Linux® Application Tuning Guide

007–4639–010

New Features in This Manual

This rewrite of the *Linux Application Tuning Guide* supports the SGI ProPack 6 for Linux operating system.

Major Documentation Changes

- Updated links in the Application Guides section in the preface of this manual.
- Added an example and updated the information in "Resetting the Default Stack Size" on page 104.
- Added information about `csh` and `bash` shell commands to "Resetting Virtual Memory Size" on page 105.
- Minor updates and editing changes throughout the manual.

Record of Revision

Version	Description
001	October 2003 Original publication.
002	May 2004 Updated to support the SGI ProPack 3 for Linux release.
003	August 2004 Updated to support the SGI ProPack 3 for Linux Service Pack 1 release.
004	January 2005 Updated to support the SGI ProPack 3 for Linux Service Pack 3 release.
005	August 2005 Updated to support the SGI ProPack 4 for Linux Service Pack 2 release.
006	April 2007 Updated to support the SGI ProPack 5 for Linux Service Pack 1 release.
007	June 2007 Updated to support the SGI ProPack 5 for Linux Service Pack 2 release.
008	October 2007 Updated to support the SGI ProPack 5 for Linux Service Pack 3 release.
009	July 2008 Updated to support the SGI ProPack 6 for Linux release.
010	January 2009 Updated to support the SGI ProPack 6 for Linux Service Pack 2 release.

Contents

About This Document . . . . . **xiii**
Related Publications xiii
Related Operating System Documentation xiii
Hardware Reference Manuals xiv
Application Guides xv
Conventions xv
Obtaining Publications xvi
Reader Comments xvii

1. System Overview **1**
Scalable Computing 1
An Overview of Altix Architecture 1
Altix 3000 Series Systems 2
Altix 4000 Series Systems 3
The Basics of Memory Management 5

2. The SGI Compiling Environment **7**
Compiler Overview 7
Modules 8
Library Overview 9
Static Libraries 9
Dynamic Libraries 9
C/C++ Libraries 10
SHMEM Message Passing Libraries 10
Other Compiling Environment Features 11

3. Performance Analysis and Debugging 13

Determining System Configuration 13

Sources of Performance Problems 14

Profiling with `pfmon` 15

Profiling with `profile.pl` 15

`profile.pl` with MPI programs 16

Using `histx` 16

`histx` Data Collection 16

`histx` Filters 19

`histx` Event Sources and Types of Sampling 19

Using VTune for Remote Sampling 20

Using GuideView 20

Other Performance Tools 21

Debugging Tools 22

Using `ddd` 23

4. Monitoring Tools 27

System Monitoring Tools 27

Hardware Inventory and Usage Commands 27

`hwinfo`(1) Command 28

`topology`(1) Command 28

`gtopology`(1) Command 29

Performance Co-Pilot Monitoring Tools 32

`pmshub`(1) Command 33

`shubstats`(1) Command 34

`linkstat`(1) Command 34

Other Performance Co-Pilot Monitoring Tools 34

System Usage Commands 36

5. Data Placement Tools 43

Data Placement Tools Overview 43

`taskset` Command 45

`dplace` Command 48

Using the `dplace` Command 48

`dplace` for Compute Thread Placement Troubleshooting Case Study 53

`dlook` Command 55

Using the dlook Command 56

Installing NUMA Tools 62

6. Performance Tuning 63

Single Processor Code Tuning 63

Getting the Correct Results 64

Managing Heap Corruption Problems 64

Using Tuned Code 66

Determining Tuning Needs 66

Using Compiler Options Where Possible 67

Tuning the Cache Performance 68

Managing Memory 70

Multiprocessor Code Tuning 70

Data Decomposition 71

Parallelizing Your Code 72

Use MPT 73

Use XPMEM DAPL Library with MPI 73

Use OpenMP 74

OpenMP Nested Parallelism 74

Use Compiler Options 75
Identifying Parallel Opportunities in Existing Code 75
Fixing False Sharing 76
Using `dplace` and `taskset` 77
Environment Variables for Performance Tuning 77
Understanding Parallel Speedup and Amdahl's Law 78
Adding CPUs to Shorten Execution Time 78
Understanding Parallel Speedup 79
Understanding Superlinear Speedup 80
Understanding Amdahl's Law 80
Calculating the Parallel Fraction of a Program 81
Predicting Execution Time with n CPUs 82
Floating-point Programs Performance 82

7. Flexible File I/O 85
FFIO Operation 85
Environment Variables 86
Simple Examples 87
Multithreading Considerations 90
Application Examples 91
Event Tracing 92
System Information and Issues 92

8. I/O Tuning 93
Layout of Filesystems and XVM for Multiple RAIDs 93

9. Suggested Shortcuts and Workarounds 95
Determining Process Placement 95

Example Using pthreads 96
Example Using OpenMP 98
Combination Example (MPI and OpenMP) 99
Resetting System Limits 102
Resetting the File Limit Resource Default 103
Resetting the Default Stack Size 104
Resetting Virtual Memory Size 105
Linux Shared Memory Accounting 106

Index 109

About This Document

This publication provides information about tuning application programs on the SGI Altix 3000 and SGI Altix 4000 families of servers and superclusters running the Linux operating system. Application programs includes Fortran and C programs written with the Intel-provided compilers on SGI Linux systems.

This document does not include information about configuring or tuning your system. For details about those topics, see the *Linux Configuration and Operations Guide.*

This guide is written for experienced programmers, familiar with Linux commands and with either the C or Fortran programming languages. The focus in this document is on achieving the highest possible performance by exploiting the features of your SGI Altix system. The material assumes that you know the basics of software engineering and that you are familiar with standard methods and data structures. If you are new to programming or software design, this guide will **not** be of use to you.

Related Publications

The following publications provide information that can supplement the information in this document.

Related Operating System Documentation

The following documents provide information about Linux implementations on SGI systems:

- *Linux Installation and Getting Started*
- *Linux Configuration and Operations Guide*

  Provides information on how to perform system configuration and operations for SGI ProPack servers.

- *Linux Resource Administration Guide*

  Provides a reference for people who manage the operation of SGI ProPack servers and contains information needed in the administration of various system resource management features such as Comprehensive System Accounting (CSA), Array Services, CPU memory sets and scheduling, and the Cpuset System.

- *SGI ProPack 6 for Linux Service Service Pack 2 Start Here*

  Provides information about the SGI ProPack 6 for Linux Service Pack 2 release.

- *Message Passing Toolkit (MPT) User's Guide*

  Describes industry-standard message passing protocol optimized for SGI computers.

See the release notes which are shipped with your system for a list of other documents that are available. All books are available on the Tech Pubs Library at http://docs.sgi.com.

Release notes for Linux systems are stored in `/usr/share/doc/sgi-scsl-`*versionnumber*`/README.relnotes`.

Hardware Reference Manuals

The following documents provide information about Altix system hardware. For a complete list of current SGI software and hardware manuals, see the *SGI ProPack 6 for Linux Start Here* available at http://docs.sgi.com.

- *SGI Altix 330 System User's Guide*

  Provides an overview of the Altix 330 system components, and it describes how to set up and operate this system.

- *SGI Altix 350 System User's Guide*

  Provides an overview of the Altix 350 system components, and it describes how to set up and operate this system.

- *SGI Altix 3000 User's Guide*

  Provides an overview of the architecture and describes the major components of the SGI Altix 3000 family of servers and superclusters. It also describes the standard procedures for powering up and powering down the system, provides basic troubleshooting information, and includes important safety and regulatory specifications.

- *SGI Altix 3700 Bx2 User's Guide*

  This guide provides an overview of the architecture and descriptions of the major components that compose the SGI Altix 3700 Bx2 family of servers. It also provides

the standard procedures for powering on and powering off the system, basic troubleshooting information, and important safety and regulatory specifications.

- *SGI Altix 4700 User's Guide*

  Provides an overview of the architecture and describes the major components of the SGI Altix 4700 family of servers. It also describes the standard procedures for powering up and powering down the system, provides basic troubleshooting information, and includes important safety and regulatory specifications.

- *Silicon Graphics Prism Visualization System User's Guide*

  Provides an overview of the Silicon Graphics Prism Visualization System components, and it describes how to set up and operate this system.

- *Silicon Graphics Prism Deskside Visualization System User's Guide*

  Provides an overview of the Silicon Graphics Prism Deskside system components, and it describes how to set up and operate this system.

- *Reconfigurable Application-Specific Computing User's Guide*

  Provides information about the SGI reconfigurable application-specific software computing (RASC) program that delivers scalable, configurable computing elements for the SGI Altix family of servers and superclusters.

Application Guides

The following documentation is provided for the compilers and performance tools which run on SGI Linux systems:

- http://sources.redhat.com/gdb/onlinedocs/gdb_toc.html
- http://www.intel.com/cd/software/products/asmo-na/eng/perflib/219780.htm; documentation for Intel compiler products can be downloaded from this website.
- http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm
- Information about the OpenMP Standard can be found at http://openmp.org/wp/.

Conventions

The following conventions are used in this documentation:

[ ]	Brackets enclose optional portions of a command or directive line.
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
...	Ellipses indicate that a preceding element can be repeated.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
variable	Italic typeface denotes variable entries and words or concepts being defined.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.

Obtaining Publications

You can obtain SGI documentation as follows:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, enter `infosearch` at a command line or select **Help > InfoSearch** from the Toolchest.
- On IRIX systems, you can view release notes by entering either `grelnotes` or `relnotes` at a command line.
- On Linux systems, you can view release notes on your system by accessing the `README.txt` file for the product. This is usually located in the `/usr/share/doc/`*productname* directory, although file locations may vary.
- You can view man pages by typing `man` *title* at a command line.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  SGI
  Technical Publications
  1140 East Arques Avenue
  Sunnyvale, CA 94085–4602

SGI values your comments and will respond to them promptly.

Chapter 1

System Overview

Tuning an application involves making your program run its fastest on the available hardware. The first step is to make your program run as efficiently as possible on a single processor system and then consider ways to use parallel processing.

Application tuning is different from system tuning, which involves topics such as disk partitioning, optimizing memory management, and configuration of the system. The *Linux Configuration and Operations Guide* discusses those topics in detail.

This chapter provides an overview of concepts involved in working in parallel computing environments.

Scalable Computing

Scalability is computational power that can grow over a large number of CPUs. Scalability depends on the time between nodes on the system. *Latency* is the time to send the first byte between nodes.

A Symmetric Multiprocessor (SMP) is a parallel programming environment in which all processors have equally fast (symmetric) access to memory. These types of systems are easy to assemble and have limited scalability due to memory access times.

Another parallel environment is that of arrays, or clusters. Any networked computer can participate in a cluster. These are highly scalable, easy to assemble, but are often hard to use. There is no shared memory and there are frequently long latency times.

Massively Parallel Processors (MPPs) have a distributed memory and can scale to thousands of processors; they have large memories and large local memory bandwidth.

Scalable Symmetric Multiprocessors (S^2MPs), as in the ccNUMA environment, combine qualities of SMPs and MPPs. They are logically programmable like an SMP and have MPP-like scability.

An Overview of Altix Architecture

This section provides a brief overview of the SGI Altix 3000 and 4000 series systems.

Altix 3000 Series Systems

In order to optimize your application code, some understanding of the SGI Altix architecture is needed. This section provides a broad overview of the system architecture.

The SGI Altix 3000 family of servers and superclusters can have as many as 256 processors and 2048 gigabytes of memory. It uses Intel's Itanium 2 processors and uses nonuniform memory access (NUMA) in SGI's NUMAflex global shared-memory architecture. An SGI Altix 350 system can have as many as 16 processors and 96 gigabytes of memory.

The NUMAflex design permits modular packaging of CPU, memory, I/O, graphics, and storage into components known as *bricks*. The bricks can then be combined and configured into different systems, based on customer needs.

On Altix 3700 systems, two Itanium processors share a common frontside bus and memory. This constitutes a node in the NUMA architecture. Access to other memory (on another node) by these processors has a higher latency, and slightly different bandwidth characteristics. Two such nodes are packaged together in each computer brick. For a detailed overview, see the *SGI Altix 3000 User's Guide*.

On an SGI Altix 3700 Bx2 system, the CR-brick contains the processors (8 processors per CR-brick) and two internal high-speed routers. The routers connect to other system bricks via NUMAlink cables and expand the compute or memory capacity of the Altix 3700 Bx2. For a detailed overview, see the *SGI Altix 3700 Bx2 User's Guide*.

All Altix 350 systems contain at least one base compute module that contains the following components:

- One or two Intel Itanium 2 processors; each processor has integrated L1, L2, and L3 caches
- Up to 24 GB of local memory
- Four PCI/PCI-X slots
- One IO9 PCI card that comes factory-installed in the lowermost PCI/PCI-X slot

For a detailed overview, see the *SGI Altix 350 System User's Guide*.

The system software consists of a standard Linux distribution (Red Hat) and SGI ProPack, which is an overlay providing additional features such as optimized libraries and enhanced kernel support. See Chapter 2, "The SGI Compiling Environment" on page 7, for details about the compilers and libraries included with the distribution.

Altix 4000 Series Systems

In the new SGI Altix 4000 series systems, functional blades - interchangeable compute, memory, I/O, and special purpose blades in an innovative blade-to-NUMAlink architecture are the basic building blocks for the system. Compute blades with a bandwidth configuration have one processor socket per blade. Compute blades with a density configuration have two processor sockets per blade. Cost-effective compute density is one advantage of this compact blade packaging.

The Altix 4000 series is a family of multiprocessor distributed shared memory (DSM) computer systems that currently scales from 8 to 512 CPU sockets (up to 1,024 processor cores) and can accommodate up to 6TB of globally shared memory in a single system while delivering a teraflop of performance in a small-footprint rack. The SGI Altix 450 currently scales from 2 to 76 cores as a cache-coherent single system image (SSI). In a DSM system, each processor board contains memory that it shares with the other processors in the system. Because the DSM system is modular, it combines the advantages of low entry-level cost with global scalability in processors, memory, and I/O. You can install and operate the Altix 4700 series system in a rack in your lab or server room. Each 42U SGI rack holds from one to four 10U high enclosures that support up to ten processor and I/O sub modules known as "blades." These blades are single printed circuit boards (PCBs) with ASICS, processors, and memory components mounted on a mechanical carrier. The blades slide directly in and out of the Altix 4700 1RU enclosures. Each individual rack unit (IRU) is 10U in height (see Figure 1-1 on page 4).

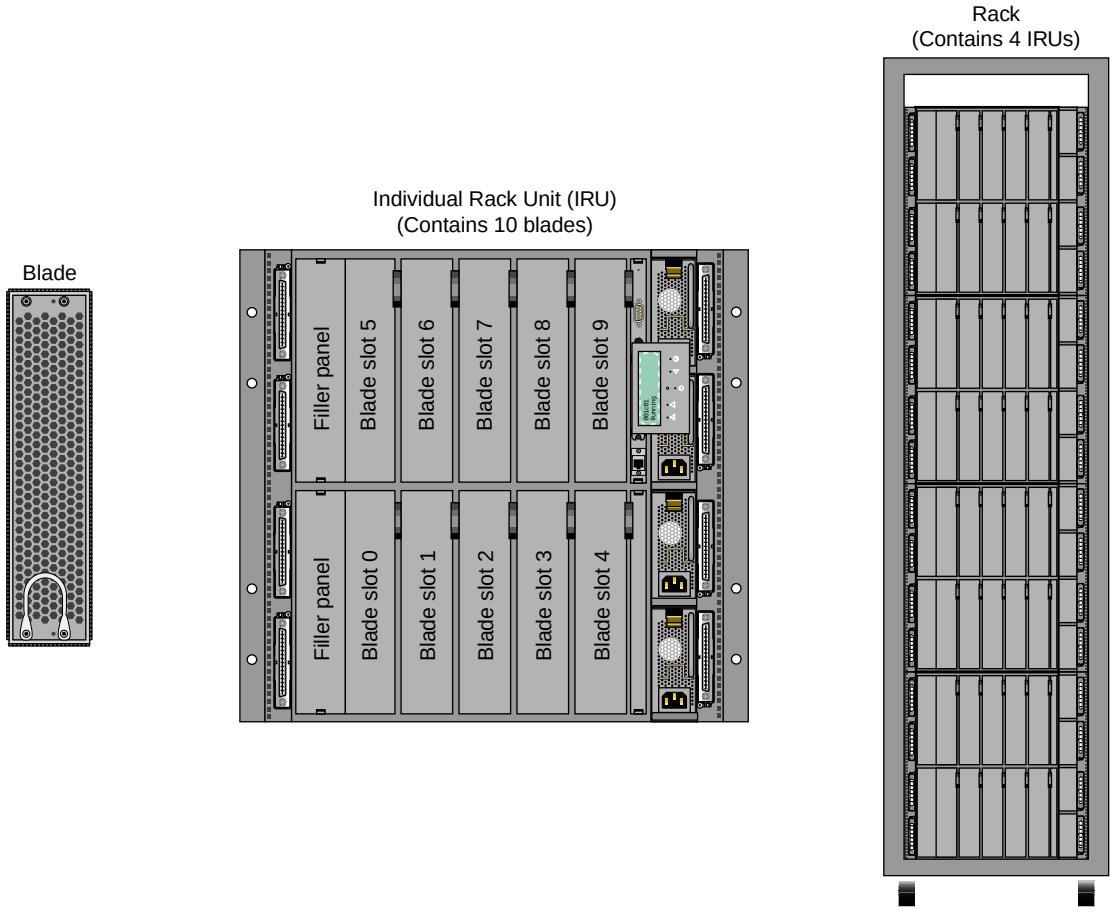

Figure 1-1 Altix 4700 Blade, Individual Rack Unit, and Rack

For more information on this system, see the *SGI Altix 4700 System User's Guide* available on the SGI Technical Publications Library. It provides a detailed overview of the SGI Altix 4700 system components and it describes how to set up and operate the system. For an overview of the new SGI Altix 450 system, see Chapter 3, "System Overview" in the *SGI Altix 450 System User's Guide.*

The Basics of Memory Management

Virtual memory (VM), also known as virtual addressing, is used to divide a system's relatively small amount of physical memory among the potentially larger amount of logical processes in a program. It does this by dividing physical memory into *pages*, and then allocating pages to processes as the pages are needed.

A page is the smallest unit of system memory allocation. Pages are added to a process when either a validity fault occurs or an allocation request is issued. Process size is measured in pages and two sizes are associated with every process: the total size and the resident set size (RSS). The number of pages being used in a process and the process size can be determined by using either the `ps`(1) or the `top`(1) command.

Swap space is used for temporarily saving parts of a program when there is not enough physical memory. The swap space may be on the system drive, on an optional drive, or allocated to a particular file in a filesystem. To avoid swapping, try not to overburden memory. Lack of adequate swap space limits the number and the size of applications that can run simultaneously on the system, and it can limit system performance.

Linux is a demand paging operating system, using a least-recently-used paging algorithm. On a validity fault, pages are mapped into physical memory when first referenced and pages are brought back into memory if swapped out.

The SGI Compiling Environment

This chapter provides an overview of the SGI compiling environment on the SGI Altix family of servers and superclusters and covers the following topics:

- "Compiler Overview" on page 7
- "Modules" on page 8
- "Library Overview" on page 9
- "Other Compiling Environment Features" on page 11

The remainder of this book provides more detailed examples of the use of the SGI compiling environment elements.

Compiler Overview

The Intel Fortran and C/C++ compilers are provided with the SGI Altix distribution. The Fortran compiler supports OpenMP 2.0 and the C/C++ compiler is compatible with `gcc` and the C99 standard.

In addition, the GNU Fortran and C compilers are available on Altix systems.

The following is the general form of the compiler command line (note that the Fortran command is used in this example):

`% ifort [options]` *`filename`*`.extension`

An appropriate filename extension is required for each compiler, according to the programming language used (Fortran, C, C++, or FORTRAN 77).

Some common compiler options are:

- `-o` *filename*: renames the output to *filename*.
- `-g`: produces additional symbol information for debugging.
- `-O`[*level*]: invokes the compiler at different optimization *levels*, from `0` to `3`.
- `-l`*directory_name*: looks for `include` files in *directory_name*.
- `-c`: compiles without invoking the linker; this options produces an `a.o` file only.

Many processors do not handle denormalized arithmetic (for gradual underflow) in hardware. The support of gradual underflow is implementation-dependent. Use the `-ftz` option with the Intel compilers to force the flushing of denormalized results to zero.

Note that frequent gradual underflow arithmetic in a program causes the program to run very slowly, consuming large amounts of system time (this can be determined with the `time` command). In this case, it is best to trace the source of the underflows and fix the code; gradual underflow is often a source of reduced accuracy anyway.. `prctl`(1) allows you to query or control certain process behavior. In a program, `prctl` tracks where floating point errors occur.

Some applications can generate an excessive number of kernel `KERN_WARN` "floating point assist" warning messages. For more information, see http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/srch21@floating%20point%20assist/linux/bks/SGI_Admin/books/LX_ConfigOps_

Modules

A *module* is a user interface that provides for the dynamic modification of a user's environment. By changing the module, a user does not have to change environment variables in order to access different versions of the compilers, loaders, libraries and utilities that are installed on the system.

Modules can be used in the SGI compiling environment to customize the environment. If the use of modules is not available on your system, its installation and use is highly recommended.

To view which modules are available on your system, use the following command (for any shell environment):

```
% module avail
```

To load modules into your environment (for any shell), use the following commands:

```
% module load intel-compilers-latest mpt-1.7.1rel
% module load scsl-1.4.1-1
```

Note: The above commands are for example use only; the actual release numbers may vary depending on the version of the software you are using. See the release notes that are distributed with your system for the pertinent release version numbers.

For details about using modules, see the `module` man page (you must have the `module` man page loaded).

Library Overview

Libraries are files that contain one or more object (`.o`) files. Libraries are used to simplify local software development by hiding compilation details. Libraries are sometimes also called *archives*.

The SGI compiling environment contains several types of libraries; an overview about each library is provided in this subsection.

Static Libraries

Static libraries are used when calls to the library components are satisfied at link time by copying text from the library into the executable. To create a static library, use the `ar`(1), or an archiver command.

To use a static library, include the library name on the compiler's command line. If the library is not in a standard library directory, be sure to use the `-L` option to specify the directory and the `-l` option to specify the library filename.

To build an appplication to have all static versions of standard libraries in the application binary, use the `-static` option on the compiler command line.

Dynamic Libraries

Dynamic libraries are linked into the program at run time and when loaded into memory can be accessed by multiple programs. Dynamic libraries are formed by creating a Dynamic Shared Object (DSO).

Use the link editor command (`ld`(1)) to create a dynamic library from a series of object files or to create a DSO from an existing static library.

To use a dynamic library, include the library on the compiler's command line. If the dynamic library is not in one of the standard library directories, use the `-rpath` compiler option during linking. You must also set the `LD_LIBRARY_PATH` environment variable to the directory where the library is stored before running the executable.

C/C++ Libraries

The following C/C++ libraries are provided with the Intel compiler:

- `libguide.a`, `libguide.so`: for support of OpenMP-based programs.
- `libsvml.a`: short vector math library
- `libirc.a`: Intel's support for Profile-Guided Optimizations (PGO) and CPU dispatch
- `libimf.a`, `libimf.so`: Intel's math library
- `libcprts.a`, `libcprts.so`: Dinkumware C++ library
- `libunwind.a`, `libunwind.so`: Unwinder library
- `libcxa.a`, `libcxa.so`: Intel's runtime support for C++ features

SHMEM Message Passing Libraries

The Shared Memory Access Library (`libsma`) is part of the Message Passing Toolkit (MPT) product on SGI Altix systems. The SHMEM programming model consists of library routines that provide low-latency, high-bandwidth communication for use in highly parallelized, scalable programs. The routines in the SHMEM application programming interface (API) provide a programming model for exchanging data between cooperating parallel processes. The resulting programs are similar in style to Message Passing Interface (MPI) programs. The SHMEM API can be used either alone or in combination with MPI routines in the same parallel program.

A SHMEM program is SPMD (single program, multiple data) in style. The SHMEM processes, called processing elements or PEs, all start at the same time, and they all run the same program. Usually the PEs perform computation on their own subdomains of the larger problem, and periodically communicate with other PEs to exchange information on which the next computation phase depends.

The SHMEM routines minimize the overhead associated with data transfer requests, maximize bandwidth, and minimize data latency. Data latency is the period of time that starts when a PE initiates a transfer of data and ends when a PE can use the data.

SHMEM routines support remote data transfer through put operations, which transfer data to a different PE, get operations, which transfer data from a different PE, and remote pointers, which allow direct references to data objects owned by another PE. Other operations supported are collective broadcast and reduction, barrier

synchronization, and atomic memory operations. An atomic memory operation is an atomic read-and-update operation, such as a fetch-and-increment, on a remote or local data object.

For details about using the SHMEM routines, see the `intro_shmem`(3) man page or the *Message Passing Toolkit (MPT) User's Guide*.

Other Compiling Environment Features

The SGI compiling environment includes several other products as part of its distribution:

- `idb`: the Intel debugger (available if your system is licensed for the Intel compilers). This is a fully symbolic debugger and supports Fortran, C, and C++ debugging.
- `gdb`: the GNU project debugger, which supports C, C++ and Modula-2. It also supports Fortran 95 debugging when the gdbf95 patch is installed.
- `ddd`: a graphical user interface to `gdb` and the other debuggers.
- TotalView: a licensed graphical debugger useful in an MPI environment (see http://www.totalviewtech.com/)

These and other performance analysis tools are discussed in Chapter 3, "Performance Analysis and Debugging" on page 13.

Performance Analysis and Debugging

Tuning an application involves determining the source of performance problems and then rectifying those problems to make your programs run their fastest on the available hardware. Performance gains usually fall into one of three categories of mesured time:

- User CPU time: time accumulated by a user process when it is attached to a CPU and is executing.
- Elapsed (wall-clock) time: the amount of time that passes between the start and the termination of a process.
- System time: the amount of time performing kernel functions like system calls, `sched_yield`, for example, or floating point errors.

Any application tuning process involves:

1. Analyzing and identifying a problem
2. Locating where in the code the problem is
3. Applying an optimization technique

This chapter describes the process of analyzing your code to determine performance bottlenecks. See Chapter 6, "Performance Tuning" on page 63, for details about tuning your application for a single processor system and then tuning it for parallel processing.

Determining System Configuration

One of the first steps in application tuning is to determine the details of the system that you are running. Depending on your system configuration, different options may or may not provide good results.

To determine the details of the system you are running, you can browse files from the `/proc` pseudo-filesystem (see the `proc`(5) man page for details). Following is some of the information you can obtain:

- `/proc/cpuinfo`: displays processor information, one entry per processor. Use this to determine clock speed and processor stepping.

- `/proc/meminfo`: provides a global view of system memory usage, such as total memory, free memory, swap space, and so on.
- `/proc/discontig`: shows memory usage (in pages).
- `/proc/pal/cpu0/cache_info`: provides detailed information about L1, L2, and L3 cache structure, such as size, latency, associativity, line size, and so on. Other files in `/proc/pal/cpu0` provide information about the Translation Lookaside Buffer (TLB) structure, clock ratios, and other details.
- `/proc/version`: provides information about the installed kernel.
- `/proc/perfmon`: if this file does not exist in`/proc` (that is, if it has not been exported), performance counters have not been started by the kernel and none of the performance tools that use the counters will work.
- `/proc/mounts`: provides details about the filesystems that are currently mounted.
- `/proc/modules`: contains details about currently installed kernel modules.

You can also use the `uname` command, which returns the kernel version and other machine information. In addition, the `topology` command displays system configuration information. See Chapter 4, "Monitoring Tools" on page 27 for more information.

Sources of Performance Problems

There are usually three areas of program execution that can have performance slowdowns:

- CPU-bound processes: processes that are performing slow operations (such as `sqrt` or floating-point divides) or non-pipelined operations such as switching between add and multiply operations.
- Memory-bound processes: code which uses poor memory strides, occurrences of page thrashing or cache misses, or poor data placement in NUMA systems.
- I/O-bound processes: processes which are waiting on synchronous I/O, formatted I/O, or when there is library or system level buffering.

Several profiling tools can help pinpoint where performance slowdowns are occurring. The following sections describe some of these tools.

Profiling with pfmon

The pfmon tool is a performance monitoring tool designed for Linux. It uses the Itanium Performance Monitoring Unit (PMU) to count and sample unmodified binaries. In addition, it can be used for the following tasks:

- To monitor unmodified binaries in its per-CPU mode.
- To run system-wide monitoring sessions. Such sessions are active across all processes executing on a given CPU.
- Launch a system-wide session on a dedicated CPU or a set of CPUs in parallel.
- Monitor activities happening at the user level or at the kernel level.
- Collect basic hardware event counts (There are 477 hardware events.)
- Sample program or system execution, monitoring up to four events at a time.

To see a list of available options, use the pfmon -help command. You can only run pfmon one CPU or conflict at a time.

Profiling with profile.pl

The profile.pl script handles the entire user program profiling process. Typical usage is as follows:

% **profile.pl -c0-3 -x6** *command args*

This script designates processors 0 through 3. The -x6 option is necessary only for OpenMP codes.

The result is a profile taken on the CPU_CYCLES PMU event and placed into profile.out. This script also supports profiling on other events such as IA64_INST_RETIRED, L3_MISSES, and so on; see pfmon -l for a complete list of PMU events. The script handles running the command under the performance monitor, creating a map file of symbol names and addresses from the executable and any associated dynamic libraries, and running the profile analyzer.

See the profile.pl(1), analyze.pl(1), and makemap.pl(1) man pages for details. You can run profile.pl one at a time per CPU or conflict. Profiles all processes on the specified CPUs.

`profile.pl` with MPI programs

For MPI programs, use the `profile.pl` command with the `-s1` option, as in the following example:

```
% mpirun -np 4 profile.pl -s1 -c0-3 test_prog </dev/null
```

The use of `/dev/null` ensures that MPI programs run in the background without asking for TTY input.

Using `histx`

The `histx` software is a set of tools used to assist with application performance analysis. It includes three data collection programs and three filters for performance data post-processing and display. The following sections describe this set of tools.

`histx` Data Collection

Three programs can be used to gather data for later profiling:

- `histx`: A profiling tool that can sample either the program counter or the call stack.

  The `histx` data collection programs monitors child processes only, not all proccesses on a CPU like `pfmon`. It will not show the profile conflicts that the `pfmon` command shows.

  The syntax of the `histx` command is as, as follows:

`histx [-b` *width*`] [-f] [-e` *source*`] [-h] [-k] -o` *file* `[-s` *type*`] [-t` *signo*`] command args...`

The `histx` command accepts the following options:

`-b` *width*	Specifies bin bits when using instruction pointer sampling: 16,32 or 64 (default: 16).
`-e` *source*	Specifies event source (default: timer@1).
`-f`	Follow fork (default: off).
`-h`	This message (command not run).
`-k`	Also count kernel events for program source (default: off).
`-o` *file*	Sends output to `file`*.prog.pid*. (REQUIRED).

`-s` *type*	Includes line level counts in instruction pointer sampling report (default: off).
`-t` *signo*	'Toggles' signal number (default: none).

- `lipfpm`: Reports counts of desired events for the entire run of a program.

  The syntax of the `lipfpm` command is as, as follows:

```
lipfpm [-c name] [-e name]* [-f] [-i] [-h] [-k] [-l] [-o path] [-p] command args...
```

  The `lipfpm` command accepts the following options:

`-c` *name*	Requests named collection of events; may not be used with `-i` or `-e` arguments.
`-e` *name*	Specifies events to monitor (for event names see Intel documents).
`-f`	Follow fork (default: off).
`-i`	Specify events interactively, as follows: – Use space bar or `Tab` key to display next event. – Use `Backspace` key to display previous event. – Use `Enter` key to select event. – Type letters to skip to events starting with the same letters – Note that `Ctrl - c`, and so on, are treated as letters. – Use the `Esc` key to finish.
`-h`	This message (command not run)
`-k`	Counts at privilege level 0 as well (default: off)
`-l`	Lists names of all events (other arguments are ignored).
`-o` *path*	Send output to path.*cmd.pid* instead of standard output.

`-p`	Produces easier to parse output.

When using the `lipfpm` command, you can specify up to four events at a time. For MPI codes, the -f option is required. Event names are specified slightly differently than in the `pfmon` command.The `-c` options shows the named collection of events, as follows:

Event	Description
`mi`	Retired M and I type instructions
`mi_nop`	Retired M and I type NOP instructions
`fb`	Retired F and B type instructions
`fb_nop`	Retired F and B type NOP instructions
`dlatNNN`	Times L1D miss latency exceeded NNN
`dtlb`	DTLB misses
`ilatNNN`	Times L1I miss latency exceeded NNN
`itlb`	ITLB misses
`bw`	Counters associated with (read) bandwidth

Sample output from the `lipfpm` command is, as follows:

```
% lipfpm -c bw stream.1
Function    Rate (MB/s)    Avg time     Min time     Max time
Copy:        3188.8937       0.0216       0.0216       0.0217
Scale:       3154.0994       0.0218       0.0218       0.0219
Add:         3784.2948       0.0273       0.0273       0.0274
Triad:       3822.2504       0.0270       0.0270       0.0272

lipfpm summary
====== =======
L1 Data Cache Read Misses -- all L1D read misses will be
counted..................................................... 10791782
L2 Misses................................................... 55595108
L3 Reads -- L3 Load Misses (excludes reads for ownership
used to satisfy stores)..................................... 55252613
CPU Cycles.................................................. 3022194261
Average read MB/s requested by L1D.......................... 342.801
Average MB/s requested by L2................................ 3531.96
Average data read MB/s requested by L3...................... 3510.2
```

- `samppm`: Samples selected counter values at a rate specified by the user.

`histx` Filters

Three programs can be used to generate reports from the `histx` data collection commands:

- `iprep`: Generates a report from one or more raw sampling reports produced by `histx`.
- `csrep`: Generates a butterfly report from one or more raw call stack sampling reports produced by `histx`.
- `dumppm`: Generates a human-readable or script-readable tabular listing from binary files produced by `samppm`.

`histx` Event Sources and Types of Sampling

The following list describes the event sources and types of sampling for the `histx` program.

Event Sources	Description
timer@N	Profiling timer events. A sample is recorded every N ticks.
pm:*event*@N	Performance monitor events. A sample is recorded whenever the number of occurrences of *event* is N larger than the number of occurrences at the time of the previous sample.
dlatM@N	A sample is recorded whenever the number of loads whose latency exceeded M cycles is N larger than the number at the time of the previous sample. M must be a power of 2 between 4 and 4096.

Types of sample are, as follows:

Types of Sampling	Description
ip	Sample instruction pointer

callstack[N]	Sample callstack. N, if given, specifies the maximum callstack depth (default: 8)

Using VTune for Remote Sampling

The Intel VTune performance analyzer does remote sampling experiments. The VTune data collector runs on the Linux system and an accompanying GUI runs on an IA-32 Windows machine, which is used for analyzing the results. The version of VTune that runs on Linux does not have the full set of options of the Windows GUI.

For details about using VTune, see the following URL:

http://developer.intel.com/software/products/vtune/vpa/

Note: VTune may not be available for this release. Consult your release notes for details about its availability.

Using GuideView

GuideView is a graphical tool that presents a window into the performance details of a program's parallel execution. GuideView is part of the KAP/Pro Toolset, which also includes the Guide OpenMP compiler and the Assure Thread Analyzer. GuideView is not a part of the default software installation with your system. GuideView is part ot Intel compilers.

GuideView uses an intuitive, color-coded display of parallel performance bottlenecks which helps pinpoint performance anomalies. It graphically illustrates each processor's activity at various levels of detail by using a hierarchical summary.

Statistical data is collapsed into relevant summaries that indicate where attention should be focused (for example, regions of the code where improvements in local performance will have the greatest impact on overall performance).

To gather programming statistics, use the `-O3`, `-openmp`, and `-openmp_profile` compiler options. This causes the linker to use `libguide_stats.a` instead of the default `libguide.a`. The following example demonstrates the compiler command line to produce a file named `swim`:

```
% efc -O3 -openmp -openmp_profile -o swim swim.f
```

To obtain profiling data, run the program, as in this example:

```
% export OMP_NUM_THREADS=8
% ./swim < swim.in
```

When the program finishes, the `swim.gvs` file is produced and it can be used with GuideView. To invoke GuideView with that file, use the following command:

```
% guideview -jpath=your_path_to_Java -mhz=998 ./swim.gvs.
```

The graphical portions of GuideView require the use of Java. Java 1.1.6-8 and Java 1.2.2 are supported and later versions appear to work correctly. Without Java, the functionality is severely limited but text output is still available and is useful, as the following portion of the text file that is produced demonstrates:

```
Program execution time (in seconds):
cpu            :            0.07 sec
elapsed        :           69.48 sec
  serial       :            0.96 sec
  parallel     :           68.52 sec
cpu percent    :            0.10 %
end
Summary over all regions (has 4 threads):
# Thread                      #0       #1        #2       #3
  Sum Parallel        :   68.304   68.230   68.240    68.185
  Sum Imbalance       :    1.020    0.592    0.892     0.838
  Sum Critical Section:    0.011    0.022    0.021     0.024
  Sum Sequential      :    0.011  4.4e-03  4.6e-03   1.6e-03
  Min Parallel        : -5.1e-04 -5.1e-04  4.2e-04  -5.2e-04
  Max Parallel        :    0.090    0.090    0.090     0.090
  Max Imbalance       :    0.036    0.087    0.087     0.087
  Max Critical Section:  4.6e-05  9.8e-04  6.0e-05   9.8e-04
  Max Sequential      :  9.8e-04  9.8e-04  9.8e-04   9.8e-04
end
```

Other Performance Tools

The following performance tools also can be of benefit when you are trying to optimize your code:

- *Guide OpenMP Compiler* is an OpenMP implementation for C, C++, and Fortran from Intel.

- *Assure Thread Analyzer* from Intel locates programming errors in threaded applications with no recoding required.

For details about these products, see the following website:

http://developer.intel.com/software/products/threading

Note: These products have not been thoroughly tested on SGI systems. SGI takes no responsibility for the correct operation of third party products described or their suitability for any particular purpose.

Debugging Tools

Three debuggers are available to help you analyze your code:

- `gdb`: the GNU project debugger. This is useful for debugging programs written in C, C++, and Fortran 95. When compiling with C and C++, include the `-g` option on the compiler command line to produce the `dwarf2` symbols database used by `gdb`.

  When using `gdb` for Fortran debugging, include the `-g` and `-O0` options. Do not use `gdb` for Fortran debugging when compiling with `-O1` or higher.

  The debugger to be used for Fortran 95 codes can be downloaded from http://sourceforge.net/project/showfiles.php?group_id=56720 . (Note that the standard `gdb` compiler does not support Fortran 95 codes.) To verify that you have the correct version of `gdb` installed, use the `gdb -v` command. The output should appear similar to the following:

  ```
  GNU gdb 5.1.1 FORTRAN95-20020628 (RC1)
  Copyright 2002 Free Software Foundation, Inc.
  ```

  For a complete list of `gdb` commands, see the `gdb` user guide online at http://sources.redhat.com/gdb/onlinedocs/gdb_toc.html or use the `help` option. Note that current instances of `gdb` do not report `ar.ec` registers correctly. If you are debugging rotating, register-based, software-pipelined loops at the assembly code level, try using `idb` instead.

- `idb`: the Intel debugger. This is a fully symbolic debugger for the Linux platform. The debugger provides extensive support for debugging programs written in C, C++, FORTRAN 77, and Fortran 90.

Running `idb` with the `-gdb` option on the shell command line provides `gdb`-like user commands and debugger output.

- `ddd`: a GUI to a command line debugger. It supports `gdb` and `idb`. For details about usage, see the following subsection.
- TotalView: a licensed graphical debugger useful in an MPI environment (see http://www.totalviewtech.com/)

Using ddd

The DataDisplayDebugger `ddd`(1) tool is a GUI to an arbitrary command line debugger as shown in Figure 3-1 on page 24. When starting `ddd`, use the `--debugger` option to specify the debugger used (for example, `--debugger "idb"`). The default debugger used is `gdb`.

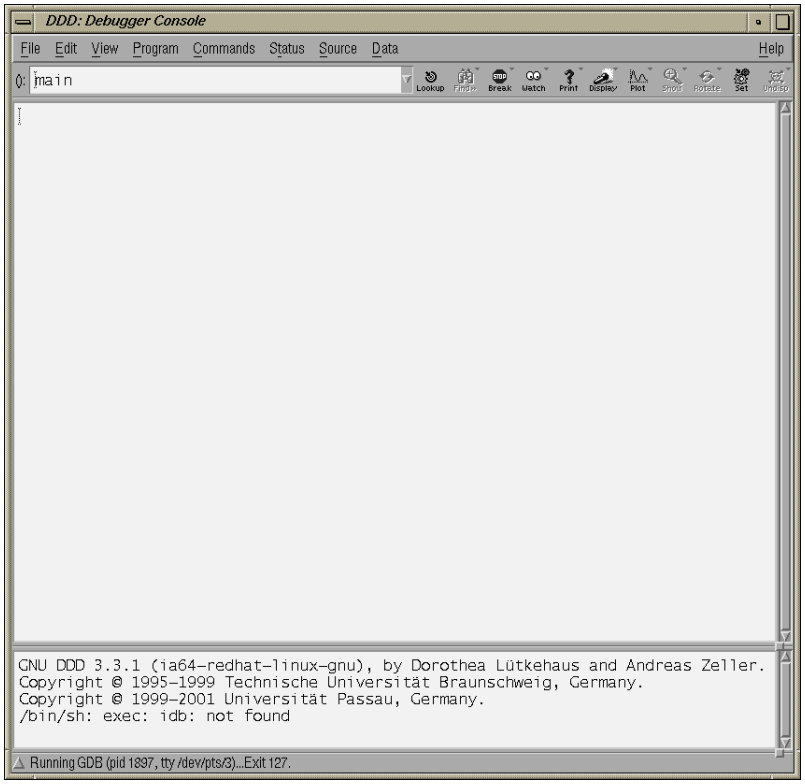

Figure 3-1 DataDisplayDebugger(ddd)(1)

When the debugger is loaded the DataDisplayDebugger screen appears divided into panes that show the following information:

- Array inspection
- Source code
- Disassembled code
- A command line window to the debugger engine

These panes can be switched on and off from the **View** menu.

Some commonly used commands can be found on the menus. In addition, the following actions can be useful:

- Select an address in the assembly view, click the right mouse button, and select `lookup`. The `gdb` command is executed in the command pane and it shows the corresponding source line.
- Select a variable in the source pane and click the right mouse button. The current value is displayed. Arrays are displayed in the array inspection window. You can print these arrays to PostScript by using the **Menu>Print Graph** option.
- You can view the contents of the register file, including general, floating-point, NaT, predicate, and application registers by selecting **Registers** from the **Status** menu. The **Status** menu also allows you to view stack traces or to switch OpenMP threads.

Monitoring Tools

This chapter describes several tools that you can use to monitor system performance. The tools are divided into two general categories: system monitoring tools and nonuniform memory access (NUMA) tools.

System monitoring tools include the `hwinfo`(1), `topology`(1), `top`(1) commands and the Performance Co-Pilot `pmchart`(1) commmand and other operating system commands such as the `vmstat`(1) , `iostat`(1) command and the `sar`(1) commands that can help you determine where system resources are being spent.

The `gtopology`(1) command displays a 3D scene of the system interconnect using the output from the `topology`(1) command.

System Monitoring Tools

You can use system utilities to better understand the usage and limits of your system. These utilities allow you to observe both overall system performance and single-performance execution characteristics. This section covers the following topics:

- "Hardware Inventory and Usage Commands" on page 27
- "Performance Co-Pilot Monitoring Tools" on page 32
- "System Usage Commands" on page 36

Hardware Inventory and Usage Commands

This section descibes hardware inventory and usage commands and covers the following topics:

- "`hwinfo`(1) Command" on page 28
- "`topology`(1) Command" on page 28
- "`gtopology`(1) Command" on page 29

`hwinfo`(1) Command

The `hwinfo`(8) command is used to probe for the hardware present in the system. It can be used to generate a system overview log which can be later used for support. To see the version installed on your system, perform the following command:

```
% rpm -qf /usr/sbin/hwinfo
hwinfo-12.55-0.3
```

For more information, see the `hwinfo`(8) man page.

`topology`(1) Command

The `topology`(1) command provides topology information about your system.

Applications programmers can use the `topology` command to help optimize execution layout for their applications. For more information, see the `topology`(1) man page.

Note: The `topology` command is bundled with SGI ProPack for Linux. It is only available if you are running SGI ProPack on your system.

Output from the `topology` command is similar to the following: (Note that the following output has been abbreviated.)

```
% topology
Machine parrot.americas.sgi.com has:
64 cpu's
32 memory nodes
8 routers
8 repeaterrouters

The cpus are:
cpu 0 is /dev/hw/module/001c07/slab/0/node/cpubus/0/a
cpu 1 is /dev/hw/module/001c07/slab/0/node/cpubus/0/c
cpu 2 is /dev/hw/module/001c07/slab/1/node/cpubus/0/a
cpu 3 is /dev/hw/module/001c07/slab/1/node/cpubus/0/c
cpu 4 is /dev/hw/module/001c10/slab/0/node/cpubus/0/a
                          ...
The nodes are:
node 0 is /dev/hw/module/001c07/slab/0/node
node 1 is /dev/hw/module/001c07/slab/1/node
```

```
node 2 is /dev/hw/module/001c10/slab/0/node
node 3 is /dev/hw/module/001c10/slab/1/node
node 4 is /dev/hw/module/001c17/slab/0/node
                        ...
The routers are:
/dev/hw/module/002r15/slab/0/router
/dev/hw/module/002r17/slab/0/router
/dev/hw/module/002r19/slab/0/router
/dev/hw/module/002r21/slab/0/router
                        ...
The repeaterrouters are:
/dev/hw/module/001r13/slab/0/repeaterrouter
/dev/hw/module/001r15/slab/0/repeaterrouter
/dev/hw/module/001r29/slab/0/repeaterrouter
/dev/hw/module/001r31/slab/0/repeaterrouter
                       ...
The topology is defined by:
/dev/hw/module/001c07/slab/0/node/link/1 is /dev/hw/module/001c07/slab/1/node
/dev/hw/module/001c07/slab/0/node/link/2 is /dev/hw/module/001r13/slab/0/repeaterrouter
/dev/hw/module/001c07/slab/1/node/link/1 is /dev/hw/module/001c07/slab/0/node
/dev/hw/module/001c07/slab/1/node/link/2 is /dev/hw/module/001r13/slab/0/repeaterrouter
/dev/hw/module/001c10/slab/0/node/link/1 is /dev/hw/module/001c10/slab/1/node
/dev/hw/module/001c10/slab/0/node/link/2 is /dev/hw/module/001r13/slab/0/repeaterrouter
```

`gtopology`(1) Command

The `gtopology`(1) command is included as part of the `pcp-sgi` package of the SGI ProPack for Linux software. It displays a 3D scene of the system interconnect using the output from the `topology`(1) command. See the man page for more details.

Note: The `gtopology` command is bundled with SGI ProPack for Linux. It is only available if you are running SGI ProPack on your system.

Figure 4-1 on page 30, shows the ring topology (the eight nodes are shown in pink, the NUMAlink connections in cyan) of an Altix 350 system with 16 CPUs.

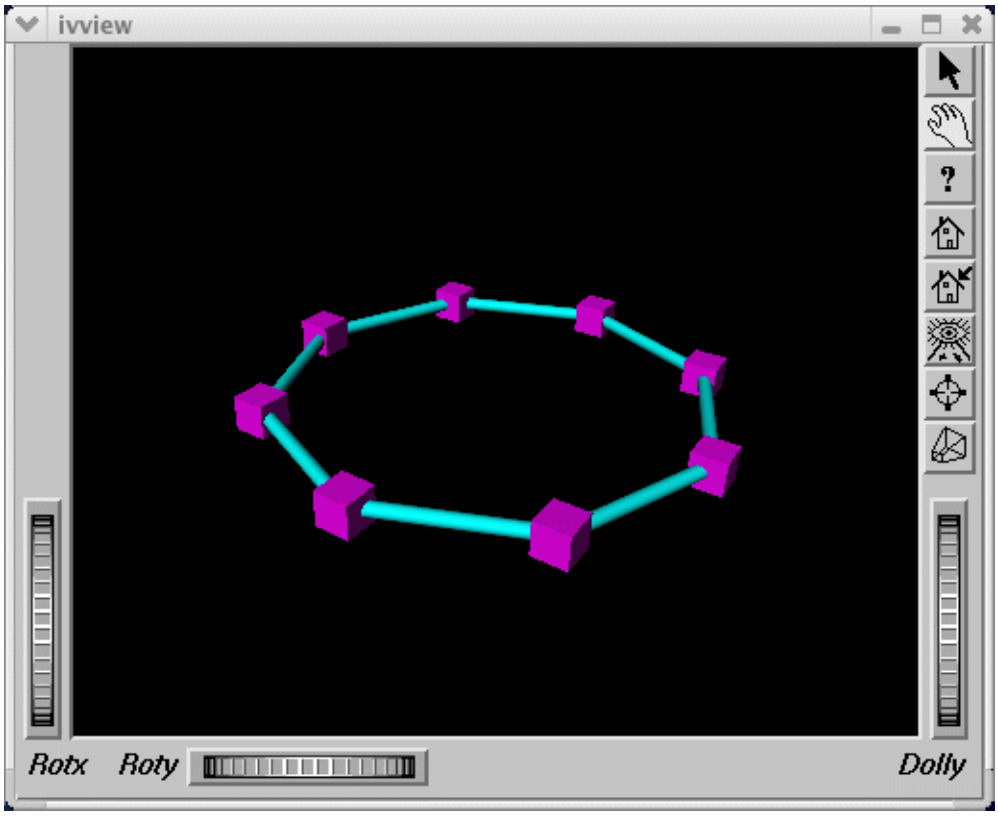

Figure 4-1 Ring Topology of an Altix 350 System with 16 CPUs

Figure 4-2 on page 31, shows the fat-tree topology of an Altix 3700 system with 32 CPUs. Again, nodes are the pink cubes. Routers are shown as blue spheres (if all ports are used) otherwise, yellow.

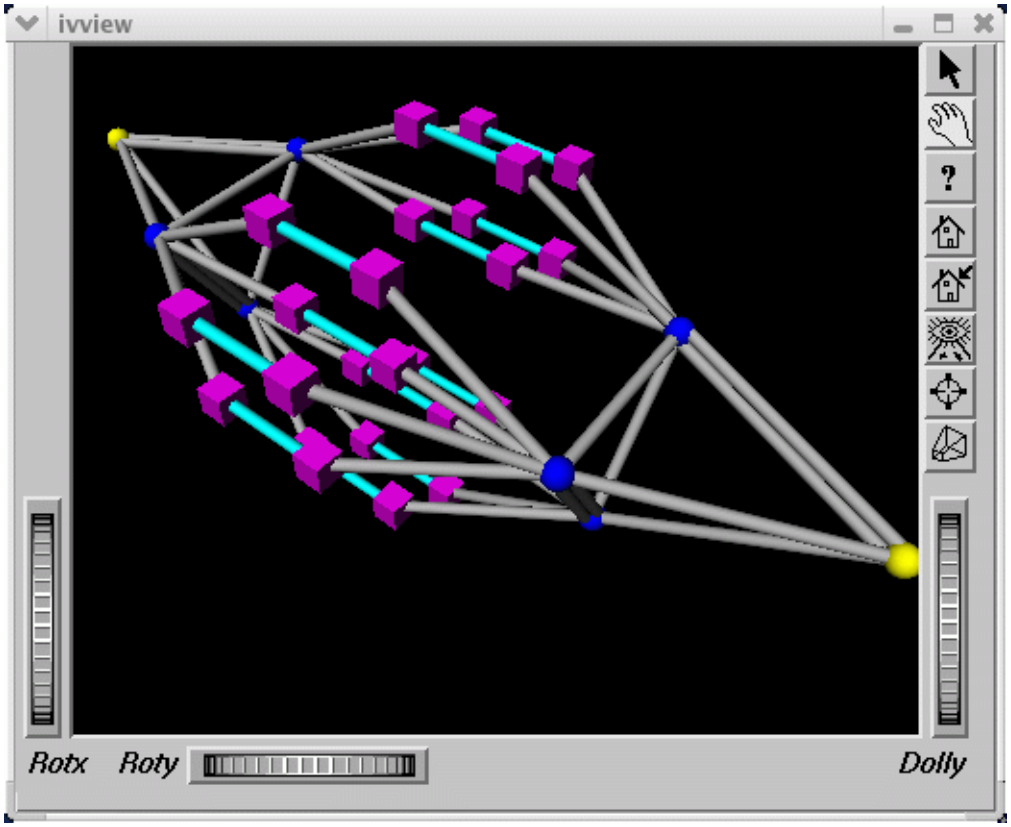

Figure 4-2 An Altix 3700 System with 32 CPUs Fat-tree Topology

Figure 4-3 on page 32, shows an Altix 3700 system with 512 CPUs. The dual planes of the fat-tree topology are clearly visible.

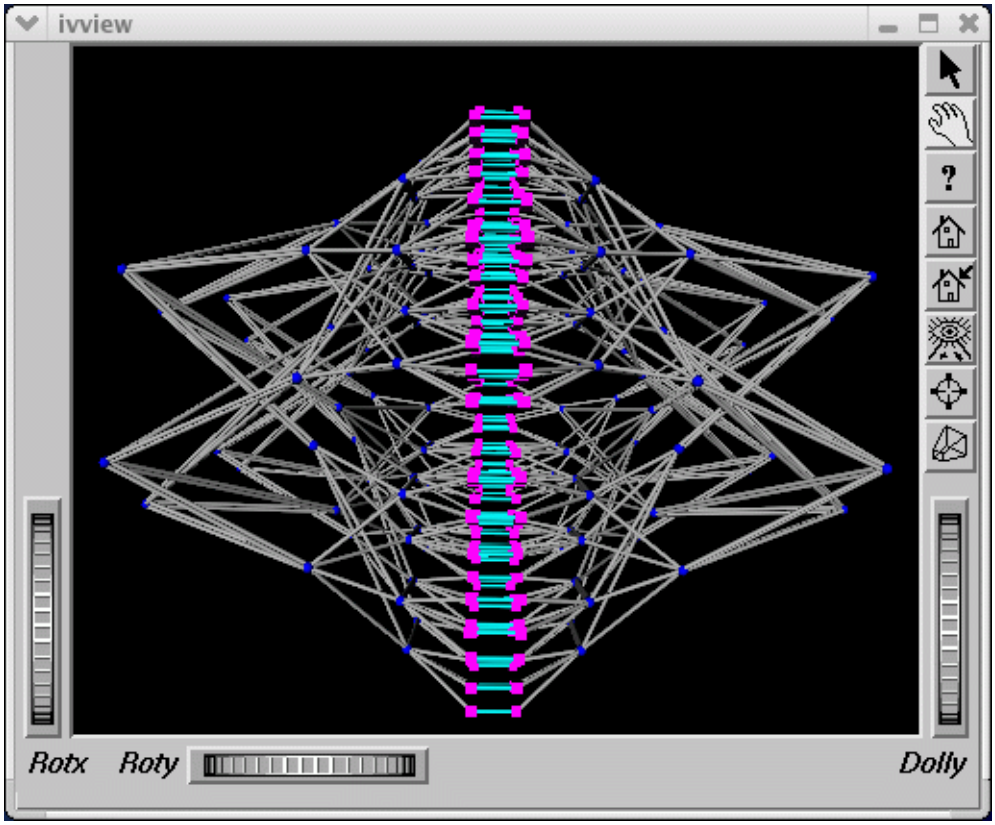

Figure 4-3 An Altix 3700 System with 512 CPUs

Performance Co-Pilot Monitoring Tools

This section describes Performance Co-Pilot monitoring tools and covers the following topics:

- "`pmshub`(1) Command" on page 33
- "`shubstats`(1) Command" on page 34
- "`linkstat`(1) Command" on page 34
- "Other Performance Co-Pilot Monitoring Tools" on page 34

Note: pmshub(1), shubstats(1), and linkstat(1) are bundled with SGI ProPack for Linux. They are only available if you are running SGI ProPack on your system.

pmshub(1) Command

The pmshub(1) command is an Altix system-specific performance monitoring tool that displays ccNUMA architecture cacheline traffic, free memory, and CPU usage statistics on a per-node basis.

Figure 4-4 on page 33, shows a four-node Altix 3700 system with eight CPUs. A key feature of pmshub is the ability to distinguish between local verses remote cacheline traffic statistics. This greatly helps you to diagnose whether the placement of threads on the CPUs in your system has been correctly tuned for memory locality (see the dplace(1) and taskset(1) man pages for information on thread placement.). It also shows undesirable anomalies such as hot cachelines (for example, due to lock contention) and other effects such as cacheline "ping-pong". For details about the intrepretation of each component of the pmshub display, see the pmshub(1) man page.

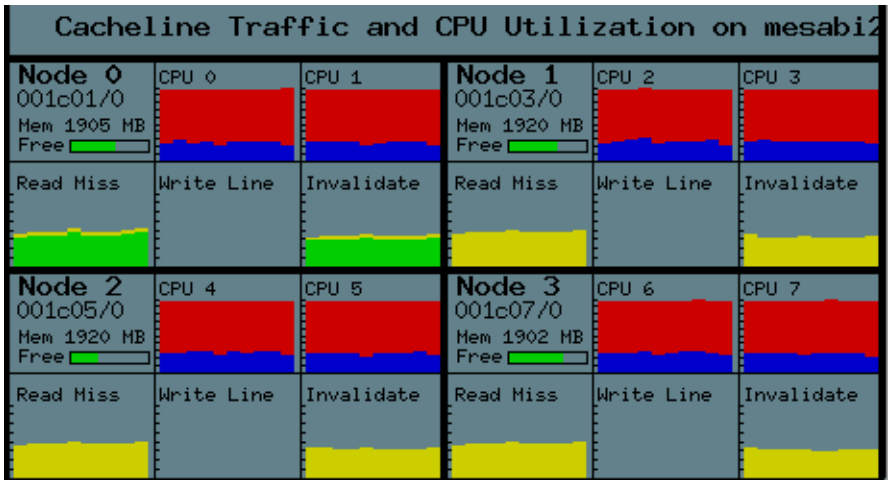

Figure 4-4 Four Node Altix 3700 System with Eight CPUs

shubstats(1) Command

The shubstats(1) command is basically a command-line version of the pmshub(1) command (see "pmshub(1) Command" on page 33). Rather than showing a graphical display, the shubstats command allows you to measure absolute counts (or rate/time converted) ccNUMA-related cacheline traffic events, on a per-node basis. You can also use this tool to obtain per-node memory directory cache hit rates.

linkstat(1) Command

The linkstat(1) command is a command-line tool for monitoring NUMAlink traffic and error rates on SGI Altix systems. This tool shows packets and Mbytes sent/received on each NUMAlink in the system, as well as error rates. It is useful as a performance monitoring tool, as well as, a tool for helping you to diagnose and identify faulty hardware. For more information, see the linkstat(1) man page.

Other Performance Co-Pilot Monitoring Tools

In addition to the Altix specific tools described above, the pcp and pcp-sgi packages also provide numerous other performance monitoring tools, both graphical and text-based. It is important to remember that all of the performance metrics displayed by any of the tools described in this chapter can also be monitored with other tools such as pmchart(1), pmval(1), pminfo(1) and others. Additionally, the pmlogger(1) command can be used to capture Performance Co-Pilot archives, which can then be "replayed" during a retrospective performance analysis.

A very brief description of other Performance Co-Pilot monitoring tools follows. See the associated man page for each tool for more details.

- pmchart(1) — graphical stripchart tool, chiefly used for investigative performance analysis.
- pmgsys(1) — graphical tool showing miniature CPU, Disk, Network, LoadAvg and memory/swap in a miniature display, for example, useful for permanent residence on your desktop for the servers you care about.
- pmgcluster(1) — pmgsys, but for multiple hosts and thus useful for monitoring a cluster of hosts or servers.
- mpvis(1) — 3D display of per-CPU usage.
- clustervis(1) — 3D display showing per-CPU and per-Network performance for multiple hosts.

- `nfsvis`(1) — 3D display showing NFS client/server traffic, grouped by NFS operation type
- `nodevis`(1) — 3D display showing per-node CPU and memory usage.
- `webvis`(1) — 3D display showing per-httpd traffic.
- `dkvis`(1) - 3D display showing per-disk traffic, grouped by controller.
- `diskstat`(1) — command line tool for monitoring disk traffic.
- `topdisk`(1) — command line, curses-based tool, for monitoring disk traffic.
- `topsys`(1) — command line, curses-based tool, for monitoring processes making a large numbers of system calls or spending a large percentage of their execution time in system mode using assorted system time measures.
- `pmgxvm`(1) — miniature graphical display showing XVM volume topology and performance statistics.
- `osvis`(1) — 3D display showing assorted kernel and system statistics.
- `mpivis`(1) — 3D display for monitoring multithreaded MPI applications.
- `pmdumptext`(1) — command line tool for monitoring multiple performance metrics with a highly configurable output format. Therefore, it is a useful tools for scripted monitoring tasks.
- `pmval`(1) — command line tool, similar to `pmdumptext`(1), but less flexible.
- `pminfo`(1) — command line tool, useful for printing raw performance metric values and associated help text.
- `pmprobe`(1) — command line tool useful for scripted monitoring tasks.
- `pmie`(1) — a performance monitoring inference engine. This is a command line tool with an extraordinarily powerful underlying language. It can also be used as a system service for monitoring and reporting on all sorts of performance issues of interest.
- `pmieconf`(1) — command line tool for creating and customizing "canned" `pmie`(1) configurations.
- `pmlogger`(1) — command line tool for capturing Performance Co-Pilot performance metrics archives for replay with other tools.

- pmlogger_daily(1) and pmlogger_check(1) — cron driven infrastructure for automated logging with pmlogger(1).
- pmcd(1) — the Performance Co-Pilot metrics collector daemon
- PCPIntro(1) — introduction to Performance Co-Pilot monitoring tools, generic command line usage and environment variables
- PMAPI(3) — introduction to the Performance Co-Pilot API libraries for developing new performance monitoring tools
- PMDA(3) — introduction to the Performance Co-Pilot Metrics Domain Agent API, for developing new Performance Co-Pilot agents

System Usage Commands

Several commands can be used to determine user load, system usage, and active processes.

To determine the system load, use the uptime(1) command, as follows:

```
[user@profit user]# uptime
  1:56pm  up 11:07, 10 users,  load average: 16.00, 18.14, 21.31
```

The output displays time of day, time since the last reboot, number of users on the system, and the average number of processes waiting to run.

To determine who is using the system and for what purpose, use the w(1) command, as follows:

```
[user@profit user]# w
  1:53pm  up 11:04, 10 users,  load average: 16.09, 20.12, 22.55
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
user1    pts/0    purzel.geneva.sg  2:52am  4:40m  0.23s  0.23s  -tcsh
user1    pts/1    purzel.geneva.sg  2:52am  4:29m  0.34s  0.34s  -tcsh
user2    pts/2    faddeev.sgi.co.j  6:03am  1:18m 20:43m  0.02s  mpirun -np 16 dplace -s1 -c0-15
/tmp/ggg/GSC_TEST/cyana-2.0.17
user3     pts/3    whitecity.readin  4:04am  9:48m  0.02s  0.02s  -csh
user2    pts/4    faddeev.sgi.co.j 10:38am  2:00m  0.04s  0.04s  -tcsh
user2    pts/5    faddeev.sgi.co.j  6:27am  7:19m  0.36s  0.32s  tail -f log
user2    pts/6    faddeev.sgi.co.j  7:57am  1:22m 25.95s 25.89s  top
user1    pts/7    mtv-vpn-hw-richt 11:46am 39:21  11.20s 11.04s  top
user1    pts/8    mtv-vpn-hw-richt 11:46am 33:32   0.22s  0.22s  -tcsh
```

```
user     pts/9    machine007.americas  1:52pm  0.00s  0.03s  0.01s  w
```

The output from this command shows who is on the system, the duration of user sessions, processor usage by user, and currently executing user commands.

To determine active processes, use the ps(1) command, which displays a snapshot of the process table. The ps –A command selects all the processes currently running on a system as follows:

```
[user@profit user]# ps -A
  PID TTY          TIME CMD
    1 ?        00:00:06 init
    2 ?        00:00:00 migration/0
    3 ?        00:00:00 migration/1
    4 ?        00:00:00 migration/2
    5 ?        00:00:00 migration/3
    6 ?        00:00:00 migration/4
                  ...
 1086 ?        00:00:00 sshd
 1120 ?        00:00:00 xinetd
 1138 ?        00:00:05 ntpd
 1171 ?        00:00:00 arrayd
 1363 ?        00:00:01 amd
 1420 ?        00:00:00 crond
 1490 ?        00:00:00 xfs
 1505 ?        00:00:00 sesdaemon
 1535 ?        00:00:01 sesdaemon
 1536 ?        00:00:00 sesdaemon
 1538 ?        00:00:00 sesdaemon
```

To monitor running processes, use the top(1) command. This command displays a sorted list of top CPU utilization processes as shown in Figure 4-5 on page 38.

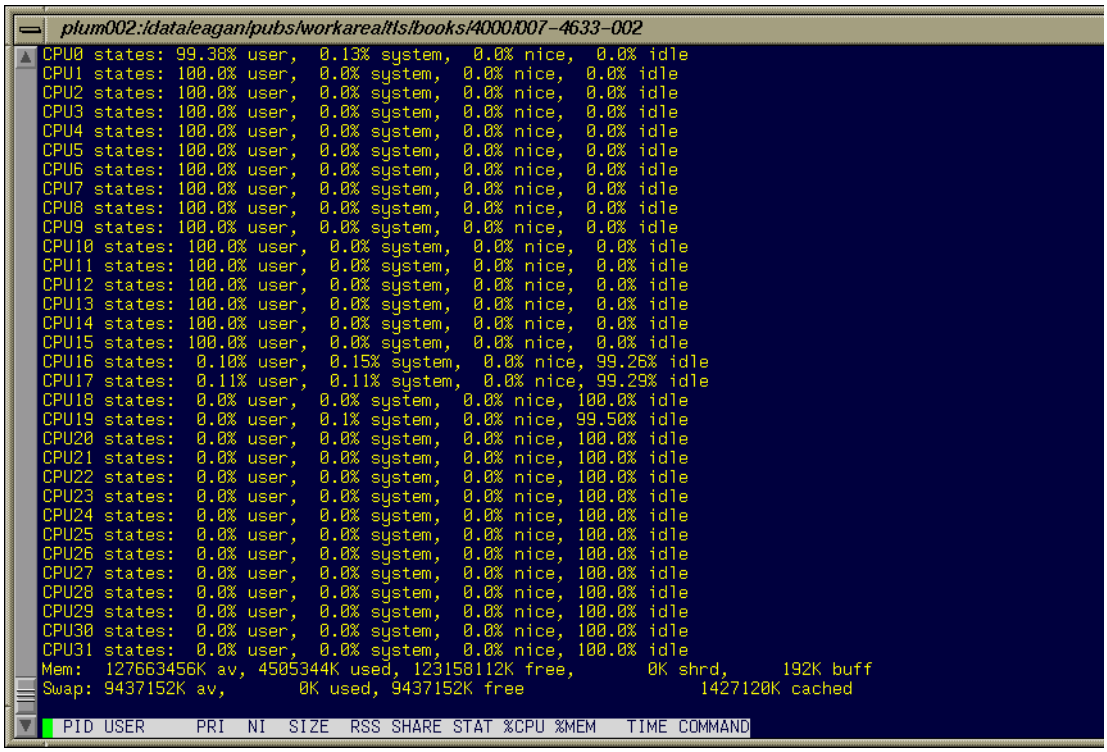

Figure 4-5 Using `top`(1) to Show Top CPU Utilization processes

The `vmstat`(1) command reports virtual memory statistics. It reports information about processes, memory, paging, block IO, traps, and CPU activity. For more information, see the `vmstat`(1) man page.

```
[user@machine3 user]# vmstat
procs                  memory      swap          io     system         cpu
 r  b   swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 1  0      0 81174720     80 11861232    0    0     0     1    1     1  0  0  0  0
```

The first report produced gives averages since the last reboot. Additional reports give information on a sampling period of length delay. The process and memory reports are instantaneous in either case.

The iostat(1) command is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. The iostat command generates reports that can be used to change system configuration to better balance the input/output load between physical disks. For more information, see the iostat(1) man page.

```
user@machine3 user]# iostat
Linux 2.4.21-sgi302c19 (revenue3.engr.sgi.com)  11/04/2004

avg-cpu:  %user   %nice    %sys   %idle
          40.46    0.00    0.16   59.39

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
```

The sar(1) command writes to standard output the contents of selected cumulative activity counters in the operating system. The accounting system, based on the values in the count and interval parameters, writes information the specified number of times spaced at the specified intervals in seconds. For more information, see the sar(1) man page.

```
[user@machine3 user]# sar
Linux 2.4.21-sgi302c19 (revenue3.engr.sgi.com)  11/04/2004

12:00:00 AM       CPU     %user     %nice   %system     %idle
12:10:00 AM       all     49.85      0.00      0.19     49.97
12:20:00 AM       all     49.85      0.00      0.19     49.97
12:30:00 AM       all     49.85      0.00      0.18     49.97
12:40:00 AM       all     49.88      0.00      0.16     49.97
12:50:00 AM       all     49.88      0.00      0.15     49.97
01:00:00 AM       all     49.88      0.00      0.15     49.97
01:10:00 AM       all     49.91      0.00      0.13     49.97
01:20:00 AM       all     49.88      0.00      0.15     49.97
01:30:00 AM       all     49.88      0.00      0.16     49.97
01:40:00 AM       all     49.91      0.00      0.13     49.97
01:50:00 AM       all     49.87      0.00      0.16     49.97
02:00:00 AM       all     49.91      0.00      0.13     49.97
02:10:00 AM       all     49.91      0.00      0.13     49.97
02:20:00 AM       all     49.90      0.00      0.13     49.97
02:30:00 AM       all     49.90      0.00      0.13     49.97
02:40:00 AM       all     49.90      0.00      0.13     49.97
```

```
02:50:00 AM       all     49.90      0.00      0.14     49.96
03:00:00 AM       all     49.90      0.00      0.13     49.97
03:10:00 AM       all     49.90      0.00      0.13     49.97
03:20:00 AM       all     49.90      0.00      0.14     49.97
03:30:01 AM       all     49.89      0.00      0.14     49.97
03:40:00 AM       all     49.90      0.00      0.14     49.96
03:50:01 AM       all     49.90      0.00      0.14     49.96
04:00:00 AM       all     49.89      0.00      0.14     49.97
04:10:00 AM       all     50.18      0.01      0.66     49.14
04:20:00 AM       all     49.90      0.00      0.14     49.96
04:30:00 AM       all     49.90      0.00      0.14     49.96
04:40:00 AM       all     49.94      0.00      0.10     49.96
04:50:00 AM       all     49.89      0.00      0.15     49.96
05:00:00 AM       all     49.94      0.00      0.09     49.97
05:10:00 AM       all     49.89      0.00      0.16     49.96
05:20:00 AM       all     49.94      0.00      0.10     49.96
05:30:00 AM       all     49.89      0.00      0.16     49.96
05:40:00 AM       all     49.94      0.00      0.10     49.96
05:50:00 AM       all     49.93      0.00      0.11     49.96
06:00:00 AM       all     49.89      0.00      0.15     49.96
06:10:00 AM       all     49.94      0.00      0.10     49.96
06:20:01 AM       all     49.88      0.00      0.17     49.95
06:30:00 AM       all     49.93      0.00      0.10     49.96
06:40:01 AM       all     49.93      0.00      0.11     49.96
06:50:00 AM       all     49.88      0.00      0.16     49.96
07:00:00 AM       all     49.93      0.00      0.10     49.96
07:10:00 AM       all     49.93      0.00      0.11     49.96
07:20:00 AM       all     49.87      0.00      0.17     49.96
07:30:00 AM       all     49.99      0.00      0.13     49.88
07:40:00 AM       all     50.68      0.00      0.14     49.18
07:50:00 AM       all     49.94      0.00      0.11     49.94
08:00:00 AM       all     49.92      0.00      0.13     49.94
08:10:00 AM       all     49.88      0.00      0.18     49.95
08:20:00 AM       all     49.93      0.00      0.13     49.95
08:30:00 AM       all     49.93      0.00      0.12     49.95
08:40:00 AM       all     49.93      0.00      0.12     49.95
08:50:00 AM       all     25.33      0.00      0.08     74.59
09:00:00 AM       all      0.02      0.00      0.04     99.95
09:10:00 AM       all      1.52      0.00      0.05     98.43
09:20:00 AM       all      0.41      0.00      0.10     99.49
09:30:00 AM       all      0.01      0.00      0.02     99.97
```

```
09:40:00 AM       all      0.01      0.00      0.02     99.97
09:50:00 AM       all      0.01      0.00      0.02     99.97
10:00:00 AM       all      0.01      0.00      0.02     99.97
10:10:00 AM       all      0.01      0.00      0.08     99.91
10:20:00 AM       all      2.93      0.00      0.55     96.52
10:30:00 AM       all      3.13      0.00      0.02     96.84

10:30:00 AM       CPU     %user     %nice   %system     %idle
10:40:00 AM       all      3.13      0.00      0.03     96.84
10:50:01 AM       all      3.13      0.00      0.03     96.84
Average:          all     40.55      0.00      0.13     59.32
```

Data Placement Tools

This chapter describes data placement tools you can use on an SGI Altix system. It covers the following topics:

- "Data Placement Tools Overview" on page 43
- "`taskset` Command" on page 45
- "`dplace` Command" on page 48
- "`dlook` Command" on page 55
- "Installing NUMA Tools" on page 62

Data Placement Tools Overview

On an SMP machine, all data is visible from all processors. Special optimization applies to SGI Altix systems to exploit multiple paths to memory, as follows:

- By default, all pages are allocated with a “first touch” policy.
- The initialization loop, if executed serially, will get pages from single node.
- In the parallel loop, multiple processors will access that one memory.

So, perform initialization in parallel, such that each processor initializes data that it is likely to access later for calculation.

Figure 5-1 on page 44, shows how to code to get good data placement.

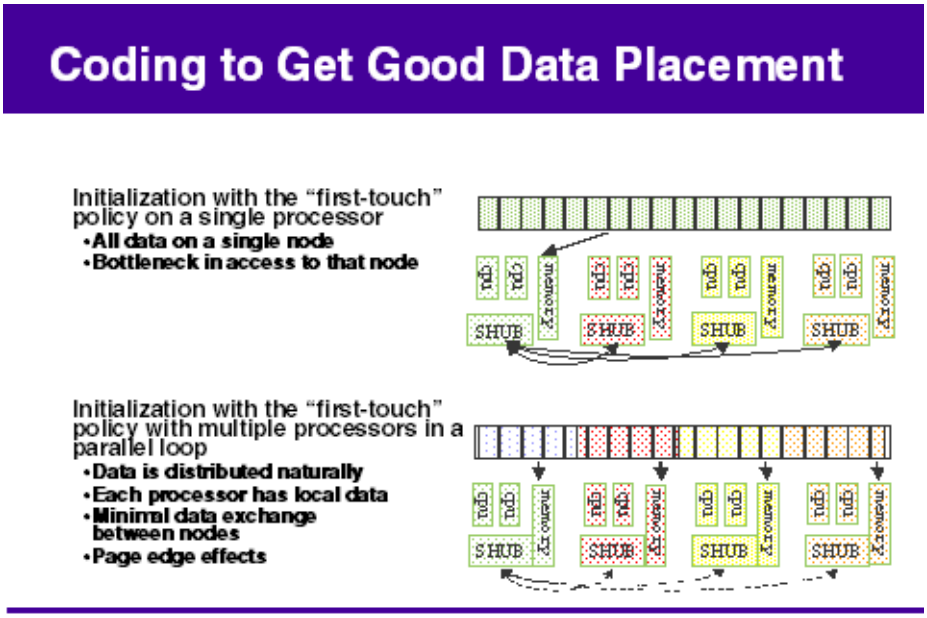

Figure 5-1 Coding to Get Good Data Placement

Placement facilities include cpusets, `taskset`(1), and `dplace`(1), all built upon CpuMemSets API:

- cpusets — Named subsets of system cpus/memories, used extensively in batch environments.
- `taskset` and `dplace` — Avoid poor data locality caused by process or thread drift from CPU to CPU.
  - `taskset` restricts execution to the listed set of CPUs (see the `taskset -c --cpu-list` option); however, processes are still free to move among listed CPUs.
  - `dplace` binds processes to specified CPUs in round-robin fashion; once pinned, they do not migrate. Use this for high performance and reproducibility of parallel codes.

For more information on CpuMemSets and cpusets, see chapter 4, “CPU Memory Sets and Scheduling” and chapter 5, “Cpuset System”, respectively, in the *Linux Resource Administration Guide*.

`taskset` Command

The `taskset`(1) command retrieves or sets a CPU affinity of a process, as follows:

```
taskset [options] [mask | list ] [pid | command [arg]...]
```

The `taskset` command is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new command with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. Note that the Linux scheduler also supports natural CPU affinity; the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. Therefore, forcing a specific CPU affinity is useful only in certain applications.

The CPU affinity is represented as a bitmask, with the lowest order bit corresponding to the first logical CPU and the highest order bit corresponding to the last logical CPU. Not all CPUs may exist on a given system but a mask may specify more CPUs than are present. A retrieved mask will reflect only the bits that correspond to CPUs physically on the system. If an invalid mask is given (that is, one that corresponds to no valid CPUs on the current system) an error is returned. The masks are typically given in hexadecimal. For example:

`0x00000001`	is processor #0
`0x00000003`	is processors #0 and #1
`0xFFFFFFFF`	is all processors (#0 through #31)

When `taskset` returns, it is guaranteed that the given program has been scheduled to a legal CPU.

The `taskset` command does not pin a task to a specific CPU. It only restricts a task so that it does not run on any CPU that is not in the `cpulist`. For example, if you use `taskset` to launch an application that forks multiple tasks, it is possible that multiple tasks will initially be assigned to the same CPU even though there are idle CPUs that are in the `cpulist`. Scheduler load balancing software will eventually distribute the tasks so that CPU bound tasks run on different CPUs. However, the exact placement is not predictable and can vary from run-to-run. After the tasks are

evenly distributed (assuming that happens), nothing prevents tasks from jumping to different CPUs. This can affect memory latency since pages that were node-local before the jump may be remote after the jump.

If you are running an MPI application, SGI recommends that you do not use the `taskset` command. The `taskset` command can pin the MPI shepherd process (which is a waste of a CPU) and then putting the remaining working MPI rank on one of the CPUs that already had some other rank running on it. Instead of `taskset`, SGI recommnds using the the `dplace`(1) (see "`dplace` Command" on page 48) or the environment variable `MPI_DSM_CPULIST`. The following example assumes a job running on eight CPUs. For example:

`# mpirun -np 8 dplace -s1 -c10,11,16-21` *myMPIapplication* `...`
To set `MPI_DSM_CPULIST` variable, perform a command similar to the following:

`setenv MPI_DSM_CPULIST 10,11,16-21 mpirun -np 8` *myMPIapplication* `...`
If they are using a batch scheduler that creates and destroys cpusets dynamically, you should use `MPI_DSM_DISTRIBUTE` environment variable instead of either `MPI_DSM_CPULIST` enviroment variable or the `dplace` command.

For more detailed information, see the `taskset`(1) man page.

To run an executable on CPU 1 (the cpumask for CPU 1 is 0x2), perform the following:

`taskset 0x2` ***executable name***

To move pid 14057 to CPU 0 (the cpumask for cpu 0 is 0x1), perform the following:

`taskset -p 0x1 14057`

To run an MPI Abaqus/Std job on Altix 4000 series system with eight CPUs, perform the following:

`taskset -c 8-15 ./runme < /dev/null &`

The stdin is redirected to `/dev/null` to avoid a `SIGTTIN` signal for MPT applications.

The following example uses the `taskset` command to lock a given process to a particular CPU (CPU5) and then uses the `profile`(1) command to profile it. It then shows how to use `taskset` to move the process to another CPU (CPU3).

```
# taskset -p -c 5 16269
pid 16269's current affinity list: 0-15
```

```
pid 16269’s new affinity list: 5

# profile.pl  -K -KK -c 5 /bin/sleep 60

The analysis showed

=====================================================================
user ticks:             0               0 %
kernel ticks:           6001            100 %
idle ticks:             5999            99.97 %

Using /boot/System.map-2.6.5-7.282-rtgfx as the kernel map file.
=====================================================================
                           Kernel

      Ticks     Percent  Cumulative   Routine
                          Percent
---------------------------------------------------------------------
       5999       99.97    99.97      default_idle
          2        0.03   100.00      run_timer_softirq
=====================================================================

Looking at the analysis for the processor, every 100th of a second,
the process has pretty much the same ip.
This might tell us that the process is in a pretty tight infinite loop.

63       16269    5  0x2000000005c3cc00 0x0005642bd60d9c5f   4 16000000
64       16269    5  0x2000000005c3cc00 0x0005642bd701c36c   4 16000000
65       16269    5  0x2000000005c3cc00 0x0005642bd7f5ea7c   4 16000000
66       16269    5  0x2000000005c3cc00 0x0005642bd8ea178a   4 16000000
67       16269    5  0x2000000005c3cc00 0x0005642bd9de3ea5   4 16000000
68       16269    5  0x2000000005c3cc00 0x0005642bdad265cb   4 16000000
69       16269    5  0x2000000005c3cbe0 0x0005642bdbc68ce6   4 16000000
70       16269    5  0x2000000005c3cc00 0x0005642bdcbab3fe   4 16000000
71       16269    5  0x2000000005c3cc00 0x0005642bddaedb13   4 16000000
72       16269    5  0x2000000005c3cc00 0x0005642bdea3021c   4 16000000
73       16269    5  0x2000000005c3cc00 0x0005642bdf97292f   4 16000000
74       16269    5  0x2000000005c3cc00 0x0005642be08b503f   4 16000000
```

```
# taskset -p 16269 -c 3
pid 16269's current affinity list: 5
pid 16269's new affinity list: 3
```

dplace Command

You can use the dplace(1) command to bind a related set of processes to specific CPUs or nodes to prevent process migration. This can improve the performance of your application since it increases the percentage of memory accesses that are local.

Using the dplace Command

The dplace command allows you to control the placement of a process onto specified CPUs, as follows:

```
dplace [-c cpu_numbers] [-s skip_count] [-n process_name] [-x skip_mask]
 [-p placement_file] command [command-args]

dplace -q
```

Scheduling and memory placement policies for the process are set up according to dplace command line arguments.

By default, memory is allocated to a process on the node on which the process is executing. If a process moves from node to node while it running, a higher percentage of memory references are made to remote nodes. Because remote accesses typically have higher access times, process performance can be diminished. CPU instruction pipelines also have to be reloaded.

You can use the dplace command to bind a related set of processes to specific CPUs or nodes to prevent process migrations. In some cases, this improves performance since a higher percentage of memory accesses are made to local nodes.

Processes always execute within a CpuMemSet. The CpuMemSet specifies the CPUs on which a process can execute. By default, processes usually execute in a CpuMemSet that contains all the CPUs in the system (for detailed information on CpusMemSets, see the *Linux Resource Administration Guide*).

The dplace command invokes an SGI kernel hook (module called numatools) to create a placement container consisting of all the CPUs (or a or a subset of CPUs) of a cpuset. The dplace process is placed in this container and by default is bound to the first CPU of the cpuset associated with the container. Then dplace invokes exec to execute the command.

The command executes within this placement container and remains bound to the first CPU of the container. As the command forks child processes, they inherit the container and are bound to the next available CPU of the container.

If you do not specify a placement file, dplace binds processes sequentially in a round-robin fashion to CPUs of the placement container. For example, if the current cpuset consists of physical CPUs 2, 3, 8, and 9, the first process launched by dplace is bound to CPU 2. The first child process forked by this process is bound to CPU 3, the next process (regardless of whether it is forked by parent or child) to 8, and so on. If more processes are forked than there are CPUs in the cpuset, binding starts over with the first CPU in the cpuset.

For more information on dplace(1) and examples of how to use the command, see the dplace(1) man page.

The dplace(1) command accepts the following options:

- -c *cpu_numbers*: The *cpu_numbers* variable specifies a list of CPU ranges, for example: "-c1", "-c2-4", "-c1, 4-8, 3". CPU numbers are **not** physical CPU numbers. They are logical CPU numbers that are relative to the CPUs that are in the set of allowed CPUs as specified by the current cpuset or taskset(1) command. CPU numbers start at 0. If this option is not specified, all CPUs of the current cpuset are available.

- -s *skip_count*: Skips the first *skip_count* processes before starting to place processes onto CPUs. This option is useful if the first *skip_count* processes are “shepherd" processes that are used only for launching the application. If *skip_count* is not specified, a default value of 0 is used.

- -n *process_name*: Only processes named *process_name* are placed. Other processes are ignored and are not explicitly bound to CPUs.

  The *process_name* argument is the basename of the executable.

- -x *skip_mask*: Provides the ability to skip placement of processes. The *skip_mask* argument is a bitmask. If bit N of skip_mask is set, then the N+1th process that is forked is not placed. For example, setting the mask to 6 prevents the second and third processes from being placed. The first process (the process named by

the command) will be assigned to the first CPU. The second and third processes are not placed. The fourth process is assigned to the second CPU, and so on. This option is useful for certain classes of threaded applications that spawn a few helper processes that typically do not use much CPU time.

Note: OpenMP with Intel applications runnint on ProPack 2.4, should be placed using the `-x` option with a `skip_mask` of `6` (`-x6`). For applications compiled on ProPack 3 (or later) using the Native Posix Thread Library (NPTL), use the `-x2` option.

- `-p` *placement_file*: Specifies a placement file that contains additional directives that are used to control process placement. (Implemented in SGI ProPack 3 Sevice Pack 2).
- `command` [*command-args*]: Specifies the command you want to place and its arguments.
- `-q`: Lists the global count of the number of active processes that have been placed (by `dplace`) on each CPU in the current cpuset. Note that CPU numbers are logical CPU numbers within the cpuset, **not** physical CPU numbers.

Example 5-1 Using the `dplace` command with MPI Programs

You can use the `dplace` command to improve placement of MPI programs on NUMA systems and verify placement of certain data structures of a long running MPI program by running a command such as the following:

```
mpirun -np 64 /usr/bin/dplace -s1 -c 0-63 ./a.out
```

You can then use the `dlook`(1) command to verify placement of certain data structures of a long running MPI program by using the `dlook` command in another window on one of the slave thread PIDs to verify placement. For more information on using the `dlook` command, see "`dlook` Command" on page 55 and the `dlook`(1) man page.

Example 5-2 Using `dplace` command with OpenMP Programs

To run an OpenMP program on logical CPUs 4 through 7 within the current cpuset, perform the following:

```
%efc -o prog -openmp -O3 program.f
%setenv OMP_NUM_THREADS 4
```

```
%dplace -x6 -c4-7 ./prog
```

The dplace(1) command has a static load balancing feature so that you do not necessarily have to supply a CPU list. To place prog1 on logical CPUs 0 through 3 and prog2 on logical CPUs 4 through 7, perform the following:

```
%setenv OMP_NUM_THREADS 4
%dplace -x6 ./prog1 &
%dplace -x6 ./prog2 &
```

You can use the dplace -q command to display the static load information.

Example 5-3 Using the dplace command with Linux commands

The following examples assume that the command is executed from a shell running in a cpuset consisting of physical CPUs 8 through 15.

Command	Run Location
dplace -c2 date	Runs the date command on physical CPU 10.
dplace make linux	Runs gcc and related processes on physical CPUs 8 through 15.
dplace -c0-4,6 make linux	Runs gcc and related processes on physical CPUs 8 through 12 or 14.
taskset 4,5,6,7 dplace app	The taskset command restricts execution to physical CPUs 12 through 15. The dplace command sequentially binds processes to CPUs 12 through 15.

To use the dplace command accurately, you should know how your placed tasks are being created in terms of the fork, exec, and pthread_create calls. Determine whether each of these worker calls are an MPI rank task or are they groups of pthreads created by rank tasks? Here is an example of two MPI ranks, each creating three threads:

```
cat <<EOF > placefile
firsttask cpu=0
exec name=mpiapp cpu=1
fork   name=mpiapp cpu=4-8:4 exact
thread name=mpiapp oncpu=4 cpu=5-7 exact thread name=mpiapp oncpu=8
cpu=9-11 exact EOF

#  mpirun is placed on cpu 0 in this example #  the root mpiapp is
```

```
placed on cpu 1 in this example

# or, if your version of dplace supports the "cpurel=" option:
# firsttask cpu=0
# fork   name=mpiapp cpu=4-8:4 exact
# thread name=mpiapp oncpu=4 cpurel=1-3 exact


# create 2 rank tasks, each will pthread_create 3 more # ranks will be
on 4 and 8
#  thread children on 5,6,7   9,10,11
dplace -p placefile mpirun -np 2 ~cpw/bin/mpiapp -P 3 -l


exit
```

You can use the debugger to determine if it is working. It should show two MPI rank applications, each with three pthreads, as follows:

```
>> pthreads | grep mpiapp
px *(task_struct *)e00002343c528000   17769   17769   17763  0       mpiapp
      member task: e000013817540000   17795   17769   17763  0     5 mpiapp
      member task: e000013473aa8000   17796   17769   17763  0     6 mpiapp
      member task: e000013817c68000   17798   17769   17763  0       mpiapp
px *(task_struct *)e0000234704f0000   17770   17770   17763  0       mpiapp
      member task: e000023466ed8000   17794   17770   17763  0     9 mpiapp
      member task: e00002384cce0000   17797   17770   17763  0       mpiapp
      member task: e00002342c448000   17799   17770   17763  0       mpiapp
```

And you can use the debugger, to see a root application, the parent of the two MPI rank applications, as follows:

```
>> ps | grep mpiapp
0xe00000340b300000   1139  17763  17729      1  0xc800000   -  mpiapp
0xe00002343c528000   1139  17769  17763      0  0xc800040   -  mpiapp
0xe0000234704f0000   1139  17770  17763      0  0xc800040   8  mpiapp
```

Placed as specified:

```
>> oncpus e00002343c528000 e000013817540000 e000013473aa8000
>> e000013817c68000 e0
000234704f0000 e000023466ed8000 e00002384cce0000 e00002342c448000
```

```
task: 0xe00002343c528000  mpiapp cpus_allowed: 4
task: 0xe000013817540000  mpiapp cpus_allowed: 5
task: 0xe000013473aa8000  mpiapp cpus_allowed: 6
task: 0xe000013817c68000  mpiapp cpus_allowed: 7
task: 0xe0000234704f0000  mpiapp cpus_allowed: 8
task: 0xe000023466ed8000  mpiapp cpus_allowed: 9
task: 0xe00002384cce0000  mpiapp cpus_allowed: 10
task: 0xe00002342c448000  mpiapp cpus_allowed: 11
```

dplace for Compute Thread Placement Troubleshooting Case Study

This section describes common reasons why compute threads do not end up on unique processors when using commands such a `dplace`(1) or `profile.pl` (see "Profiling with `profile.pl`" on page 15).

In the example that follows, a user used the `dplace -s1 -c0-15` command to bind 16 processes to run on 0-15 CPUs. However, output from the `top`(1) command shows only 13 CPUs running with CPUs 13, 14, and 15 still idle and CPUs 0, 1 and 2 are shared with 6 processes.

```
263 processes: 225 sleeping, 18 running, 3 zombie, 17 stopped
CPU states:  cpu    user    nice  system    irq  softirq  iowait    idle
             total  1265.6%    0.0%   28.8%   0.0%    11.2%    0.0%  291.2%

             cpu00  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu01   90.1%    0.0%    0.0%   0.0%     9.7%    0.0%    0.0%

             cpu02   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu03   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu04  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu05  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu06  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

             cpu07   88.4%    0.0%   10.6%   0.0%     0.8%    0.0%    0.0%

             cpu08  100.0%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%
```

```
            cpu09   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

            cpu10   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

            cpu11   88.1%    0.0%   11.2%   0.0%     0.6%    0.0%    0.0%

            cpu12   99.7%    0.0%    0.2%   0.0%     0.0%    0.0%    0.0%

            cpu13    0.0%    0.0%    2.5%   0.0%     0.0%    0.0%   97.4%

            cpu14    0.8%    0.0%    1.6%   0.0%     0.0%    0.0%   97.5%

            cpu15    0.0%    0.0%    2.4%   0.0%     0.0%    0.0%   97.5%
 Mem:  60134432k av, 15746912k used, 44387520k free,       0k shrd,
672k buff
         351024k active,            13594288k inactive

  Swap: 2559968k av,       0k used, 2559968k free
  2652128k cached

    PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME CPU COMMAND

   7653 ccao      25   0  115G 586M  114G R    99.9  0.9   0:08   3 mocassin

  7656 ccao      25   0  115G 586M  114G R    99.9  0.9   0:08   6 mocassin

   7654 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   4 mocassin

   7655 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   5 mocassin

   7658 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   8 mocassin

   7659 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   9 mocassin

   7660 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08  10 mocassin

   7662 ccao      25   0  115G 586M  114G R    99.7  0.9   0:08  12 mocassin

   7657 ccao      25   0  115G 586M  114G R    88.5  0.9   0:07   7 mocassin
```

```
7661 ccao      25   0  115G 586M  114G R    88.3  0.9   0:07  11 mocassin

7649 ccao      25   0  115G 586M  114G R    55.2  0.9   0:04   2 mocassin

7651 ccao      25   0  115G 586M  114G R    54.1  0.9   0:03   1 mocassin

7650 ccao      25   0  115G 586M  114G R    50.0  0.9   0:04   0 mocassin

7647 ccao      25   0  115G 586M  114G R    49.8  0.9   0:03   0 mocassin

7652 ccao      25   0  115G 586M  114G R    44.7  0.9   0:04   2 mocassin

7648 ccao      25   0  115G 586M  114G R    35.9  0.9   0:03   1 mocassin
```

An application can start some threads executing for a very short time yet the threads still have taken a token in the CPU list. Then, when the compute threads are finally started, the list is exhausted and restarts from the beginning. Consequently, some threads end up sharing the same CPU. To bypass this, try to eliminate the "ghost" thread creation, as follows:

- Check for a call to the "system" function. This is often responsible for the placement failure due to unexpected thread creation.
- When all the compute processes have the same name, you can do this by issuing a command, such as the following:

  ```
  dplace -c0-15 -n compute-process-name ...
  ```

- You can also run `dplace -e -c0-32` on 16 CPUs to understand the pattern of the thread creation. If by chance, this pattern is the same from one run to the other (unfortunately race between thread creation often occurs), you can find the right flag to `dplace`. For example, if you want to run on CPU 0-3, with `dplace -e -C0-16` and you see that threads are always placed on CPU 0, 1, 5, and 6, then `dplace -e -c0,1,x,x,x,2,3` or `dplace -x24 -c0-3` (24 =11000, place the 2 first and skip 3 before placing) should place your threads correctly.

`dlook` Command

You can use `dlook`(1) to find out where in memory the operating system is placing your application's pages and how much system and user CPU time it is consuming.

Using the dlook Command

The `dlook`(1) command allows you to display the memory map and CPU usage for a specified process as follows:

`dlook [-a] [-c] [-h] [-l] [-o` *outfile*`] [-s` *secs*`] command [`*command-args*`]`
`dlook [-a] [-c] [-h] [-l] [-o` *outfile*`] [-s` *secs*`]` *pid*

For each page in the virtual address space of the process, `dlook`(1) prints the following information:

- The object that owns the page, such as a file, SYSV shared memory, a device driver, and so on.
- The type of page, such as random access memory (RAM), FETCHOP, IOSPACE, and so on.
- If the page type is RAM memory, the following information is displayed:
  - Memory attributes, such as, SHARED, DIRTY, and so on
  - The node on which the page is located
  - The physical address of the page (optional)
- Optionally, the `dlook`(1) command also prints the amount of user and system CPU time that the process has executed on each physical CPU in the system.

Two forms of the `dlook`(1) command are provided. In one form, `dlook` prints information about an existing process that is identified by a process ID (PID). To use this form of the command, you must be the owner of the process or be running with root privilege. In the other form, you use `dlook` on a command you are launching and thus are the owner.

The `dlook`(1) command accepts the following options:

- `-a`: Shows the physical addresses of each page in the address space.
- `-c`: Shows the user and system CPU time, that is, how long the process has executed on each CPU.
- `-h`: Explicitly lists holes in the address space.
- `-l`: Shows libraries.
- `-o`: Outputs to file name (*outfile*). If not specified, output is written to stdout.

- `-s`: Specifies a sample interval in seconds. Information about the process is displayed every second (`secs`) of CPU usage by the process.

An example for the `sleep` process with a PID of 4702 is as follows:

Note: The output has been abbreviated to shorten the example and bold headings added for easier reading.

```
dlook 4702

Peek:  sleep
Pid: 4702       Thu Aug 22 10:45:34 2002

Cputime by cpu (in seconds):
                 user    system
  TOTAL          0.002     0.033
  cpu1           0.002     0.033

Process memory map:
  2000000000000000-2000000000030000 r-xp 0000000000000000 04:03 4479 /lib/ld-2.2.4.so
        [2000000000000000-2000000000002c000]        11 pages on node   1  MEMORY|SHARED

  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-200000000003c000]         3 pages on node   0  MEMORY|DIRTY

                                     ...

  2000000000128000-2000000000370000 r-xp 0000000000000000 04:03 4672       /lib/libc-2.2.4.so
        [2000000000128000-2000000000164000]        15 pages on node   1  MEMORY|SHARED
        [2000000000174000-2000000000188000]         5 pages on node   2  MEMORY|SHARED
        [2000000000188000-2000000000190000]         2 pages on node   1  MEMORY|SHARED
        [200000000019c000-20000000001a8000]         3 pages on node   1  MEMORY|SHARED
        [20000000001c8000-20000000001d0000]         2 pages on node   1  MEMORY|SHARED
        [20000000001fc000-2000000000204000]         2 pages on node   1  MEMORY|SHARED
        [200000000020c000-2000000000230000]         9 pages on node   1  MEMORY|SHARED
        [200000000026c000-2000000000270000]         1 page  on node   1  MEMORY|SHARED
        [2000000000284000-2000000000288000]         1 page  on node   1  MEMORY|SHARED
        [20000000002b4000-20000000002b8000]         1 page  on node   1  MEMORY|SHARED
        [20000000002c4000-20000000002c8000]         1 page  on node   1  MEMORY|SHARED
        [20000000002d0000-20000000002d8000]         2 pages on node   1  MEMORY|SHARED
        [20000000002dc000-20000000002e0000]         1 page  on node   1  MEMORY|SHARED
```

```
      [2000000000340000-2000000000344000]         1 page  on node   1  MEMORY|SHARED
      [200000000034c000-2000000000358000]         3 pages on node   2  MEMORY|SHARED

                                          ....

  20000000003c8000-20000000003d0000 rw-p 0000000000000000 00:00 0
      [20000000003c8000-20000000003d0000]         2 pages on node   0  MEMORY|DIRTY
```

The `dlook` command gives the name of the process (`Peek:  sleep`), the process ID, and time and date it was invoked. It provides total user and system CPU time in seconds for the process.

Under the heading **Process memory map**, the `dlook` command prints information about a process from the `/proc/`*pid*`/cpu` and `/proc/`*pid*`/maps` files. On the left, it shows the memory segment with the offsets below in decimal. In the middle of the output page, it shows the type of access, time of execution, the PID, and the object that owns the memory (in this case, `/lib/ld-2.2.4.so`). The characters `s` or `p` indicate whether the page is mapped as sharable (`s`) with other processes or is private (`p`). The right side of the output page shows the number of pages of memory consumed and on which nodes the pages reside. A page is 16, 384 bytes. *Dirty memory* means that the memory has been modified by a user.

In the second form of the `dlook` command, you specify a command and optional command arguments. The `dlook` command issues an `exec` call on the command and passes the command arguments. When the process terminates, `dlook` prints information about the process, as shown in the following example:

`dlook` ***`date`***

```
Thu Aug 22 10:39:20 CDT 2002
```

```
Exit:  date
Pid: 4680       Thu Aug 22 10:39:20 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
      [2000000000030000-200000000003c000]         3 pages on node   3  MEMORY|DIRTY

  20000000002dc000-20000000002e4000 rw-p 0000000000000000 00:00 0
      [20000000002dc000-20000000002e4000]         2 pages on node   3  MEMORY|DIRTY
```

```
  2000000000324000-2000000000334000 rw-p 0000000000000000 00:00 0
        [2000000000324000-2000000000328000]         1 page  on node   3  MEMORY|DIRTY

  4000000000000000-400000000000c000 r-xp 0000000000000000 04:03 9657220    /bin/date
        [4000000000000000-400000000000c000]         3 pages on node   1  MEMORY|SHARED

  6000000000008000-6000000000010000 rw-p 0000000000008000 04:03 9657220    /bin/date
        [600000000000c000-6000000000010000]         1 page  on node   3  MEMORY|DIRTY

  6000000000010000-6000000000014000 rwxp 0000000000000000 00:00 0
        [6000000000010000-6000000000014000]         1 page  on node   3  MEMORY|DIRTY

  60000fff80000000-60000fff80004000 rw-p 0000000000000000 00:00 0
        [60000fff80000000-60000fff80004000]         1 page  on node   3  MEMORY|DIRTY

  60000fffffff4000-60000fffffffc000 rwxp ffffffffffffc000 00:00 0
        [60000fffffff4000-60000fffffffc000]         2 pages on node   3  MEMORY|DIRTY
```

If you use the `dlook` command with the `-s` *secs* option, the information is sampled at regular internals. The output for the command **`dlook -s 5 sleep 50`** is as follows:

```
Exit:  sleep
Pid: 5617       Thu Aug 22 11:16:05 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-200000000003c000]            3 pages on node   3  MEMORY|DIRTY

  2000000000134000-2000000000140000 rw-p 0000000000000000 00:00 0

  20000000003a4000-20000000003a8000 rw-p 0000000000000000 00:00 0
        [20000000003a4000-20000000003a8000]            1 page  on node   3  MEMORY|DIRTY

  20000000003e0000-20000000003ec000 rw-p 0000000000000000 00:00 0
        [20000000003e0000-20000000003ec000]            3 pages on node   3  MEMORY|DIRTY

  4000000000000000-4000000000008000 r-xp 0000000000000000 04:03 9657225    /bin/sleep
        [4000000000000000-4000000000008000]            2 pages on node   3  MEMORY|SHARED

  6000000000004000-6000000000008000 rw-p 0000000000004000 04:03 9657225    /bin/sleep
        [6000000000004000-6000000000008000]            1 page  on node   3  MEMORY|DIRTY
```

```
  6000000000008000-600000000000c000 rwxp 0000000000000000 00:00 0
        [6000000000008000-600000000000c000]            1 page  on node   3  MEMORY|DIRTY

  60000fff80000000-60000fff80004000 rw-p 0000000000000000 00:00 0
        [60000fff80000000-60000fff80004000]            1 page  on node   3  MEMORY|DIRTY

  60000fffffff4000-60000fffffffc000 rwxp ffffffffffffc000 00:00 0
        [60000fffffff4000-60000fffffffc000]            2 pages on node   3  MEMORY|DIRTY
```

You can run a Message Passing Interface (MPI) job using the `mpirun` command and print the memory map for each thread, or redirect the ouput to a file, as follows:

Note: The output has been abbreviated to shorten the example and bold headings added for easier reading.

```
mpirun -np 8 dlook -o dlook.out ft.C.8

Contents of dlook.out:
_____________________________________________________________________________
Exit:  ft.C.8
Pid: 2306       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node  21  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
        [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
                                        ...
_____________________________________________________________________________
_____________________________________________________________________________
Exit:  ft.C.8
Pid: 2310       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node  25  MEMORY|DIRTY
```

```
       [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
       [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
       [2000000000050000-2000000000054000]            1 page  on node  25  MEMORY|DIRTY

                                        ...
_____________________________________________________________________________
_____________________________________________________________________________
Exit:  ft.C.8
Pid: 2307       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
       [2000000000030000-2000000000034000]            1 page  on node  30  MEMORY|DIRTY
       [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
       [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
       [2000000000050000-2000000000054000]            1 page  on node  30  MEMORY|DIRTY
                                        ...
_____________________________________________________________________________
_____________________________________________________________________________
Exit:  ft.C.8
Pid: 2308       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
       [2000000000030000-2000000000034000]            1 page  on node   0  MEMORY|DIRTY
       [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

  2000000000044000-2000000000060000 rw-p 0000000000000000 00:00 0
       [2000000000044000-2000000000050000]            3 pages on node  12  MEMORY|DIRTY|SHARED
       [2000000000050000-2000000000054000]            1 page  on node   0  MEMORY|DIRTY
                                        ...
```

For more information on the `dlook` command, see the `dlook` man page.

Installing NUMA Tools

To use the dlook(1), dplace(1), and topology(1) commands, you must load the numatools kernel module. Perform the following steps:

1. To configure numatools kernel module to be started automatically during system startup, use the chkconfig(8) command as follows:

   ```
   chkconfig --add numatools
   ```

2. To turn on numatools, enter the following command:

   ```
   /etc/rc.d/init.d/numatools start
   ```

   This step will be done automatically for subsequent system reboots when numatools are configured on by using the chkconfig(8) utility.

The following steps are required to disable numatools:

1. To turn off numatools, enter the following:

   ```
   /etc/rc.d/init.d/numatools stop
   ```

2. To stop numatools from initiating after a system reboot, use the chkconfig(8) command as follows:

   ```
   chkconfig --del numatools
   ```

Performance Tuning

After analyzing your code to determine where performance bottlenecks are occurring, you can turn your attention to making your programs run their fastest. One way to do this is to use multiple CPUs in parallel processing mode. However, this should be the last step. The first step is to make your program run as efficiently as possible on a single processor system and then consider ways to use parallel processing.

This chapter describes the process of tuning your application for a single processor system, and then tuning it for parallel processing in the following sections:

- "Single Processor Code Tuning"
- "Multiprocessor Code Tuning" on page 70

It also describes how to improve the performance of floating-point programs in

- "Floating-point Programs Performance" on page 82

Single Processor Code Tuning

Several basic steps are used to tune performance of single-processor code:

- Get the expected answers and then tune performance. For details, see "Getting the Correct Results" on page 64.
- Use existing tuned code, such as that found in math libraries and scientific library packages. For details, see "Using Tuned Code" on page 66.
- Determine what needs tuning. For details, see "Determining Tuning Needs" on page 66.
- Use the compiler to do the work. For details, see "Using Compiler Options Where Possible" on page 67.
- Consider tuning cache performance. For details, see "Tuning the Cache Performance" on page 68.
- Set environment variables to enable higher-performance memory management mode. For details, see "Managing Memory" on page 70.

Getting the Correct Results

One of the first steps in performance tuning is to verify that the correct answers are being obtained. Once the correct answers are obtained, tuning can be done. You can verify answers by initially disabling specific optimizations and limiting default optimizations. This can be accomplished by using specific compiler options and by using debugging tools.

The following compiler options emphasize tracing and porting over performance:

- `-O`: the `-O0` option disables all optimization. The default is `-O2`.
- `-g`: the `-g` option preserves symbols for debugging.
- `-mp`: the `-mp` option limits floating-point optimizations and maintains declared precision.
- `-IPF_fltacc`: the `-IPF_fltacc` option disables optimizations that affect floating-point accuracy.
- `-r`:, `-i`: the `-r8` and `-i8` options set default real, integer, and logical sizes to 8 bytes, which are useful for porting codes from Cray, Inc. systems. **This explicitly declares intrinsic and external library functions.**

Some debugging tools can also be used to verify that correct answers are being obtained. See "Debugging Tools" on page 22 for more details.

Managing Heap Corruption Problems

Two methods can be used to check for heap corruption problems in programs that use `glibc malloc/free` dynamic memory management routines: environment variables and Electric Fence.

Set the `MALLOC_CHECK_` environment variable to `1` to print diagnostic messages or to `2` to abort immediately when heap corruption is detected.

Electric Fence is a `malloc` debugger. It aligns either the start or end of an allocated block with an invalid page, causing segmentation faults to occur on buffer overruns or underruns at the point of error. It can also detect accesses to previously freed regions of memory.

Overruns and underruns are circumstances where an access to an array is outside the declared boundary of the array. Underruns and overruns cannot be simultaneously detected. The default behavior is to place inaccessible pages immediately after

allocated memory, but the complementary case can be enabled by setting the EF_PROTECT_BELOW environment variable. To use Electric Fence, link with the libefence library, as shown in the following example:

```
% cat foo.c
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
int i;
int * a;
float *b;
a = (int *)malloc(1000*sizeof (int));
b = (float *)malloc(1000*sizeof (float));

a[0]=1;
for (i=1 ; i<1001;i++)
 {
   a[i]=a[i-1]+1;
}
for (i=1 ; i<1000;i++)
{
  b[i]=a[i-1]*3.14;
}

printf(‘‘answer is %d %f \n’’,a[999],b[999]);

}
```

Compile and run the program as follows (note the error when it is compiled with the library call):

```
% ecc foo.c
% ./a.out
answer is 1000 3136.860107
% ecc foo.c -lefence
% ./a.out
Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens
Segmentation fault
% dbg
```

To avoid potentially large core files, the recommended method of using Electric Fence is from within a debugger. See the `efence` man page for additional details.

Using Tuned Code

Where possible, use code that has already been tuned for optimum hardware performance.

The following mathematical functions should be used where possible to help obtain best results:

- SCSL: SGI's Scientific Computing Software Library. This library includes BLAS, LAPACK, FFT, convolution/correlation, and iterative/direct sparse solver routines. Documentation is available via online man pages (see `intro_scsl`) and the *SCSL User's Guide* available on the SGI Technical Publications Library at http://docs.sgi.com.
- MKL: Intel's Math Kernel Library. This library includes BLAS, LAPACK, and FFT routines.
- VML: the Vector Math Library, available as part of the MKL package (`libmkl_vml_itp.so`).
- Standard Math library

  Standard math library functions are provided with the Intel compiler's `libimf.a` file. If the `-lm` option is specified, `glibc libm` routines are linked in first.

Documentation is available for MKL and VML, as follows:
http://intel.com/software/products/perflib/index.htm?iid=ipp_home+software_libraries&
.

Determining Tuning Needs

Use the following tools to determine what points in your code might benefit from tuning:

- `time`: Use this command to obtain an overview of user, system, and elapsed time.
- `gprof`: Use this tool to obtain an execution profile of your program (a `pcsamp` profile). Use the `-p` compiler option to enable `gprof` use.

- VTune: This Intel performance monitoring tool is a Linux-server, Windows-client application. It supports remote sampling on all Itanium and Linux systems.
- `pfmon`: This performance monitoring tool is designed for Itanium and Linux. It uses the Itanium Performance Monitoring Unit (PMU) to do counting and sampling on unmodified binaries.

For information about other performance analysis tools, see Chapter 3, "Performance Analysis and Debugging" on page 13.

Using Compiler Options Where Possible

Several compiler options can be used to optimize performance. For a short summary of `ifort` or `ecc` options, use the `-help` option on the compiler command line. Use the `-dryrun` option to show the driver tool commands that `ifort` or `ecc` generate. This option does not actually compile.

Use the following options to help tune performance:

- `-ftz`: Flushes underflow to zero to avoid kernel traps. Enabled by default at `-O3` optimization.
- `-fno-alias`: Assumes no pointer aliasing. Pointer aliasing can create uncertainty about the possibility that two unrelated names might refer to the identical memory; because of this uncertainty, the compiler will assume that any two pointers can point to the same location in memory. This can remove optimization opportunities, particularly for loops.

  Other aliasing options include `-ansi_alias` and `-fno_fnalias`. Note that incorrect alias assertions may generate incorrect code.

- `-ip`: Generates single file, interprocedural optimization; `-ipo` generates multifile, interprocedural optimization.

  Most compiler optimizations work within a single procedure (like a function or a subroutine) at a time. This **intra**-procedural focus restricts optimization possibilities because a compiler is forced to make worst-case assumptions about the possible effects of a procedure. By using **inter**-procedural analysis, more than a single procedure is analyzed at once and code is optimized. It performs two passes through the code and requires more compile time.

- `-O3`: Enables `-O2` optimizations plus more aggressive optimizations, including loop transformation and prefetching. *Loop transformation* are found in a

transformation file created by the compiler; you can examine this file to see what suggested changes have been made to loops. *Prefetch instructions* allow data to be moved into the cache before their use. A prefetch instruction is similar to a load instruction.

Note that Level 3 optimization may not improve performance for all programs.

- `-opt_report`: Generates an optimization report and places it in the file specified in `-opt_report_file`.
- `-override_limits`: This is an undocumented option that sometimes allows the compiler to continue optimizing when it has hit an internal limit.
- `-prof_gen` and `-prof_use`: Generates and uses profiling information. These options require a three-step compilation process:
  1. Compile with proper instrumentation using `-prof_gen`.
  2. Run the program on one or more training datasets.
  3. Compile with `-prof_use`, which uses the profile information from the training run.
- `-S`: Compiles and generates an assembly listing in the `.s` files and does not link. The assembly listing can be used in conjunction with the output generated by the `-opt_report` option to try to determine how well the compiler is optimizing loops.

Tuning the Cache Performance

There are several actions you can take to help tune cache performance:

- Avoid large power-of-2 (and multiples thereof) strides and dimensions that cause *cache thrashing*. Cache thrashing occurs when multiple memory accesses require use of the same cache line. This can lead to an unnecessary number of cache misses.

  To prevent cache thrashing, redimension your vectors so that the size is not a power of two. Space the vectors out in memory so that concurrently accessed elements map to different locations in the cache. When working with two-dimensional arrays, make the leading dimension an odd number; for multidimensional arrays, change two or more dimensions to an odd number.

Consider the following example: a cache in the hierarchy has a size of 256 KB (or 65536 4—byte words). A Fortran program contains the following loop:

```
real data(655360,24)
...
do i=1,23
   do j=1,655360
      diff=diff+data(j,i)-data(j,i+1)
   enddo
enddo
```

The two accesses to `data` are separated in memory by 655360*4 bytes, which is a simple multiple of the cache size; they consequently load to the same location in the cache. Because both data items cannot simultaneously coexist in that cache location, a pattern of replace on reload occurs that considerably reduces performance.

- Use a memory stride of 1 wherever possible. A loop over an array should access array elements from adjacent memory addresses. When the loop iterates through memory by consecutive word addresses, it uses every word of every cache line in sequence and does not return to a cache line after finishing it.

  If memory strides other than 1 are used, cache lines could be loaded multiple times if an array is too large to be held in memory at one time.

- Cache bank conflicts can occur if there are two accesses to the same 16-byte-wide bank at the same time. Try different padding of arrays if the output from the `pfmon -e L2_OZQ_CANCELS1_BANK_CONF` command and the output from the `pfmon -e CPU_CYCLES` command shows a high number of bank conflicts relative to total CPU cycles. These can be combined into one command:

  ```
  % pfmon -e CPU_CYCLES,L2_OZQ_CANCELS1_BANK_CONF a.out
  ```

  A maximum of four performance monitoring events can be counted simultaneously.

- Group together data that is used at the same time and do not use vectors in your code, if possible. If elements that are used in one loop iteration are contiguous in memory, it can reduce traffic to the cache and fewer cache lines will be fetched for each iteration of the loop.

- Try to avoid the use of temporary arrays and minimize data copies.

Managing Memory

Nonuniform memory access (NUMA) uses hardware with memory and peripherals distributed among many CPUs. This allows scalability for a shared memory system but a side effect is the time it takes for a CPU to access a memory location. Because memory access times are nonuniform, program optimization is not always straightforward.

Codes that frequently allocate and deallocate memory through `glibc malloc/free` calls may accrue significant system time due to memory management overhead. By default, `glibc` strives for system-wide memory efficiency at the expense of performance.

In compilers up to and including version 7.1.x, to enable the higher-performance memory management mode, set the following environment variables:

```
% setenv MALLOC_TRIM_THRESHOLD_ -1
% setenv MALLOC_MMAP_MAX_ 0
```

Because allocations in `ifort` using the `malloc` intrinsic use the `glibc malloc` internally, these environment variables are also applicable in Fortran codes using, for example, Cray pointers with `malloc/free`. But they do not work for Fortran 90 allocatable arrays, which are managed directly through Fortran library calls and placed in the stack instead of the heap. The example, above, applies only to the `csh` shell and the `tcsh` shell.

Multiprocessor Code Tuning

Before beginning any multiprocessor tuning, first perform single processor tuning. This can often obtain good results in multiprocessor codes also. For details, see "Single Processor Code Tuning" on page 63.

Multiprocessor tuning consists of the following major steps:

- Determine what parts of your code can be parallelized. For background information, see "Data Decomposition" on page 71.
- Choose the parallelization methodology for your code. For details, see "Parallelizing Your Code" on page 72.
- Analyze your code to make sure it is parallelizing properly. For details, see Chapter 3, "Performance Analysis and Debugging" on page 13.

- Check to determine if false sharing exists. For details, see "Fixing False Sharing" on page 76.
- Tune for data placement. For details, see "Using `dplace` and `taskset`" on page 77.
- Use environment variables to assist with tuning. For details, see "Environment Variables for Performance Tuning" on page 77.

Data Decomposition

In order to efficiently use multiple processors on a system, tasks have to be found that can be performed at the same time. There are two basic methods of defining these tasks:

- Functional parallelism

  Functional parallelism is achieved when different processors perform different functions. This is a known approach for programmers trained in modular programming. Disadvantages to this approach include the difficulties of defining functions as the number of processors grow and finding functions that use an equivalent amount of CPU power. This approach may also require large amounts of synchronization and data movement.

- Data parallelism

  Data parallelism is achieved when different processors perform the same function on different parts of the data. This approach takes advantage of the large cumulative memory. One requirement of this approach, though, is that the problem domain be *decomposed*. There are two steps in data parallelism:

  1. Data decomposition

     Data decomposition is breaking up the data and mapping data to processors. Data can be broken up explicitly by the programmer by using message passing (with MPI) and data passing (using the SHMEM library routines) or can be done implicitly using compiler-based MP directives to find parallelism in implicitly decomposed data.

     There are advantages and disadvantages to implicit and explicit data decomposition:

     – **Implicit decomposition advantages**: No data resizing is needed; all synchonization is handled by the compiler; the source code is easier to

develop and is portable to other systems with OpenMP or High Performance Fortran (HPF) support.

- **Implicit decomposition disadvantages**: The data communication is hidden by the user; the compiler technology is not yet mature enough to deliver consistent top performance.
- **Explicit decomposition advantages**: The programmer has full control over insertion of communication and synchronization calls; the source code is portable to other systems; code performance can be better than implicitly parallelized codes.
- **Explicit decomposition disadvantages**: Harder to program; the source code is harder to read and the code is longer (typically 40% more).

2. The final step is to divide the work among processors.

Parallelizing Your Code

The first step in multiprocessor performance tuning is to choose the parallelization methodology that you want to use for tuning. This section discusses those options in more detail.

You should first determine the amount of code that is parallelized. Use the following formula to calculate the amount of code that is parallelized:

```
p=N(T(1)-T(N)) / T(1)(N-1)
```

In this equation, T(1) is the time the code runs on a single CPU and T(N) is the time it runs on N CPUs. Speedup is defined as T(1)/T(N).

If *speedup*/N is less than 50% (that is, N>(2-*p*)/(1-*p*)), stop using more CPUs and tune for better scalability.

CPU activity can be displayed with the `top` or `vmstat` commands or accessed by using the Performance Co-Pilot tools (for example, `pmval kernel.percpu.cpu.user`) or by using the Performance Co-Pilot visualization tools `pmchart`.

Next you should focus on a parallelization methodology, as discussed in the following subsections.

Use MPT

You can use the Message Passing Interface (MPI) from the SGI Message Passing Toolkit (MPT). MPI is optimized and more scalable for SGI Altix series systems than generic MPI libraries. It takes advantage of the SGI Altix architecture and SGI Linux NUMA features. MPT is included with the SGI ProPack for Linux software.

Use the `-lmpi` compiler option to use MPI. For a list of environment variables that are supported, see the `mpi` man page.

`MPIO_DIRECT_READ` and `MPIO_DIRECT_WRITE` are supported under Linux for local XFS filesystems in SGI MPT version 1.6.1 and beyond.

MPI provides the MPI-2 standard MPI I/O functions that provide file read and write capabilities. A number of environment variables are available to tune MPI I/O performance. See the `mpi_io`(3) man page for a description of these environment variables.

Performance tuning for MPI applications is described in more detail in Chapter 6 of the *Message Passing Toolkit (MPT) User's Guide*.

Use XPMEM DAPL Library with MPI

A Direct Access Programming Library (DAPL) is provided for high performance communication between processes on a partitioned or single host Altix system. This can be used with Intel MPI via the remote direct memory access (RDMA) MPI device. See `dapl_xpmem`(3) for more information.

To run Intel MPI with XPMEM DAPL on a system running SGI ProPack 5 for Linux, perform the following commands:

```
setenv I_MPI_DEVICE rdma:xpmem
```

To run an example program, perform commands similar to the following:

```
setenv DAPL_VERSION on    ( to get confirmation that you are using DAPL )
mpirun -np 2 /home/yoursystem/syslib/libtools/bin_linux/mpisanity.intel
```

On SGI ProPack 5 for Linux systems, to use XPMEM DAPL make sure you have the `sgi-dapl` module loaded, as follows:

```
module load sgi-dapl
```

For more information, see the following files:

/opt/sgi-dapl/share/doc/README
/opt/sgi-dapl/share/doc/README.sgi-dapl-api
See the following man pages: dapl_xpmem(3), libdat(3), and dat.conf(4).

Use OpenMP

OpenMP is a shared memory multiprocessing API, which standardizes existing practice. It is scalable for fine or coarse grain parallelism with an emphasis on performance. It exploits the strengths of shared memory and is directive-based. The OpenMP implementation also contains library calls and environment variables.

To use OpenMP directives with C, C++, or Fortran codes, you can use the following compiler options:

- ifort -openmp or ecc -openmp: These options use the OpenMP front-end that is built into the Intel compilers. The resulting executable file makes calls to libguide.so, which is the OpenMP run-time library provided by Intel.
- guide: An alternate command to invoke the Intel compilers to use OpenMP code. Use guidec (in place of ifort), guideefc (in place of ifort), or guidec++ to translate code with OpenMP directives into code with calls to libguide. See "Other Performance Tools" on page 21 for details.

  The -openmp option to ifort is the long-term OpenMP compiler for Linux provided by Intel. However, if you have performance problems with this option, using guide might provide improved performance.

For details about OpenMP usage see the OpenMP standard, available at http://www.openmp.org/specs.

OpenMP Nested Parallelism

This section describes OpenMP nested parallelism. For additional information, see the dplace(1) man page.

Here is a simple example for OpenMP nested parallelism with 2 "top" threads and 4 "bottom" threads that are called master/nested below:

```
% cat place_nested
firsttask cpu=0
thread name=a.out oncpu=0 cpu=4 noplace=1 exact onetime thread name=a.out oncpu=0
```

```
cpu=1-3 exact thread name=a.out oncpu=4 cpu=5-7 exact

% dplace -p place_nested a.out
Master thread 0 running on cpu 0
Master thread 1 running on cpu 4
Nested thread 0 of master 0 gets task 0 on cpu 0 Nested thread 1 of master 0 gets task 1 on cpu 1
Nested thread 2 of master 0 gets task 2 on cpu 2 Nested thread 3 of master 0 gets task 3 on cpu 3
Nested thread 0 of master 1 gets task 0 on cpu 4 Nested thread 1 of master 1 gets task 1 on cpu 5
Nested thread 2 of master 1 gets task 2 on cpu 6 Nested thread 3 of master 1 gets task 3 on cpu 7
```

Use Compiler Options

Use the compiler to invoke automatic parallelization. Use the `-parallel` and `-par_report` option to the `efc` or `ecc` compiler. These options show which loops were parallelized and the reasons why some loops were not parallelized. If a source file contains many loops, it might be necessary to add the `-override_limits` flag to enable automatic parallelization. The code generated by `-parallel` is based on the OpenMP API; the standard OpenMP environment variables and Intel extensions apply.

There are some limitations to automatic parallelization:

- For Fortran codes, only `DO` loops are analyzed
- For C/C++ codes, only `for` loops using explicit array notation or those using pointer increment notation are analyzed. In addition, `for` loops using pointer arithmetic notation are not analyzed nor are `while` or `do/while` loops. The compiler also does not check for blocks of code that can be run in parallel.

Identifying Parallel Opportunities in Existing Code

Another parallelization optimization technique is to identify loops that have a potential for parallelism, such as the following:

- Loops without data dependencies; a *data dependency conflict* occurs when a loop has results from one loop pass that are needed in future passes of the same loop.
- Loops witih data dependencies because of temporary variables, reductions, nested loops, or function calls or subroutines.

Loops that do not have a potential for parallelism are those with premature exits, too few iterations, or those where the programming effort to avoid data dependencies is too great.

Fixing False Sharing

If the parallel version of your program is slower than the serial version, false sharing might be occurring. False sharing occurs when two or more data items that appear not to be accessed by different threads in a shared memory application correspond to the same cache line in the processor data caches. If two threads executing on different CPUs modify the same cache line, the cache line cannot remain resident and correct in both CPUs, and the hardware must move the cache line through the memory subsystem to retain coherency. This causes performance degradation and reduction in the scalability of the application. If the data items are only read, not written, the cache line remains in a shared state on all of the CPUs concerned. False sharing can occur when different threads modify adjacent elements in a shared array. When two CPUs share the same cache line of an array and the cache is decomposed, the boundaries of the chunks split at the chache line.

You can use the following methods to verify that false sharing is happening:

- Use the performance monitor to look at output from `pfmon` and the `BUS_MEM_READ_BRIL_SELF` and `BUS_RD_INVAL_ALL_HITM` events.
- Use `pfmon` to check `DEAR` events to track common cache lines.
- Use the Performance Co-Pilot `pmshub` utility to monitor cache traffic and CPU utilization. You can also use the `shubstats`(1) tool to monitor Altix cache and directory traffic.

If false sharing is a problem, try the following solutions:

- Use the hardware counter to run a profile that monitors storage to shared cache lines. This will show the location of the problem. You can use the `profile.pl -e` command or `histx -e` command. For more information, see "Profiling with `profile.pl`" on page 15, "Using `histx`" on page 16, and the `profile.pl`(1) man page.
- Revise data structures or algorithms.
- Check shared data, static variables, common blocks, and private and public variables in shared objects.
- Use critical regions to identify the part of the code that has the problem.

Using dplace and taskset

The dplace command binds processes to specified CPUs in a round-robin fashion. Once bound to a process, they do not migrate. This is similar to _DSM_MUSTRUN on IRIX systems. dplace numbering is done in the context of the current CPU memory set. See Chapter 4, "Monitoring Tools" on page 27 for details about dplace.

The taskset command restricts execution to the listed set of CPUs; however, processes are still free to move among listed CPUs.

Environment Variables for Performance Tuning

You can use several different environment variables to assist in performance tuning. For details about environment variables used to control the behavior of MPI, see the mpi(1) man page.

Several OpenMP environment variables can affect the actions of the OpenMP library. For example, some environment variables control the behavior of threads in the application when they have no work to perform or are waiting for other threads to arrive at a synchronization semantic; other variables can specify how the OpenMP library schedules iterations of a loop across threads. The following environment variables are part of the OpenMP standard:

- OMP_NUM_THREADS (The default is the number of CPUs in the system.)
- OMP_SCHEDULE (The default is static.)
- OMP_DYNAMIC (The default is false.)
- OMP_NESTED (The default is false.)

In addition to the preceding environment variables, Intel provides several OpenMP extensions, two of which are provided through the use of the KMP_LIBRARY variable.

The KMP_LIBRARY variable sets the run-time execution mode, as follows:

- If set to serial, single-processor execution is used.
- If set to throughput, CPUs yield to other processes when waiting for work. This is the default and is intended to provide good overall system performance in a multiuser environment. This is analogous to the IRIX _DSM_WAIT=YIELD variable.
- If set to turnaround, worker threads do not yield while waiting for work. This is analogous to the IRIX _DSM_WAIT=SPIN variable. Setting KMP_LIBRARY to

`turnaround` may improve the performance of benchmarks run on dedicated systems, where multiple users are not contending for CPU resources.

If your program gets a segmentation fault immediately upon execution, you may need to increase `KMP_STACKSIZE`. This is the private stack size for threads. The default is 4 MB. You may also need to increase your shell stacksize limit.

Understanding Parallel Speedup and Amdahl's Law

There are two ways to obtain the use of multiple CPUs. You can take a conventional program in C, C++, or Fortran, and have the compiler find the parallelism that is implicit in the code.

You can write your source code to use explicit parallelism, stating in the source code which parts of the program are to execute asynchronously, and how the parts are to coordinate with each other.

When your program runs on more than one CPU, its total run time should be less. But how much less? What are the limits on the speedup? That is, if you apply 16 CPUs to the program, should it finish in 1/16th the elapsed time?

This section covers the following topics:

- "Adding CPUs to Shorten Execution Time" on page 78
- "Understanding Parallel Speedup" on page 79
- "Understanding Amdahl's Law" on page 80
- "Calculating the Parallel Fraction of a Program" on page 81
- "Predicting Execution Time with n CPUs" on page 82

Adding CPUs to Shorten Execution Time

You can distribute the work your program does over multiple CPUs. However, there is always some part of the program's logic that has to be executed serially, by a single CPU. This sets the lower limit on program run time.

Suppose there is one loop in which the program spends 50% of the execution time. If you can divide the iterations of this loop so that half of them are done in one CPU

while the other half are done at the same time in a different CPU, the whole loop can be finished in half the time. The result: a 25% reduction in program execution time.

The mathematical treatment of these ideas is called Amdahl's law, for computer pioneer Gene Amdahl, who formalized it. There are two basic limits to the speedup you can achieve by parallel execution:

- The fraction of the program that can be run in parallel, *p*, is never 100%.
- Because of hardware constraints, after a certain point, there is less and less benefit from each added CPU.

Tuning for parallel execution comes down to doing the best that you are able to do within these two limits. You strive to increase the parallel fraction, *p*, because in some cases even a small change in *p* (from 0.8 to 0.85, for example) makes a dramatic change in the effectiveness of added CPUs.

Then you work to ensure that each added CPU does a full CPU's work, and does not interfere with the work of other CPUs. In the SGI Altix architectures this means:

- Spreading the workload equally among the CPUs
- Eliminating false sharing and other types of memory contention between CPUs
- Making sure that the data used by each CPU are located in a memory near that CPU's node

Understanding Parallel Speedup

If half the iterations of a `DO`-loop are performed on one CPU, and the other half run at the same time on a second CPU, the whole `DO`-loop should complete in half the time. For example, consider the typical C loop in Example 6-1.

Example 6-1 Typical C Loop

```
for (j=0; j<MAX; ++j) {
   z[j] = a[j]*b[j];
}
```

The compiler can automatically distribute such a loop over *n* CPUs (with *n* decided at run time based on the available hardware), so that each CPU performs `MAX`/*n* iterations.

The speedup gained from applying *n* CPUs, *Speedup(n)*, is the ratio of the one-CPU execution time to the *n*-CPU execution time: *Speedup(n)* = T*(1)* ÷ T*(n)*. If you measure the one-CPU execution time of a program at 100 seconds, and the program runs in 60 seconds with two CPUs, *Speedup*(2) = 100 ÷ 60 = 1.67.

This number captures the improvement from adding hardware. T(*n*) ought to be less than T(1); if it is not, adding CPUs has made the program slower, and something is wrong! So *Speedup(n)* should be a number greater than 1.0, and the greater it is, the better. Intuitively you might hope that the speedup would be equal to the number of CPUs (twice as many CPUs, half the time) but this ideal can seldom be achieved.

Understanding Superlinear Speedup

You expect *Speedup(n)* to be less than *n*, reflecting the fact that not all parts of a program benefit from parallel execution. However, it is possible, in rare situations, for *Speedup(n)* to be larger than *n*. When the program has been sped up by more than the increase of CPUs it is known as *superlinear speedup*.

A superlinear speedup does not really result from parallel execution. It comes about because each CPU is now working on a smaller set of memory. The problem data handled by any one CPU fits better in cache, so each CPU executes faster than the single CPU could do. A superlinear speedup is welcome, but it indicates that the sequential program was being held back by cache effects.

Understanding Amdahl's Law

There are always parts of a program that you cannot make parallel, where code must run serially. For example, consider the DO-loop. Some amount of code is devoted to setting up the loop, allocating the work between CPUs. This housekeeping must be done serially. Then comes parallel execution of the loop body, with all CPUs running concurrently. At the end of the loop comes more housekeeping that must be done serially; for example, if *n* does not divide MAX evenly, one CPU must execute the few iterations that are left over.

The serial parts of the program cannot be speeded up by concurrency. Let *p* be the fraction of the program's code that can be made parallel (*p* is always a fraction less than 1.0.) The remaining fraction (1–*p*) of the code must run serially. In practical cases, *p* ranges from 0.2 to 0.99.

The potential speedup for a program is proportional to *p* divided by the CPUs you can apply, plus the remaining serial part, 1-*p*. As an equation, this appears as Example 6-2.

Example 6-2 Amdahl's law: *Speedup*(*n*) Given *p*

$$Speedup(n) = \frac{1}{(p/n)+(1-p)}$$

Suppose p = 0.8; then *Speedup*(2) = 1 / (0.4 + 0.2) = 1.67, and *Speedup*(4)= 1 / (0.2 + 0.2) = 2.5. The maximum possible speedup (if you could apply an infinite number of CPUs) would be 1 / (1-*p*). The fraction *p* has a strong effect on the possible speedup.

The reward for parallelization is small unless *p* is substantial (at least 0.8); or to put the point another way, the reward for increasing *p* is great no matter how many CPUs you have. The more CPUs you have, the more benefit you get from increasing *p*. Using only four CPUs, you need only *p*= 0.75 to get half the ideal speedup. With eight CPUs, you need *p*= 0.85 to get half the ideal speedup.

Calculating the Parallel Fraction of a Program

You do not have to guess at the value of *p* for a given program. Measure the execution times T(1) and T(2) to calculate a measured *Speedup*(2) = T(1) / T(2). The Amdahl's law equation can be rearranged to yield *p* when *Speedup* (2) is known, as in Example 6-3.

Example 6-3 Amdahl's law: p Given *Speedup*(2)

$$p = \frac{2}{1} * \frac{SpeedUp(2) - 1}{SpeedUp(2)}$$

Suppose you measure T(1) = 188 seconds and T(2) = 104 seconds.

```
SpeedUp(2) = 188/104 = 1.81
p = 2 * ((1.81-1)/1.81) = 2*(0.81/1.81) = 0.895
```

In some cases, the *Speedup*(2) = T(1)/T(2) is a value greater than 2; in other words, a superlinear speedup ("Understanding Superlinear Speedup" on page 80). When this occurs, the formula in Example 6-3 returns a value of *p* greater than 1.0, which is clearly not useful. In this case you need to calculate *p* from two other more realistic timings, for example T(2) and T(3). The general formula for *p* is shown in Example 6-4, where *n* and *m* are the two CPU counts whose speedups are known, *n*>*m*.

Example 6-4 Amdahl's Law: *p* Given *Speedup*(*n*) and *Speedup*(*m*)

$$p = \frac{Speedup(n) - Speedup(m)}{(1 - 1/n)*Speedup(n) - (1 - 1/m)*Speedup(m)}$$

Predicting Execution Time with n CPUs

You can use the calculated value of *p* to extrapolate the potential speedup with higher numbers of CPUs. The following example shows the expected time with four CPUs, if *p*=0.895 and T(1)=188 seconds:

```
Speedup(4)= 1/((0.895/4)+(1-0.895)) = 3.04
T(4)= T(1)/Speedup(4) = 188/3.04 = 61.8
```

The calculation can be made routine using the computer by creating a script that automates the calculations and extrapolates run times.

These calculations are independent of most programming issues such as language, library, or programming model. They are not independent of hardware issues, because Amdahl's law assumes that all CPUs are equal. At some level of parallelism, adding a CPU no longer affects run time in a linear way. For example, on some architectures, cache-friendly codes scale closely with Amdahl's law up to the maximum number of CPUs, but scaling of memory intensive applications slows as the system bus approaches saturation. When the bus bandwidth limit is reached, the actual speedup is less than predicted.

Floating-point Programs Performance

Certain floating-point programs experience slowdowns due to excessive floating point traps called Floating-Point Software Assist (FPSWA).

This happens when the hardware cannot complete a floating point operation and requests help (emulation) from software. This happens, for instance, with denormals numbers. See the following document for more details:

http://www.intel.com/design/itanium/downloads/245415.htm

The symptoms are a slower than normal execution, FPSWA message in the system log (run dmesg). The average cost of a FPSWA fault is quite high around 1000 cycles / fault.

By default, the kernel prints a message similar to the following in the system log:

```
foo(7716): floating-point assist fault at ip 40000000000200e1
           isr 0000020000000008
```

The kernel throttles the message in order to avoid flooding the console.

It is possible to control the behavior of the kernel on FPSWA faults using the `prctl`(1) command. In particular, it is possible to get a signal delivered at the first FPSWA. It is also possible to silence the console message.

Using `pfmon` you can count `fp_true_sirstall` to test for FPSWA faults, as follows:

```
$ pfmon --no-qual-check -ku --drange=fpswa_interface \
-eloads_retired,ia64_inst_retired,fp_true_sirstall -- test-fpswa

      1 LOADS_RETIRED
2615140 IA64_INST_RETIRED
```

To see a list of available options, use the `pfmon - help` command.

Flexible File I/O

Flexible File I/O (FFIO) provides a mechanism for improving the file I/O performance of existing applications without having to resort to source code changes, that is, the current executable remains unchanged. Knowledge of source code is not required, but some knowledge of how the source and the application software work can help you better interpret and optimize FFIO results. To take advantage of FFIO, all you need to do is to set some environment variables before running your application. This chapter covers the following topics:

- "FFIO Operation" on page 85
- "Environment Variables" on page 86
- "Simple Examples" on page 87
- "Multithreading Considerations" on page 90
- "Application Examples " on page 91
- "Event Tracing " on page 92
- "System Information and Issues " on page 92

FFIO Operation

The FFIO subsystem allows you to define one or more additional I/O buffer caches for specific files to augment the Linux kernel I/O buffer cache. The FFIO subsystem then manages this buffer cache for you. In order to accomplish this, FFIO intercepts standard I/O calls like open, read, and write, and replaces them with FFIO equivalent routines. These routines route I/O requests through the FFIO subsystem which utilizes the user defined FFIO buffer cache. FFIO can bypass the Linux kernel I/O buffer cache by communicating with the disk subsystem via direct I/O. This gives you precise control over cache I/O characteristics and allows for more efficient I/O requests. For example, doing direct I/O in large chunks (say 16 megabytes) allows the FFIO cache to amortize disk access. All file buffering occurs in user space when FFIO is used with direct I/O enabled. This differs from the Linux buffer cache mechanism which requires a context switch in order to buffer data in kernel memory. Avoiding this kind of overhead, helps FFIO to scale efficiently. Another important distinction is that FFIO allows you to create an I/O buffer cache dedicated to a specific application.

The Linux kernel, on the other hand, has to manage all the jobs on the entire system with a single I/O buffer cache. As a result, FFIO typically outperforms the Linux kernel buffer cache when it comes to I/O intensive throughput.

Environment Variables

There are only two environment variables that you need to set in order to use FFIO. They are LD_PRELOAD and FF_IO_OPTS.

In order to enable FFIO to trap standard I/O calls, you must set the LD_PRELOAD environment variable.

For SGI Altix 4000 series systems, perform the following:

```
setenv LD_PRELOAD /usr/lib/libFFIO.so
```

For SGI Altix XE systems, perform the following:

```
setenv LD_PRELOAD /usr/lib64/libFFIO.so
```

The LD_PRELOAD software is a Linux feature that instructs the linker to preload the indicated shared libraries. In this case, libFFIO.so is preloaded and provides the routines which replace the standard I/O calls. An application that is not dynamically linked with the glibc library will not work with FFIO, since the standard I/O calls will not be intercepted. To disable FFIO, perform the following:

```
unsetenv LD_PRELOAD
```

The FFIO buffer cache is managed by the FF_IO_OPTS environment variable. The syntax for setting this variable can be quite complex. A simple method for defining this variable is, as follows:

```
setenv FF_IO_OPTS  '<string>(eie.direct.mbytes:<size>:<num>:<lead>:<share>:<stride>:0)'
```

You can use the following parameters with the FF_IO_OPTS environment variable:

<string>	Matches the names of files that can use the buffer cache.
<size>	Number of 4k blocks in each page of the I/O buffer cache.
<num>	Number of pages in the I/O buffer cache.
<lead>	The maximum number of "read ahead" pages.
<share>	A value of 1 means a shared cache, 0 means private

<stride>	Note that the number after the stride parameter is always 0.

The following example shows a command that creates a shared buffer cache of 128 pages where each page is 16 megabytes (that is, 4096*4k). The cache has a lead of six pages and uses a stride of one, as follows:

```
setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0)'
```

Each time the application opens a file, the FFIO code checks the file name to see if it matches the string supplied by FF_IO_OPTS. The file's path name is not considered when checking for a match against the string. So in the example supplied above, file names like /tmp/test16 and /var/tmp/testit would both be a match.

More complicated usages of FF_IO_OPTS are built upon this simpler version. For example, multiple types of file names can share the same cache, as follows:

```
setenv FF_IO_OPTS 'output* test*(eie.direct.mbytes:4096:128:6:1:1:0)'
```

Multiple caches may also be specified with FF_IO_OPTS. In the example that follows, files of the form output* and test* share a 128 page cache of 16 megabyte pages. The file special42 has a 256 page private cache of 32 megabyte pages, as follows:

```
setenv FF_IO_OPTS 'output* test*(eie.direct.mbytes:4096:128:6:1:1:0) special42(eie.direct.mbytes:8192:25
```

Additional parameters can be added to FF_IO_OPTS to create feedback that is sent to standard output. Examples of doing this diagnostic output will be presented in the following section.

Simple Examples

This section walks you through some simple examples using FFIO.

Assume that LD_PRELOAD is set for the correct library and FF_IO_OPTS is defined, as follows:

```
setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0)'
```

This example uses a small C program called fio that reads four megabyte chunks from a file for 100 iterations. When the program runs it produces output, as follows:

```
./fio -n 100 /build/testit
Reading 4194304 bytes 100 times to /build/testit
Total time  = 7.383761
```

```
Throughput  = 56.804439 MB/sec
```

It can be difficult to tell what FFIO may or may not be doing even with a simple program such as shown above. A summary of the FFIO operations that occurred can be directed to standard output by making a simple addition to FF_IO_OPTS, as follows:

```
setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0, event.summary.mbytes.notrace )'
```

This new setting for FF_IO_OPTS generates the following summary on standard output when the program is run:

```
./fio -n 100 /build/testit
Reading 4194304 bytes 100 times to /build/testit
Total time  = 7.383761
Throughput  = 56.804439 MB/sec

event_close(testit)    eie <-->syscall   (496 mbytes)/( 8.72 s)=   56.85 mbytes/s
oflags=0x0000000000004042=RDWR+CREAT+DIRECT
sector size =4096(bytes)
cblks =0  cbits =0x0000000000000000
current file size =512 mbytes   high water file size =512 mbytes

function     times      wall     all     mbytes     mbytes       min        max       avg
             called     time    hidden  requested  delivered   request    request    request
    open          1     0.00
    read          2     0.61                32         32        16         16         16
    reada        29     0.01       0       464        464        16         16         16
    fcntl
       recall
       reada     29     8.11
       other      5     0.00
    flush         1     0.00
    close         1     0.00
```

Two synchronous reads of 16 megabytes each were issued (for a total of 32 megabytes) and 29 asynchronous reads (reada) were also issued (for a total of 464 megabytes). Additional diagnostic information can be generated by specifying the .diag modifier, as follows:

```
setenv FF_IO_OPTS 'test*(eie.direct.diag.mbytes:4096:128:6:1:1:0 )'
```

The `.diag` modifier may also be used in conjunction with `.event.summary`, the two operate independently from one another, as follows:

```
setenv FF_IO_OPTS 'test*(eie.diag.direct.mbytes:4096:128:6:1:1:0, event.summary.mbytes.notrace )'
```

An example of the diagnostic output generated when just the `.diag` modifier is used is, as follows:

```
./fio -n 100 /build/testit
Reading 4194304 bytes 100 times to /build/testit
Total time  = 7.383761
Throughput  = 56.804439 MB/sec

eie_close EIE final stats for file /build/testit
eie_close  Used shared eie cache 1
eie_close  128 mem pages of 4096 blocks (4096 sectors), max_lead = 6 pages
eie_close  advance reads used/started :       23/29    79.31%   (1.78 seconds wasted)
eie_close  write hits/total           :        0/0      0.00%
eie_close  read  hits/total           :       98/100   98.00%
eie_close  mbytes transferred    parent --> eie --> child      sync        async
eie_close                                 0            0        0            0
eie_close                               400          496        2           29 (0,0)
eie_close                          parent <-- eie <-- child

eie_close EIE stats for Shared cache 1
eie_close  128 mem pages of 4096 blocks
eie_close  advance reads used/started :       23/29    79.31%   (0.00 seconds wasted)
eie_close  write hits/total           :        0/0      0.00%
eie_close  read  hits/total           :       98/100   98.00%
eie_close  mbytes transferred    parent --> eie --> child      sync        async
eie_close                                 0                    0            0
eie_close                               400         496        2           29 (0,0)
```

Information is listed for both the file and the cache. An `mbytes transferred` example is shown below:

```
eie_close  mbytes transferred    parent --> eie --> child      sync        async
eie_close                                 0                    0            0
eie_close                               400         496        2           29 (0
```

The last two lines are for write and read operations, respectively. Only for very simple I/O patterns, the difference between (parent –> eie) and (eie –> child) read statistics

can be explained by the number of read aheads. For random reads of a large file over a long period of time, this is not the case. All write operations count as async.

Multithreading Considerations

FFIO will work with applications that use MPI for parallel processing. An MPI job assigns each thread a number or rank. The master thread has rank 0, while the remaining threads (called slave threads) have ranks from 1 to N-l where N is the total number of threads in the MPI job. It is important to consider that the threads comprising an MPI job do not (necessarily) have access to each others address space. As a result, there is no way for the different MPI threads to share the same FFIO cache. By default, each thread defines a separate FFIO cache based on the parameters defined by FF_IO_OPTS.

Having each MPI thread define a separate FFIO cache based on a single environment variable (FF_IO_OPTS) can waste a lot of memory. Fortunately, FFIO provides a mechanism that allows the user to specify a different FFIO cache for each MPI thread via the following environment variables:

```
setenv FF_IO_OPTS_RANK0 'result*(eie.direct.mbytes:4096:512:6:1:1:0)'
setenv FF_IO_OPTS_RANK1 'output*(eie.direct.mbytes:1024:128:6:1:1:0)'
setenv FF_IO_OPTS_RANK2 'input*(eie.direct.mbytes:2048:64:6:1:1:0)'
             .
             .
             .
setenv FF_IO_OPTS_RANKN-1 ...   (N = number of threads).
```

Each rank environment variable is set using the exact same syntax as FF_IO_OPTS and each defines a distinct cache for the corresponding MPI rank. If the cache is designated shared, all files within the same ranking thread will use the same cache. FFIO works with SGI MPI, HP MPI, and LAM MPI. In order to work with MPI applications, FFIO needs to determine the rank of callers by invoking the mpi_comm_rank_() MPI library routine . Therefore, FFIO needs to determine the location of the MPI library used by the application. This is accomplished by having the user set one (and only one) of the following environment variables:

```
setenv SGI_MPI /usr/lib   # ia64 only
         or
setenv LAM_MPI *see below
         or
setenv HP_MPI  *see below
```

```
*LAM and HP MPIs are usually distributed via a third party application.  The preci
paths  to the LAM and the HP MPI libraries are application dependent.  Please refe
application installation guide to find the correct path.
```

In order to use the rank functionality, both the MPI and FF_IO_OPTS_RANK0 environment variables must be set. If either variable is not set, then the MPI threads all use FF_IO_OPTS. If both the MPI and the FF_IO_OPTS_RANK0 variables are defined but, for example, FF_IO_OPTS_RANK2 is undefined, all rank 2 files would generate a no match with FFIO. This means that none of the rank 2 files would be cached by FFIO (in this case things DO NOT default to FF_IO_OPTS).

Fortran and C/C++ applications that use the pthreads interface will create threads that share the same address space. These threads can all make use of the single FFIO cache defined by FF_IO_OPTS.

Application Examples

FFIO has been deployed successfully with several HPC applications such as Nastran and Abaqus. In a recent customer benchmark, an eight-way Abaqus throughput job ran approximately twice as fast when FFIO was used. The FFIO cache used 16 megabyte pages (that is, page_size = 4096) and the cache size was 8.0 gigabytes. As a rule of thumb, it was determined that setting the FFIO cache size to roughly 10-15% of the disk space required by Abaqus yielded reasonable I/O performance. For this benchmark, the FF_IO_OPTS environment variable was defined by:

```
setenv FF_IO_OPTS '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
       *.nck* *.sct *.lop *.ngr *.elm *.ptn* *.stp* *.eig *.lnz* *.mass *.inp* *.scn* *.ddm
       *.dat* fort*(eie.direct.nodiag.mbytes:4096:512:6:1:1:0,event.summary.mbytes.notrace)'
```

For the MPI version of Abaqus, different caches were specified for each MPI rank, as follows:

```
setenv FF_IO_OPTS_RANK0 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
       *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
       *.dat* fort*(eie.direct.nodiag.mbytes:4096:512:6:1:1:0,event.summary.mbytes.notrace)'

setenv FF_IO_OPTS_RANK1 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
       *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
       *.dat* fort*(eie.direct.nodiag.mbytes:4096:16:6:1:1:0,event.summary.mbytes.notrace)'
```

```
setenv FF_IO_OPTS_RANK2 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
       *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
       *.dat* fort*(eie.direct.nodiag.mbytes:4096:16:6:1:1:0,event.summary.mbytes.notrace)'

setenv FF_IO_OPTS_RANK3 '*.fct *.opr* *.ord *.fil *.mdl* *.stt* *.res *.sst *.hdx *.odb* *.023
       *.nck* *.sct *.lop *.ngr *.ptn* *.stp* *.elm *.eig *.lnz* *.mass *.inp *.scn* *.ddm
       *.dat* fort*(eie.direct.nodiag.mbytes:4096:16:6:1:1:0,event.summary.mbytes.notrace)'
```

Event Tracing

By specifying the `.trace` option as part of the event parameter the user can enable the event tracing feature in FFIO, as follows:

```
setenv FF_IO_OPTS 'test*(eie.direct.mbytes:4096:128:6:1:1:0, event.summary.mbytes.trace )'
```

This option generates files of the form `ffio.events.pid` for each process that is part of the application. By default, event files are placed in `/tmp` but this destination can be changed by setting the `FFIO_TMPDIR` environment variable. These files contain time stamped events for files using the FFIO cache and can be used to trace I/O activity (for example, I/O sizes and offsets).

System Information and Issues

The SGI ProPack 5 Service Pack 1 release provided the first stable version of FFIO. Applications written in C, C++, and Fortran are supported. C and C++ applications can be built with either the Intel or gcc compiler. Only Fortran codes built with the Intel compiler will work.

The following restrictions on FFIO must also be observed:

- The FFIO implementation of `pread`/`pwrite` is not correct (the file offset advances).
- Do not use FFIO to do I/O on a socket.
- Do not link your application with the `librt` asynchronous I/O library.
- Calls that operate on files in `/proc`, `/etc`, and `/dev` are not intercepted by FFIO.
- Calls that operate on `stdin`, `stdout`, and `stderr` are not intercepted by FFIO.
- FFIO is not intended for generic I/O applications such as `vi`, `cp`, or `mv`, and so on.

I/O Tuning

This chapter describes tuning information that you can use to improve I/O throughput and latency.

Layout of Filesystems and XVM for Multiple RAIDs

There can be latency spikes in response from a RAID and such a spikes can in effect slow down all of the RAIDs as one I/O completion waits for all of the striped pieces to complete.

These latency spikes impact on throughput may be to stall all the I/O or to delay a few I/Os while others continue. It depends on how the I/O is striped across the devices. If the volumes are constructed as stripes to span all devices, and the I/Os are sized to be full stripes, the I/Os will stall, since every I/O has to touch every device. If the I/Os can be completed by touching a subset of the devices, then those that do not touch a high latency device can continue at full speed, while the stalled I/Os can complete and catch up later.

In large storage configurations, it is possible to lay out the volumes to maximize the opportunity for the I/Os to proceed in parallel, masking most of the effect of a few instances of high latency.

There are at least three classes of events that cause high latency I/O operations, as follows:

- Transient disk delays - one disk pauses
- Slow disks
- Transient RAID controller delays

The first two events affect a single logical unit number (LUN). The third event affects all the LUNs on a controller. The first and third events appear to happen at random. The second event is repeatable.

Suggested Shortcuts and Workarounds

This chapter contains suggested workarounds and shortcuts that you can use on your SGI Altix system. It covers the following topics:

- "Determining Process Placement" on page 95
- "Resetting System Limits" on page 102
- "Linux Shared Memory Accounting" on page 106

Determining Process Placement

This section describes methods that can be used to determine where different processes are running. This can help you understand your application structure and help you decide if there are obvious placement issues.

There are some set-up steps to follow before determining process placement (note that all examples use the C shell):

1. Set up an alias as in this example, changing *guest* to your username:

   ```
   % pu
   % alias pu "ps -edaf|grep guest"
   ```

   The `pu` command shows current processes.

2. Create the `.toprc` preferences file in your login directory to set the appropriate `top` options. If you prefer to use the `top` defaults, delete the `.toprc` file.

   ```
   % cat <<EOF>> $HOME/.toprc

   YEAbcDgHIjklMnoTP|qrsuzV{FWX
   2mlt
   EOF
   ```

3. Inspect all processes and determine which CPU is in use and create an alias file for this procedure. The CPU number is shown in the first column of the `top` output:

   ```
   % top -b -n 1 | sort -n | more
   % alias top1 "top -b -n 1 | sort -n "
   ```

Use the following variation to produce output with column headings:

```
% alias top1 "top -b -n 1 | head -4 | tail -1;top -b -n 1 | sort -n"
```

4. View your files (replacing *guest* with your username):

```
% top -b -n 1 | sort -n | grep guest
```

Use the following variation to produce output with column headings:

```
% top -b -n 1 | head -4 | tail -1;top -b -n 1 | sort -n grep guest
```

Example Using pthreads

The following example demonstrates a simple usage with a program name of `th`. It sets the number of desired OpenMP threads and runs the program. Notice the process hierarchy as shown by the PID and the PPID columns. The command usage is the following, where *n* is the number of threads:

```
% th n
```

```
% th 4
% pu

UID       PID   PPID   C STIME TTY          TIME CMD
root      13784 13779  0 12:41 pts/3    00:00:00 login --
guest1
guest1    13785 13784  0 12:41 pts/3    00:00:00 -csh
guest1    15062 13785  0 15:23 pts/3    00:00:00 th 4   <-- Main thread
guest1    15063 15062  0 15:23 pts/3    00:00:00 th 4   <-- daemon thread
guest1    15064 15063 99 15:23 pts/3    00:00:10 th 4   <-- worker thread 1
guest1    15065 15063 99 15:23 pts/3    00:00:10 th 4   <-- worker thread 2
guest1    15066 15063 99 15:23 pts/3    00:00:10 th 4   <-- worker thread 3
guest1    15067 15063 99 15:23 pts/3    00:00:10 th 4   <-- worker thread 4
guest1    15068 13857  0 15:23 pts/5    00:00:00 ps -aef
guest1    15069 13857  0 15:23 pts/5    00:00:00 grep guest1

% top -b -n 1 | sort -n | grep guest1

LC %CPU   PID USER     PRI  NI  SIZE  RSS SHARE STAT %MEM   TIME COMMAND
 3  0.0 15072 guest1     16   0  3488 1536  3328 S     0.0   0:00 grep
 5  0.0 13785 guest1     15   0  5872 3664  4592 S     0.0   0:00 csh
 5  0.0 15062 guest1     16   0 15824 2080  4384 S     0.0   0:00 th
 5  0.0 15063 guest1     15   0 15824 2080  4384 S     0.0   0:00 th
```

```
 5 99.8 15064 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
 7  0.0 13826 guest1     18   0  5824 3552  5632 S     0.0   0:00 csh
10 99.9 15066 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
11 99.9 15067 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
13 99.9 15065 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
15  0.0 13857 guest1     15   0  5840 3584  5648 S     0.0   0:00 csh
15  0.0 15071 guest1     16   0 70048 1600 69840 S     0.0   0:00 ort
15  1.5 15070 guest1     15   0  5056 2832  4288 R     0.0   0:00top
```

Now skip the `Main` and daemon processes and place the rest:

```
% usr/bin/dplace -s 2 -c 4-7 th 4
% pu

UID         PID  PPID  C STIME TTY          TIME CMD
root      13784 13779  0 12:41 pts/3    00:00:00 login --
guest1
guest1    13785 13784  0 12:41 pts/3    00:00:00 -csh
guest1    15083 13785  0 15:25 pts/3    00:00:00 th 4
guest1    15084 15083  0 15:25 pts/3    00:00:00 th 4
guest1    15085 15084 99 15:25 pts/3    00:00:19 th 4
guest1    15086 15084 99 15:25 pts/3    00:00:19 th 4
guest1    15087 15084 99 15:25 pts/3    00:00:19 th 4
guest1    15088 15084 99 15:25 pts/3    00:00:19 th 4
guest1    15091 13857  0 15:25 pts/5    00:00:00 ps -aef
guest1    15092 13857  0 15:25 pts/5    00:00:00 grep guest1

% top -b -n 1 | sort -n | grep guest1

LC %CPU   PID USER      PRI  NI  SIZE  RSS SHARE STAT %MEM   TIME COMMAND
 4 99.9 15085 guest1     25   0 15856 2096  6496 R     0.0   0:24 th
 5 99.8 15086 guest1     25   0 15856 2096  6496 R     0.0   0:24 th
 6 99.9 15087 guest1     25   0 15856 2096  6496 R     0.0   0:24 th
 7 99.9 15088 guest1     25   0 15856 2096  6496 R     0.0   0:24 th
 8  0.0 15095 guest1     16   0  3488 1536  3328 S     0.0   0:00 grep
12  0.0 13785 guest1     15   0  5872 3664  4592 S     0.0   0:00 csh
12  0.0 15083 guest1     16   0 15856 2096  6496 S     0.0   0:00 th
12  0.0 15084 guest1     15   0 15856 2096  6496 S     0.0   0:00 th
15  0.0 15094 guest1     16   0 70048 1600 69840 S     0.0   0:00 sort
15  1.6 15093 guest1     15   0  5056 2832  4288 R     0.0   0:00 top
```

Example Using OpenMP

The following example demonstrates a simple OpenMP usage with a program name of `md`. Set the desired number of OpenMP threads and run the program, as shown below:

```
% alias pu "ps -edaf | grep guest1
% setenv OMP_NUM_THREADS 4
% md
```

The following output is created:

```
% pu

UID          PID  PPID  C STIME TTY          TIME CMD
root       21550 21535  0 21:48 pts/0    00:00:00 login -- guest1
guest1     21551 21550  0 21:48 pts/0    00:00:00 -csh
guest1     22183 21551 77 22:39 pts/0    00:00:03 md    <-- parent / main
guest1     22184 22183  0 22:39 pts/0    00:00:00 md    <-- daemon
guest1     22185 22184  0 22:39 pts/0    00:00:00 md    <-- daemon helper
guest1     22186 22184 99 22:39 pts/0    00:00:03 md    <-- thread 1
guest1     22187 22184 94 22:39 pts/0    00:00:03 md    <-- thread 2
guest1     22188 22184 85 22:39 pts/0    00:00:03 md    <-- thread 3
guest1     22189 21956  0 22:39 pts/1    00:00:00 ps -aef
guest1     22190 21956  0 22:39 pts/1    00:00:00 grep guest1

% top -b -n 1 | sort -n | grep guest1

LC %CPU   PID USER      PRI  NI  SIZE  RSS SHARE STAT %MEM   TIME COMMAND
 2  0.0 22192 guest1     16   0 70048 1600 69840 S     0.0   0:00 sort
 2  0.0 22193 guest1     16   0  3488 1536  3328 S     0.0   0:00 grep
 2  1.6 22191 guest1     15   0  5056 2832  4288 R     0.0   0:00 top
 4 98.0 22186 guest1     26   0 26432 2704  4272 R     0.0   0:11 md
 8  0.0 22185 guest1     15   0 26432 2704  4272 S     0.0   0:00 md
 8 87.6 22188 guest1     25   0 26432 2704  4272 R     0.0   0:10 md
 9  0.0 21551 guest1     15   0  5872 3648  4560 S     0.0   0:00 csh
 9  0.0 22184 guest1     15   0 26432 2704  4272 S     0.0   0:00 md
 9 99.9 22183 guest1     39   0 26432 2704  4272 R     0.0   0:11 md
14 98.7 22187 guest1     39   0 26432 2704  4272 R     0.0   0:11 md
```

From the notation on the right of the `pu` list, you can see the `-x 6` pattern.

```
place 1, skip 2 of them, place 3 more  [ 0 1 1 0 0 0 ]
  now, reverse the bit order and create the dplace -x mask
  [ 0 0 0 1 1 0 ]  -->  [ 0x06 ]  --> decimal 6
  dplace does not currently process hex notation for this bit mask)
```

The following example confirms that a simple `dplace` placement works correctly:

```
% setenv OMP_NUM_THREADS 4
% /usr/bin/dplace -x 6 -c 4-7 md
% pu
UID         PID  PPID  C STIME TTY          TIME CMD
root      21550 21535  0 21:48 pts/0    00:00:00 login -- guest1
guest1    21551 21550  0 21:48 pts/0    00:00:00 -csh
guest1    22219 21551 93 22:45 pts/0    00:00:05 md
guest1    22220 22219  0 22:45 pts/0    00:00:00 md
guest1    22221 22220  0 22:45 pts/0    00:00:00 md
guest1    22222 22220 93 22:45 pts/0    00:00:05 md
guest1    22223 22220 93 22:45 pts/0    00:00:05 md
guest1    22224 22220 90 22:45 pts/0    00:00:05 md
guest1    22225 21956  0 22:45 pts/1    00:00:00 ps -aef
guest1    22226 21956  0 22:45 pts/1    00:00:00 grep guest1

% top -b -n 1 | sort -n | grep guest1

LC %CPU   PID USER      PRI  NI  SIZE  RSS SHARE STAT %MEM   TIME COMMAND
 2  0.0 22228 guest1     16   0 70048 1600 69840 S     0.0   0:00 sort
 2  0.0 22229 guest1     16   0  3488 1536  3328 S     0.0   0:00 grep
 2  1.6 22227 guest1     15   0  5056 2832  4288 R     0.0   0:00 top
 4  0.0 22220 guest1     15   0 28496 2736 21728 S     0.0   0:00 md
 4 99.9 22219 guest1     39   0 28496 2736 21728 R     0.0   0:12 md
 5 99.9 22222 guest1     25   0 28496 2736 21728 R     0.0   0:11 md
 6 99.9 22223 guest1     39   0 28496 2736 21728 R     0.0   0:11 md
 7 99.9 22224 guest1     39   0 28496 2736 21728 R     0.0   0:11 md
 9  0.0 21551 guest1     15   0  5872 3648  4560 S     0.0   0:00 csh
15  0.0 22221 guest1     15   0 28496 2736 21728 S     0.0   0:00 md
```

Combination Example (MPI and OpenMP)

For this example, explicit placement using the `dplace -e -c` command is used to achieve the desired placement. If an `x` is used in one of the CPU positions, `dplace` does not explicitly place that process.

If running without a cpuset, the `x` processes run on any available CPU.

If running with a cpuset, you have to renumber the CPU numbers to refer to “logical” CPUs (`0 ...` *n)* within the cpuset, regardless of which physical CPUs are in the cpuset. When running in a cpuset, the unplaced processes are constrained to the set of CPUs within the cpuset.

For details about cpuset usage, see the *Linux Resource Administration Guide.*

The following example shows a “hybrid” MPI and OpenMP job with two MPI processes, each with two OpenMP threads and no cpusets:

```
% setenv OMP_NUM_THREADS 2
% efc -O2 -o hybrid hybrid.f -lmpi -openmp

% mpirun -v -np 2 /usr/bin/dplace -e -c x,8,9,x,x,x,x,10,11 hybrid

-------------------------
# if using cpusets ...
-------------------------
# we need to reorder cpus to logical within the 8-15 set [0-7]

% cpuset -q omp -A mpirun -v -np 2 /usr/bin/dplace -e -c x,0,1,x,x,x,x,2,3,4,5,6,7 hybrid

# We need a table of options for these pairs. "x" means don't
# care. See the dplace man page for more info about the -e option.
# examples at end

  -np  OMP_NUM_THREADS  /usr/bin/dplace -e -c <as shown> a.out
  ---  ---------------  ------------------------------------
   2         2          x,0,1,x,x,x,x,2,3
   2         3          x,0,1,x,x,x,x,2,3,4,5
   2         4          x,0,1,x,x,x,x,2,3,4,5,6,7

   4         2          x,0,1,2,3,x,x,x,x,x,x,x,x,4,5,6,7
   4         3
x,0,1,2,3,x,x,x,x,x,x,x,x,4,5,6,7,8,9,10,11
   4         4
x,0,1,2,3,x,x,x,x,x,x,x,x,4,5,6,7,8,9,10,11,12,13,14,15
   Notes:               0 <- 1 -> <- 2 -> <- 3 -> <------ 4
------------------>
```

```
   Notes:
     0. mpi daemon process
     1. mpi child procs, one per np
     2. omp daemon procs, one per np
     3. omp daemon helper procs, one per np
     4. omp thread procs, (OMP_NUM_THREADS - 1) per np

-------------------------------------------
# Example -   -np 2 and OMP_NUM_THREADS 2
-------------------------------------------

% setenv OMP_NUM_THREADS 2
% efc -O2 -o hybrid hybrid.f -lmpi -openmp

% mpirun -v -np 2 /usr/bin/dplace -e -c x,8,9,x,x,x,x,10,11 hybrid

% pu

UID        PID  PPID  C STIME TTY          TIME CMD
root  21550 21535  0 Mar17 pts/0 00:00:00 login -- guest1
guest1 21551 21550  0 Mar17 pts/0 00:00:00 -csh
guest1 23391 21551  0 00:32 pts/0 00:00:00 mpirun -v -np 2

/usr/bin/dplace
guest1 23394 23391  2 00:32 pts/0 00:00:00 hybrid   <-- mpi daemon
guest1 23401 23394 99 00:32 pts/0 00:00:03 hybrid   <-- mpi child 1
guest1 23402 23394 99 00:32 pts/0 00:00:03 hybrid   <-- mpi child 2
guest1 23403 23402  0 00:32 pts/0 00:00:00 hybrid   <-- omp daemon 2
guest1 23404 23401  0 00:32 pts/0 00:00:00 hybrid   <-- omp daemon 1
guest1 23405 23404  0 00:32 pts/0 00:00:00 hybrid   <-- omp daemon hlpr 1
guest1 23406 23403  0 00:32 pts/0 00:00:00 hybrid   <-- omp daemon hlpr 2
guest1 23407 23403 99 00:32 pts/0 00:00:03 hybrid   <-- omp thread 2-1
guest1 23408 23404 99 00:32 pts/0 00:00:03 hybrid   <-- omp thread 1-1
guest1 23409 21956  0 00:32 pts/1 00:00:00 ps -aef
guest1 23410 21956  0 00:32 pts/1 00:00:00 grep guest1

% top -b -n 1 | sort -n | grep guest1

LC %CPU   PID USER      PRI  NI  SIZE  RSS SHARE STAT %MEM   TIME COMMAND
 0  0.0 21551 guest1     15   0  5904 3712  4592 S     0.0   0:00 csh
 0  0.0 23394 guest1     15   0  883M 9456  882M S     0.1   0:00 hybrid
```

```
 4  0.0 21956 guest1     15   0  5856 3616  5664 S     0.0   0:00 csh
 4  0.0 23412 guest1     16   0 70048 1600 69840 S     0.0   0:00 sort
 4  1.6 23411 guest1     15   0  5056 2832  4288 R     0.0   0:00 top
 5  0.0 23413 guest1     16   0  3488 1536  3328 S     0.0   0:00 grep
 8  0.0 22005 guest1     15   0  5840 3584  5648 S     0.0   0:00 csh
 8  0.0 23404 guest1     15   0  894M  10M  889M S     0.1   0:00 hybrid
 8 99.9 23401 guest1     39   0  894M  10M  889M R     0.1   0:09 hybrid
 9  0.0 23403 guest1     15   0  894M  10M  894M S     0.1   0:00 hybrid
 9 99.9 23402 guest1     25   0  894M  10M  894M R     0.1   0:09 hybrid
10 99.9 23407 guest1     25   0  894M  10M  894M R     0.1   0:09 hybrid
11 99.9 23408 guest1     25   0  894M  10M  889M R     0.1   0:09 hybrid
12  0.0 23391 guest1     15   0  5072 2928  4400 S     0.0   0:00 mpirun
12  0.0 23406 guest1     15   0  894M  10M  894M S     0.1   0:00 hybrid
14  0.0 23405 guest1     15   0  894M  10M  889M S     0.1   0:00 hybrid
```

Resetting System Limits

To regulate these limits on a per-user basis (for applications that do not rely on `limit.h`), the `limits.conf` file can be modified. System limits that can be modified include maximum file size, maximum number of open files, maximum stack size, and so on. You can view this file is, as follows:

```
[user@machine user]# cat /etc/security/limits.conf
# /etc/security/limits.conf
#
#Each line describes a limit for a user in the form:
#
#            #
#Where:
# can be:
#        - an user name
#        - a group name, with @group syntax
#        - the wildcard *, for default entry
#
# can have the two values:
#        - "soft" for enforcing the soft limits
#        - "hard" for enforcing hard limits
#
# can be one of the following:
#        - core - limits the core file size (KB)
```

```
#        - data - max data size (KB)
#        - fsize - maximum filesize (KB)
#        - memlock - max locked-in-memory address space (KB)
#        - nofile - max number of open files
#        - rss - max resident set size (KB)
#        - stack - max stack size (KB)
#        - cpu - max CPU time (MIN)
#        - nproc - max number of processes
#        - as - address space limit
#        - maxlogins - max number of logins for this user
#        - priority - the priority to run user process with
#        - locks - max number of file locks the user can hold
#
#                #

#*              soft    core            0
#*              hard    rss             10000
#@student       hard    nproc           20
#@faculty       soft    nproc           20
#@faculty       hard    nproc           50
#ftp            hard    nproc           0
#@student       -       maxlogins       4

# End of file
```

For instructions on how to change these limits, see "Resetting the File Limit Resource Default" on page 103.

Resetting the File Limit Resource Default

Several large user applications use the value set in the `limit.h` file as a hard limit on file descriptors and that value is noted at compile time. Therefore, some applications may need to be recompiled in order to take advantage of the SGI Altix system hardware.

To regulate these limits on a per-user basis (for applications that do not rely on `limit.h`), the `limits.conf` file can be modified. This allows the administrator to set the allowed number of open files per user and per group. This also requires a one-line change to the `/etc/pam.d/login` file.

Follow this procedure to execute these changes:

1. Add the following line to `/etc/pam.d/login`:

   ```
   session  required  /lib/security/pam_limits.so
   ```

2. Add the following line to `/etc/security/limits.conf`, where *username* is the user's login and *limit* is the new value for the file limit resource:

   ```
   [username]  hard  nofile  [limit]
   ```

The following command shows the new limit:

```
ulimit -H -n
```

Because of the large number of file descriptors that that some applications require, such as MPI jobs, you might need to increase the system-wide limit on the number of open files on your Altix system. The default value for the file limit resource is 1024. The default 1024 file descriptors allows for approximately 199 MPI processes per host.You can increase the file descriptor value to 8196 to allow for more than 512 MPI processes per host by adding adding the following lines to the `/etc/security/limits.conf` file:

```
*     soft    nofile      8196
*     hard    nofile      8196
```

For more information on setting system limits, see the Chapter 5, "Kernel Tunable Parameters on SGI ProPack Servers" in the *Linux Configuration and Operations Guide*.

Resetting the Default Stack Size

Some applications will not run well on an Altix system with a small stack size. To set a higher stack limit, follow the instructions in "Resetting the File Limit Resource Default" on page 103 and add the following lines to the `/etc/security/limits.conf` file:

```
* soft stack 300000
* hard stack unlimited
```

This sets a soft stack size limit of 300000 KB and an unlimited hard stack size for all users (and all processes).

Another method that does not require root privilege relies on the fact that many MPI implementation use `ssh`, `rsh`, or some sort of login shell to start the MPI rank processes. If you merely need to bump up the soft limit, you can modify your shell's

startup script. For example, if your login shell is `bash` then add something like the following to your `.bashrc` file:

```
% ulimit -s 300000
```

Note that SGI MPT MPI allows you to set your stack size limit larger with the `ulimit` or `limit` shell command before launching an MPI program with `mpirun`(1) or `mpiexec_mpt`(1). MPT will propagate the stack limit setting to all MPI processes in the job.

For more information on defaul settings, also see "Resetting the File Limit Resource Default" on page 103.

Resetting Virtual Memory Size

The virtual memory parameter `vmemoryuse` determines the amount of virtual memory available to your application. If you are running with `csh`, use `csh` commands, such as, the following:

```
limit
limit vmemoryuse 7128960
limit vmemoryuse unlimited
```

The following MPI program fails with a memory-mapping error because of a virtual memory parameter `vmemoryuse` value set too low:

```
% limit vmemoryuse 7128960

% mpirun -v -np 4 ./program
MPI: libxmpi.so 'SGI MPI 4.9 MPT 1.14  07/18/06 08:43:15'
MPI: libmpi.so  'SGI MPI 4.9 MPT 1.14  07/18/06 08:41:05'
MPI: MPI_MSGS_MAX = 524288
MPI: MPI_BUFS_PER_PROC= 32
mmap failed (memmap_base) for 504972 pages (8273461248

bytes) Killed n
```

The program now succeeds when virtual memory is unlimited:

```
%  limit vmemoryuse unlimited


% mpirun -v -np 4 ./program
```

```
MPI: libxmpi.so 'SGI MPI 4.9 MPT 1.14  07/18/06 08:43:15'
MPI: libmpi.so  'SGI MPI 4.9 MPT 1.14  07/18/06 08:41:05'
MPI: MPI_MSGS_MAX = 524288
MPI: MPI_BUFS_PER_PROC= 32

HELLO WORLD from Processor 0

HELLO WORLD from Processor 2

HELLO WORLD from Processor 1

HELLO WORLD from Processor 3
```

If you are running with `bash`, use `bash` commands, such as, the following:

```
ulimit -a
ulimit -v 7128960
ulimit -v unlimited
```

Linux Shared Memory Accounting

The Linux operating system does not calculate memory utilization in a manner that is useful for certain applications in situations where regions are shared among multiple processes. This can lead to over-reporting of memory and to processes being killed by schedulers erroneously detecting memory quota violation.

The `get_weighted_memory_size` function weighs shared memory regions by the number of processes using the regions. Thus, if 100 processes are each sharing a total of 10GB of memory, the weighted memory calculation shows 100MB of memory shared per process, rather than 10GB for each process.

Because this function applies mostly to applications with large shared-memory requirements, it is located in the SGI NUMA tools package and made available in the `libmemacct` library available from a new package called `memacct`. The library function makes a call to the `numatools` kernel module, which returns the weighted sum back to the library, and then returns back to the application.

The usage statement for the `memacct` call is, as follows:

```
cc ... -lmemacct
 #include <sys/types.h>
 extern int get_weighted_memory_size(pid_t pid);
```

The syntax of the memacct call is, as follows:

```
int *get_weighted_memory_size(pid_t pid);
```

Returns the weighted memory (RSS) size for a pid, in bytes. This weights the size of shared regions by the number of processes accessing it. Return -1 when an error occurs and set errno, as follows:

ESRCH	Process pid was not found.
ENOSYS	The function is not implemented. Check if numatools kernel package is up-to-date.

Normally, the following errors should not occur:

ENOENT	Can not open /proc/numatools device file.
EPERM	No read permission on /proc/numatools device file.
ENOTTY	Inappropriate ioctl operation on /proc/numatools device file.
EFAULT	Invalid arguments. The ioctl() operation performed by the function failed with invalid arguments.

For more information, see the memacct(3) man page.

Index

A

Altix architecture overview
  Altix 3000 series systems, 2
  Altix 4000 series systems, 3
Amdahl's law, 78
  execution time given n and p, 82
  parallel fraction p, 80
  parallel fraction p given speedup(n), 81
  speedup(n) given p, 80
  superlinear speedup, 80
analysis
  system configuration, 13
application tuning process, 13
automatic parallelization
  limitations, 75

C

cache bank conflicts, 69
cache performance, 68
cacheline traffic and CPU utilization, 33
cluster environment, 1
commands
  dlook, 56
  dplace, 48
  topology, 28
common compiler options, 7
compiler command line, 7
compiler libaries
  C/C++, 10
  dynamic libraries, 9
  message passing, 10
  overview, 9
compiler libraries
  static libraries, 9
compiler options
  tracing and porting, 64
compiler options for tuning, 67
compiling environment, 7
  compiler overview, 7
  debugger overview, 11
  libraries, 9
  modules, 8
CPU utilization, 33
CPU-bound processes, 14
csrep command, 19

D

data decomposition, 71
data dependency, 75
data parallelism, 71
data placement tools, 43
  cpusets, 44
  dplace, 44
  overview, 43
  taskset, 44
debugger overview, 11
debuggers, 22
  gdb, 11
  idb, 11
  TotalView, 11
denormalized arithmetic, 8
determining parallel code amount, 72
determining tuning needs
  tools used, 66
dlook command, 56
dplace command, 48
dumppm, 19

E

Electric Fence debugger, 64
Environment variables, 77
explicit data decomposition, 71

F

False sharing, 76
file limit resources
  resetting, 103
Flexible File I/O (FFIO), 90
  environment variables to set, 86
  operation, 85
  overview, 85
  simple examples, 87
floating-point programs, 82
Floating-Point Software Assist, 82
FPSWA
  See "Floating-Point Software Assist", 82
functional parallelism, 71

G

gdb tool, 22
GNU debugger, 22
gtopology command, 29
GuideView tool, 20

H

histx, 16
histx data collection, 16
histx filters, 19
hwinfo command, 28

I

I/O-bound processes, 14
idb tool, 23
implicit data decomposition, 71
iostat command, 39
iprep command, 19

L

latency, 1
limits
  system, 102
linkstat command, 34
lipfpm command, 17

M

memory management, 5, 70
memory page, 5
memory strides, 69
memory-bound processes, 14
Message Passing Toolkit
  for parallelization, 73
  using profile.pl, 16
modules, 8
  command examples, 8
MPP definition, 1

N

NUMA Tools
  command
    dlook, 55
    dplace, 48
  installing, 62

O

OpenMP, 74
  environment variables, 77
  Guide OpenMP Compiler, 22

P

parallel execution
  Amdahl's law, 78
  parallel fraction p, 80
parallel speedup, 79
parallelization
  automatic, 75
  using MPI, 73
  using OpenMP, 74
performance
  Assure Thread Analyzer, 22
  Guide OpenMP Compiler, 22
  GuideView, 20
  VTune, 20
performance analysis, 13
Performance Co-Pilot monitoring tools, 32
  linkstat, 34
  Other Performance Co-Pilot monitoring tools, 34
  pmshub, 33
  shubstats, 34
performance gains
  types of, 13
performance problems
  sources, 14
pfmon tool, 15
pmshub command, 33
process placement, 95
  MPI and OpenMP, 99
  set-up, 95
  using OpenMP, 98
  using pthreads, 96
profile.pl script, 15
profiling
  pfmon, 15
  profile.pl, 15
ps command, 37

R

resetting default system stack size, 104
resetting file limit resources, 103
resetting system limit resources, 102
resident set size, 5

S

samppm command, 19
sar command, 39
scalable computing, 1
SHMEM, 10
shortening execution time, 78
shubstats command, 34
SMP definition, 1
stack size
  resetting, 104
superlinear speedup, 80
swap space, 5
system
  overview, 1
system configuration, 13
system limit resources
  resetting, 102
system limits
  address space limit, 103
  core file siz, 103
  CPU time, 103
  data size, 103
  file locks, 103
  file size, 103
  locked-in-memory address space, 103
  number of logins, 103
  number of open files, 103
  number of processes, 103

priority of user process, 103
resetting, 102
resident set size, 103
stack size, 103
system monitoring tools, 27
command
hwinfo, 28
topology, 28
system usage commands, 36
iostat, 39
ps, 37
sar, 39
top, 38
uptime, 36
vmstat, 38
w, 37

T

taskset command, 45
tools
Assure Thread Analyzer, 22
Guide OpenMP Compiler, 22
GuideView, 20
pfmon, 15
profile.pl, 15, 16
VTune, 20
top command, 38
topology command, 28
tuning
cache performance, 68
debugging tools
Electric Fence, 64
idb, 23
dplace, 77
Electric Fence, 64
environment variables, 77
false sharing, 76
heap corruption, 64
managing memory, 70
multiprocessor code, 70
parallelization, 72
profiling
GuideView, 20
histx command, 16
mpirun command, 16
pfmon, 15
profile.pl script, 15
VTune analyzer, 20
single processor code, 63
using compiler options, 67
using dplace, 77
using math functions, 66
using taskset, 77
verifying correct results, 64

U

uname command, 14
unflow arithmetic
effects of, 8
uptime command, 36

V

virtual addressing, 5
virtual memory, 5
vmstat command, 38
VTune performance analyzer, 20

W

w command, 37