sgi®

# Linux® Application Tuning Guide

007–4639–001

CONTRIBUTORS

Written by Jean Wilson and Terry Schultz

Illustrated by Chrystie Danzer

Edited by Susan Wilkening

Production by Karen Jacobson

Engineering and other contributions by Rich Altmaier, John Baron, Beverly Bernard, Ray Bryant, Gerardo Cisneros, Tom Elken, Martyn Foster, Ayad Jassim, Kevin McMahon, Steve Neuner, Jean-Pierre Panziera, Arthur Raefsky

# Record of Revision

| Version | Description |
|---|---|
| 001 | October 2003<br>Original publication. |

# Contents

# About This Document

This publication provides information about tuning application programs on the SGI Altix 3000 family 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.

## 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 IRIX and Linux implementations on SGI systems:

- *Linux Installation and Getting Started*
- *Linux Resource Administration Guide*
- *IRIX Admin: Resource Administration*
- *SGI ProPack for Linux Start Here*
- *Message Passing Toolkit: MPI Programmer's Manual*

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.`

### 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://intel.com/software/perflib; documentation for Intel compiler products can be downloaded from this website.
- http://developer.intel.com/software/products/vtune/vtune61/index.htm/
- Information about the OpenMP Standard can be found at http://www.openmp.org/specs.

## 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

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.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:

  Technical Publications
  SGI
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

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 range of performance, while retaining compatibility with other systems. 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 ($S^2$MPs), 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

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 64 processors and 512 gigabytes of memory. It uses Intel's Itanium 2 processors and uses nonuniform memory access (NUMA) in SGI's NUMAflex global shared-memory architecture.

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.

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.

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 5, for details about the compilers and libraries included with the distribution.

## 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 a malloc 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 in a filestyem. 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.

*Chapter 2*

# 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 5
- "Modules" on page 6
- "Library Overview" on page 7
- "Other Compiling Environment Features" on page 8

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 can be migrated from 32—bit platforms to Altix. These compilers are included in the standard Red Hat Linux distribution.

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

```
% efc [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`.
- `-I`*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.

## 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 the correct compilers, loader, libraries, and utilities.

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.

## 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 use the static version of standard libraries, use the full pathname of the library or use the `-static` option on the compiler command line.

### Dynamic Libraries

Dynamic libraries are linked into the program 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

### Shared Memory Libraries

The Shared Memory Access Library (`libsma`) is part of the Message Passing Toolkit (MPT) product on SGI Altix systems. The library routines operate on remote and local memory. Unlike message passing, shared memory (`shmem`) routines do not require the sender-receiver pair and have minimal overhead and latency, while providing maximum data bandwidth.

The `shmem` routines have the following supported operations:

- Remote data transfer
- Atomic swap
- Atomic increment
- Work-shared broadcast and reduction
- Synchronization

For details about using the `shmem` routines, see the `intro_shmem` man page or the *Message Passing Toolkit: MPI Programmer's Manual*.

## 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.

These and other performance analysis tools are discussed in Chapter 3, "Performance Analysis and Debugging" on page 11.

*Chapter 3*

# 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 two 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.

Any application tuning process involves:

1. Analyzing and identifying a problem
2. Evaluating the problem
3. Applying an optimization technique

This chapter describes the process of analyzing your code to determine performance bottlenecks. See Chapter 5, "Performance Tuning" on page 37, 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 is a Bourne shell script that uses information in `/dev/hw` to display system configuration information. See Chapter 4, "Monitoring Tools" on page 21 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 performing 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 on unmodified binaries. In addition, it can be used for the following tasks:

- To monitor unmodified binaries in its per-process mode.
- To run system-wide monitoring sessions. Such session 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 event counts.
- 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.

## 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.

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

## 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.

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 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.

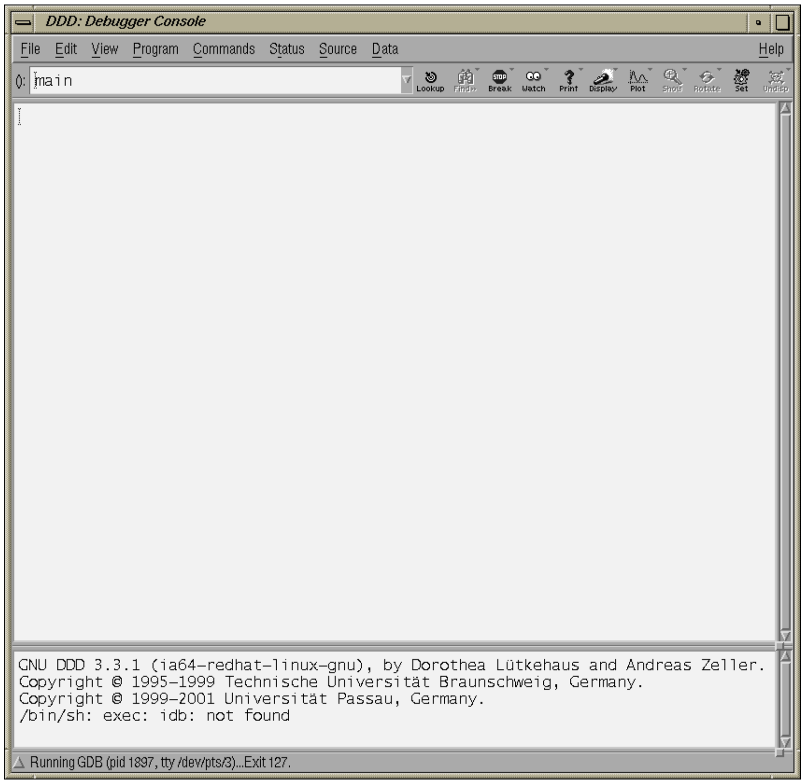
- `idb`: the Intel debugger. This is a fully symbolic debugger for the Linux platform. The debugger provides extensive support for debugging programs written in C, C++, FORTRAN 77, and Fortran 90. At this time, `idb` cannot be used to debug multithreaded or multiprocessor programs on Itanium systems.

  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.

## Using ddd

The DataDisplayDebugger `ddd`(1) tool is a GUI to an arbitrary command line debugger as shown in Figure 3-1 on page 18. To instantiate `ddd`, use the `--debugger` option to specify the debugger used (for example, `--debugger "idb"`). The default debugger used is `gdb`.



```
GNU DDD 3.3.1 (ia64-redhat-linux-gnu), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
/bin/sh: exec: idb: not found
```

Running GDB (pid 1897, tty /dev/pts/3)...Exit 127.

**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 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 `hinv`(1) command, `topology`(1) command, and other operating system commands that can help you determine where system resources are being spent.

NUMA tools include the `dlook`(1) and `dplace`(1) commands that you can use to improve the performance of processes running on your SGI Altix NUMA machine.

## 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.

### Hardware Inventory and Usage Commands

The `hinv`(1) command displays the contents of the system's hardware inventory. The information displayed includes brick configuration, processor type, main memory size, and disk drive information, as follows:

```
[user1@profit user1]# hinv
1 Ix-Brick
4 R-Brick
8 C-Brick
32  1500 MHz Itanium 2 Rev. 5 Processor
Main memory size: 121.75 Gb
Broadcom Corporation NetXtreme BCM5701 Gigabit Ethernet (rev 21). on pci01.04.0
Integral SCSI controller pci01.03.0: QLogic 12160 Dual Channel Ultra3 SCSI (Rev 6) pci01.03.0
  Disk Drive: unit   1 lun  0 on SCSI controller pci01.03.0  0
  Disk Drive: unit   2 lun  0 on SCSI controller pci01.03.0  0
  Disk Drive: unit   1 lun  0 on SCSI controller pci01.03.0  0
  Disk Drive: unit   2 lun  0 on SCSI controller pci01.03.0  0
SCSI storage controller: QLogic Corp. QLA2200 (rev 5). pci03.01.0
```

```
  Disk Drive: unit  10 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  11 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  12 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  13 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  14 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  15 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  16 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  17 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  18 lun  0 on SCSI controller pci03.01.0  0
  Disk Drive: unit  19 lun  0 on SCSI controller pci03.01.0  0
Co-processor: Silicon Graphics, Inc. IOC4 I/O controller (rev 79). on pci01.01.0
CD-ROM MATSHITADVD-ROM SR-8588 7Z20   on pci01.01.0 target0/lun0
```

The `topology`(1) command provides topology information about your system. Topology information is extracted from information in the`/dev/hw` directory. Unlike the IRIX operating system, in Linux the hardware topology information is implemented on a `devfs` filesystem rather than on a `hwgraph` filesystem. The `devfs` filesystem represents the collection of all significant hardware connected to a system, such as CPUs, memory nodes, routers, repeater routers, disk drives, disk partitions, serial ports, Ethernet ports, and so on. The `devfs` filesystem is maintained by system software and is mounted at `/hw` by the Linux kernel at system boot.

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

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
```

## 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
```

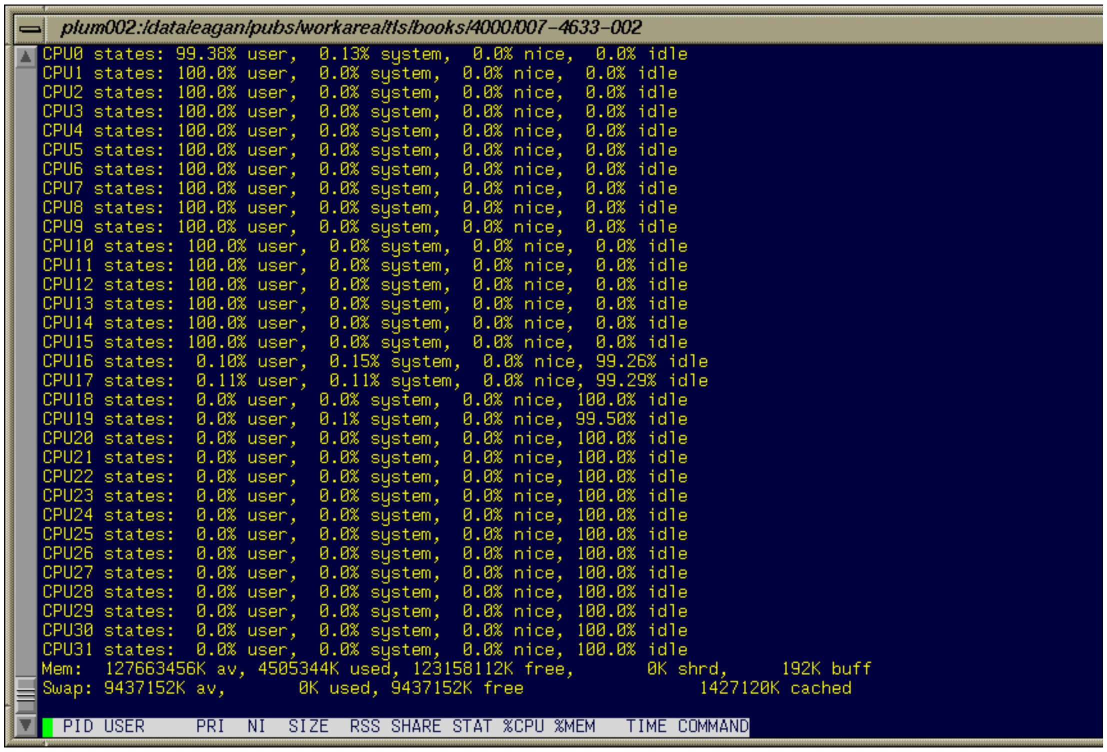
The output from this command shows who is on the system, the duration of user sessions, processor usage by user, and currently executing user commands.

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 4-1 on page 25.



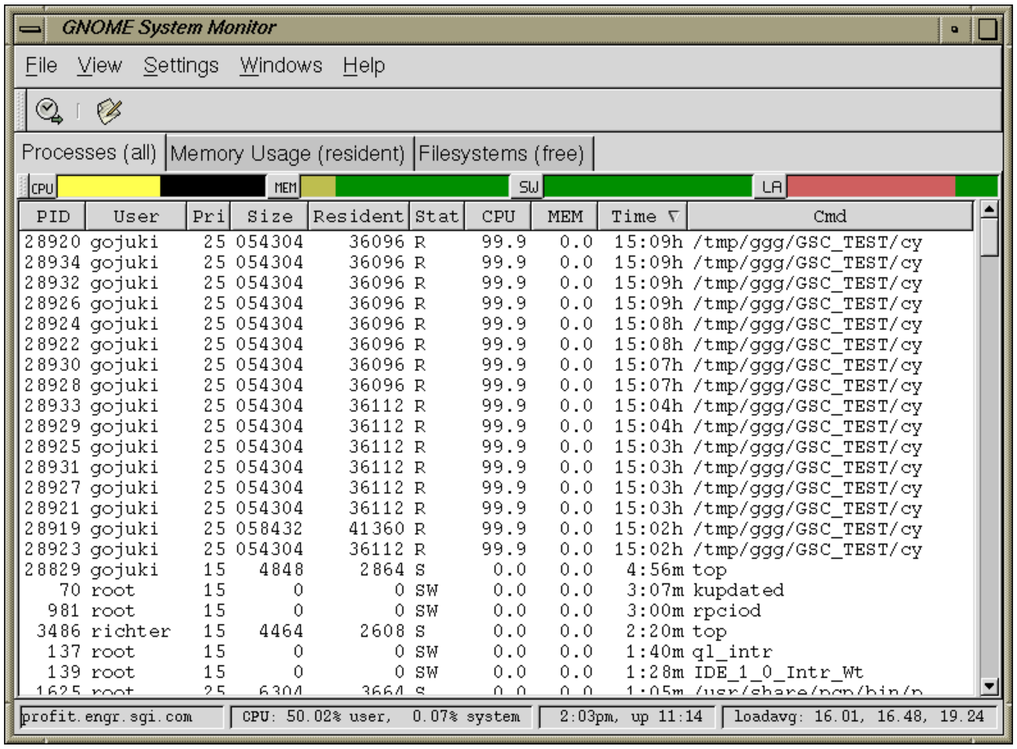
```
plum002:/data/eagan/pubs/workarea/tls/books/4000/007-4633-002
CPU0 states: 99.38% user,  0.13% system,  0.0% nice,  0.0% idle
CPU1 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU2 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU3 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU4 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU5 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU6 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU7 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU8 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU9 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU10 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU11 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU12 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU13 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU14 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU15 states: 100.0% user,  0.0% system,  0.0% nice,  0.0% idle
CPU16 states:  0.10% user,  0.15% system,  0.0% nice, 99.26% idle
CPU17 states:  0.11% user,  0.11% system,  0.0% nice, 99.29% idle
CPU18 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU19 states:  0.0% user,  0.1% system,  0.0% nice, 99.50% idle
CPU20 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU21 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU22 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU23 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU24 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU25 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU26 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU27 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU28 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU29 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU30 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
CPU31 states:  0.0% user,  0.0% system,  0.0% nice, 100.0% idle
Mem:  127663456K av, 4505344K used, 123158112K free,       0K shrd,     192K buff
Swap: 9437152K av,       0K used, 9437152K free                 1427120K cached

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

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

.

To monitor memory use, use the `gtop`(1) command. This produces a memory chart that shows how memory is used by the various procceses that are present as shown in


GNOME System Monitor
File View Settings Windows Help
Processes (all) Memory Usage (resident) Filesystems (free)
CPU
MEM
SW
LA

```
  PID User     Pri  Size  Resident Stat  CPU   MEM   Time     Cmd
28920 gojuki    25 054304    36096 R     99.9  0.0  15:09h /tmp/ggg/GSC_TEST/cy
28934 gojuki    25 054304    36096 R     99.9  0.0  15:09h /tmp/ggg/GSC_TEST/cy
28932 gojuki    25 054304    36096 R     99.9  0.0  15:09h /tmp/ggg/GSC_TEST/cy
28926 gojuki    25 054304    36096 R     99.9  0.0  15:09h /tmp/ggg/GSC_TEST/cy
28924 gojuki    25 054304    36096 R     99.9  0.0  15:08h /tmp/ggg/GSC_TEST/cy
28922 gojuki    25 054304    36096 R     99.9  0.0  15:08h /tmp/ggg/GSC_TEST/cy
28930 gojuki    25 054304    36096 R     99.9  0.0  15:07h /tmp/ggg/GSC_TEST/cy
28928 gojuki    25 054304    36096 R     99.9  0.0  15:07h /tmp/ggg/GSC_TEST/cy
28933 gojuki    25 054304    36112 R     99.9  0.0  15:04h /tmp/ggg/GSC_TEST/cy
28929 gojuki    25 054304    36112 R     99.9  0.0  15:04h /tmp/ggg/GSC_TEST/cy
28925 gojuki    25 054304    36112 R     99.9  0.0  15:03h /tmp/ggg/GSC_TEST/cy
28931 gojuki    25 054304    36112 R     99.9  0.0  15:03h /tmp/ggg/GSC_TEST/cy
28927 gojuki    25 054304    36112 R     99.9  0.0  15:03h /tmp/ggg/GSC_TEST/cy
28921 gojuki    25 054304    36112 R     99.9  0.0  15:03h /tmp/ggg/GSC_TEST/cy
28919 gojuki    25 058432    41360 R     99.9  0.0  15:02h /tmp/ggg/GSC_TEST/cy
28923 gojuki    25 054304    36112 R     99.9  0.0  15:02h /tmp/ggg/GSC_TEST/cy
28829 gojuki    15   4848     2864 S      0.0  0.0   4:56m top
   70 root      15      0        0 SW     0.0  0.0   3:07m kupdated
  981 root      15      0        0 SW     0.0  0.0   3:00m rpciod
 3486 richter   15   4464     2608 S      0.0  0.0   2:20m top
  137 root      15      0        0 SW     0.0  0.0   1:40m ql_intr
  139 root      15      0        0 SW     0.0  0.0   1:28m IDE_1_0_Intr_Wt
 1625 root      25   6304     3664 S      0.0  0.0   1:05m /usr/share/pcp/bin/p

profit.engr.sgi.com | CPU: 50.02% user,  0.07% system | 2:03pm, up 11:14 | loadavg: 16.01, 16.48, 19.24
```

**Figure 4-2** Using `gtop`(1) to Show Memory Usage of Processes

.

## NUMA Tools

This sectoin describes the NUMA tools `dlook`(1) and `dplace`(1).

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.

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.

## `dlook`

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 elapsed 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 elapsed 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 the file name. 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
        [2000000000000000-200000000002c000]        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. *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

  60000ffffffff4000-60000fffffffc000 rwxp ffffffffffffc000 00:00 0
        [60000ffffffff4000-60000fffffffc000]         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 iInterface (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.

## dplace

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] com
```

```
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.

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 a kernel hook (that is, a process aggregate or PAGG) to create a placement container consisting of all the CPUs (or a or a subset of CPUs) of the CpuMemSet. The dplace process is placed in this container and by default is bound to the first CPU of the CpuMemSet 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 CpuMemSet 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 CpuMemSet, binding starts over with the first CPU in the CpuMemSet.

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 CpuMemSet or `runon`(1) command. CPU numbers start at 0. If this option is not specified, all CPUs of the current CpuMemSet are available. Note that a previous `runon` command may be used to restrict the available CPUs.
- `-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 currently should be placed using the `-x` option with a `skip_mask` of `6` (`-x6`). This could change in future versions of OpenMP.

- `-p` *placement_file*: Specifies a placement file that contains additional directives that are used to control process placement. (Not yet implemented).
- `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 4-1** 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`" on page 27 and the `dlook`(1) man page.

**Example 4-2** Using `dplace` command with OpenMP Programs

To run an OpenMP program on logical CPUs 4 through 7 within the current CpuMemSet, 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 4-3** Using the `dplace` command with Linux commands

The following examples assume that the command is executed from a shell running in a CpuMemSet 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. |
| `runon 4-7 dplace app` | The `runon` command restricts execution to physical CPUs 12 through 15. The `dplace` command sequentially binds processes to CPUs 12 through 15. |

## 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
   ```

*Chapter 5*

# 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 45

## 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 38.
- Use existing tuned code, such as that found in math libraries and scientific library packages. For details, see "Using Tuned Code" on page 40.
- Determine what needs tuning. For details, see "Determining Tuning Needs" on page 40.
- Use the compiler to do the work. For details, see "Using Compiler Options Where Possible" on page 41.
- Consider tuning cache performance. For details, see "Tuning the Cache Performance" on page 42.
- Set environment variables to enable higher-performance memory management mode. For details, see "Managing Memory" on page 44.
- Change stack and data segments. For details, see "Using `chatr` to Change Stack and Data Segments" on page 44.

## 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 17 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 shipped with Red Hat Linux. 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
%
```

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 11.

## Using Compiler Options Where Possible

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

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.
- `-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 locatio, a pattern of deletion and 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 can 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. A detailed discussion of this issue can be found at http://www.linuxshowcase.org/ezolt.html .

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 `efc` 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.

## Using `chatr` to Change Stack and Data Segments

The `chatr` command changes the ELF (Executable and Linking Format) header of a program so that when the program is run, instructions in the stack and data segments are either executable or are not executable.

The default is to allow the stack and data segments to be executable. For certain workloads, a performance improvement can be obtained by turning off the default behavior and making the stack and code segments unexecutable. This typically benefits workloads that do a large number of `fork()` calls. Workloads without a large number of `fork()` calls will probably not see a performance improvement.

You can change this behavior system-wide by using the `sysctl` command. The following command disallows executable stacks and data segments:

```
% sysctl vm.executable_stacks=0
```

The following command allows executable stacks and data segments:

```
% sysctl vm.executable_stacks=1
```

Note that some programs may now fail with a `SEGV` error even though they worked correctly before. Usually, these are programs that store instructions in the stack or data segment and then branch to those instructions (for example, older versions of the X server that load graphics drivers into the data segment, or Java JIT compilers). You can use the `chatr` command to make these programs work correctly, regardless of the value of `vm.executable_stacks`.

For details about usage, see the `chatr`(1) man page.

## 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 37.

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 45.
- Choose the parallelization methodology for your code. For details, see "Parallelizing Your Code" on page 47.
- Analyze your code to make sure it is parallelizing properly. For details, see Chapter 3, "Performance Analysis and Debugging" on page 11.
- Check to determine if false sharing exists. For details, see "Fixing False Sharing" on page 49.
- Tune for data placement. For details, see "Using `dplace` and `runon`" on page 50.
- Use environment variables to assist with tuning. For details, see "Environment Variables for Performance Tuning" on page 50.

### 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 on a remote IRIX workstation. Hardware configuration information can be displayed using the `/proc/cpuinfo` or `/proc/meminfo` script. Detailed cache (and other) information can be displayed using the `/proc/pal/cpu`*X* script, where *X* is a CPU number.

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 3000 series systems than generic MPI libraries. It takes advantage of the SGI Altix 3000 architecture and SGI Linux NUMA features.

Use the `-lmpi` compiler option to use MPI. For a list of environment variables that are supported, see the `mpi` man page. Those variables that are valid for IRIX systems only are so noted on the 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.

For details about the Message Passing Toolkit, see the *Message Passing Toolkit: MPI Programmer's Manual*.

### 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:

- `efc -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 `ecc`), `guideefc` (in place of `efc`), or `guidec++` to translate code with OpenMP directives into code with calls to `libguide`. See "Other Performance Tools" on page 16 for details.

  The `-openmp` option to `efc` 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.

### 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.

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.

**Note:** The `pmshub` utility may not be available for this release. Consult your release notes for information about its availability.

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.
- 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 `runon`

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 21 for details about `dplace`.

The `runon` 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

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 5-1.

**Example 5-1** 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 5-2.

**Example 5-2** Amdahl's law: *Speedup*(*n*) Given *p*

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

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 5-3.

**Example 5-3** Amdahl's law: p Given *Speedup*(2)

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

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 53). When this occurs, the formula in Example 5-3 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 5-4, where *n* and *m* are the two CPU counts whose speedups are known, *n*>*m*.

**Example 5-4** Amdahl's Law: *p* Given *Speedup*(*n*) and *Speedup*(*m*)

```
                Speedup(n) - Speedup(m)
p  =  -------------------------------------------
      (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.

*Chapter 6*

# Suggested Shortcuts and Workarounds

This chapter contains suggested workarounds and shortcuts that you can use on your SGI Altix system.

## 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_OPENMP 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 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
```

# Index