113750 size=1004
pc=0x55556892b4 location=`(memset dest)'
      /usr/lib64/libmudflap.so.0(__mf_backtrace+0x1cc) [0x555568695c]
Nearby object 1: checked region begins 0B into and ends 4B after
mudflap object 0x120113b90: name=`malloc region'
bounds=[0x120113750,0x120113b37] size=1000 area=heap check=0r/3w liveness=3
alloc time=1179835120.248191 pc=0x5555688c94
      /usr/lib64/libmudflap.so.0(__mf_backtrace+0x7c) [0x555568680c]
      [0x120208c50]
number of nearby objects: 1
Hello world
test@sc1-m0n6: ~$
```

Since MIPS doesn't have a frame-pointer, which prevents us from getting true backtraces, how can we find the offending code? We can promote the violation either to start `gdb` or to generate an `SEGV`.

```
test@sc1-m0n6: ~$ export MUDFLAP_OPTIONS="-check-initialization -viol-gdb"
test@sc1-m0n6: ~$ ./a.out
*******
mudflap violation 1 (check/write): time=1179835511.161191 ptr=0x1201138d0 size=1004
pc=0x55556892b4 location=`(memset dest)'
      /usr/lib64/libmudflap.so.0(__mf_backtrace+0x1cc) [0x555568695c]
Nearby object 1: checked region begins 0B into and ends 4B after
mudflap object 0x120113d10: name=`malloc region'
bounds=[0x1201138d0,0x120113cb7] size=1000 area=heap check=0r/3w liveness=3
alloc time=1179835511.161191 pc=0x5555688c94
      /usr/lib64/libmudflap.so.0(__mf_backtrace+0x7c) [0x555568680c]
      [0x120221e70]
number of nearby objects: 1
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "mips64el-gentoo-linux-gnu".
Attaching to process 1689
Reading symbols from /net/home/test/a.out...done.
Using host libthread_db library "/lib64/libthread_db.so.1".
```

```
Reading symbols from /usr/lib64/libmudflap.so.0...done.
Loaded symbols for /usr/lib64/libmudflap.so.0
Reading symbols from /lib64/libc.so.6...done.
Loaded symbols for /lib64/libc.so.6
Reading symbols from /lib64/libdl.so.2...done.
Loaded symbols for /lib64/libdl.so.2
Reading symbols from /lib64/ld.so.1...done.
Loaded symbols for /lib64/ld.so.1
0x00000055559eb45c in waitpid () from /lib64/libc.so.6
(gdb) where
#0  0x00000055559eb45c in waitpid () from /lib64/libc.so.6
#1  0x00000055559884bc in do_system () from /lib64/libc.so.6
#2  0x0000005555686d1c in __mf_violation () from /usr/lib64/libmudflap.so.0
#3  0x000000555568811c in __mfu_check () from /usr/lib64/libmudflap.so.0
#4  0x00000055556892b4 in __mf_check () from /usr/lib64/libmudflap.so.0
#5  0x000000555569868c in __mfwrap_memset () from /usr/lib64/libmudflap.so.0
#6  0x0000000120000cfc in main () at hello_world.c:7
(gdb) up 6
#6  0x0000000120000cfc in main () at hello_world.c:7
7         memset(a,0,1004);
(gdb)
```

If you want to just SEGV, use -viol-segv.

Chapter 5 Optimizing Application Performance

In this section:

Performance tuning is an iterative process used to optimize the efficiency of a program. Typically, it involves finding the programs's hot spots (code that uses a disproportionate amount of processor time) and then eliminating the bottlenecks (code that uses processor resources inefficiently, causing unnecessary delays) in them.

General Procedure for Optimizing an Application

In general, the steps for tuning applications for optimal performance are:

1. Compile the application using the appropriate optimization options. Include the `-g` option to help with profiling and correlating data with the source code.

   ☼ Using `-g` with `-O#` does not degrade optimization; it merely increases the number of symbols the compiler must keep.

2. Characterize the application.

   Run `papiex` using the -a option to establish baseline performance data.

   If the data suggests the application contains hot spots, profile the application using the tool appropriate for the area of concern (for example, run `mpipex` on an application that appears to spend an inordinate amount of time in MPI tasks).

3. Profile the application.

   Profilers measure a program's runtime characteristics and resource utilization. Use these tools to determine which areas of code present the greatest potential for increasing performance before you actually begin tuning code.

   - Run `mpipex` on an application to detect MPI bottlenecks.
   - Run `hpcex` on an application to detect computational bottlenecks.
   - Run `ioex` on an application to detect I/O bottlenecks.

4. Take advantage of compiler and preprocessor optimizations whenever possible and appropriate. (See *Using Compiler Options* on page 26 and *Build Optimizations on* page 53.)

5. To get to the root of persistent bottlenecks, instrument and run the application with a tool (`TAU/tauex`, and `Vampir/vampirtrace`) that provides detailed profile and trace data that can identify specific bottlenecks.

For small kernels of code, consider using `papiex`, enclosing the code of interest within software calipers.

6. Tune application code blocks that dominate execution time.

   Consider optimizing your underlying algorithm. For example, even a finely tuned `O(N*N)` sorting algorithm may perform significantly worse than an untuned `O(N log N)` algorithm.

   For data dependent computations, base benchmarks on a variety of realistic (both size and values) input data sets. Maintain consistent input data during the fine-tuning process.

7. Know when to stop.

   Successive optimizations carry diminishing returns.

Build Optimizations

You can do some simple things up front to optimize your application's performance.

- Compiler optimization flags

  The compilers default to `-O0` optimization when debugging is enabled. With debugging disabled, the GNU compilers default to `-O0`, but the PathScale compilers default to `-O2`. (For more details, see *Using Compiler Options* on page 26.)

  The PathScale compiler

  At level `-O2`, the PathScale compiler performs optimizations on inner loops, instruction scheduling, global register allocation, and function scopes. Table 4 shows option settings for each optimization level as defined for the PathScale compilers. This list is not all inclusive. For details on these options, see the PathScale compiler's documentation and man pages.

Table 4. Effects of PathScale compilers' optimization flags

Option Name	**-O0**	**-O1**	**-O2**	**-O3**	**Notes**
-OPT:div_split	off	off	off	off	on if IEEE_arithmatic = 3
-OPT:fast_complex	off	off	off	off	on if roundoff = 3
-OPT:fast_exp	off	off	off	on	on if roundoff ≥ 1
-OPT:fast_nint	off	off	off	off	on if roundoff = 3
-OPT:fast_sqrt	off	off	off	off	
-OPT:fast_trunc	off	off	off	on	on if roundoff ≥ 1
-OPT:fold_reassociate	off	off	off	off	on if roundoff ≥ 2

Table 4. Effects of PathScale compilers' optimization flags (Cont'd)

Option Name	-O0	-O1	-O2	-O3	Notes
-OPT:fold_unsafe_relops	on	on	on	on	
-OPT:fold_insigned_relops	off	off	off	off	
-OPT:IEEE_arithmatic	1	1	1	2	
-OPT:IEEE_NaN_inf	on	on	on	on	
-OPT:recip	off	off	off	off	on if roundoff ≥ 2
-OPT:roundoff	0	0	0	1	
-OPT:fast_math	off	off	off	off	on if roundoff ≥ 2
-OPT:rsqrt	0	0	0	0	1 if roundoff ≥ 2
-OPT:OLimit	6000	6000	6000	9000	
-OPT:align_unsafe	off	off	off	on	
-OPT:goto	off	on	on	on	
-OPT:reorg_common	off	off	off	on	on when all files referencing the common block are compiled at -O3; else off
-LNO:opt	0	–	–	1	

At level `-O3`, the PathScale compiler performs additional optimizations that usually increase the performance of applications, but occasionally, these additional optimizations slow some applications down. If this occurs, try using the PathScale compiler's `-LNO:prefetch=0` flag, which turns off prefetching and helps mainly codes that fit in cache.

⚠ At level `-O3`, the PathScale compiler performs certain math optimizations that, on rare occasions, cause incorrect results.

☼ The `pathopt2` tool (see page 28) can facilitate the process of progressively compiling and comparing results.

For more details, see *PathScale Compiler Options* on page 27, and the PathScale documentation and man pages.

The gcc compiler

At level `-O1`, the gcc compiler attempts to reduce code size and execution time of the resulting executable, without increasing compilation time.

At level `-O2`, the gcc compiler applies optimizations that do not involve a trade-off between space and speed, so it does no loop unrolling or function inlining. It performs all `-O1` level optimizations plus many additional optimizations, including alignment of functions, jumps, loops, and labels; global and common subexpression elimination; and function, block, and instruction reordering and rescheduling optimizations. While these optimizations increase compilation time, they also increase the performance of the resulting executable.

☼ If your application uses computed gotos, you may get better runtime performance by disabling (`-fno-gcse`) global common subexpression elimination.

At level `-O3`, the gcc compiler performs all `-O1` and `-O2` optimizations plus performs loop unrolling, function inlining, and cleanup of redundant spilling.

At level `-Os`, the gcc compiler optimizes for code size. It performs all `-O1` and `-O2` optimizations that do not typically increase code size, plus it disables the `align-` and `reorder-` optimizations enabled at `-O2`, the default `prefetch-loop-arrays`, and the `tree-vect-loop-version` optimization.

For more details, see *GNU Compiler Options* on page 29, and gcc documentation and man pages.

General Optimization Tips

These are some general things you can do to optimize your application's performance:

- Understand how the hardware is implementing your code and what code/algorithms are causing delays. It's easy to hyperfocus on one or two bottlenecks, continuing to optimize them long after the critical path has moved elsewhere.
- Balance the load by distributing the work, which includes data transfers. For example, don't set up one node to simultaneously receive data from many nodes.
- Plan how to use each level of the memory hierarchy, getting as much reuse out of the data at each level as possible, including registers, L1 and L2 caches, and main memory.

☼ For background information on how the memory caches work, see *Memory System Operation on* page 132.

Memory Access Optimizations

These are some specific things you can do to make sure your code optimally accesses memory:

- Loop optimization

  Unroll critical loops to keep independent operations flowing through the processor. Generally, you can issue a dual floating-point operation instruction together with a floating-point load or store operation or with an integer instruction. The processor will issue one instruction (sometimes two) every cycle, but floating-point instructions usually take four cycles to compute their result. You can issue other instructions in the mean time, as long as those instructions do not depend on the result of the first.(For more details, see *Memory System Operation on* page 132.)

- Array optimizations
  - Unit stride (stride1) memory accesses make the most efficient use of caches.
  - For larger arrays, increase their row dimension to a prime so that rows, columns and diagonals don't cause frequent cache collisions. Depending on the ratio of column stride to cache way size (L1 data cache is 4-way; L2 cache is 2-way), walking down an array column could displace a row. (For more details, see *L1 Data Cache Stalls on* page 133.)
  - Integer operations, specifically array indexing, cost as much as floating point operations or cache hits. Therefore it's advantageous to keep address computations very simple. In the extreme, it may be more efficient to use separately named variables (`A0, A1, A2,…`) instead of an array (`A[0,…,2]`) for values you want to keep in registers.But doing so can impact code readability.

SCTICK Fast Timers

The fast timers are architecture-specific and therefore not portable. They use a fast access path to minimize the overhead of context switching into the kernel. Use them to compare and fine tune optimizations you make to your code, or to measure small intervals, as specific regions of code, by calling the timer before and after the target block of code.

- `sc_tick()` measures time in clock ticks
- `sc_tick_nsec()` measures time in nanoseconds

Both timers measure time from an arbitrary point of reference in the past.

To use them, you must include `sctick.h` in your C/C++ program.

☼ Fortran is not supported, but you can use the PAPI fast timers, `PAPI_get_real_cyc()` and `PAPI_get_us()`. For details, see the appropriate PAPI man pages.

Function prototypes:

```
long long sc_tick(void)

double sc_tick_nsec(void)
```

For example, `use_sctick.c`:

```
$ cat use_sctick.c
#include <linux/sicortex/sctick.h>
#include <stdio.h>

int main(void) {
  int i;
  float a = 1.0;

  // get starting time
  long long t1 = sc_tick();

  // do some work
  for (i=0; i<100; i++)
    a = a* 1.000001;

  // get ending time
  long long t2= sc_tick();

  printf("a=%f\n", a);
  printf("100 FP adds took %d ticks\n", t2-t1);
  return 0;
}
```

Compile `use_sctick.c` and run the executable:

```
$ pathcc -g -O3 use_sctick.c -o use_sctick
$ ./use_sctick
a=1.000095
100 FP adds took 790 ticks
```

Application Performance Tools

The SiCortx software provides a rich set of tools for optimizing application performance and for finding and resolving performance issues in application code, such as:

- How well is my code running?
- Which function(s) are using the most processor cycles?
- Which compiler flags might provide better optimization? For details, see *Build Optimizations on* page 53, and x on page x (compiler chapter).

- Where are the bottlenecks—MPI, I/O, or memory stalls?
- Where are memory stalls occurring (L1 or L2 caches, faulty loops)?

All of the performance tools run on the System's nodes. All are open source, re-engineered for the SiCortex architecture, but you can download the original source code from the internet

Table 5. Tool Descriptions

Tool	Description
papiex	Based on PAPI, `papiex` provides an overall view of the application's performance. `It` measures the aggregate totals and related metrics from the processor performance counters. Metrics include statistics across all tasks, memory usage, and time spent in I/O and MPI operations. `Papiex` does not require users to instrument or recompile applications. It works on executables linked with shared libraries only. For quickstart usage, see page 61. For details, see *Using Papiex on* page 64.
mpipex	Based on MpiP, `mpipex` measures the application's MPI performance. It measures the time an MPI application spends communicating, then aggregates statistics on the number and duration of MPI calls, average message size, and so on. It works only with dynamically-linked MPI program executables. Using `mpipex` on dynamically-linked MPI applications, you can avoid the link step that using `mpiP` requires. For quickstart usage, see page 61. For details, see *Using Mpipex on* page 71.
hpcex	The `hpcex` tool is a front-end to the HPCToolkit. It performs event-based sampling and statistical profiling of serial and parallel applications and outputs the data with source code, similarly to `gprof`. It can correlate event statistics with the source code. Besides `hpcex`, the toolkit also includes `hpcstruct`, `hpcproftt`, `hpcprof-flat`, and `hpcviewer`. For quickstart usage, see page 61. For details, see *Using HPCex on* page 74.
tauex	The TAU toolkit is an instrumentation profiling and tracing system for parallel and serial applications. It supports both tracing and profiling models for C, C++, Fortran, MPI, OpenMP, and pthread applications. Using PAPI, it reports exact function-, block-, and statement-level hardware counts. For profiling, it summarizes aggregate statistics for routines and statements and generates callpath profiles for routines. For tracing, it logs all MPI interprocess communications and events. Use `tauex` at runtime to add or change TAU profiling and tracing options and to specify which hardware events to count during execution of an instrumented executable. Vampir can analyze trace data (`.OTF` files) output by TAU. For quickstart usage, see page 62. For details, see *Using TAU on* page 79.

Table 5. Tool Descriptions (Cont'd)

Tool	Description
Vampir	`Vampirtrace` provides a convenient measurement infrastructure for program performance tracing. It records the data that Vampir analyzes and graphically displays. `Vampirtrace` provides instrumentation and tracing facilities tailored for parallel and HPC applications. It covers user code, MPI, and OpenMP instructions. Vampir, available from ParaTools, provides a visualization tool that graphically depicts time-based traces of hardware performance data (for example, correlation of application behavior with the flow of MPI messages and data). For quickstart usage, see page 62. For details, see *Using Vampirtrace on* page 82.
gptlex	Based on the General Purpose Timing Library (GPTL), the `gptlex` tool provides control over GPTL functionality at run time, through command line arguments. The General Purpose Timing Library is a simple instrumentation package that provides easy access to wall clock timers, processor timers, and PAPI counters. For quickstart usage, see page 62. For details, see *Using TAU on* page 79.
ioex	`Ioex` uses the `papiex` driver to provide simple I/O statistics for dynamically linked executables (not static executables). It measures the performance of the application's I/O operations by intercepting calls to System I/O, C library functions, and MPI I/O, and then collecting the call statistics. It works with threaded and MPI applications. For quickstart usage, see page 62. For details, see *Using Ioex on* page 92.
pfmon	A simple monitoring tool that collects simple event counts or samples from unmodified binaries or for an entire system. It uses software breakpoints and produces repeatable counts for events, such as instruction counts, and so on. For quickstart usage, see page 63. For details, see *Using Pfmon on* page 94.
oprofile	Oprofile runs on a single node. It is a node-wide statistical profiler that samples performance counter interrupt events. Once started, it profiles all code (hardware and software interrupt handlers, kernel modules, shared libraries, applications, etc.) running on the processors until it is explicitly stopped. For details, see *Using Oprofile on* page 94.

The "ex" tools (e.g. `papiex`) do not require you to manually instrument your source code. They provide high-level aggregate and statistical performance measurements of an application. All work the same way. Using similar arguments and shared libraries, they are dynamically loaded, and because they do not require instrumented code, you need not recompile your application to use them. Use these tools to get an overall picture of an application and to identify problem code. An exception, `hpcex` can also provide low-level information.

The library tools (e.g.TAU) require you to manually instrument your source code and link with the tool's library. They provide detailed profile and trace data, which can identify specific bottlenecks in parallel applications, including OpenMP and MPI point-to-point and collective operations, and they can correlate application behavior with the flow of MPI messages and data. Because the library tools require instrumented code, you must recompile your application to use them. Use these tools to examine problem code at the function-, block-, and statement-level.

Use the compiler's `-g` option to generate symbols that enable the performance tools to provide more information than just function names.

Memory corruption errors can be diagnosed using memory debugging tools. See *Memory Debugging with DUMA* on page 45 and *Memory Debugging with Mudflap* on page 47.

Invoking the Tools

First you need to connect to the head node. For details, see *Connecting to a Head Node* on page 16.

Using `srun`, run all applications under the tools on multiple nodes this way:

```
$ srun -p <partition> -n <ntasks> toolname [args] <executable> [args]
```

or run all applications under the tools on a single node this way:

```
$ srun -p <partition> -N 1 toolname [args] <executable> [args]
```

Table 6 on page 61 provides typical usage examples for running the tools on multiple processors.

In Table 6, assume that **`$ srun -p <partition> -n <ntasks>`** immediately precedes tool commands that run an application executable. These example commands begin with the tool name.

Table 6. Tool Usage Examples

Tool	**Commands**

papiex	• List all hardware events available on the System:
	`$ papiex -l`
	• Measure the overall performance of an application, using all useful events:
	`papiex -a <executable> [args]`
	• Measure the overall performance of an application, using specific events:
	`papiex -e CPU_DCMISS -e CPU_MSTALL <executable> [args]`
	• Measure how much memory an application uses:
	`papiex -x <executable> [args]`

mpipex	Profile an MPI application: `mpiex <executable> [args]`

hpcex	• Gather profile data at the default sampling rate: every 999,999 cycles
	`hpcex <executable> [args]`
	• Gather profile data simultaneously on two events, using a different sampling interval for each event:
	`hpcex -e CPU_DCMISS:654321 -e CPU_MSTALL:456789 <executable> [args]`
	• Generate loop-level data for `hpcproftt`:
	`hpcstruct <executable> > <executable>.psxml`
	• Display profiles, broken down into files, functions, and lines:
	`hpcproftt -e <executable>.hpcex.*`
	• Display profiles and instrumented source code:
	`hpcproftt --src=all -e <executable>.hpcex.*`
	• Display summary profiles:
	`hpcproftt -M sum-only -e <executable>.hpcex.*`
	• (optional) Intermingle source and assembly code:
	`hpcproftt --obj=s -S <executable>.psxml <executable>.hpcex.*`
	• Display summary profiles, broken down into files, functions, loops, and lines:
	`hpcproftt -S <executable>.psxml -M sum-only -e <executable>.hpcex.*`
	• Create a database viewable in `hpcviewer` (Java GUI) or in `paraprof` (from TAU):
	`hpcprof-flat -S <executable>.psxml <executable>.hpcex.*`
	`$ ls experiment-db`
	`experiment-db:`
	`config.xml experiment.xml`
	To run `hpcviewer` on an x86 workstation running `X` and the Java Runtime and view the results:
	`$ hpcviewer experiment-db/experiemnt.xml`

Table 6. Tool Usage Examples (Cont'd)

Tool	Commands

tauex	• Instrument the source code:
	`$ make CC=taucc F77=tauf90 FC=tauf90 CXX=taucxx …`
	• Generate the callpath profiles, include events `CPU_DCMISS` (L1 data cache misses) and `CPU_MSTALL` (dependency stalls):
	`tauex -T CALLPATH -e CPU_DMISS -e CPU_MSTALL <executable> [args]`
	• OpenMP and C++ sources must be compiled using the `-tau:` options that correspond to those used by tauex.
	For example, first compile a mixed Fortran + MPI + OpenMP application:
	`$ make F77="tauf90 -tau:MPI,OPENMP"`
	Then generate the flat profiles, include events `CPU_DCMISS` (L1 data cache misses) and `CPU_MSTALL` (dependency stalls):
	`tauex -T OPENMP -e CPU_DMISS -e CPU_MSTALL <executable> [args]`
	• Display the profiles and call paths on the x86_64 workstation:
	`$ cd <executable>.tau.<slurm_job_id>`
	`$ paraprof *`

Vampir	• Generate Vampir traces, including `CPU_DCMISS` (L1 data cache misses) and `CPU_MSTALL` (dependency stalls) events
	`$ make CC=taucc F77=tauf90 FC=tauf90 CXX=taucxx …`
	`$ export VT_METRICS=CPU_DCMISS:CPU_MSTALL`
	`$ srun -p <partition> -n <ntasks> <executable> [args]`
	• Display Vampir .oft traces on nodes:
	`$ srun -p <partition> -n <ntasks> vngd`
	• Display Vampir .oft traces on the x86_64 workstation:
	`$ vngd`
	then connect the GUI to the address returned from the `vngd` command.

gptlex	Generate a text call tree and count floating-point arithmetic.
	• First, auto-instrument your source code using the compiler's `-finstrument-functions` option:
	`pathcc -g -O3 -finstrument-functions [compiler_opts] -o <executable> -c \`
	`<executable>.c`
	• Generate the tree and count floating-point arithmetic:
	`gptlex -G -e CPU_FPARITH <executable> [args]`

ioex	Profile I/O: `ioex <executable> [args]`

Table 6. Tool Usage Examples (Cont'd)

Tool	Commands

pfmon	• Measure L1 data cache misses and dependency stalls for a particular function or address range in an application:
	`pfmon -e CPU_DCMISS, CPU_MSTALL --trigger-code-start-address=<fn/addr> --trigger-code-stop-address=<fn/addr> --trigger-code-repeat <executable> [args]`
	• Profile L1 data cache misses for a particular function or address range in an application:
	`pfmon -e CPU_DCMISS --trigger-code-start-address=<fn/addr> --trigger-code-stop-address=<fn/addr> --trigger-code-repeat --smpl-periods-random=0xff:5 --long-smpl-period=100000 --resolve-addr <executable> [args]`
	• Count cycles in interrupt and kernel modes:
	`pfmon -K -I <executable> [args]`
	• Profile system-wide in user and kernel modes, ignoring all context switches, for the duration of process `<myapp>`:
	`pfmon --system-wide -U -K --smpl-module=inst-hist -e CPU_CYCLES --smpl-periods-random=0xff:5 --long-smpl-period=100000 --resolve-addr <myapp> [args]`

Displaying Available Hardware Performance Counter Events

For each of the performance tools, you can display a list of the hardware performance counter events it can count by using, for example:

```
papiex -l
```

and you can display the full description of individual hardware performance counter events using, for example:

```
papiex -L <event_name>
```

To list the PAPI events, use the `papi_avail` command.

The System provides two types of hardware performance counters:

- Processor counters—Each processor is equipped with two internal performance counters that measure processor cycles, instructions issued, L1 cache operations, L2 cache operations, Translation Lookaside Buffer (TLB) misses, and so on.

  The `CPU_*` prefix identifies processor counter events. For a list of these events, see Table 15 *on page* 97. For a list of the PAPI preset events that map to the processor counter events, see Table 16 on page 98.

- Node counters—Each node is equipped with a block of 256 external counters (42 counters per processor) that are sampled in hardware, two at a time, for a period for 4096 cycles.

  The `SCB_*` prefix identifies node counter events. For a description of these events, see *Node Counter Events* on page 95

Using Papiex

First, measure all events in your application:

```
$ srun -p <partition> -n <ntasks> papiex -a <executable>
```

The `-a` option enables multiplexing, which instructs papiex to select and simultaneously sample all useful events available on the System. The samples are scaled for the entire run to provide an approximate value for each event.

Specify `-a` only on reasonably long runs of several seconds or more. On short runs, it may yield misleading data since it does statistical multiplexing on multiple events using the two available counters.

Command synopsis

```
papiex [args] <executable> [args]
```

Default mode:

```
papiex -U -e PAPI_TOT_CYC -e PAPI_FP_INS
```

Output

Unless instructed otherwise, papiex writes the output to `<executable>.papiex.<size>.<host>.<proc-id>.<instance>.txt`.

See the `papiex(1)` man page for complete details.

Example

For example, compile the `su3imp` test program (part of the SpecHPC benchmark suite) and run it under `papiex`:

```
$ srun -p sc1-comp -n 128 papiex -a su3imp.pathscale-O3

$ ls su3imp.pathscale-O3.papiex.128.sc1-m0n7.9826.1
job_summary.txt  task_108.txt  task_119.txt  task_15.txt
task_26.txt  task_37.txt  task_48.txt ... task_127.txt
```

You can see this is a 128-processor run. Job summary statistics are in the `su3imp.pathscale-O3.papiex.128.sc1-m0n7.9826.1/job_summary.txt` file.

Derived Metrics

Looking at some of the metrics from the *Derived Metrics* section of the `job_summary.txt` file (lines are numbered for ease of discussion):

```
1. MFLOPS Aggregate (wallclock) ................        12992.20
2. MFLOPS ......................................          105.07
3. IPC .........................................            0.34
4. Running Time % ..............................           99.87
5. Running Time in Domain % ....................           98.30
```

- Line 1 represents the total number of megaflops per second, computed with wallclock (real) time, of the application.

  All remaining lines in the summary represent averages across all tasks in the run.

- Line 2 represents the average number of megaflops as computed with the hardware cycle counter.
- Line 3 represents instructions per cycle; the peak for the processors is 2, for limited combinations of instructions.
- Line 4 is the percentage of time that the application actually ran without being blocked by I/O or preempted by another process.
- Line 5 is the percentage of time that the application ran in the requested domain without being blocked by I/O or preempted by another process.

Characterizing Mixed Instructions

```
6.  Memory Instructions % ........................          47.85
7.  Est.Int.Arith.Instructions % .................          10.23
8.  Non-FP Instructions % ........................          30.54
9.  FP Instructions % ............................          69.46
10. FP Arith. Instructions % .....................          26.39
11. FMA Instructions % ...........................          20.13
12. Branch Instructions % ........................           3.80
13. Load/Store Ratio .............................           3.42
```

- Line 6 represents the percentage of instructions of any type doing loads and stores.

Regarding lines 7 through 10: on the MIPs architecture, floating-point loads, stores, and conditionals are counted as floating-point instructions, so precise characterization of mixed instructions is somewhat difficult.

- Line 7 represents an estimate of integer arithmetic instructions. The estimate also includes moves to floating-point registers and synchronizing instructions, but these contribute minimally to it.

- Line 8 represents the percentage of everything that is not a floating-point instruction, which includes integer loads, stores, conditionals, moves, synchronizations, and all forms of integer arithmetic.
- Line 9 represents the percentage of floating-point instructions, which includes any instruction that involves the floating-point unit.
- Line 10 represents the percentage of instructions that actually compute a result (get you closer to your answer)
- Line 11 represents the percentage of all instructions that are variants of the highly efficient, fused multiply/add (madd) instruction.

  A madd instruction counts as one instruction, but two flops.
- Line 13 represents the balance of loads to stores.

This data alerts us to the overabundance of floating-point loads/stores in relation to the number arithmetic instructions. It also shows that most arithmetic instructions are multiply-adds (madd).

Compute Density Metrics

```
14. Flops per Load/Store ..........................          0.97
15. Flops per L1 D-cache Miss .....................         15.87
```

- Line 14 represents the number of floating-point operations (not instructions) for every load/store in the application.

  This number is also known as the *Computational Intensity*.
- Line 15 represents the number of floating-point operations (not instructions) for every miss of the L1 data cache.

The denominators of both metrics include integer instructions, so be sure to view both numbers in conjunction with the number of non floating-point instructions. For both metrics, higher values are better than lower ones.

Cache Metrics

The caches are one of the most important resources for achieving good performance.

```
16. L1 D-cache Misses Per Thousand Ins...........     29.32
17. L1 D-cache Hit % ............................     93.87
18. L1 I-cache Hit % ............................    100.00
19. Private L2 Cache Hit % ......................     40.82
20. Other L2 Cache Hit % ........................      1.64
21. L2 Miss % ...................................     58.21
```

```
22. L2 Bandwidth ...............................    221.94
23. Memory Bandwidth ...........................    258.37
```

- Line 19 and 20 represent the percentage of L2 data cache hits.

  Each of six processors in a node has a private L2 cache segment to which it can read and write. However, misses in one processor's private L2 cache segment, can be satisfied in any of the other five processor's L2 cache segment.

  For nonthreaded code, we expect this number to be near zero; however, the OS is free to share read-only pages of data, even among statically linked processes.

- Line 21 represents the total L2 miss rate of all the L2 data cache segments.

- Line 22 and 23 represent the bandwidth demands placed on the L2 cache and main memory, respectively.

From this data, we see that due to the high L2 miss rate of this code the demand on main memory is rather high.

TLB Statistics

The Table Lookaside Buffer (TLB) is an address cache for reference data and instructions. Missing in this cache is quite expensive, so you want the hit rates as high as possible.

```
24. L1 D-TLB Hit % ..............................           96.99
25. L1 I-TLB Hit % ..............................           99.98
26. L2 TLB Hit % ................................           99.67
27. Branch Misprediction % ......................           26.72
28. Dual Issue % ................................           49.07
```

- Line 27 represents the percentage of mispredicted branches.

- Line 28 represents the percentage of instructions that were issued (not retired) at the same time.

Line 27 shows a high percentage of mispredictions, but as we'll see later, mispredicted branches carry very little penalty. The rest of the results (lines 24-26, 28) are unremarkable.

Estimated Loss in Application Performance

Papiex estimates the amount of performance lost in you application code, based on a cost model shown at the end of the `job_summary.txt` file.

```
29. Est. L2 Private Hit Stall % .................            6.51
30. Est. L2 Other Hit Stall % ...................            0.57
```

```
31. Est. L2 Miss (private,other) Stall % .........          28.99
32. Total Est. Memory Stall % ....................          36.07
33. Est. D-TLB Miss Stall % ......................           0.98
34. Est. I-TLB Miss Stall % ......................           0.01
35. Est. TLB Trap Stall % ........................           0.06
36. Total Est. TLB Stall % .......................           1.05
37. Est. Mispred. Branch Stall % .................           0.35
38. Dependency (M-stage) Stall % .................           4.18
39. Total Measured Stall % .......................           5.58
40. Total Underestimated Stall % .................          37.47
41. Total Overestimated Stall % ..................          41.65
```

- Line 32 represents the percentage of time the application stalled on various levels of the cache hierarchy and main memory.
- Line 37 represents the percentage contributed by the `CPU_MSTALL event`, which counts cycles that the processor stalls on certain types of dependencies, including some, but not all, cache misses.

  ☼ Note that back-to-back misses count in this metric, but a miss followed by a dependent instruction may not.
- Line 39 represents the total amount of time spent in stalls for which papiex could actually count (not estimate), including TLB misses, branch mispredictions, and dependency stalls.
- Line 40 represents the total time lost due to estimated stalls on memory, TLB, and branch mispredictions. It does not include the dependency stall metric included in Line 38.

  This value is the lower bound due to the overlap of the dependency stall metric (`CPU_MSTALL`) and papiex's estimate for memory stalls.
- Line 41 represents the total time lost due to estimated memory stalls and on dependency stalls.

  This value is an overestimate because it includes both the memory stall time and the dependency stall metric.

From Line 32, we see the application lost 36% of cycles to various memory and cache stalls. Furthermore, the difference between dependency stalls and memory stalls indicates that most misses were not back-to-back, which always count as dependency stalls. It's likely that the processor more often stalled waiting on the cache to return a data item.

Estimated Ideal Time

Papiex can estimate how fast the application should run if all stalls, identified from the previous results, are removed.

Here are two estimates, each based on a different model of the instruction mix.

```
42. Actual/Ideal Cyc (max. dual) .................            3.80
43. Ideal IPC (max. dual) ........................            1.29
44. Ideal MFLOPS (max. dual) .....................          399.02
45. Actual/Ideal Cyc (cur. dual) .................            3.90
46. Ideal IPC (cur. dual) ........................            1.33
47. Ideal MFLOPS (cur. dual) .....................          409.33
```

- Lines 42 through 44 are based on a reordering of the instruction mix.
- Lines 45 through 47 are based on the current ordering of the instruction mix.

The difference between the estimates of the two models is typically very small. Inhouse testing has shown that these estimates are highly reliable when tuning small regions of code. For these results, we see that if this application ran free of all stalls, it would run about 400 Mflops (4× times) faster.

MPI, I/O, and Threaded Functions

Papiex knows about some MPI, I/O, and threaded functions and can account for time lost there. Not all functions are instrumented, but most of the common cases are covered:

```
48. MPI cycles % ................................             1.64
49. MPI Sync cycles % ...........................             0.00
50. I/O cycles % ................................             0.00
51. Thr Sync cycles % ...........................             0.00
```

From this data, we see that despite being a 128-processor run, the application lost only 1.64 percent of time due to MPI.

Line 49 shows that the application spent no time waiting on MPI completions, or otherwise in a barrier, and no time doing I/O or thread synchronization.

All metrics examined up to this point are contained in both the job_summary.txt file, the task_*.txt files, and the thread_*.txt files.

Task Memory Usage

This data is gathered when you specify `-a` to papiex. It is included in only the `task_*.txt` and `thread_*.txt` output files.

```
Mem virtual peak KB..........................           143296
Mem resident peak KB.........................           125824
Mem text KB...................................              192
Mem library KB................................             7680
Mem heap KB...................................           117312
Mem stack KB..................................             1344
Mem shared KB.................................                5
Mem locked KB.................................                0
```

MEM resident peak KB is the only important measurement in this run. It represents the total physical memory touched by the task.

You can download versions of the su3_imp test case at http://www.nersc.gov/projects/SDSA/software/?benchmark=MILC&action=general.

Using Calipers to Measure Specific Code Regions

You can also monitor performance counts and derived metrics on specific regions of your code by enclosing the regions within the papiex calipers, papiex_start() and papiex_stop(), then compiling and linking your program with the papiex library.

In the output file, papiex indents the resulting counts and derived metrics for each instrumented region under the label you pass to the papiex_start() function at the start of the region.

For example, instrument the caliper.c program:

```
#include <stdio.h>
#include <papiex.h>
void flops(int count)
{
  int retval;
   double a = 1.001;
   int i;
   papiex_start(1, "Flops-loop");
   for (i=0; i< count;i++) {
     a = a * 1.10;
     if (a > 10000.0) {
       a = 1.001;
     }
   }
   papiex_stop(1);
   printf("a=%f\n", a);
}
int main(int argc, char **argv)
{
   int c = 200000000;
   flops(c);
   return 0;
}
```

Run the executable under papiex and measure two events:

```
$ papiex -e CPU_FPARITH -e CPU_CYCLES caliper
Derived Metrics:
```

```
CPU Utilization .............................            1.00
MFLIPS ......................................          136.43
% I/O Cycles ................................            0.00
Cycles ......................................     2.20654e+09
FP Instructions .............................     6.02062e+08
    Flops-loop
    Derived Metrics:
    CPU Utilization .........................            1.00
    MFLIPS ..................................          136.44
    Cycles ..................................     2.20637e+09 [100.0%]
    FP Instructions .........................     6.02062e+08 [100.0%]
Real usecs ..................................        4.43e+06
Real cycles .................................       2.215e+09
Proc usecs ..................................        4.44e+06
Proc cycles .................................       2.215e+09
I/O cycles ..................................               0
PAPI_TOT_CYC ................................     2.20654e+09
PAPI_FP_INS .................................     6.02062e+08
    Flops-loop
    Executions ..............................               1
    Real cycles .............................      2.2145e+09
    PAPI_TOT_CYC ............................     2.20637e+09 [100.0%]
    PAPI_FP_INS .............................     6.02062e+08 [100.0%]
Event descriptions:
PAPI_TOT_CYC                  : Total cycles
PAPI_FP_INS                   : Floating point instructions
Derived event descriptions:
CPU Utilization               : Virtual cycles / Real cycles
```

You can use the `-a` option to sample numerous metrics within instrumented regions.

Using Mpipex

To measure the time your MPI applications spends communicating, run `mpipex` in default mode this way:

```
$ srun -p <partition> -n <ntasks> mpipex <executable>
```

Command synopsis

```
mpipex [options] <executable> [args]
```

Output

Unless instructed otherwise, `mpipex` writes the output to `<executable>.mpipex.<size>.<host>.<proc-id>.<instance>`.

The output provides these types of information:

- Environment
- MPI time per task

- Callsite listing per MPI call
- Aggregate times of top twenty callsites
- Callsite statistics per function

See the mpipex man page for complete details.

Working with Static Executables

If you're working with static executables, you can relink using `-lmpiP` to generate the same output that `mpipex` does. For example,

```
$ pathcc myapp.c -lmpiP -lbfd -lscmpi
```

Then, simply run the executable without `mpipex`:

```
$ srun -p sc1 -n 4 <myapp>
```

The output is written to the default location:
`myapp.mpipex.<size>.<host>.<proc-id>.<instance>.txt`.

Example

For example, compile the `su3imp` test program (part of the SpecHPC benchmark suite) and run it under mpiex:

```
$ srun -p sc1-comp -n 128 mpipex su3imp
```

```
...
mpiP:
mpiP: Storing mpiP output in [./su3imp.mpipex.128.sc1-m0n0.scsystem.7213.1.txt].
mpiP:

$ less su3imp.mpipex.128.sc1-m0n0.7213.1.txt
```

MPI Time

```
---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task  AppTime  MPITime   MPI%
   0       527      19.7     3.75
   1       527      19.8     3.75
   2       527      19.8     3.76
   3       527      18.3     3.46
   4       527      19.6     3.71
   5       527        20     3.79
   6       527      19.6     3.72
   7       527      19.7     3.74
```

This data shows how much time each task spent in MPI and in the application code. Load balances of either compute or communicate are easily visible here.

Callsites Identification

```
---------------------------------------------------------------------------
@--- Callsites: 25 --------------------------------------------------------
---------------------------------------------------------------------------
 ID Lev File/Address        Line  Parent_Funct                 MPI_Call
  1  0  com_mpi.c           2709  cleanup_general_gather       Wait
  2  0  com_mpi.c           1594  do_gather                    Isend
  3  0  setup.c             211   readin                       Bcast
  4  0  com_mpi.c           480   g_complexsum                 Allreduce
```

This data shows how mpipex uses identifiers to map the output to the callsite, the location of the MPI call in the source code.

In this output example, the function `do_gather` on line 1594 in file `com_mpi.c` called `MPI_Isend()`.

Aggregate Time of MPI Calls

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Barrier                18   1.64e+06    2.46   60.11    0.09
Wait                   21   6.51e+05    0.98   23.93    0.66
Wait                    3    2.5e+05    0.38    9.20    0.41
Allreduce              10   5.15e+04    0.08    1.89    0.31
```

This data shows the most expensive MPI calls and their variance across the nodes the job used.

From this data, the most expensive MPI call is a barrier operation, which consumed 60% of the application's MPI time. It's location is callsite 18, which is included in the Callsites Identification table, but not shown in the example.

`COV` stands for coefficient of variance, and numbers near zero indicate very good balance.

Aggregate Size of Sent Messages

```
-----------------------------------------------------------------------
@--- Aggregate Sent Message Size (top twenty, descending, bytes) ----------
-----------------------------------------------------------------------
Call                Site      Count      Total       Avrg  Sent%
Isend                 12    1992060   1.91e+11   9.59e+04  94.00
Isend                  2      16632   1.22e+10   7.33e+05   6.00
Allreduce             10      62496      5e+05          8   0.00
```

This data shows the amount of data exchanged with each MPI call, sorted by total number of bytes exchanged.

As with all MPI implementations, longer messages are more efficient than shorter ones.

Callsite Time Statistics

```
---------------------------------------------------------------------------
@--- Callsite Time statistics (all, milliseconds): 3030 -------------------
---------------------------------------------------------------------------
Name              Site Rank  Count      Max     Mean      Min   App%   MPI%
Allreduce            4    0      8     34.8     4.85    0.174   0.01   0.20
Allreduce            4    1      8     19.4     3.57    0.115   0.01   0.14
Allreduce            4    2      8     19.4     2.94     0.11   0.00   0.12
```

This data shows for each callsite, the time statistics of each MPI call sorted by rank. This section can grow quite large, but it is useful for localizing poor synchronization.

Callsite Message Sent Statistics

```
---------------------------------------------------------------------------
@--- Callsite Message Sent statistics (all, sent bytes) -------------------
---------------------------------------------------------------------------
Name              Site Rank   Count       Max      Mean       Min       Sum
Allreduce            4    0       8         4         4         4        32
Allreduce            4    1       8         4         4         4        32
Allreduce            4    2       8         4         4         4        32
Allreduce            4    3       8         4         4         4        32
Allreduce            4    4       8         4         4         4        32
```

This data shows the amount of data exchanged at each callsite.

Using HPCex

To profile your application, run `hpcex` this way:

```
$ srun -p <partition> -n <ntasks> hpcex -e <event> \
                                       <executable> [args]
```

Command synopsis

- `hpcex -e <event> <executable> [args]`
- `hpcstruct <executable> > <executable>.psxml`
- `hpcproftt -S <executable>.psxml <executable>.hpcex.*`
- `hpcviewer <experiment-db/experiment.xml>`

Output

Unless instructed otherwise, hpcex writes the output to `<executable>.hpcex.<size>.<host>.<proc-id>.<instance>`.

Typical Workflow

1. Run `hpcex` to create a statistical profile (binary) of a PAPI or native event (for listings see page 63). (For usage details, see page 61.)

2. Optionally, run `hpcstruct` to recover static program structure. (For usage details, see page 61.)

   The toolkit uses structure information to identify loops and inlining and to correlate at the instruction level. Without it, correlation is naive and line-level only.

3. Interpret and visualize the data.

   a. Run `hpcproftt` to visualize flat textual data. For example:

   ```
   $ hpcproftt --src-all --metric=sum-only -l <srcpath>\
               -S <executable>.psxml <executable>.hpcex>*
   ```

   b. Alternatively, run `hpcprof-flat` to visualize flat or callpath data, which you can view using `hpcviewer`. For example:

   ```
   $ hpcprof-flat -l <srcpath> -S <executable>.psxml
                                     <executable>.hpcex.*
   ```

4. Run `hpcviewer`, a Java-based GUI, to display the output data from `hpcprof-flat`. (For usage details, see page 61.)

   `Hpcviewer` requires an x86 workstation running `X` and the Java runtime.

Serial Run Example

To run natively on the System, prepend to the commands:

```
$ srun -p <partition> -N 1
```

1. Compile the simple `float.c` program (see *hpcex float example* on page 99 for the complete listing) with these optimizations:

   ```
   $ pathcc -g -O2 float.c -o float
   ```

   Compile using the -g flag to generate symbol information that enables the hpc* tools to provide file names, line numbers and function names.

2. Profile `CPU_CYCLES`, with a lower sampling interval to get more samples in short runs.

   ```
   $ hpcex -e CPU_CYCLES:9999 ./float
   [snipped]
   $ ls float*
   ```

```
float float.c float.hpcex.1.sc1-m3n6.13419.0x0.dat
```

By default, `hpcex` runs in user mode (`-U`). If you want to capture floating-point exceptions, add `-K` (kernel mode) and `-I` (interrupt mode) to the `hpcex` command line.

3. Generate the program structure to get better source correlation.

```
$ hpcstruct float > float.psxml
$ ls float*
float float.c float.hpcex.1.sc1-m3n6.13419.0x0.dat \
                                             float.psxml
```

4. Generate flat textual data to get all summaries and annotated source code.

```
$ hpcproftt --src=all -S float.psxml \
                    float.hpcex.1.sc1-m3n6.13419.0x0.dat
```

5. Examine the output in detail.

   The following output snippet states that one sample contains 9999 events and that there were 1462386 samples in the run:

```
Metric definitions. column: name (nice-name) [units]
{details}:
   1: CPU_CYCLES [events] {CPU Cycles:9999 ev/smpl}
Program summary (row 1: sample count for raw metrics):
 1462386
```

   If the sample count is too low, the results will be statistically ambiguous, and may significantly increase profiling overhead. To increase the sample count, increase the sampling interval (in this case `CPU_CYCLES` to something like `CPU_CYCLES:999999`).

   The next output snippet confirms that only one software module, `float`, took all of the cycles:

```
 Load module summary:
  100.00%  /net/home/user/test/float/float
 2.1e-04%  /lib64/libc-2.5.so
```

   So, in the simple `float.c` example program, a single function—`float`—and a single loop within it, took all of the cycles.

```
Procedure summary:
 100.00%  [/net/home/user/test/float/float]</net/home/user/test/src/float.c>flops
1.4e-04%  [/lib64/libc-2.5.so]<~~~<unknown-file>~~~>__printf_fp
[snipped]
Loop summary (dependent on structure information):
 100.00%  [/net/home/user/test/float/float]</net/home/user/test/src/ float.c>7-9
```

We can see that the floating-point comparison was expensive, most likely because it depended on the multiply operation:

```
Annotated file (statement/line level):
[/net/home/user/test/float/float]/net/home/user/test/src/float.c
    6             int i;
    7   0.12%     for (i=0; i< count;i++) {
    8               a = a * 1.10;
    9  99.88%       if (a > 10000.0)
   10                 a = 1.1;
```

6. Use `hpcprof-flat`, instead of `hpcproftt` (Step 4), to create a flat database.

```
$ hpcprof-flat -S float.psxml float.hpcex.1.sc1-m3n6.13419.0x0.dat
[snipped]
$ ls exper*
experiment-db:
config.xml  experiment.xml
```

7. View the results in `hpcviewer` on a workstation running `X` and the Java Runtime.

```
$ hpcviewer experiment-db/experiment.xml
```

You can also get assembly code listings for the program. Here's how:

8. Repeat steps 2 and 3 to profile the program, and generate the program structure for better source correlations.

9. Then run the `.dat` output file from `hpcstruct` under `hpcproftt` like this:

```
$ hpcproftt --obj=s -S float.psxml float.hpcex.1.sc1-m3n6.13419.0x0.dat
[snipped]
Procedure: flops (flops)
Metric summary for procedure (percents relative to load module):
   1462380
   100.00%
/net/home/user/test/src/float.c:9
0x120000a84:          movf.d    $f0,$f9,$fcc2
0x120000a88:          sra       v1,a0,0x2
0x120000a8c:          beqz      v1,0x120000ad8
0x120000a90:   11.36% mul.d     $f13,$f0,$f2
0x120000a94:    0.63% c.lt.d    $fcc6,$f6,$f13
[snipped]
```

MPI Run Example

To run applications natively on the System, prepend to the commands:

```
$ srun -p <partition> -n <ntasks>
```

1. Compile the `pi.c` program (see *hpcex pi example* on page 100 for the complete listing) with these optimizations:

   ```
   $ mpicc -g -O2 pi.c -o pi
   ```

   ☼ Compile using the -g flag to generate symbol information that enables the hpc* tools to provide file names, line numbers and function names.

2. Create the profiles.

   ```
   $ srun -p sc1-comp -n 4 hpcex -e CPU_CYCLES ./pi
   [snipped]
   $ ls pi.hpcex.*
   pi.hpcex.4.sc1-m0n5.5009.0x0.dat pi.hpcex.4. \
                                      sc1-m0n5.5011.0x0.dat
   pi.hpcex.4.sc1-m0n5.5010.0x0.dat pi.hpcex.4. \
                                      sc1-m0n5.5012.0x0.dat
   ```

3. Generate the program structure to get better source correlation.

   ```
   $ hpcstruct pi > pi.psxml
   ```

4. Generate flat output data; pass all profile files as arguments to `hpcproftt`.

   ```
   $ hpcproftt --src=all -S pi.psxml pi.hpcex.*
   ```

5. Examine the output.

   Because the number of output columns increases with the number of tasks, instead of showing one entry per task, show aggregate metrics using `hpcproftt`’s `-M sum-only` argument:

   ```
    $ hpcproftt --src=all -S pi.psxml -M sum-only pi.hpcex.*
    [snipped]
     999999  122.47%       0 2999997
      [/usr/lib64/libscmpi_optimized.so]~~~<unknown-file>~~~
      749999   57.74%       0  999999 [/net/home/usr/test/
      examples/pi/pi]/net/home/usr/test/src/pi.c
   [snipped]
   ```

   The columns are, in order, `mean`, `rdev`, `min`, and `max`. `rdev` is the relative standard deviation (stddev/mean).

See the `hpcex`, `hpcstruct`, `hpcproftt`, `hpcprof-flat`, and `hpcviewer` man pages for complete details.

Using TAU

Using a TAU compiler script, compile your source code to instrument it, then run the executable under `tauex` to generate profile and trace data.

```
$ srun -p <partition> -n <ntasks> taucc <myapp>.c -o <myapp>
$ srun -p <partition> -n <ntasks> taucxx <myapp>.C -o <myapp>
$ srun -p <partition> -n <ntasks> tauf90 <myapp>.F -o <myapp>
```

Compiling and Instrumenting Source Code

The TAU compiler scripts—`taucc`, `taucxx`, or `tauf90`— default to using the corresponding PathScale compiler. (The TAU cross-compilers are `sctaucc`, `sctaucxx`, and `sctauf90`.) The TAU compiler scripts specify the libraries and TAU options to use for compiling and instrumenting the source code. The default instrumentation configuration is `profile`, `MPI`, and `pthread`.

Table 7. Example TAU auto-instrumentation scenarios

To autoinstrument	Do this

Everything in a single-file test program for profiling	`$ taucc mpi_test.c -o mpi_test`
	`$ taucxx mpi_test.C -o mpi_test`
	`$ tauf90 mpi_test.F -o mpi_test`

Everything in a single-file hybrid MPI/OpenMP test program for profiling	`$ taucc -tau:openmp, mpi mpi-omp-test.c -o mpi-omp-test`
	`$ taucxx -tau:openmp, mpi mpi-omp-test.C -o mpi-omp-test`
	`$ taucf90 -tau:openmp, mpi mpi-omp-test.F -o mpi-omp-test`

Everything in a single-file of a multiple source-file C program	`$ pathcc -c mpi-test.c`
	`$ taucc -c mpi-test-two.c`
	`$ taucc mpi-test.o mpi-test-two.o -o mpi-test`

Generate Vampir .OFT traces in a single-file combined C++/OpenMP program	`$ taucxx -tau:openmp, vampirtrace omp.cpp -o omp`

Only C++ and OpenMP codes require special instrumentation. For those codes, you must supply `-tau:` flags on the command line to specify what kind of code the compiler is compiling. This means that C++ and OpenMP codes cannot be retargeted at runtime using `tauex`, whereas all other configurations can.

Table 8. Some useful TAU options

To	Use this option[1]
Display TAU options	`-tau:help`
Show me what the command does, but don't do it	`-tau:showme`
Enable verbose mode	`-tau:verbose`

Specify code type and instrumentation	`-tau:<option>`
	`[openmp, pthread, mpi, profile, vampirtrace, epilog, trace, callpath, disable]`

Table 8. Some useful TAU options (Cont'd)

To	Use this option[1]

Throttle instrumentation	• Enable and use the default throttle threshold—disable profiling when function has executed > 100000 times with an inclusive time per call < 10 µsecs:
	`export TAU_THROTTLE <any_int>`
	• Also set these environment variables to change the default threshold:
	`export TAU_THROTTLE_NUMCALLS <value>`
	`export TAU_THROTTLE_PERCALL <value>`

Selectively instrument source code	`-tau:options="-optPreProcess -optTauSelectFile=<file>"`
	The `-optPreProcess` option directs the compiler to preprocess the source code before parsing it.
	The Select File contains a list of files, functions, loops, etc. to include or exclude in the instrumentation:
	`BEGIN_FILE_INCLUDE_LIST`
	`<file1>`
	`<file2>`
	`<file3>`
	`END_FILE_INCLUDE_LIST`
	`BEGIN_INCLUDE_LIST  # can _EXCLUDE_ functions too`
	`<func_name1>`
	`<func_name2>`
	`END_INCLUDE_LIST`

Compile a 32-bit application	Add to the compiler command line: `-tau_makefile=/usr/share/TAU/32/Makefile.tau-multiplecounters-pathcc-mpi-papi-pdt`

[1] See the TAU man pages for a complete list of the options.

TAU Facts to Consider

The TAU compilers use automatic compiler instrumentation and function call interposition to measure applications. Some things to note:

- Running your executable under `tauex`, without recompiling it with a TAU script, produces profiles of MPI activity vs application activity. The results can provide you useful information.
- Unless you explicitly exclude functions or throttle instrumentation, the Tau compiler instruments every function in a source file (see Table 8).
- You need not compile every source file in your application with the TAU compiler.
- If you use the TAU compiler on every source file in your application, you can expect a mild dilation at runtime.

- If you run an instrumented executable without `tauex`, the Tau library automatically uses the default options: `-U -T MPI, PTHREAD, PROFILE, -e P_WALL_CLOCK_TIME`[*].

Using Tauex

To generate profile and trace data, run tauex on TAU instrumented executables this way:

```
$ srun -p <partition> -n <ntasks> tauex [options] <executable> [args]
```

Command synopsis

```
tauex [options] <executable> [args]
```

Input

Table 9 lists the tauex command line options.

Table 9. Tauex command options

To	Use this option[1]
Enable debugging output, use repeatedly for more output	`-d`
Display help	`-h`
Display host information	`-i`
Dump the shell environment variables and exit	`-s`
User mode counts	`-U`
Kernel mode counts	`-K`
Supervisor mode counts	`-S`
Interrupt mode counts	`-I`
List events	`-l`
Describe the event	`-L <event>`
Specify PAPI preset or native event	`-e <event>`
Specify TAU options	`-T <option1, option2, …>`[2] `[MPI, OPENMP, PTHREAD, SERIAL, PROFILE, CALLPATH, TRACE, VAMPIRTRACE, EPILOG, DIASABLE]`
Enable debug/verbose mode	`-v`
Specify TAU library directly	`-XrunTAU-<options>`

[1] See the tauex man pages for a complete list of options and descriptions.

[2] Some options are mutually exclusive.

[*] Implemented using the fast PAPI timer, a 250 Mhz, 64-bit cycle timer.

Output

By default, `tauex` outputs the Tau trace data to the `./<executable>.tau.<slurm_job_id>` directory. Output filenames are: `a.0.def.z a.1.events.z a.2.events.z` … `a.#.events.z  a.otf`.

Profile data is output to one or more directories named: `./<executable>.tau.<slurm_job_id>/MULTI__<metric_name>`.

Viewing Results

- TAU profile data

  Use either the `pprof` text-based viewer or the `paraprof` Java GUI viewer. (To use `paraprof`, you must download, install, and run it on a workstation that has Java installed.) For details, see the `pprof` and `paraprof` man pages.

- Vampir trace data

  To import Tau trace files into Vampir to analyze and display the results, first compile the application with the appropriate Tau compiler script, then run `tauex` on the instrumented executable using the `-T VAMPIRTRACE` option to output `.OTF` trace files.

  See the tau and tauex man pages for complete details.

Using Vampirtrace

☼ To run the Analysis Server (`vngd`) and the Visualization Client (`vng`), you need to have purchased and installed licenses. Contact your System Administrator to find out whether you already have or need to purchase licenses.

Using a Vampirtrace compiler script, compile your MPI source code to instrument and link it with the Vampirtrace library. Run the resulting executable, then view the resulting traces using the Vampir GUI viewer, `vng`.

```
$ make CC=vtcc F77=vtf90 FC=vtf90 CXX=vtcxx …
$ srun -p <partition> -n <ntasks> <vt_executable> [args]
```

Start the `vngd` analysis server on the nodes, then display the `.oft` traces in the Vampir `vng` GUI on the workstation (see *Visualizing Results* on page 86).

Output

By default, Vampirtrace writes trace output to the current working directory. The output files are named:

```
<executable>.0.def.z  <executable>.1.events.z
executable>.2.events.z … <executable>.nth.events.z
<executble>.otf
```

Compiling and Instrumenting Source Code

The Vampirtrace compiler scripts—`vtcc`, vtcxx, and `vtf90`—instruct the underlying PathScale compiler where to find all required Vampirtrace libraries (static) and configuration files and to use the compiler's `-finstrument-functions` option to instrument the source code. Vampirtrace then records all entries and exits from functions, all calls made to the MPI library, and all point-to-point and collective communication operations.

Table 10 shows examples of how to use the Vampirtrace compiler scripts to autoinstrument source code.

Table 10. Example Vampirtrace autoinstrumentation scenarios[1]

To autoinstrument	**Do this**
All functions and MPI calls in a single-file program	`vtcc mpi_test.c -o mpi_test`
All functions in a single-file sequential program	`vtcc -vt:seq seq_test.c -o seq_test`
All functions in a single-file OpenMP program	`vtcc -vt:omp omp_test.c -o omp_test`
All functions in a single-file hybrid OpenMP/MPI program	`vtcc -vt:hyb mpi_omp_test.c -o mpi_omp_test`

Only MPI calls to contrast MPI vs application time	`pathcc -c mpi_test.c`
	`vtcc mpi_test.o -o mpi_test`

One source in a multisource application	`pathcc -c mpi_test.c`
	`vtcc -c mpi_test_two.c`
	`vtcc mpi_test.o mpi_test_two.o -o mpi_test`

[1] For C++ and Fortran codes, substitute the appropriate compiler script.

Table 11 shows some useful vampirtrace options:

Table 11. Sampling of vampirtrace `-vt:` options

To	**Use this option**
Display a list of the vampirtrace options	`-vt:help`
Show me what this command does, but don't execute it	`-vt:showme`
Enable debug/verbose mode	`-vt:verbose`

Specify code type and instrumentation	`-vt:<option>`
	`[seq, omp, mpi, hyb]`

Change the underlying compiler	`-vt:<cc|cxx> <gcc|g++>`
Disable auto-instrumentation to manually instrument source code	`-vt:inst manual`

Vampirtrace Facts to Consider

The Vampirtrace compilers use automatic instrumentation and function call interposition to measure your application. Some things to note:

- You must compile and instrument your source code natively on the System.
- You need not compile every source file in your application using the Vampirtrace compiler scripts.
- Every function in a source file compiled with a Vampirtrace compiler is instrumented to generate a trace record.
- If you link but don't compile your application using a Vampirtrace compiler script, Vampirtrace produces traces of MPI activity only.
- If you compile every source file in your application using a Vampirtrace compiler, you can expect a significant dilation in your application's runtime, depending on your I/O configuration.

Output Data Buffering

By default, Vampirtrace generates and temporarily buffers the total number of trace records needed before flushing them all to disk. This behavior can result in huge trace files. You can change the default behavior by setting one or both environment variables (Table 12) before running your application:

Table 12. Buffer control environment variables

To	Set this environment variable
Limit or increase the number of times to flush the trace buffer	`VT_MAX_FLUSHES=<value>`
Set the size of the trace buffer	`VT_BUFFER_SIZE=<value>`

For example:

```
$ export VT_MAX_FLUSHES=1
$ export VT_BUFFER_SIZE=1
$ srun -p sc1-comp -n 2 ./pi
[1]VampirTrace: Maximum number of buffer flushed reached
   (VT_MAX_FLUSHES=1)
[1]VampirTrace: tracing turned off permanently
[0]VampirTrace: Maximum number of buffer flushed reached
   (VT_MAX_FLUSHES=1)
[0]VampirTrace: tracing turned off permanently
pi is approximately 3.1415926535896905, error is
0.0000000000001026
```

Manually Instrumenting Source Code

To collect more detailed information about an application, such as user-defined events or recording the location of subroutine calls in the source code, you must include the appropriate header file in your program and manually instrument the source code with calls to the Vampirtrace API. For example:

- For C or C++ applications

```
#include "vt_user.h"
VT_USER_START("name");
...
VT_USER_END("name");
```

- For Fortran applications

```
#include "vt_user.inc"
VT_USER_START('name')
...
VT_USER_END('name')
```

- For OpenMP profiling directives (Pathscale compilers only)

```
!POMP$ INST INIT   # must be first executable statement of
                   # the main program

# Use INST BEGIN and INST END to mark any user-defined
# sequences; if the block has multiple exit points, use
# INST ALTEND on all but the last one.

!POMP$ INST BEGIN(name)
...
  [ !POMP$ INST ALTEND(name) ]
...
!POMP$ INST END(name)
```

Preprocess Fortran source files, and then for any code, include `-vt:inst manual` on the Vampirtrace compiler's command line to prevent the Vampir compilers from autoinstrumenting the source code; for example:

```
$ vtcc -vt:inst manual myapp.c -o myapp
```

To turn off traces from these calls (and decrease profiling overhead), you need only relink your application's object file with the dummy library `-lVTnull`; for example, on the System:

```
sc1-mono:~$ vtcc myapp.o -lVTnull -lscm -lm -o myapp
```

Using the Hardware Performance Counters

By default, Vampirtrace collects timing information using `P_WALLCLOCK_TIME`, implemented using the fast PAPI timer, a 250 Mhz, 64-bit cycle timer.

Vampirtrace can also use the System's hardware performance counters using PAPI. To do so, set the environment variable `VT_METRICS` to the counters you want sampled:

```
$ export VT_METRICS="CPU_MSTALL:CPU_FPMADD"
```

For a list of the hardware performance counters, run `papiex -l`.

See the Vampirtrace man pages and user documentation for complete details.

Visualizing Results

Vampir enables you to visualize and debug MPI issues that arise during program execution. The two major components of Vampir run on different machines.

- The Analysis Server (`vngd`), which analyzes the trace data, runs on the System.
- The Visualization Client (`vng`), which displays the results, runs on the cross-development workstation.

1. On the System, start a `vngd` parallel server job. For example:

   ```
   sc1-mono:~$ srun -p sc1-comp1 -n 180 vngd
   ```

   ☼ For big traces, we recommend that you use a Lustre file system and an appropriate number of processors.

   The `vngd` Analysis Server returns the address and port of the node on which it is running. For example:

   ```
   Server listens on: sc1-m0n0.scsystem:3000
   ```

   Where `sc1-m0n0.scsystem` is the address of the node.

2. On the cross-development workstation, start the Visualization Client by typing `vng` on the command line.

3. Connect to the Analysis Server.

   On the toolbar, click **File>Server**. In the Server field, enter the address of the node returned by the Analysis Server in Step1, then click **OK**.

   (In the example, the address returned was `sc1-m0n0.scsytem`.)

4. When connected to the server, on the toolbar, click **File>Open Tracefile**, then select from the list the`.otf` trace file you want to view.

   You can select how you want the trace data displayed. In profile summary mode, you can group flat profile charts much like `gprof` does. You select whether to profile the entire program run or a specified time interval. In event time line mode, you can delve deeper into the inner working of the application, visualizing the behavior of individual processes over time.

5. When you are done, be sure to kill the vngd trace daemon using scancel or ^C to free up the resources.

See the Vampir man pages and user documentation for complete details.

Using GPTL

The General Purpose Timer Library provides an API to simplify the gathering of timing statistics for C and Fortran codes. The API also provides an optional interface to PAPI counters. The gptlex tool (see page 90) can provides function-level profiling without modifying the source code.

The Fortran GPTL entry point names are identical to their C counterparts and take identical arguments, except that, in Fortran, names are case-insensitive.

The GPTL library allows you to instrument codes with an unlimited number of user-named *timers*. A call to GPTLstart() starts a given timer, and a call to GPTLstop() stops it. Within the same code, you can start and stop a timer an arbitrary number of times.

General Calling Sequence

The general sequence for calling the GPTL library is:

- Include the appropriate gptl.* header file.
- Call GPTLsetoption() as often as needed to set GPTL options, such as specifying the output format and enabling PAPI counters.
- Call GPTLinitialize() to initialize the GPTL library.
- Enclose regions of source code that you want to time within calls to GPTLstart(region_name) and GPTLstop(region_name).

  The argument region_name is a user-defined string.
- Call GPTLpr(int) to specify the extension for the output file, timing.*int*, where *int* is an arbitrary integer. For MPI codes, it is convenient to use the MPI rank.

Thread Safety

The library is thread-safe, which means that calls to GPTLstart() and GPTLstop() can safely occur within threaded regions. In that case, the results for each thread are printed separately.

Default Timer Output

With a single call to `GPTLpr` (see code example on page 89), you can dump the current state of all timers to an output file. The default output for each timer includes:

- Number of calls
- Wall-clock time
- Maximum time
- Minimum time
- Estimated overhead incurred by the underlying timing routine (UTR) that the library employed

  The default UTR is `papi_get_real_usec`, but you can change this setting at run time.

The library also supports an arbitrary number of *nesting* levels. In this case, the names of timers that are nested inside of other timers are indented in the output file. This formatting makes it easy to see which timers are subsumed by other timers.

Accessing PAPI Counters

The library has optional access to the `PAPI` performance counter library. If one or more `PAPI` counters are enabled during the run, when `GPTLpr()` is called, the `PAPI` counter values are printed for each timer, with the other timing statistics.

Example

The `ugex` program, an instrumented code example, is a threaded Fortran program with OpenMP enabled (using the PathScale compiler's `-mp` flag). The program was run on two threads. The default estimated overhead statistic has been disabled by a simple call to the gptl library:

```
ret = gptlsetoption (gptloverhead, 0)
```

Example GPTL instrumented code:

```
      program ugex
      implicit none

#include "gptl.inc"
#include "f90papi.h"

      integer, parameter :: nompiter = 128 ! iteration count for threaded loop
      integer, parameter :: ny = 9         ! iteration count for middle loop
      integer, parameter :: nx = 1000000   ! iteration count for inner loop

      integer :: i, j, iter        ! loop indices
      integer :: ret               ! return code
      integer*8 :: papicounters(3) ! PAPI counter values

      real*8 :: sums(nompiter)     ! summation array
      if (gptlsetoption (gptlverbose, 0) < 0) call exit (1)       ! turn off verbosity
      if (gptlsetoption (gptlabort_on_error, 1) < 0) call exit(2) ! abort on error

      ret = gptlsetoption (PAPI_FP_INS, 1)     ! count FP instructions
      ret = gptlsetoption (PAPI_TOT_INS, 1)    ! count total instructions
      ret = gptlsetoption (gptloverhead, 0)    ! don't print overhead stats
      ret = gptlsetoption (gptlnarrowprint, 1) ! print fewer sig figs

      ret = gptlinitialize ()   ! initialize GPTL
      ret = gptlstart ('total') ! start a timer for the entire program


      ret = gptlstart ('init')  ! start a timer
      do i=1,nompiter
         sums(i) = 0.
      end do
      ret = gptlstop ('init')   ! stop a timer
! Invoke a threaded loop, and gather timing info

!$OMP PARALLEL DO PRIVATE (i, j, iter, ret)

      do iter=1,nompiter
         ret = gptlstart ('Jloop')
         do j=1,ny
            ret = gptlstart ('Iloop1')
            do i=1,nx
               sums(iter) = sums(iter) + 0.0001*i
            end do
            ret = gptlstop ('Iloop1')

            ret = gptlstart ('Iloop2')
            do i=1,nx
               sums(iter) = sums(iter) + i
            end do
            ret = gptlstop ('Iloop2')
         end do
         ret = gptlstop ('Jloop') ! stop timer
      end do

      ret = gptlstop ('total')  ! stop the timer for the entire program

                           ! Retrieve the PAPI counters for timer 'total' and print them

      ret = gptlquerycounters ('total', -1, papicounters)
      write(6,*)'total PAPI_FP_INS=    ', papicounters(1)
      write(6,*)'total PAPI_TOT_CYC=   ', papicounters(2)

      ret = gptlpr (0)          ! print the timing results to timing.0
      ret = gptlfinalize ()     ! clean up

      stop 0
      end program ugex
```

Example GPTL output from the call to gptlpr:

```
PAPI event multiplexing was OFF
PAPI events enabled:
  Floating point instructions executed
  Total instructions executed
```

```
Underlying timing routine was PAPI_get_real_usec.
Per-call utr overhead est: 1.47e-06 sec.

If overhead stats are printed, roughly half the estimated number is
embedded in the wallclock (and/or PAPI counter) stats for each timer

An asterisk in column 1 below means that timer had multiple indentation
levels. Only the first is printed, though printed timing info is complete.

Stats for thread 0:
          Called Recurse Wallclock max      min      % of total   FP_INS e6 / sec  TOT_INS e6 / sec
  total          1    -   20.690    20.690    20.690     100.00 4.90e+09   236.64 6.34e+09   306.35
    init         1    -    0.000     0.000     0.000       0.00      136    12.36     3294   299.45
    Jloop       64    -   20.689     0.325     0.319     100.00 4.90e+09   236.65 6.34e+09   306.36
      Iloop1   576    -   11.549     0.021     0.019      55.82 3.02e+09   261.85 3.74e+09   324.24
      Iloop2   576    -    9.120     0.016     0.015      44.08 1.87e+09   205.26 2.59e+09   284.26
.
.
Stats for thread 1:
          Called Recurse Wallclock max     min       % of total   FP_INS e6 / sec  TOT_INS e6 / sec
  Jloop         64    -   19.705     0.308     0.307      95.24 4.90e+09   248.47 6.34e+09   321.66
    Iloop1     576    -   11.002     0.020     0.019      53.18 3.02e+09   274.86 3.74e+09   340.35
    Iloop2     576    -    8.684     0.016     0.015      41.97 1.87e+09   215.57 2.59e+09   298.54
.
.
Same stats sorted by timer for threaded regions:
Thd       Called Recurse Wallclock max       min        FP_INS e6 / sec  TOT_INS e6 / sec
000 Jloop       64    -   20.689     0.325     0.319     100.00 4.90e+09   236.65 6.34e+09   306.36
.
000 Iloop1     576    -   11.549     0.021     0.019      55.82 3.02e+09   261.85 3.74e+09   324.24
.
000 Iloop2     576    -    9.120     0.016     0.015      44.08 1.87e+09   205.26 2.59e+09   284.26
.
```

See the `GPTL(3)` man page for complete details.

Using Gptlex

To control the functionality of the GPTL library at runtime, run `gptlex` in default mode this way:

```
$ srun -p <partition> -n <ntasks> gptlex -G <executable> [args]
```

By default, `gptlex` provides wall clock timings for an application. It also automatically produces a dynamic call tree that preserves parent-child calling relationships, but only if the application was manually instrumented with GPTL library calls, or it was compiled using the compiler's auto-instrumentation flag.

Command synopsis

```
gptlex [options] <executable> [args]

gptlex <command>
```

Output

Output files are written to the current working directory in the file `<executable>.gptlex.<node_name>.<MPI_RANK>`.

If the executable is not an MPI program, MPI_RANK is 0.

Measuring Hardware Performance Events

The gptlex tool can also count hardware performance events using PAPI—both PAPI preset events and native events. It supports multiple threads of execution and works seamlessly with MPI programs.

To measure hardware performance events, you must supply the -e <papi-event> option (for a listing, see page 95) to specify which events to measure. You can specify more than one event per run. If you specify more events than the number of physical registers (listed using the -i option), you must also supply the -m (multiplexing) option.

Autoinstrumenting Source code

To auto-instrument source code, you must use the compilers' auto-instrument flag -finstrument-functions at compile time and enable gptlex to use the compiler's output by passing it -G. This enables gptlex to detect the instrumentation points inserted at function entry and exit points in the executable by the compiler and to count the number of times each instrumented function is executed.

☼ If your source code is already instrumented with calls to the GPTL library, don't pass the -G option to the compiler. This informs gptlex that your program is writing the timing output files. Running gptlex this way enables you to add a PAPI counter without recompiling.

☼ If you run gptlex on a noninstrumented executable, without using the compilers' auto-instrument flag with gptlex's auto-instrument option, gptlex measures the time of the entire executable.

Example

Compile the count.c program:

```
int main ()
{
    void A(int);
    void B(void);

    int i;

    for (i = 0; i < 99; ++i) {
       A(100000);
    }

    for (i = 0; i < 8 ; ++i) {
       B();
    }
}

void A (int n)
{
    int i;
    double x = 0;
    for (i = 1; i <= n; ++i) {
       x += 1. / i;
    }
}

void B ()
{
    void C(void);

    C();
}

void C ()
{
}
```

using: `pathcc -finstrument-functions -o count count.c`

then run this `gptlex` command on the executable:

```
gptlex -G -e PAPI_TOT_INS ./count
```

to get this output:

```
PAPI event multiplexing was OFF
PAPI events enabled:
  Total instructions executed

Underlying timing routine was PAPI_get_real_usec.
Per-call utr overhead est: 0 sec.

If overhead stats are printed, roughly half the estimated number is
embedded in the wallclock (and/or PAPI counter) stats for each timer

An asterisk in column 1 below means that timer had multiple indentation
levels. Only the first is printed, though printed timing info is complete.

If a '% of' field is present, it is w.r.t. the first timer for thread 0.
If a 'e6 per sec' field is present, it is in millions of PAPI counts per sec.

Stats for thread 0:
           Called Recurse Wallclock max      min      % of gptlex  UTR Overhead  TOT_INS e6 / sec
  gptlex        1   -         1.257    1.257     1.257     100.00         0.000 1.59e+08   126.38
    main        1   -         1.256    1.256     1.256      99.92         0.000 1.59e+08   126.47
      A        99   -         1.253    0.013     0.012      99.68         0.000 1.59e+08   126.56
      B         8   -         0.000    0.000     0.000       0.00         0.000    45253     0.00
        C       8   -         0.000    0.000     0.000       0.00         0.000    14168     0.00
Overhead sum          =     0.000 wallclock seconds
Total calls           = 117
Total recursive calls = 0
```

See the `gptlex(1)` man page for complete details.

Using Ioex

To measure I/O statistics on an application, run ioex this way:

```
srun -p <partition> -n <ntasks> ioex <executable> [args]
```

Command synopsis

```
ioex [options] <inputfile> [args]
```

Output

The statistics are stored in the output file `<executable>.ioex.<size>.<host>.<proc_id>.<instance>`, located in the current working directory.

For threaded and MPI applications, `ioex` creates separate files for each thread or task, but it does not aggregate the data across threads or tasks. It reports these data:

- Generates statistics (such as block sizes, time/call, etc.) for read, write, and seek operations.
- Prints flags passed to the various ‘open’ operations.
- Detects strided/sequential and random access patterns.
- Captures and generates statistics for MPI I/O calls.

Example

Compile and run the `seek_strided.c` application:

```
$ pathcc -g -O seek_strided seek_strided.c

$ srun -p sc1 ioex ./seek_strided 10 2
```

(For a listing of the `seek_strided` source code, see *ioex seek_strided example* on page 101.)

to generate the following statistics:

```
ioex output is in seek_strided.ioex.1.scx14n0.29322
ioex version:       0.99rc9
Executable:         /net/home/tester/test/bin/ioex_seek_strided
Arguments:          10 2
Processor:          ICE9A
Clockrate:          500.000000
Hostname:           scx14n0
Options:            IOEX,NO_SUMMARY_STATS,NO_DERIVED_STATS,NO_MPI_PROF,NO_IO_PROF
Domain:             User
Parent process id:  29321
Process id:         29322
Start:              Thu Apr 26 13:24:13 2007
Finish:             Thu Apr 26 13:24:15 2007
I/O stats:
File: zero
  fopen
    calls                             :            1
    args                              :            w+
  lseek
    calls                             :            4
    rewinds                           :            0
    abs seek/call                     :            2097152
    bytes accessed between seeks      :            1048576
    access type                       :            STRIDED
    stride                            :            2097152
  fread
    calls                             :            4
    usecs                             :             49000
    usecs/call                        :             12250
    bytes                             :             4194304
    bytes/call                        :             1048576
    MB/s                              :             85
  fwrite
    calls                             :             10
    usecs                             :             60000
    usecs/call                        :             6000
    bytes                             :             10485760
    bytes/call                        :             1048576
     MB/s                             :             174
```

See the ioex man page for complete details.

Using Pfmon

Originally designed as a test harness for Perfmon2, `pfmon` is a low-level tool that has full access to all of the System's performance monitoring features. It provides highly accurate measurements through the use of software breakpoints, `ptrace`, and counting domains.

Though `papiex` and `hpcex` provide most of the functions that `pfmon` does, `pfmon` is the tool of choice for monitoring the performance of applications that use statically-linked binaries and for very accurately measuring very small sequences of instructions, in any of the many supported modes.

You can find thorough documentation for `pfmon` at http://perfmon2.sourceforge.net/.

Using Oprofile

Oprofile runs on a single node. It is a system-wide statistical profiler that samples performance counter interrupt events. Once started, `oprofile` continues, until explicitly stopped, to profile all code running on the node's processors.

⚠ Before other tools can use the node's performance counters, the oprofile daemon must be shutdown.

⚠ Initializing and controlling the oprofile daemon requires root privileges. Once running, regular users can retrieve and display the profile data.

Oprofile is typically used to profile the interactions of many independent processes and kernel tasks. As such, it is particularly useful on systems that run multiple independent oprofile daemon processes. (For parallel profiling (in user mode), use `hpcex`.)

Oprofile's `opcontrol` utility enables root users to set up and control the oprofile daemon. The `opcontrol` utility controls data collection with a control script. Root users edit the control script directly, or pass it arguments specified with the command line option, `--setup`.

The oreport utility enables regular users to retrieve and display profile data in image summaries, which lists the number of samples for individual binary samples (e.g. libraries or applications), and symbol summaries, which provide per-symbol profile data.You can create reports that contain both data types.

Oprofile does not provide callgraph profiling on MIPS systems.

The `ophelp` utility lists and describes the events that are available for profiling on the System.

See the `oprofile` man pages for complete details.

Hardware Performance Counter Events

Hardware performance counter events include node and processor counter events (and the PAPI preset events the map to them).

Node Counter Events

The node counters can measure events across the entire node. These events include DMA traffic, fabric switch packets, and additional processor events.

Like the processor counters, you call the node counters by name (prefixed by `SCB_`). Table 13 lists some of the SCB counters. For a complete list, run papiex `-l`.

Table 13. Sampling of node counter events

Event	Maps to processor counter event
SCB_CPU1_CYCLES	SCB CPU1 cpu cycles
SCB_CPU1_DATAWT	SCB CPU1 cycles of data fetch wait
SCB_CPU1_DATAWT24	SCB CPU1 A data fetch wait ≥ 24 cycles
SCB_CPU1_DATAWT32	SCB CPU1 A data fetch wait ≥ 32 cycles
SCB_CPU1_DATAWT48	SCB CPU1 A data fetch wait ≥ 48 cycles

Table 13. Sampling of node counter events (Cont'd)

Event	Maps to processor counter event
SCB_CPU1_DATAWT64	SCB CPU1 A data fetch wait ≥ 64 cycles
SCB_CPU1_DATAWT8	SCB CPU1 A data fetch wait ≥ 8 cycles
SCB_CPU1_DATAWT96	SCB CPU1 A data fetch wait ≥ 96 cycles
SCB_CPU1_DCHIT	SCB CPU1 L1 data cache hits
SCB_CPU1_DCMISS	SCB CPU1 L1 data cache misses
SCB_CPU1_DTLBHIT	SCB CPU1 data TLB hits
SCB_CPU1_DTLBMISS	SCB CPU1 data TLB misses

The node counters are sampled in pairs for a period of 4096 cycles. They are slower than the processor counters, so their values cannot be compared to processor counter values without appropriate scaling (multiply by a factor of 128).

Because the node counters are sampled in pairs (determined by the order in which they appear on the command line) and can take qualifiers, you can use them to do conditional counting (see Table 14). A qualifier appended to the first counter event refers to the second counter event.

You can combine qualifiers, but certain combinations are SCB_ event-dependent. For a list of qualifiers available for a particular event, run `papiex -L <SCB_name>`.

SCB_* events default to IFOTHER_NONE:HIST_NONE, if you do not specify a qualifier. This sets each counter to independently count the number of cycles its event is active.

To do conditional counting, use the SCB_ event qualifiers this way:

Table 14. SCB_ event qualifier usage

Qualifier	Description
IFOTHER_NONE	Both counters count their events independently of each other
IFOTHER_AND	Count cycles/events only when both counter events occur at the same time
IFOTHER_ANDNOT	Count cycles/events only when the 2nd counter event does not occur
HIST_NONE	Count number of cycles
HIST_EDGE	Count number of events

For example:

- To count the number of CPU cycles in which there are data cache misses:

```
$ srun -p <partition> -n <ntasks> papiex
          -e SCB_CPU0_CYCLES:IFOTHER_AND
          -e SCB_CPU0_DCMISS <executable> [args]
```

- To count the number of CPU cycles in which there are no data cache misses:

```
$ srun -p <partition> -n <ntasks> papiex
       -e SCB_CPU0_CYCLES:IFOTHER_ANDNOT
       -e SCB_CPU0_DCMISS <executable> [args]
```

- To count the number of level 1 data cache misses in which no level 1 data TLB misses occur:

```
$ srun -p <partition> -n <ntasks> papiex
        -e SCB_CPU0_DCMISS:IFOTHER_ANDNOT:HIST_EDGE
        -e SCB_CPU_DTLBMISS:HIST_EDGE <executable> [args]
```

Processor Counter Events and PAPI Preset Events

Table 15 and Table 16 list the processor counter events and the PAPI preset events that map to them, respectively.

Table 15. Processor counter events

Event	Description
CPU_BRANCH	CPU Branches executed
CPU_COP2	CPU COP2 and COP2X instructions executed
CPU_CYCLES	CPU Cycles
CPU_DCEVICT	CPU Data cache line evicted
CPU_DCMISS	CPU Data cache misses
CPU_DTLBMISS	CPU DTLB misses
CPU_FLOAT	CPU Floating point instructions executed (includes loads/stores)
CPU_FPARITH	CPU Floating point arithmetic instructions
CPU_FPMADD	CPU Floating point multiply-add instructions
CPU_ICMISS	CPU I-Cache misses
CPU_INSDUAL	CPU Dual issued instructions
CPU_INSEXEC	CPU Instructions executed
CPU_INSFETCH	CPU Instructions fetched
CPU_INSSCHED	CPU Instructions scheduled
CPU_ITLBMISS	CPU ITLB misses
CPU_L2MISS	CPU Cachable L2 Cache requests that miss in local L2

Table 15. Processor counter events (Cont'd)

Event	Description
CPU_L2MISSALL	CPU Cachable L2 Cache requests that miss in all caches and fill from memory
CPU_L2REQ	CPU Cachable L2 Cache requests
CPU_LOAD	CPU Load/pref/sync/cache ops
CPU_MISPRED	CPU Branches mispredicted
CPU_MSTALL	CPU Scheduling conflict M-stage stalls
CPU_SC	CPU Conditional stores
CPU_SCFAIL	CPU Conditional stores that fail
CPU_STORE	CPU Stores
CPU_TLBTRAP	CPU TLB miss exception traps

Table 16. PAPI preset events

Event	Maps to processor counter event
PAPI_L1_ICA	CPU_INSFETCH
PAPI_LD_INS	CPU_LOAD
PAPI_SR_INS	CPU_STORE
PAPI_CSR_FAL	CPU_SCFAIL
PAPI_CSR_TOT	CPU_SC
PAPI_FP_INS	CPU_FPARITH
PAPI_BR_INS	CPU_BRANCH
PAPI_TLB_IM	CPU_ITLBMISS
PAPI_TLB_TL	CPU_TLBTRAP
PAPI_TLB_DM	CPU_DTLBMISS
PAPI_BR_MSP	CPU_MISPRED
PAPI_L1_ICM	CPU_ICMISS
PAPI_L1_DCM	CPU_DCMISS
PAPI_MEM_SCY	CPU_MSTALL
PAPI_FUL_ICY	CPU_INSDUAL
PAPI_L2_TCM	CPU_L2MISSALL
PAPI_L2_TCA	CPU_L2REQ

Performance Tool Program Examples

hpcex float example

```
#define TYPE double
#include <stdio.h>

void flops(int count)
{
int retval;
```

```
 TYPE a = 1.001;
 int i;
 TYPE old_a;
 for (i =0; i < count; i++){
  a = a * 1.10;
  if (a > 10000.0){
   a = 1.001;
  }
 }
 print("a=%f\n", a);

}

int main(int argc, char **argv)
{
 int c = 200000000;
 unsigned long mask = 0x1;
 //if (sched_setaffinity(getpid(),1,&mask)==-1)
 //    perror("");
 flops(c);
 return 0;
}
```

to examine this ASCII output:

```
[snipped]
=============================================================================
Procedure summary:
-----------------------------------------------------------------------------
 100.00%  [/net/home/work/float]</net/home/work/test/src/float.c>flops
=============================================================================
Loop summary (dependent on structure information):
-----------------------------------------------------------------------------
 100.00%  [/net/home/work/float]</net/home/work/test/src/float.c>11-13
=============================================================================
Statement summary:
-----------------------------------------------------------------------------
  66.50%  [/net/home/work/float]</net/home/work/test/src/float.c>13
  33.09%  [/net/home/work/float]</net/home/work/test/src/float.c>11
   0.41%  [/net/home/work/float]</net/home/work/test/src/float.c>12
=============================================================================
Annotated file (statement/line level):
[/net/home/work/float]/net/home/work/test/src/float.c
-----------------------------------------------------------------------------
    1          #define TYPE double
    2          #include <stdio.h>
    3
    4
    5          void flops(int count)
    6          {
    7            int retval;
    8             TYPE a = 1.001;
    9             int i;
   10             TYPE old_a;
   11  33.09%     for (i=0; i< count;i++) {
   12   0.41%       a = a * 1.10;
   13  66.50%       if (a > 10000.0) {
   14                 a = 1.001;
   15               }
   16             }
   17             printf("a=%f\n", a);
   18
   19          }
```

```
[snipped]
```

hpcex pi example

```
#ifdef PAPI
#include "fpapi.h"
#endif
      program main

      include 'mpif.h'

      double precision  PI25DT
      parameter        (PI25DT = 3.141592653589793238462643d0)

      integer   INTSIZ , DBLSIZ,  ALLNODES,   ANYNODE
      parameter(INTSIZ=4,DBLSIZ=8,ALLNODES=-1,ANYNODE=-1)

      double precision  pi, h, sum, x, f, a, temp
      integer n, myid, numnodes, i, rc
      integer sumtype, sizetype, masternode
      integer status(MPI_STATUS_SIZE)
#ifdef PAPI
      integer retval, es
#endif
c     function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)

#ifdef PAPI
      retval = PAPI_VER_CURRENT
      es = PAPI_NULL
      call PAPIf_library_init(retval)
      if ( retval.NE.PAPI_VER_CURRENT) then
         print *, "papi failed"
      endif
      call PAPIf_create_eventset(es, retval)
      if ( retval.NE.PAPI_OK) then
         print *, "papi eventset failed"
      end if
      call PAPIf_add_event(es, PAPI_TOT_CYC, retval)
      if ( retval .NE. PAPI_OK ) then
         print *, "papi event failed"
      endif
#endif
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numnodes, ierr )
c     print *, "Process ", myid, " of ", numnodes, " is alive"

      sizetype   = 10
      sumtype    = 17
      masternode =  0

 10      if ( myid .eq. 0 ) then

n = 8000

         do i=1,numnodes-1
            call MPI_SEND(n,1,MPI_INTEGER,i,sizetype,MPI_COMM_WORLD,rc)
         enddo

      else
```

```
         call MPI_RECV(n,1,MPI_INTEGER,masternode,sizetype,
     +        MPI_COMM_WORLD,status,rc)

      endif

c     check for quit signal
      if ( n .le. 0 ) goto 30

c     calculate the interval size
      h = 1.0d0/n

      sum  = 0.0d0
      do 20 i = myid+1, n, numnodes
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
 20         continue
      pi = h * sum

      if (myid .ne. 0) then

         call MPI_SEND(pi,1,MPI_DOUBLE_PRECISION,masternode,sumtype,
     +        MPI_COMM_WORLD,rc)

      else

         do i=1,numnodes-1
            call MPI_RECV(temp,1,MPI_DOUBLE_PRECISION,i,sumtype,
     +           MPI_COMM_WORLD,status,rc)
            pi = pi + temp
         enddo
      endif

c     node 0 prints the answer.
      if (myid .eq. 0) then
         write(6, 97) pi, abs(pi - PI25DT)
 97            format('  pi is approximately: ', F18.16,
     +        '  Error is: ', F18.16)
      endif

 30      call MPI_FINALIZE(rc)
      end
```

ioex seek_strided example

```
#include <unistd.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

char zero_buf[1024*1024];
int main(int argc, char **argv){
 int i;
 unlink("zero");
 FILE* myfile = fopen("zero", "w+");
 if (argc < 3){
  fprintf(stderr, "usage: ./writer <write><stride>\n"
           "where <write> is number of MB to write, and
           <stride> is seek between reads\n");
  return(1);
 }
 int num = atoi(argv[1]);
 int stride = atoi(argv[2]);
 for (i = 0; i < num; i++){
  fwrite(zero_buf, 1024*1024, 1, myfile);
 }
 fflush(myfile);
 rewind(myfile);
 int fd = fileno(myfile);
 for (i = 0; i < num; i+=stride+1){
  fread(zero_buf, 1024*1024, 1, myfile);
  lseek(fd, stride * 1024 * 1024, SEEK_CUR);
 }
 fclose(myfile);
 exit(0);
}
```

Chapter 6 Using the Optimized Math and Science Libraries

In this section:

The SiCortex software suite includes the standard GNU math library, `libm`; a tuned version of libm, `libscm`; a tuned scientific library, `libscs`; two versions of a tuned string library, `libscstr`; and an array of math and science libraries.

All optimized libraries are provided in static and dynamic versions for both n32 and n64 ABIs.

The compilers default to searching for and linking the dynamic version of the libraries. To use the static version, you must supply the `-static` option to the linker.

The `pathf95` compiler follows the name-mangling rules described in *Compiler Name Mangling* on page 140. Because the libraries supplied with the SiCortex software comply with these rules, we strongly recommend that you avoid using compiler flags that alter the compiler's default method of applying underscores to symbol names.

Libscm Tuned Math Library

The `libscm` library contains tuned versions of many of the `libm` math functions: `atan`, `ceil`, `ceilf`, `expf`, `exp2f`, `exp10f`, `floor`, `floorf`, `fmax`, `fmaxf`, `fmin`, `fminf`, `rint`, `rintf`, `round`, `roundf`, `trunc`, `truncf`, `hypot`, `hypotf`, `sin`, `sinf`, `cos`, `cosf`, `sinecos`, `sincosf`, `log`, `log10`, `log2`, `logf`, `log10f`, and `log2f`.

Libscm functions typically provide a speedup in performance that is, at least, several times over their libm (glibc) counterparts and at equal or near equal accuracy.

The `libscm` library supports applications built with either the n32 or n64 ABI. The linker automatically selects the correct version of the `libscm` library.

To preserve cached data, `libscm` avoids using memory as much as possible.

For best, predictable results, make sure your program calls IEEE compliant data and uses *round-to-nearest* mode. Remember that `libscm` functions do not always report exceptions.

The functions do not handle `inexact` or `overflow/underflow` exceptions.

Accuracy of libscm Functions

Except as described in Table 17, most of the functions are usually accurate to 1 unit of least precision (ulp), but accuracy decreases when the functions are called in other than *round-to-nearest* floating point rounding mode. None of the functions sets `errno`.

Table 17. Accuracy specifications

Function	Description
hypot hypotf	Hypotenuse. Computes the value of the square root of $x^2 + y^2$ without undue overflow or underflow. Fully IEEE compliant for NANs and infinities.
sin cos	Sine and cosine of x in radians. Domain ≈ closed interval (`-2^49, 2^49`); domain base 10 (more accurate) `-3.968253968253968254e14` to `3.968253968253968254e14`. The base 10 domain is considered the correct one, although error bounds have not been rigorously determined. No bounds checking is done, so the caller must provide values in the appropriate domain. These functions occasionally (rarely) return answers that differ by 2 ulps from the correctly rounded one.

<func_name>f = single precision version

Table 17. Accuracy specifications (Cont'd)

Function	Description
sinf cosf	Domain accuracy (-2^25, 2^25) Domain accuracy (-2^24, 2^24) Maximum error has not been rigorously proven, but has been tested for many millions of values, with no error exceeding 1 ulp found. No bounds checking is done, so the caller must provide values in the appropriate domain. These functions return accuracy in all floating-point rounding modes.
sincos	Simultaneous sine and cosine of x in radians. Domain ≈ closed interval (-2^49, 2^49); domain base 10 (more accurate) -3.968253968253968254e14 to 3.968253968253968254e14. The base 10 domain is considered the correct one, although error bounds have not been rigorously determined. No bounds checking is done, so the caller must provide values in the appropriate domain. This function occasionally (rarely) returns answers that differ by 2 ulps from the correctly rounded one.
sincosf	Domain accuracy (-2^25, 2^25) Maximum error has not been rigorously proven, but has been tested for many millions of values, with no error exceeding 1 ulp found. No bounds checking is done, so the caller must provide values in the appropriate domain. This function returns accuracy in all floating-point rounding modes.
log log10 log2	Natural logarithm. Accuracy 1 ulp across the entire domain, except it does not return infinity for log (0). Base 10/Base 2. Guaranteed accurate only for 2 or 3 ulps, although most values fall within 1 ulp of correctly rounded results. These functions are not IEEE-compliant. Use *round-to-nearest* mode.
logf log10f log2f	Same description as the double-precision counterpart. Usually accurate to 1 ulp, with no cases found where error exceeded 1 ulp. Otherwise, same description as double-precision counterparts.
expf exp2f exp10f	Exponential function. Returns the value of e (the base of the natural logarithms) raised to the power of x, two to the power of x, and ten to the power of x, respectively. Although error bounds have not been rigorously determined, no cases were found in which bounds exceeded 1 ulp. These functions do not return the same range of subnormals in the limit g(x)⇒0; instead they flush to zero. The libm expf function incorrectly promotes values at the top of the representable domain to infinity. Because the libscm version does not emulate this behavior, it may sometimes give different answers for the highest values in the range. For example, at the input point 88.72202301025390625, the libm expf function returns infinity, whereas the libscm expf function correctly returns 3.4000478186e+38. Although the highest range limit for infinity differs between exp10f and exp2f, since these functions are provided through accurate base translation, they use the same limit as expf.
<func_name>f = single precision version	

Accessing the libscm Library

The libscm library is available from Fortran, C, and C++ applications. Access to libscm from Fortran is available only through the PathScale compilers' `-ffast-math` flag.

Fortran access to libscm

The `-ffast-math` flag causes the linker to link automatically with libscm. It should be used for compiling as well as linking to get the best in application performance.

```
pathf95 -ffast-math -o myapp myappa.f90 myappb.f90
```

or

```
pathf95 -Ofast -o myapp myappa.f77 myappb.f77
```

☼ The `Ofast` option incorporates `-ffast-math`. To use `-Ofast` without linking with libscm, include `-fno-fast-math` on the link line.

C/C++ access to libscm

Because the functions in libscm are aliased with and replace those in libm, users need only include `<math.h>` in their source files.

The `libscm` function names supersede those in `libm`, so to use both libraries with compilers, insert `-lscm` before `-lm` on the compiler/linker command line, for example:

```
gcc -o myapp myappa.o myappb.o -lscm -lm
```

```
pathcc -o myapp myappa.c myappb.c -lscm -lm
```

You can also use the PathScale compilers' `-ffast-math` flag to autolink libscm with C and C++ applications. If you specify both `-ffast-math` and `-lm`, the compilers link, in order, `-lscm -lmpath -lm`. Libmpath contains Open64 implementations of some libm functions and some math functions (such as `acos`) not yet implemented in libscm.

```
pathcc -ffast-math -o myapp myappa.c myappb.c -lm
```

or

```
pathcc -Ofast -o myapp myappa.c myappb.c -lm
```

Libscs Tuned Scientific Library

The libscs library provides some miscellaneous functions not included in other science libraries. It contains tuned anisotropic correlation/convolution routines: conv2d_dp, conv2d_sp (2-dimensional, double- and single-precision, respectively) and conv3d_dp, conv3d_sp (3-dimensional, double- and single-precision, respectively).

The libscs library supports applications built with either the n32 or n64 ABI. The linker automatically selects the correct version of the libscs library.

Applications written in C/C++ must include <scslib.h> in their source files. Those written in Fortran must include scslib.inc.

To call libscs routines from Fortran applications, simply use the routines as if all variables are passed by reference. For example:

```
CALL CONV2D_SP(A, NRA, NCA, K, NRK, NCK, C)
```

The libscs routines expect row major order. However, for Fortran applications, no special precautions are needed as long as the order of kernels and matrices match.

The libscs library is independent of other libraries, so you can include it anywhere on the link line by specifying -lscs.

The anisotropic convolution routines actually implement correlation. The only difference between convolution and correlation is the order in which the filter kernel is traversed to yield the sum for each input datum. For convolution, the filter kernel is traversed backwards, from end to beginning; for correlation, from beginning to end. This means the order in which the application loads the filter coefficients determines whether the function performs convolution or correlation. Typically, such stencil functions are referred to generically as *convolutions*, a convention used hereafter in this manual. Table 18 on page 108 describes the convolution routines.

Table 18. libscs tuning details

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td>conv2d_sp</td><td>

Single-precision anisotropic 2-dimensional convolution. Calling prototype:

```
void conv2d_sp(
                float *a,       // pointer to input matrix a
                int nra,        // number of rows in a _and_ c
                int nca,        // number of cols in a _and_ c
                float *k,       // pointer to stencil kernel k
                int nrk,        // number of taps per row in the kernel
                int nck,        // number of taps per column in the kernel
                float *c;       // pointer to output matrix c
                )
```

Aside from the difference in the declared type of the pointer arguments, the description of this routine is the same as its double-precision counterpart, `conv2d_dp`.

</td></tr>
<tr><td>conv3d_sp</td><td>

Single-precision anisotropic 3-dimensional convolution. Calling prototype:

```
void conv3d_sp(
                float *a,       // input volume a
                int nsa,        // number of slices a _and_ c
                int nra,        // number of rows per slice a _and_ c
                int nca,        // number of cols per slice a _and_ c
                float *k,       // stencil volume k
                int nsk,        // number of slices k
                int nrk,        // number of rows per slice
                int nck,       // number of columns per slice
                float *c       // output volume c
                )
```

Aside from the difference in the declared type of the pointer arguments, the description of this routine is the same as its double-precision counterpart, `conv3d_dp`.

</td></tr>
</table>

Table 18. libscs tuning details (Cont'd)

<table>
<tr><th>Function</th><th>Description</th></tr>
<tr><td>conv2d_dp</td><td>

Double-precision anisotropic 2-dimensional convolution. Calling prototype:

```
void conv2d_dp(
                double *a,      // pointer to input matrix a
                int nra,        // number of rows in a _and_ c
                int nca,        // number of cols in a _and_ c
                double *k,      // pointer to stencil kernel k
                int nrk,        // number of taps per row in the kernel
                int nck,        // number of taps per column in the kernel
                double *c;     // pointer to output matrix c
                )
```

Arguments a, c, and k require only natural alignment. However, performance may vary if, for example, all memory is page aligned, which may increase TLB or cache activity, depending on the dimensions.

Input a and output c must have the same dimensions. However, the number of real outputs in c is less than the inputs in a because of the nature of convolution. The actual number of outputs is given by (nra – nrk + 1) * (nca – nck + 1) with the remaining points constituting a guard band around the *edge* of c. The actual output runs from [c + nck/2, c – nck/2] in each row, and from [c + nrk/2, c – nrk/2] in each column.

If the number of output columns (nca – nck + 1) is evenly divisible by 4, no erroneous data is written into the guard band. Otherwise, the guard band is not guaranteed to be uncorrupted. Likewise, keeping the number of output columns to an even multiple of four produces the best performance.

The minimum dimensions nra and nca are determined to ensure a positive nonzero number of output rows and columns.

The algorithm requires that the kernel dimensions nrk and nck be multiples of 3. This enforces kernels of 9x9, 15x15, 21x21, 33x33, and so on. The kernel data has no symmetry requirements and is fully anisotropic with regards to the implementation. Also, there is no nrk = nck requirement.

If you need a non multiple-of-three kernel (for example, 11x11), zero-pad the kernel in a symmetrical guard band around the edges. For this 11x11 example, the first and last rows and the first and last columns of a 12x12-defined kernel would be zero. In such scenarios, the loss in performance is proportional to the percentage of the kernel that is zero-padded.

</td></tr>
</table>

Table 18. libscs tuning details (Cont'd)

Function	Description

conv3d_dp	Double-precision anisotropic 3-dimensional convolution. Calling prototype:
	`void conv3d_dp(`
	`          double *a,     // input volume a`
	`          int nsa,       // number of slices a _and_ c`
	`          int nra,       // number of rows per slice a _and_ c`
	`          int nca,       // number of cols per slice a _and_ c`
	`          double *k,     // stencil volume k`
	`          int nsk,       // number of slices k`
	`          int nrk,       // number of rows per slice`
	`          int nck,       // number of columns per slice`
	`          double *c      // output volume c`
	`          )`
	Arguments `a`, `c`, and `k` require only natural alignment.
	Input `a` and output `c` must have the same dimensions. The actual number of outputs is given by `(nsa - nsk + 1) * (nra - nrk + 1) * (nca - nck + 1)` with the remaining points constituting a guard band around the *edge* of volume `c`. The actual output runs from slices `[c + nsk/2, c - nsk/2]`, from `[c + nck/2, c - nck/2]` in each row, and from `[c + nrk/2, c - nrk/2]` in each column.
	If the number of output columns `(nca - nck + 1)` is evenly divisible by 4, no erroneous data is written into the guard band. Otherwise, the guard band is not guaranteed to be uncorrupted.
	Since the number of points written into the guard band may become large enough to adversely affect performance, it behooves you to select output sizes in which the number of output columns `(nca - nck + 1)` is evenly divisible by 4.
	The minimum dimensions `nsa`, `nra`, and `nca` are determined to ensure a positive nonzero number of output slices, rows and columns. The minimum dimensions depend on `nsk`, `nrk`, and `nck`.
	The memory layout of volumes `a`, `c`, and `k` is traditional C row major—columns, rows, and slices—so column `p` of row `q` of slice `l` can be accessed this way: `l*nra*nca + q*nca + p`.
	The number of slices in the kernel `k` can begin at 1 and grow to any size. When `nsk = 1`, 3D convolution devolves to a series of 2D convolutions, making it more efficient to use `conv2d_dp` for this case.
	The number of rows and columns, `nrk` and `nck`, in `k` must be multiples of 3. There is no `nrk = nck` requirement, though nonsymmetric dimensions have not been tested.

Libscstr and Libscfstr Tuned String Libraries

The `libscstr` and `libscfstr` libraries provide tuned versions of the `libc` functions `strcpy`, `memset`, and `memcpy`. In general, the tuned functions provide a significant increase in performance over their `libc` counterparts. Use of the two tuned libraries differs only in the calling convention. Table 19 on page 111 describes each of the tuned functions.

The libscstr and libscfstr libraries work only with C/C++ applications.

- Libscstr

  To use the libscstr library, you must modify your source code to call the functions by their libscstr names: sc_strcpy, sc_memset, and sc_memcpy. The PathScale compilers automatically link in the libscstr library, but the gcc compiler does not, so you need to add -lscstr to the compile/link line when using gcc.

  For any functions called by their standard libc names, the linker links in those functions from libc. This feature enables you to retain calls to libc functions for small copy operations, which is slightly faster for string lengths less than eight bytes, while replacing larger copy operations with calls to the tuned sc_* functions.

- Libscfstr

  To use the libscfstr (*f* stands for *fast*) library, you call the functions by their standard libc names in your source code, and include -lscfstr on the link line. The standard libc functions are replaced by the fast functions in libscfstr.

- Furthermore, you can speed up existing executables by setting LD_PRELOAD={/usr/lib64 or /usr/lib32}/libscfstr.so. This results in all executables, whether or not built with libscfstr, using the fast functions in libscfstr.

To use the static versions of the tuned string libraries, you must supply the full path to the library on the link line, for example:

```
-l /usr/lib32/libscstr.a
```

Table 19. libscstr/libscfstr tuning details

Function	Description
sc_strcpy	Replaces strcpy. Functionality is identical to strcpy. Faster than strcpy for string lengths greater than eight bytes, comparable to strcpy for string lengths less than eight bytes. You need not pad to the next largest multiple of 64 to obtain optimal performance.
sc_memcpy	Replaces memcpy. Functionality is identical to memcpy, with the exception that it sometimes fails to return the pointer to the original destination. Therefore, when using sc_memcpy, do not count on the pointer to the original destination being returned. Much faster than memcpy at small sizes. Provides nearly monotonic performance increases, and you need not pad to the next largest multiple of 64 to obtain optimal performance. Uses a hit-under-miss algorithm to maximize bus bandwidth to and from the L2 cache and main memory.

Table 19. libscstr/libscfstr tuning details (Cont'd)

Function	Description
sc_memset	Replaces `memset`. Functionality is identical to `memset`. Much faster than `memset` for all lengths and alignments, and it has a much smoother–nearly monotonic–performance profile. Uses a hit-under-miss algorithm to maximize the bus bandwidth to and from the L2 cache and main memory for larger lengths.

Math and Science Libraries

Table 20 lists and describes the supplied math and science libraries.

Table 20. Descriptions of the math and science libraries

Library	Description
AtlasBLAS	Automatically Tuned Linear Algebra Software/Basic Linear Algebra Subroutines. (Serial) Atlas provides portably optimal linear algebra software based on BLAS, which provides standard building blocks for performing basic vector and matrix operations. • Level 1 routines perform scalar, vector and vector-vector operations. • Level 2 routines perform matrix-vector operations. • Level 3 routines perform matrix-matrix operations. Linking: see *Linking the Optimized Atlas Library for Fast BLAS* on page 114.
GotoBLAS	GotoBLAS provides portably optimal linear algebra software based on BLAS, but focuses on minimizing TLB misses, rather than on optimizing cache utilization. Linking for Fortran, C, C++: `-lgotoblas`
BLACS	Basic Linear Algebra Communications Subprograms. A linear algebra-oriented, message-passing interface that makes linear algebra applications easier to program and port. Needed by ScaLAPACK.
FFTW	Fast Fourier Transform. (Serial and parallel) FFTW2 for MPI applications FFTW3 for serial and multithreaded applications
LAPACK	Linear Algebra Package. (Serial) Designed for solving systems of simultaneous linear equations, least squares solutions of linear systems, eigenvalue problems, singular value problems. Provides the associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur), reordering of Schur factorizations, and estimating condition numbers. Handles dense and banded matrices, but not sparse matrices. Provides similar functionality for real and complex matrices, in single and double precision. Linking for C, C++, Fortran: `-llapack <-lgotoblas|-lcblas -latlas|-lf77blas -latlas>`

Table 20. Descriptions of the math and science libraries (Cont'd)

Library	Description
PETSc	Portable, Extensible Toolkit for Scientific computation. (Serial and parallel) Provides tools for both serial and parallel numerical solutions of PDEs that require solving large-scale, sparse nonlinear systems of equations.Includes: • Nonlinear solvers–Newton-based methods: Line Search, Trust Region, other • Time steppers–Euler, Backward Euler, Pseudo Time Stepping, other • Krylov Subspace Methods–GM RES, CG, Bi-CG-STAB, TFQMR, Richardson, Chebychev, other • Preconditioners–Additive Schwartz, Block Jacobi, Jacobi, ILU, ICC, LU (sequential only), other • Matrices–Compressed Sparse Row (AIJ), Blocked COmpressed Sparse Row (BAIJ), Block Diagonal (BDIAG), Dense, other • Vectors • Index sets–Indices, Block Indices, Stride, other Linking: see *Linking the PETSc Library* on page 114.
ScaLAPACK	Scalable LAPACK. (Parallel) Designed for distributed memory, MIMD parallel computers. Routines for solving linear systems of equations, symmetric positive definite branded linear systems of equations; condition estimation and iterative refinement, for LU and Cholesky factorization, matrix inversion, full-rank least squares problems, orthogonal and generalized orthogonal factorization, orthogonal transformation routines, reductions to upper Hessenberg, bidiagonal and tridiagonal form, reduction of symmetric-definite generalized eigenproblem to standard form, the symmetric, generalized symmetric and the nonsymmetric eigenproblem. Linking for C, C++, Fortran: `-lscalapack -lblacs -llapack <-lgotoblas|-lcblas -latlas|-lf77blas -latlas>`
SPRNG	Scalable Parallel Pseudo Random Number Generator. (Serial and parallel) Supports ASC Monte Carlo computations. Linking for Fortran, C, C++: `-lsprng -lstdc++`

Linking the Optimized Atlas Library for Fast BLAS

To use the optimized version of the Atlas library for fast BLAS, you must link with the BLAS and Atlas libraries.

Linking is the same on the nodes and the cross-development workstation. For example, using the n64 version of the library:

- Fortran `-lf77blas -latlas`
- C `-lcblas -latlas`
- Combined Fortran/C
  `-lf77blas -lcblas -latlas`

The path to the include files differs between the native and cross-development environments. For applications that make explicit use of Atlas, the location of the include files are:

- Nodes `/usr/include/atlas`.
- Cross-development workstation
  `/opt/sicortex/rootfs/default/usr/include/atlas`.

The `pathf95` compiler follows the name-mangling rules described in *Compiler Name Mangling* on page 140. Since neither `pathcc` nor `gcc` have compiler mechanisms to append these underscores, C applications using the BLAS interface directly need to follow the Fortran underscoring convention, either explicitly or through preprocessing, to produce the correct symbol form.

Linking the PETSc Library

The way you link the PETSc library differs between the native and cross-development environments and varies according to whether you are building an n64 or n32 application.

Building Natively on the Nodes

- n64 applications

  There are no variables to set, so you can build your application normally, using PETSc makefiles.

- n32 applications

Make sure the System Administrator has mounted the n32 buildroot on the nodes where you intend to build your n32 application.

```
export PETSC_DIR=/PathToRootfs*/build.n32/usr/share/petsc
export PETSC_ARCH=linux-mips-n32
```

Then edit your makefile to add to CFLAGS, FFLAGS, and all linker flags:

```
-isysroot/PathToRootfs*/build.n32 -mabi=n32
```

Building on the Cross-Development Workstation

- n64 applications

  ```
  export PETSC_DIR=/PathToRootfs†/build/usr/share/petsc
  export PETSC_ARCH=linux-mips-n64
  ```

  Then build your application normally, using PETSc makefiles.

- n32 applications

  ```
  export PETSC_DIR=/PathToRootfs†/build.n32/usr/share/petsc
  export PETSC_ARCH=linux-mips-n32
  ```

  Then modify your makefile to add to CFLAGS, FFLAGS, and all linker flags:

  ```
  --sysroot=/PathToRootfs†/build.n32 -mabi=n32
  ```

  Then build your application normally, using PETSc makefiles.

- SC072 workstation

  ```
  export PETSC_DIR=/PathToRootfs†/usr/share/petsc
  export PETSC_ARCH=linux-mips-<n32|n64>
  ```

  If building an n64 application, build it normally, using PETSc makefiles.

  If building an n32 application, modify your makefile to add to CFLAGS, FFLAGS, and all linker flags:

  ```
  --sysroot=/PathToRootfs†/build.n32 -mabi=n32
  ```

Linking Interdependent Libraries

Many of the math and science libraries use routines contained in other libraries. When you use such interdependent libraries, make sure you link them in the correct order. A library containing routines used by another must be linked after the library that uses it. So, for example, where liba

[*] Typically, /.root0/opt/sicortex/rootfs

[†] Typically, /opt/sicortex/rootfs

depends on routines in `libb`, and both depend on the MPI library, the linker command line might look like this:

```
gcc -o myapp ab.o ac.o -la -lb -lscmpi_debug*
```

Alternatively, you can use GNU's `libtool` to order the libraries correctly for linking. For details, see the `libtool` documentation.

* To use the MPI optimized library instead, replace `-lscmpi_debug` with `-lscmpi`.

Chapter 7 Developing MPI Applications

In this section:

For details on submitting and running MPI applications on the System, see *Chapter 2, Running Applications* on page 15.

The SiCortex MPI library implements the Message Passing Interface (MPI) for SiCortex systems. SiCortex MPI uses the nodes' DMA engine and the System's Interconnect Fabric to implement high-bandwidth, low-latency node-to-node communication. This scheme provides a direct user-mode communication path, enabling MPI processes to pass data between one another without making system calls.

SiCortex MPI Implementation

The SiCortex MPI implementation is based on the MPICH2 software from Argonne National Laboratory:
http://www.mcs.anl.gov/research/projects/mpich2/

Because the SiCortex MPI library uses the DMA engine directly, MPI applications can take immediate advantage of the System's communication architecture, without writing machine-specific code.

MPI Feature Support

SiCortex MPI provides the standard MPI language bindings for C, C++, Fortran 77, and Fortran 90 applications. It supports all MPI-1 features and these selected MPI-2 features:

- MPI I/O

  Supports parallel I/O, in which a parallel MPI application does all I/O operations, and collective I/O, in which many processes concurrently access the same file.

- Intercommunicator collective operations

  Enables collective operations between processes that belong to different communicator groups.

- One-sided communication

  Enables one process to specify all communication parameters for both the sending and receiving sides.

- External interfaces

  Supports generalized requests, named objects, error classes, and so forth.

Compiling and Linking MPI Applications

The SiCortex MPI library, `libscmpi`, implements the Message Passing Interface (MPI) for SiCortex systems. It provides interfaces for C, C++, and Fortran programs.

For more information on compiling, see *Chapter 3, Compiling and Linking Applications* on page 25 and *General Procedure for Optimizing an Application* on page 52.

MPI Library

There are two versions of the MPI library: scmpi and`scmpi_debug`. We recommend that you use `scmpi_debug` until you have finished debugging your application, and then switch to the optimized version, `scmpi`.

The `scmpi_debug` version performs internal and external safety checking on MPI library usage. Though it runs more slowly, it can detect errors in code that calls the MPI library and errors internal to the MPI library.

The MPI library supports applications built with either the n32 or n64 ABI. The linker automatically selects the correct version of the MPI library and links with other libraries that it needs to interface with the DMA engine and with the SLURM process management.

MPI Header Files

MPI programs written in C and C++ must include `<mpi.h>` in their source files. Those written in Fortran must include `mpif.h`.

Because of a name conflict between `stdio.h` and the MPI C++ binding that involves `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`, you must either include `mpi.h` before `stdio.h` and `iostream.h` in MPI programs written in `C++`, or add `-DMPICH_IGNORE_CXX_SEEK` to the compiler command line to force it to skip the MPI versions of the `SEEK_*` routines.

MPI Library Linking Order

Though no special switches are required for compiling, you must link your program with the MPI library, either `-lscmpi` or `-lscmpi_debug.` When using other libraries that depend on it, add the MPI library to the end of the linker's command line.

For example, the linker command line might look like this, while you are debugging your application:

```
pathcc -o mympiapp -lscaLAPACK -lscmpi_debug
```

then, after your application is running smoothly and you want to do some performance runs, like this:

```
pathcc -o mympiapp -lscaLAPACK -lscmpi
```

For C++ programs that use either `g++` or `pathCC`, you must add `-lscmpicxx -lscmpi` to the end of the link line (to use the debug version of the MPI library, add `-lscmpicxx -lscmpi_debug` instead).

MPI Compiler Scripts

SiCortex MPI software provides two sets of compiler scripts (Table 21) that automatically invoke the appropriate PathScale compiler and link with the optimized version of the MPI library. In general, the `mpi*` compiler scripts run in the native environment, and the `scmpi*` compiler scripts run in the cross-development environment (see page 31).

Table 21. MPI compiler scripts[*]

Native Environment	Cross-Development Environment
mpicc	scmpicc
mpicxx	scmpicxx

Table 21. MPI compiler scripts* (Cont'd)

Native Environment	Cross-Development Environment
mpif77	scmpif77
mpif90	scmpif90

* If you are building n32 applications, see page 33 and page 35 for compiling instructions. If you are building your n32 application on the nodes, make sure your System Administrator has mounted the n32 buildroot on the nodes where you intend to build your application.

Because the compiler scripts automatically link in the MPI library, you needn't add it to the link line. However, if you want to use the debug version of the library instead, you must add `--mpilib=-lscmpi_debug` to the end of the link line.

MPI Debugging Hook

Before you run your MPI application, you can set the `SCMPI_DEBUG_WAIT` environment variable to help debug it. Setting `SCMPI_DEBUG_WAIT` to any value causes the MPI code to pause in an infinite loop during `MPI_INIT()` operations. This behavior enables the user to attach a debugger to processes spawned in the program. Then, for each paused process, the user calls `MPIDI_Debug_start()` from the debugger to continue the process.

MPI Performance Tips

- Globally synchronized time

  `MPI_Wtime()` reports time values that are globally synchronized. So, if two processes of an MPI program call `MPI_Wtime()` at the same time, `MPI_Wtime()` will report the same value. The accuracy of synchronization is better than one microsecond.

  The `MPI_WTIME_IS_GLOBAL` attribute of `MPI_COMM_WORLD` indicates that `MPI_Wtime()` reports a global value.

- Early send and posted receive queue sizes

  Like other MPI implementations, SiCortex MPI implements software message queues that track:

  - Send operations yet to be matched to a receive call (early send queue)
  - Receive operations yet to be matched to a sender (posted receive queue)

Very long queues increase the time the MPI library must spend traversing them to find matches, and thus can impact the performance of ordinary send and receive operations. This issue is of considerable importance for applications that scale to thousands of processes.

Mitigating this issue can be tricky, but sometimes using barrier operations to keep processes in phase can actually increase performance by keeping the queues short.

- Using `MPI_Sendrecv()`

  Calling `MPI_Sendrecv()` can be more efficient than posting individual send and receive operations, for both short and long messages.

- Short message latency

  The blocking forms of send and receive operations, `MPI_Send()` and `MPI_Recv()`, have the shortest latency, and therefore the best performance for short messages, particularly those less than 1KB.

- Long message bandwidth

  In MPI, longer messages achieve better bandwidth by amortizing fixed communication overheads and by allowing the library to use multiple paths through the Interconnect Fabric.

  In practice, point-to-point message sizes in the range of 100KB to 1MB can achieve peak bandwidth.

- Overlapping communication with computation

  SiCortex MPI allows applications some opportunity to overlap communication and computation, particularly for very large messages (≥100KB). Because the MPI library does not maintain a thread independent of the application, it makes no software progress when no MPI function calls are made. Opportunities for an application to overlap computations occur only when the MPI library can return to the application while hardware RDMA operations are progressing.

  Once the MPI library software establishes a match at the receiver end, it commands the DMA engine to initiate up to six RDMA operations. It can return to the application from a nonblocking call until these communication operations end. A carefully coded application could take this opportunity to perform computation operations while the communication operations proceed.

  ☼ In some cases, particularly with very large messages, the application must invoke the MPI library to schedule additional

RDMA operations before the entire operation finishes. Calling `MPI_Iprobe()` at both the send and receive ends can enable the library to schedule more RDMA operations.

- Data alignment

  The alignment of data can affect communication performance. For short messages (<1KB), aligning the source and destination buffers on 8-byte boundaries improves performance slightly.

  For long messages (>1KB), communication is most efficient when the source and destination buffers are relatively* aligned to 64 bytes, and least efficient (incurring a temporary buffer copy), when they are not relatively aligned to 4 bytes.

  The effect of alignment on long messages is most apparent for very large messages (>100KB), where performance can be limited by the DMA engine's access to memory.

  Applications written in C can control the alignment of variables by using the `gcc aligned` attribute, and control the alignment of dynamically allocated buffers by using `posix_memalign()` instead of `malloc()`.

- Noncontiguous, derived data types

  Calling MPI routines using derived data types that specify noncontiguous data impacts performance. For these cases, SiCortex MPI copies data in and out of temporary buffers, incurring the costs of data copying and of interpreting the data types. In general, it's better for an application to move data in and out of its own contiguous temporary data buffers because that allows the compiler to generate efficient code for moving the data.

Thread Support

SiCortex supports `MPI_THREAD_FUNNELED` semantics, which allows multithreaded processes, but only the main thread in a process can access the MPI library. All other threads in the process funnel their MPI calls to the main thread. Only the main thread initializes and finalizes MPI.

* Buffers are relatively aligned when the difference between the source and destination addresses is divisible by the alignment. This is guaranteed when both buffers have the required absolute alignment.

This means that codes built for hybrid OpenMP/MPI operation can safely use the SiCortex MPI library if all MPI operations are done outside threaded regions.

(For details on MPI thread-compliance, see MPI documentation. For a list of resources, see *MPI Reference Information* .)

MPI Reference Information

- Reference to the online man pages:
  - http://www-unix.mcs.anl.gov/mpi/index.htm MPI home
  - http://www.mcs.anl.gov/research/projects/mpich2/ MPICH2 home
- References to MPI and MPICH2 tutorials and standards information:
  - http://www-unix.mcs.anl.gov/mpi/tutorial/
  - http://ci-tutor.ncsa.uiuc.edu/login.php
  - http://www.mpi-forum.org/
- References to books:
  - Sweetman, Dominic (2006). *See MIPS Run; Second Edition*. San Francisco. Morgan Kaufmann Publishers.
  - Pacheco, Peter S. (1997). *Parallel Programming with MPI*. San Francisco. Morgan Kaufmann Publishers.
  - Snir, Marc, et al. *MPI-The Complete reference; Volume I, The MPI Core*. The MIT Press, 1998.
  - Gropp, William, et al. *MPI-The Complete reference; Volume II, The MPI-2 Extensions*. The MIT Press, 1998.
  - Gropp, William, et al. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1999.
  - Gropp, William, et al. *Using MPI-2, Advanced Features of the Message-Passing Interface*. The MIT Press, 1999.

Chapter 8 Writing Threaded Applications

In this section:

Though the SiCortex software uses MPI to implement the interface between user applications and the System's communication architecture, MPI is not necessary for writing parallel applications for the System.

Some parallel applications may have little or no need for communication between processes, so wouldn't need to use MPI. Yet users might still find MPI a convenient tool for communication or coordination tasks (for details, see *Chapter 7, Developing MPI Applications* on page 117).

Some parallel applications use TCP/IP methods, such as sockets, to communicate between processes. The System's nodes support this model. Indeed, the operating system transmits IP traffic between nodes directly over the Interconnect Fabric (see page 8). This method can provide substantial bandwidth for large messages, but generally, MPI provides better communication bandwidth for long messages and much shorter latency for short messages.

Some parallel applications use multithreading. SiCortex supports the POSIX Pthreads interface (explicit threads programming) and the OpenMP threads model (but only for PathScale compilers).

Multithreaded Programming Considerations

Software developers who write multithreaded applications without using OpenMP or POSIX Pthreads, need to understand how to use memory barriers and sync operations to guarantee the correct ordering of events any time one processor must communicate data in a specific order to another processor. For details on these synchronization tools, see the *Linux Kernel Memory Barriers* document at:

http://kerneltrap.org/node/6431

Optimizing Code Parallelization

These are some ways you can optimize parallelization of your code:

- Data dependencies

  Different tasks in an application that use the same data storage locations are one of the primary inhibitors to programming parallelism.

- Load balancing

  The goal of load balancing is to distribute work among all processors such that they remain busy all of the time.

  - Partition the work equally among all processors

    For array/matrix operations in which each processor performs similar computations, distribute the data evenly among the processors. (See *Memory System Operation on* page 132.)

    For loop iterations during which similar operations are performed, distribute the iterations across all processors.

  - Assign work dynamically

    Certain classes of applications tend toward load imbalance regardless of whether data is evenly distributed across all processors:

    sparse arrays – some processors work while others idle.

    adaptive grid methods – some processors need to refine their mesh while others don’t.

    N-body simulations – some particles migrate between their original processor domain and another processor’s domain, or the particles owned by some processors require more work than those owned by others.

    When you know that the amount of work each processor performs will vary or is unpredictable, use a scheduler-task pool approach. As each processor finishes its work, have it queue to get more work.

    You may need to design an algorithm that detects and handles load imbalances as they occur dynamically within the code.

Ordering Rules: Memory and I/O Operations

The System's ordering rules for memory and I/O operations are:

- To guarantee that memory operation A is visible to other processors or to an I/O device before memory operation B, a sync operation must intervene between A and B. Otherwise, if A and B are not in the same 32-byte L1 data cache block, B may be executed before A.
- All I/O writes complete in the correct order. Although the instruction sequence `read_io(x); write_io(y)` may reorder inversely, the instruction sequence `write_io(y); read_io(x)` never reorders. This ensures that I/O read operations are deferred until all I/O and memory write operations have completed and are visible to all of the node's processors.
- I/O writes and reads to the L2 cache's control registers (interrupt and cache ECC properties) may reorder with respect to each other and to I/O writes to other targets. So, to guarantee correct ordering of I/O operations on the local control registers, a sync operation must intervene between such I/O instructions. Memory writes are always ordered with respect to I/O writes to any of the local control registers.
- The processors implement a *hits under misses* policy, which allows memory reads to reorder with respect to each other in the absence or a sync operation or other ordering event (e.g. the second read operation depends on the result of the first).

OpenMP and Hybrid OpenMP/MPI Applications

The PathScale compilers support OpenMP, but the SiCortex GNU compilers do not. If your code contains OpenMP directives or library calls, compile with the compilers' `-mp` option.

Hybrid OpenMP/MPI applications run on Sicortex systems as long as only the main thread makes MPI calls. `MPI_THREAD_FUNNELED` is the technical term that describes the model to which the SiCortex MPI library adheres. The bottom line is that if your code contains no MPI calls within threaded loops, your application should run successfully.

Compiler Support

The PathScale compiler suite implements OpenMP with supplied parallel directives, runtime libraries, and environment variables.

Compiling and linking

Use the PathScale compiler's `-mp` flag on both the compile and link lines. Otherwise, the compilers will not honor OpenMP directives embedded in the program or process the source code delimited by the OpenMP conditional compilation sentinels (`[!|C|c|*]$OMP` or `# pragma`).

You can experiment with the compiler's `-OPT:early_mp` option, which forces the compiler to perform loop nest optimization (LNO) after it transforms the code to run under multiple threads, so that the optimization operates on the multithreaded forms of the loops.

When you compile and link with the `mpi*` compiler scripts (see page 119), you needn't specify the MPI library, unless you want to use the debug version. However, when you compile and link with the PathScale compilers directly, you must specify the MPI library, `-lscmpi`, on the link line. For more information, see *Compiling and Linking MPI Applications* on page 118.

Setting the stack size limit for threads

If your threaded Fortran application fails with a `segfault` or similar signal, it may have exceeded the default stack size.

The Fortran compiler allocates function-local data on the stack by default. OpenMP programs have a stack for the main thread of execution and separate stacks for each additional thread. The Fortran runtime environment automatically resizes these stacks as necessary, but the memory reserves may be inadequate. You can set separate stack size limits for the main thread, using the `PSC_STACK_LIMIT` environment variable, and for the additional threads, using the `PSC_OMP_STACK_SIZE` environment variable. These environment variables control the amount of memory the Fortran runtime environment can use to resize the stacks.

☼ `PSC_STACK_LIMIT` applies to Fortran programs only. For C/C++ programs, the stack size for the main thread is typically set using the `ulimit` command. `PSC_OMP_STACK_SIZE` applies to both Fortran and C/C++ programs.

To print out the details on how the runtime environment is computing the stack size to use, set the `PSC_STACK_VERBOSE` environment variable before you run your application. It works for both `PSC_STACK_LIMIT` and `PSC_OMP_STACK_SIZE`.

For complete details on using these environment variables, see the *PathScale Compiler Suite User Guide* at www.pathscale.com/docs.html.

Setting the number of threads

The OMP_NUM_THREADS environment variable specifies the number of threads to spawn per process. If you don't specify OMP_NUM_THREADS at runtime, the effective value is 6—the number of processors per node.

OMP_NUM_THREADS values greater than six indicate oversubscription, and increasing values beyond six will probably degrade performance. However, setting OMP_NUM_THREADS=N, where 1 ≤ N ≤ 6, enables you to experiment with scaling. To do so, set the OMP_NUM_THREADS environment variable before you run your application.

```
$ export OMP_NUM_THREADS=4  # bash shell

$ setenv OMP_NUM_THREADS 4  # csh shell
```

Launching a hybrid OpenMP/MPI job using srun

You launch hybrid OpenMP/MPI jobs the same as you do MPI jobs using srun. First you need to determine how many -n tasks to specify for the job, which is obtained by:

total number of nodes × 6/OMP_NUM_THREADS

For example, to run a hybrid OpenMP/MPI application that runs thirty-two MPI processes, one per node, with three threads per process:

```
$ export OMP_NUM_THREADS=3
$ srun -p sc1-mon6 -N 32 -n 64 ./myomp_mpi_app
```

To run a hybrid OpenMP/MPI application that runs thirty-two MPI processes, one per node, with six threads per process:

```
$ export OMP_NUM_THREADS=6
$ srun -p sc1-mon6 -N 32 -n 32 ./myomp_mpi_app
```

SLURM's resource manager imports whatever environment you export before running the job, then starts the MPI processes on the nodes. OpenMP takes care of the threads.

Chapter 9 Processor and Memory System Functional Features

In this section:

- Node Details
- Memory System Operation
  - L1 Data Cache Stalls
  - L2 Cache Memory Mapping Process

This chapter describes the functional features of node's components and how they impact application performance.

Node Details

The SiCortex node (Figure 4) is the heart of every SiCortex System. Each node contains six MIPS64® processors. These processors support both n32 and n64 ABI modes (for details, see page 135), and they provide data compatibility with x86_64 Linux systems: little-endian data (see page 136), IEEE 754 floating point data, and x86_64 C integer sizes.

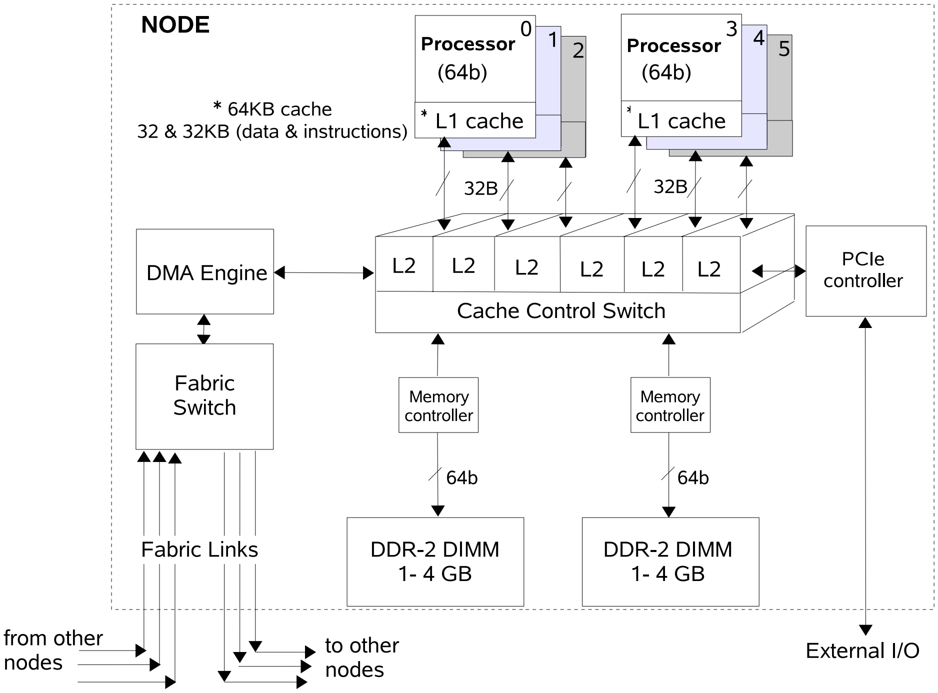

Figure 4. Architecture of the SiCortex node

- Each 64-bit MIPS processor runs at 500MHz. Every cycle, each processor can issue two instructions
  - One integer computation and one floating point computation
  - One floating point computation and one floating point load/store instruction

    One of the dual instructions can be a double precision floating point multiply/add computation.
- Each L1 instruction and data cache is 4-way set associative. Instructions and data each use 32KB of the aggregate 64KB cache.
- The shared L2 cache is 2-way set associative and implements cache coherency between the processors. Each processor has 256KB of the aggregate 1.5MB cache.
- Depending on the memory option selected, each node may have 2, 4, or 8 GB of main memory.
- The Fabric Links transfer aligned 64-bit data, via the DMA engine and the Interconnect Fabric (not shown), to and from each node. Data is transferred in packets, each of which can contain up to 128 bytes of data.

Memory System Operation

Cache operation significantly impacts application performance, so it’s important to understand how the memory system works.

The memory system enforces a policy of data exclusivity that prevents multiple processors from simultaneously accessing an application’s data structures. To enable simultaneous access of an application’s read-only, common data, that data needs to be loaded separately into each processor’s L2 cache.

Because each processor’s L1 data cache is a subset of its L2 cache:

- Reads of main memory always fill both the L2 cache and the L1 data cache.
- Any data that is in the L1 data cache is also in the L2 cache.
- Data removed from the L2 cache is also removed from the L1 data cache, if it is there.

L1 Data Cache Stalls

The processor's L1 data cache operates under a *hits under misses* policy. This means that successful accesses (hits) to the L1 data cache can continue while a single access remains pending (miss). A second access that misses in the L1 data cache causes a stall that lasts until that access finishes successfully. The wait time depends on whether the processor must access the L2 cache (≈12 cycles/24 ns) or main memory (≈45 cycles/90 ns). This policy can impact compiler behavior (for details, see page 29).

Because the L2 cache works with 64-byte data and the L1 data cache works with 32-byte data (that is doubleword aligned), looping to fill arrays has an associated hit/miss pattern. For example, a simple loop that reads each byte to fill an array has this repeating hit/miss pattern:

$$\text{L2/L1 miss}_{\text{B0}}, \text{ L1 hit}_{\text{B1-31}}, \text{ L1 miss/L2 hit}_{\text{B32}}, \text{ L1 hit}_{\text{B33-63}}$$

Data returning to the L1 data cache blocks the cache, stalling other cache accesses for three cycles, until the read/write operation completes and the data is available in the cache. So it's much more efficient to keep the processor busy by issuing compute operations* between read/write operations than to issue consecutive read/write operations guaranteed to stall and cause the processor to idle.

L2 Cache Memory Mapping Process

The L2 cache uses a hash function to calculate the L2 cache index. To do so, the hash function uses physical address bits `16:7 XOR 26:17` concatenated with bit `6`, which identifies which of the two main memory modules is the source/target.

Among other things, hashing reduces array collisions. For example, imagine computing `a=b+c` across two 64KB arrays and storing the results in a third. Without hashing, both of the original arrays map to the same index, but with hashing, each maps to a different index.

Knowing how the L2 cache hash function operates, you can use the processors' performance counters to identify access patterns in your code responsible for excessive L2 misses and fix them.

* The compute operations must not depend on the result of the preceding memory operation.

Chapter 10 Understanding the Application Binary Interfaces

In this section:

What is an ABI?

An Application Binary Interface (ABI) is a set of runtime conventions that all tools which deal with a program’s binary representations follow. These conventions include the way programs use processor registers, represent data types and memory addresses, and pass arguments when calling functions.

An ABI is specific to its processor’s instruction set. A well-defined ABI ensures that compiled programs work properly with the operating system and that object code created by different compilers can interoperate.

The tools that must conform to the ABI include compilers, assemblers, linkers, and language runtime support. Different sets of tools are interoperable if they implement the same ABI and generate files that can be used in the same program.

A compiler conforms to an ABI if it generates code that follows all of the specifications defined by the ABI. A library conforms to an ABI if its calling interfaces follow all of the specifications defined by the ABI. An application conforms to an ABI if it was built using tools that conform to the ABI, and it does not contain source code that specifically changes the behavior specified by the ABI.

The System supports two ABIs: n32 and n64 (default). The main difference between them is the number of bits each uses to represent addresses. The n64 ABI uses 64 bits (the pointer size is 8 bytes), and the n32 ABI uses 32 bits. So n64-built applications can use a larger virtual address space. Both ABIs use the 64-bit registers available in the MIPS64 architecture, and the n32 ABI imposes no limitations on the use of 64-bit data types.

The SiCortex compiler suites can generate code conforming to either ABI, selected at compile time. By default, the compilers use the n64 ABI. When linking, all of the objects must have been compiled using the same ABI. The System includes both n64 and n32 versions of the libraries.

Programmers who have used other MIPS systems may be familiar with the o32 MIPS ABI. This is a 32-bit ABI and predates the availability of 64-bit registers in the MIPS architecture family. The System does not support the o32 ABI.

For an in-depth treatise on the MIPS software standards implemented by the ABIs see, Sweetman, Dominic (2006), *See MIPS® Run; Second Edition.* San Francisco. Morgan Kaufmann Publishers.

Data Formats

The System's processors operate on little-endian formatted data, which stores the least significant bits of the data in the lowest byte address. Floating-point data uses the IEEE 754 representation.

The System's representation of binary data is identical to that used by the x86_64 ABI.

The pathf95 compiler's `-byteswapio` option enables an application to transparently read and write big-endian data.

Table 22 shows the differences between the n32 and n64 ABIs.

Table 22. Feature comparisons between ABIs

Feature	**n32**	**n64**[a]
Compiler	gcc/PathScale[b]	gcc/PathScale[b]
# FP registers	32	32
# Argument registers	8	8
Debug format	dwarf	dwarf
ISAs supported	MIPS 64	MIPS 64
32/64 mode	64b regs/32b addrs	64b

[a] the n64 ABI is the default for the supplied tools.

[b] Includes scgcc and scpathscale compiler variants.

Table 23 shows the differences in size between n32 and n64 ABIs for C and C++ data types.

Table 23. C and C++ data type size differences in bytes

Data Type	**n32**	**n64**
char	1	1
short int	2	2
int	4	4
long int	**4**	**8**
long long int	8	8
pointer	**4**	**8**
float	4	4
double	8	8
long double	16	16
enum	4	4

Table 24 shows the differences in size between n32 and n64 ABIs for Fortran data types:

Table 24. Fortran data type size differences in bytes

Data Type	**n32**	**n64**
LOGICAL[a]	4	4
BYTE	1	1
INTEGER[a]	4	4
REAL[a]	4	4

Table 24. Fortran data type size differences in bytes (Cont'd)

Data Type	n32	n64
COMPLEX[a]	4 real +4 imaginary	4 real +4 imaginary
POINTER	4	8

[a] For data types LOGICAL, INTEGER, REAL, and COMPLEX, you can specify variable size (in bytes 1|2|4|8) by appending *<bytes> to the end of the data type name; for example REAL*8.

Table 25 shows data type correspondence between Fortran, C, and C++ languages.

Table 25. Data type compatibility

Fortran	C/C++
LOGICAL, LOGICAL*4[a]	int/int, bool
BYTE	char
INTEGER, INTEGER*4[a]	int
REAL, REAL*4[a]	float
COMPLEX, COMPLEX*4[a]	struct {float r, i;}
CHARACTER	char
CHARACTER*n	see the code example for Fortran Calling C in *Interlanguage Coding Examples on* page 144
TYPE	typedef struct

[a] For data types LOGICAL, INTEGER, REAL, and COMPLEX, you can specify variable size (in bytes 1|2|4|8) by appending *<bytes> to the end of the data type name; for example REAL*8.

Register Usage

The n64 and n32 ABIs implement these rules of register usage:

- Up to eight arguments can be passed in registers.
- Argument slots and registers are 64 bits. Shorter arguments are promoted to 64 bits exactly as if they were loaded into a register.
- For arguments passed in registers, the caller allocates no stack space.
- Any floating point value occupying by itself one of the first eight argument slots is passed in a floating point register. This rule includes aligned double fields in arrays and structures, as long as

the fields are neither members of a union nor variable arguments to printf() or similar variable-argument functions.

Function prototypes are required to satisfy the rule that caller and callee code must be compiled knowing the exact number of arguments and data types they pass to each other.

Alignment Rules

The n64 and n32 ABIs implement these alignment rules:

- All stack areas are quadword (16-byte) aligned.

  Since both n32 and n64 ABIs recognize 16-byte basic objects (such as long double floating point), and these objects are 16-byte aligned, the stack must be realigned to a multiple of 16 bytes for each function's frame.

- All parameter registers are doubleword (8-byte) aligned.
- All basic data types are aligned in memory on boundaries that match their byte-size.

  Quad-precision floating-point parameters (long double and REAL*16) are quadword (16-byte) aligned.

Overriding the Default ABI

The compilers and linkers generate n64 objects by default. To create n32 objects instead, specify that to the compiler by appending 32 to the end of the compiler name: for example pathcc32, pathf9532, pathCC32, scpathcc32, scpathf9532, scpathCC32, and so on.

☼ If building natively on the nodes, the System Administrator must first have mounted the n32 buildroot on the nodes where you intend to build your n32 application.

The System provides sets of libraries built for both the n64 and n32 ABIs. On the nodes the n64 libraries are in /lib64 and /usr/lib64, and the n32 libraries are visible in /lib32 and /usr/lib32, if the System Administrator has mounted the n32 buildroot on the nodes. Both /lib and /usr/lib are symbolic links to the n64 directories.

See *Running n32 Applications* on page 22 for instructions on how to run n32 applications.

Interlanguage Programming Considerations

Though all of the compilers provided with the SiCortex Development Software Suites generate object files that are interoperable, users need to be aware of some compiler behaviors that can impact mixed language applications.

Compiler Name Mangling

When generating linker symbols, both the Fortran and C++ compilers *mangle* or *decorate* identifiers by adding underscore characters to them. The Fortran compiler does this so that a Fortran function with a name like `open` doesn't collide with the C library function `open` during linking.

- Linking C++ and Fortran code

  Use the `"extern C"` declaration, as described here, to make the C++ compiler behave like C with respect to that declaration, and then proceed as if linking C and Fortran.

  The C++ compiler mangles symbol names to implement overloading, and it adds to data structures information (such as virtual table pointers) that other languages cannot understand. Using the `extern “C”` declaration within C++ source code to generate a C-compatible interface is the easiest way to interface with C and Fortran.

  To call a C function from a C++ program, declare the C function with the `extern “C”` keyword. Likewise, to call a C++ function from a C program, use the same `extern “C”` keyword to declare the C++ function. To call a Fortran procedure, declare the procedure with the `extern “C”` keyword, then follow the Fortran naming conventions for appending underscores to global names (see C++ code example on page 142).

- Linking C and Fortran code

  Use one of three options:

  - Add underscores to the name within the C source code to match the Fortran-generated name that the linker will see. By default, Fortran appends one underscore to a name that contains no underscore and two underscores to a name that already contains an underscore. For example, `x` becomes `x_`, `x_y` becomes `x_y__`, and `x_` becomes `x___`.

  Because the libraries supplied with the SiCortex software comply with these name-mangling rules, we strongly recommend that you avoid using compiler flags that alter the compiler's default method of applying underscores to symbol names. If you encounter link problems that you cannot resolve any other way, try the compiler's `-fno-second-underscore` option.

- Compile with the Fortran `-fdecorate` option and provide a file that instructs the compiler how to map specific Fortran identifiers onto linker symbols. For example, an `-fdecorate` file containing the lines:

  ```
  x x
  cprovided_y y
  ```

  causes a Fortran identifier `x` to generate a linker name `x`, and a Fortran identifier `cprovided_y` to generate a linker name `y`.

- Disable decoration by compiling with the Fortran `-fno-underscoring` option. While a simple solution because it forces identical Fortran-generated and C-generated linker names, it is also risky because it makes it possible for a Fortran identifier to collide with a name in the C library or in the Fortran runtime libraries.

  In general, all Fortran-generated code in an application must use the same underscoring strategy, so this option is not feasible unless certain Fortran-generated libraries cannot be recompiled using the `-fno-underscoring` option.

Named Common Blocks in Fortran

- Fortran named common blocks are represented in C/C++ by a `struct` whose members correspond to those in the common block.

- The name of the C/C++ struct must match the Fortran-generated common block name (using one of the three *Linking C and Fortran code* options described on page 140). By default, Fortran generates `_BLNK__` for blank common.

  Fortran Common Block:

  ```
    INTEGER I
    COMPLEX C
    DOUBLE COMPLEX CD
    DOUBLE PRECISION D
    COMMON /COM/ i, c, cd, d
  ```

  Represented in C:

  ```
  extern struct{
    int i;
    struct {float real, imag;} c;
    struct {double real, imag;} cd;
    double d;
  }com_;
  ```

Represented in C++:

```
extern "C" struct{
  int i;
  struct {float real, imag;} c;
  struct {double real, imag;} cd;
  double d;
}com_;
```

Mismatching Data Types

- Make sure to match your argument data types. See Table 25 on page 138.
- Neither Fortran nor C programs can call C++ functions that contain objects with constructors/destructors, unless the initialization in the main program is performed from a C++ program in which constructors/destructors are properly initialized.
- To use the same header file in a mixed C and C++ application, enclose the contents of the header file in an `extern "C" {}` declaration, and then test for the `__cplusplus` macro:

```
#ifndef STDIO_H
#define STDIO_H

#ifdef __cplusplus
extern "C"{
#endif /*__cplusplus */
.
./* Functions and data types defined */
.
#ifdef __cplusplus
}
#endef /* __cplusplus */

# endif
```

Passing Arguments

- Fortran passes arguments by reference. C/C++ passes variables by value, except for C++ variables that are explicitly declared as references.

When calling from C/C++ to Fortran, use the `&` operator to pass variables by address. When calling from Fortran to C/C++, declare the C/C++ parameters with the * operator to specify that an address was passed.

Alternatively, you can use the Fortran `%VAL` function to pass Fortran arguments explicitly by value to C/C++ functions. To do so, in the Fortran program you enclose the parameter(s) to pass within `%VAL()`:

```
INTEGER*!    I
LOGICAL*1   BVAR
CALL CVALUE (%VAL (I), %VAL (BVAR))
```

- For strings declared as `CHARACTER` in Fortran, both a pointer to the first character in the string and an integer argument representing the length of the string are passed to the calling function. (The Fortran `CHARACTER` type is represented as an array of the C `char` type, but it is not guaranteed to be `\0` terminated.)

  The compiler places the length argument at the end of the parameter list, following all other formal arguments. The length argument is passed by value, not by reference.

- Fortran Cray pointers, declared with the `pointer` *statement*, correspond to C pointers, but Fortran 90 pointers, declared with the `pointer` *attribute*, are unique to Fortran. (Cray pointers are data type extensions used to specify dynamic objects.) You declare these pointer types this way:

Cray:	`POINTER (<POINTER>, <POINTEE>)`
Fortran 90:	`POINTER :: [OBJECT_NAME]`

Returning Values

- Fortran, C, and C++ define functions and subroutines differently. Fortran distinguishes between functions and subroutines according to whether or not a value is returned. A Fortran subroutine is equivalent to a C void function, and a Fortran function is equivalent to a C function that returns a value.

  To call a C or C++ function from a Fortran program, call the C/C++ function as a function if it returns a value, but as a subroutine if it doesn't.

  To call a Fortran function from a C/C++ program, make sure the called function returns the same data type. (See Table 25, *Data type compatibility, on* page 138)

- Use these C/C++ constructs for working with the Fortran `COMPLEX` data type:

Fortran	**C/C++**
COMPLEX – COMPLEX*8[a]	struct {float r, i;} x;
COMPLEX*16	struct {double d, di;} x;

  [a] See Table 25 on page 138 for details.

Array and Structure Considerations

- The default initial index value for arrays differs between Fortran and C/C++. Fortran array indexes start at `1`, and C/C++ array indexes start at `0`. You can declare your Fortran 90 array indexes to start at `0` to comply with the C/C++ convention.

- Wherever possible, use Fortran 77 type arrays because Fortran 90 arrays contain information that C cannot understand. For example, Fortran arguments `a(5,6)`, `a(n)`, or `a(1:*)` pass a simple pointer that corresponds well to a C array, but neither `a(:,:)` nor an allocatable array nor a Fortran 90 pointer array corresponds to anything in C.
- Use the Fortran `sequence` keyword to increase the likelihood that the layout of a Fortran 90 structure will match that of a C structure.

Interlanguage Coding Examples

- Fortran Calling C

In the C function, `csub_`, called by the Fortran main program, `fcallsc`, each argument is defined as a pointer, and the C function name, `csub_`, is in lower case and has a trailing underscore character.

```
  PROGRAM FCALLSC
    INTEGER :: IVAR = 7
    INTEGER :: RET
    REAL :: RVAR = 7.1
    CHARACTER (LEN=8) :: CVAR= 'A STRING'

    INTEGER, EXTERNAL :: CSUB

    WRITE(6,*) 'IN FCALLSC: IVAR, RVAR, CVAR=', IVAR
&   RVAR, CVAR
    RET = CSUB (IVAR,%VAL(IVAR), RVAR, CVAR)
    WRITE(6,*) 'IN FCALLSC: GOT RETURN CODE=', RET

    STOP
    END PROGRAM FCALLSC
```

```
  /* csub_ */
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>

    int csub_(int *ivar, int ivarval, float *rval,
               char *cvar, int lencvar)
    {
    char *ccvar;

    printf("In csub input length of char=%d\n",
           lencvar);
    if (!(ccvar = malloc(lenccvar+1))){
         printf("csub: malloc failure\n");
         exit -1;
        }
    strncpy(ccvar, cvar, lencvar);
    ccvar[lencvar] = '\0';
    printf("In csub: ivar, ivarval, rvar, ccvar=%d %d %f
              %s\n", *ivar, ivarval, *rvar, ccvar);
    free(ccvar);
```

```
return 0;
}
```

- C Calling Fortran

  In the C main program, `ccallsf`, the call uses the `&` operator to reference the variables, and the Fortran subroutine name, `fsub`, is in lower case and has a trailing underscore character:

```
/* ccallsf */
#include <stdio.h>
#include <string.h>

int main()
{
  int ivar = 7;
  float rvar = 7.1;
  char *cvar = "A string";
  int lencvar = strlen(cvar);

  extern void fsub_(int *, float *, char *, int);

  printf("In ccallsf: ivar, rvar, cvar=%d %f
           %s\n", ivar, rvar, cvar);
  fsub_(&ivar, &rvar, cvar, lencvar);
  return 0;
}
```

```
    SUBROUTINE FSUB(IVAR, RVAR, CVAR)
    INTEGER, INTENT(IN) :: IVAR
    REAL, INTENT(IN) :: RVAR
    CHARACTER(LEN=*), INTENT(IN) :: CVAR

    WRITE(6,*) 'IN FUB, LENGTH OF CVAR =', LEN(CVAR)
    WRITE(6,*) 'IN FUB: IVAR, RVAR, CVAR=', IVAR,
&   RVAR, CVAR

    RETURN
    END SUBROUTINE FSUB
```

Appendix A SLURM I/O Buffering

In this Appendix:

SLURM I/O Paths

When you start a job with `srun`, SLURM reads `stdin` from your terminal and broadcasts it to each task. SLURM also collects `stdout` and `stderr` from each task and returns it to your terminal. (A task is a user program that runs on one or more compute nodes. In this context, *task* is synonymous with *process*.)

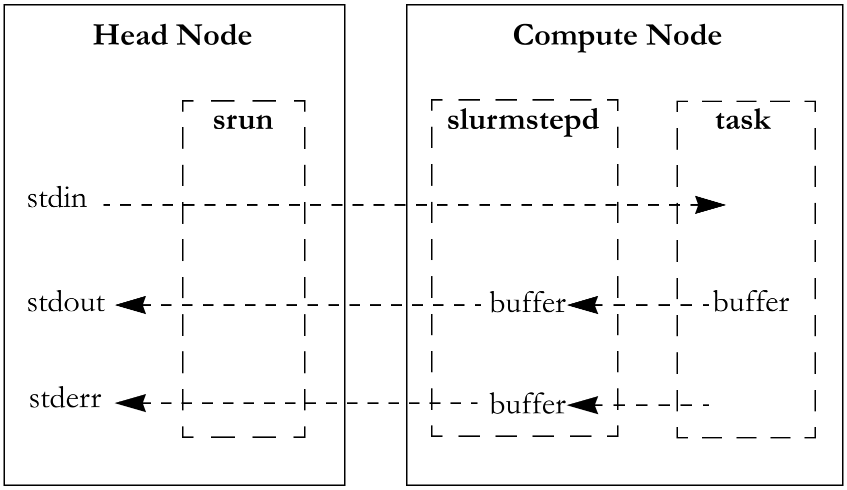

Figure 5. SLURM default I/O paths

Figure 5 shows the SLURM I/O paths for `stdin`, `stdout`, and `stderr`. srun, the program that starts SLURM jobs, runs on the head node, and `slurmstepd`, a daemon program that runs on each compute node, manages the tasks that run on the node.

As shown in Figure 5 on page 147, `stdin`, `stdout`, and `stderr` do not pass directly between `srun` and the task. Instead, each passes through `slurmstepd` on the compute node. And, if data is written to `stdout`, by default, it is buffered by both the task and `slurmstepd`.

Buffering Basics

Buffering is a strategy for managing I/O. When an I/O stream (like stdout) is buffered, delivery of data written to the stream is delayed because the data is stored in a buffer in memory and delivered later, when it is more efficient or convenient to complete the I/O operation.

By default, `stdin` is not buffered, but `stdout` is.

Buffering stdout in the task

By default, all Linux programs buffer `stdout`. This is true regardless of whether the program is running under SLURM. There are two kinds of buffering:

- Line buffering

  When an I/O stream is line buffered, the Linux I/O libraries collect bytes written to the stream in a buffer until a complete line has been written (indicated by a newline (`\n`) character). Then the entire line is delivered to its destination.

- Block buffering

  When an I/O stream is block buffered, the I/O libraries collect data written to the stream in a buffer until some fixed amount (for example, 4096 bytes) has been collected. Then the entire block of data is delivered to its destination.

By default, `stdout` is line buffered when it is connected to a terminal and block buffered when it is connected to something else, such as a file or another program.

Because `stdout` from the task is not directly connected to the user's terminal, but instead to `slurmstepd`, it is, by default, block buffered in the task.

Buffering stdout in slurmstepd

When `slurmstepd` on the compute node receives data from the task's `stdout`, it does not immediately send it on to `srun`. Instead, `slurmstepd` buffers the data internally until it has a complete line, then sends the entire line to `srun`.

stderr By default, the task does not buffer `stderr`. Characters written to `stderr` are immediately sent to `slurmstepd`. However, `slurmstepd` line buffers `stderr` from the task, just as it does `stdout`.

Complications of Buffering

- Lost output

  If a task crashes, any output buffered in the task, but not yet delivered to `slurmstepd`, is lost.

- Delayed output

  In the default case, `stdout` from the task is block buffered. The size of the buffer is likely to be quite a bit larger than a line of output from a task. So a program could write many lines to `stdout` before a block of data is delivered and displayed on the screen. This behavior can cause confusion.

  Some users have concluded that SLURM does not deliver any output to the screen until it has obtained all its input from `stdin`. This is not the case, but appears so if the task generates less than one block of output before all input has been received.

Controlling Buffering

There are several things you can do to control buffering.

- Flush the task buffers

  You can call `fflush(stdout)` at any point in the program to deliver immediately to slurmstepd the data currently buffered in the task.

- Disable task buffering

  You can call `setvbuf (stdout, 0, _IONBF, 0)` at the beginning of a C/C++ program to disable any buffering of `stdout` in the task. With buffering disabled, each byte written to `stdout` is delivered immediately to `slurmstepd`.

  For Fortran programs, call a C routine to invoke `setvbuf (stdout, 0, _IONBF, 0)` at program start. Make sure you add an underscore character to the function's name in your C routine, for example, `extern void setbuf_ ()`, to accommodate the Fortran compiler's name mangling rules (for details, see *Compiler Name Mangling* on page 140).

- Disable line buffering in `slurmstepd`

  You can pass the `--unbuffered` switch to `srun`:

  ```
  srun -p sc1 -N 108 --unbuffered my_program
  ```

  to suppress line buffering in `slurmstepd`. With line buffering disabled, slurmstepd immediately passes all data it receives from the task on `stdout` or `stderr` to `srun`.

Recommended Strategy

- Program/task crashes

  If your program crashes, you need unbuffered output. Either write to `stderr` or disable buffering in the task. Either method prevents loss of output when a program crashes.

  When a task crashes, `slurmstepd` still delivers all output that it receives, so you don't need to use the `--unbuffered` switch.

  When a task hangs, but does not crash, you may need to `scancel` it on the compute node to flush `slurmstepd`'s line buffers (for details, see *Canceling a Job: scancel and ^C on* page 20).

- Program monitoring

  For new program development, the best way to generate monitoring output is to write it to `stderr`.

  For existing programs that write monitoring output to `stdout`, disable buffering in the task as previously described.

- Non line-oriented output

  Output that is not line-oriented, such as spinners and curses escapes, must be completely unbuffered. A program that generates such output should either:

  - Write to `stderr`

  or

  - Disable buffering in the task and then invoke srun using the `--unbuffered` switch to suppress line buffering in `slurmstepd`.

Index