# Linux® Application Tuning Guide for SGI® X86-64 Based Systems

007–5646–004

## COPYRIGHT

## LIMITED RIGHTS LEGEND

## TRADEMARKS AND ATTRIBUTIONS

# New Features in This Manual

This rewrite of the *Linux Application Tuning Guide for SGI X86-64 Based Systems* supports the SGI Performance Suite 1.4 release.

## Major Documentation Changes

Performed the following:

- Updated information about using the `cpumap` command in "Determining System Configuration" on page 11.
- Updated "`dlook` Command" on page 55.
- Updated "Performance Tuning Running MPI on SGI UV Systems" on page 97.
- Added "OFED Tuning Requirements for UPC and SHMEM" on page 125.
- Added "Setting Java Enviroment Variables" on page 126.

# Record of Revision

| Version | Description |
|---|---|
| 001 | November 2010<br>Original publication. |
| 002 | February 2011<br>Supports the SGI Performance Suite 1.1 release. |
| 003 | November 2011<br>Supports the SGI Performance Suite 1.3 release. |
| 004 | May 2012<br>Supports the SGI Performance Suite 1.4 release. |

# Contents

# About This Document

This publication mainly provides information about tuning application programs on the SGI® UV™ series systems running the Linux operating system. Application programs includes Fortran and C programs written with the Intel-provided compilers on SGI Linux systems. Some parts of this manual are also applicable to other SGI X86-64 based systems, such as, the SGI® ICE™ and SGI® Rackable™ systems.

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 system. The material assumes that you know the basics of software engineering and that you are familiar with standard methods and data structures. If you are new to programming or software design, this guide will **not** be of use to you.

## Related Publications

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

### Related Operating System Documentation

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

- *SGI UV Systems Linux Configuration and Operations Guide*

  Provides information on how to perform system configuration and operations for SGI UV series systems.

- *Linux Resource Administration Guide*

  Provides a reference for people who manage the operation of SGI servers and contains information needed in the administration of various system resource management features such as Array Services, Cpusets on Linux, NUMA tools, and the Cpuset library functions.

- *SGI Performance Suite 1.4 Start Here*

  Provides information about the SGI Performance Suite 1.4 release.

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

  Describes industry-standard message passing protocol optimized for SGI computers. Chapter 7, "Run-time Tuning" discusses ways in which the you can tune the run-time environment to improve the performance of an MPI message passing application on SGI computers. None of these ways involve application code changes.

- *MPInside Reference Guide*

  Documents the SGI MPInside MPI profiling tool.

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 Technical Publications Library at http://docs.sgi.com.

For the latest information about software and documentation in this release, see the release notes that are in a file named `README.TXT` that is available in `/docs` directory on the CD.

### Hardware Reference Manuals

The following documents provide information about Altix system hardware.

- *SGI Altix UV 2000 System User Guide*

  Provides an overview of the architecture and describes the major components of the SGI Altix UV 2000 series systems. It also describes the standard procedures for powering up and powering down the system, provides basic troubleshooting information, and includes important safety and regulatory specifications.

- *SGI Altix UV 1000 System User's Guide*

  Provides an overview of the architecture and describes the major components of the SGI UV 1000 series systems. It also describes the standard procedures for powering up and powering down the system, provides basic troubleshooting information, and includes important safety and regulatory specifications.

- *SGI Altix UV 100 System User's Guide*

  Provides an overview of the architecture and describes the major components of the SGI UV 100 series systems. It also describes the standard procedures for powering up and powering down the system, provides basic troubleshooting information, and includes important safety and regulatory specifications.

- *SGI Altix ICE 8200 Series System Hardware User's Guide*

  Provides an overview of the architecture and describes the major components of the SGI ICE 8200 series systems. It also describes the standard procedures for powering up and powering down the system, provides basic troubleshooting information, and includes important safety and regulatory specifications.

- *SGI Altix ICE 8400 Series System Hardware User's Guide*

  Provides an overview of the architecture and describes the major components of the SGI ICE 8400 series systems. It also describes the standard procedures for powering up and powering down the system, provides basic troubleshooting information, and includes important safety and regulatory specifications.

- *SGI ICE X System Hardware User Guide*

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

**Note:** SGI has a large number of compute platforms. For a complete list of current SGI software and hardware manuals, see the *SGI Performance Suite 1.4 Start Here* available at http://docs.sgi.com.

## Application Guides

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

- http://sourceware.org/gdb/documentation/

  *GDB: The GNU Project Debugger* website with documentation, such as, *Debugging with GDB*, *GDB User Manual*, and so on.

- http://www.intel.com/cd/software/products/asmo-na/eng/perflib/219780.htm; documentation for Intel compiler products can be downloaded from this website.

  Intel Software Network page with links to Intel documentation, such as, *Intel Professional Edition Compilers*, *Intel Thread Checker*, *Intel VTune Performance Analyzer*, and various Intel cluster software solutions.

- Intel provides detailed application tuning information including the Intel Xeon processor 5500 at http://www.intel.com/Assets/en_US/PDF/manual/248966.pdf?wapkw= Intel Xeon processor 5500 Series tuning manual
- Intel provides specific tuning information tutorial for Nehalem (Intel Xeon 5500) at http://software.intel.com/sites/webinar/tuning-your-application-for-nehalem/.
- Intel provides information for Westmere (Intel Xeon 5600) at http://www.intel.com/itcenter/products/xeon/5600/index.htm
- http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-for-linux-documentation/

  Intel Software Network page with information specific to *Intel VTune Performance Analyzer* including links to documentation.
- Intel provides information about the Intel Performance Tuning Utility (PTU) at http://software.intel.com/en-us/articles/intel-performance-tuning-utility/.
- Information about the OpenMP Standard can be found at http://openmp.org/wp/.

  The OpenMP API specification for parallel programming website is found here.

## 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.
- You can view release notes on your system by accessing the `README.txt` file for the product. This is usually located in the `/usr/share/doc/`*productname* directory, although file locations may vary.
- You can view man pages by typing `man` *title* at a command line.

## Reader Comments

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

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

  SGI  
  Technical Publications  
  46600 Landing Parkway  
  Fremont, CA 94538

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. See the *SGI UV Systems Linux Configuration and Operations Guide* for SGI UV series systems.

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

## Scalable Computing

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

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

On an symmetric multiprocessor (SMP) machine, all data is visible from all processors. NonUniform Memory Access (NUMA) machines also have a shared address space. In both cases, there is a single shared memory space and a single operating system instance. However, in an SMP machine, each processor is functionally identical and has equal time access to every memory address. In contrast, a NUMA system has a shared address space, but the access time to memory vary over physical address ranges and between processing elements. The Intel® Xeon® 7500 series processor (Nehalem i7 architecture) and the Intel Xeon processor E5 family platform are examples of NUMA architecture. Each processor has its own memory and can address the memory attached to another processor through the Quick Path Interconnet (QPI). For more information, see the system architecture overview in "Data Placement Tools Overview" on page 39.

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 SGI System Architecture

See the appropriate hardware manual for system architecture overviews. The *SGI Performance Suite 1.4 Start Here* lists all the current SGI hardware manuals. The *SGI Management Center for SGI ICE* provides system architecture overviews for the SGI® ICE™ 8200 and SGI ICE 8400 systems. The *SGI Management Center for SGI ICE X* provides a system architecture overview for the SGI® ICE™ X system. All books are available on the Tech Pubs Library at http://docs.sgi.com

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

*Swap space* is used for temporarily saving parts of a program when there is not enough physical memory. The swap space may be on the system drive, on an optional drive, or allocated to a particular file in a filesystem. To avoid swapping, try not to overburden memory. Lack of adequate swap space limits the number and the size of applications that can run simultaneously on the system, and it can limit system performance. Access time to disk is orders of magnitude slower than access to random access memory (RAM). A system that runs out of memory and uses swap to disk while running a program will have its performance seriously affected, as swapping will become a major bottleneck. Be sure your system is configured with enough memory to run your applications.

Linux is a demand paging operating system, using a least-recently-used paging algorithm. Pages are mapped into physical memory when first referenced and pages are brought back into memory if swapped out. In a system that uses demand paging, the operating system copies a disk page into physical memory only if an attempt is made to access it, that is, a page fault occurs. A page fault handler algorithm does the necessary action. For more information, see the `mmap`(2) man page.

*Chapter 2*

# The SGI Compiling Environment

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

- "Compiler Overview" on page 5
- "Environment Modules" on page 6
- "Library Overview" on page 8
- "Other Compiling Environment Features" on page 10

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 available from Intel Corporation or can be ordered from SGI on a reseller basis. The Fortran compiler supports OpenMP 3.0 and the C/C++ compiler is compatible with `gcc` and the C99 standard. Both the C/C++ and Fortran Intel compilers support OpenMP 3.0. It should be noted that Intel has renamed their compilers, as follows:

- Intel® C++ Compilers are now called Intel® C++ Composer XE.
- Intel® Fortran Compilers gcc, g++, and g77 compilers are now called Intel® Visual Fortran Composer XE and Intel® Visual Fortran Composer XE with IMSL.

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

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

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

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

Some common compiler options are:

- `-o` *filename*: renames the output to *filename*.

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

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

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

Some applications can generate an excessive number of kernel KERN_WARN "floating point assist" warning messages. For more information, see the *SGI UV Systems Configuration and Operations Guide*.

## Environment Modules

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

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

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

```
% module avail
```

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

```
% module load intel-compilers-latest mpt/2.04
```

**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.

The module help command provides a list of all arguements accepted, as follows:

```
sys:~> module help

  Modules Release 3.1.6 (Copyright GNU GPL v2 1991):
  Available Commands and Usage:
        + add|load              modulefile [modulefile ...]
        + rm|unload             modulefile [modulefile ...]
        + switch|swap           modulefile1 modulefile2
        + display|show          modulefile [modulefile ...]
        + avail                 [modulefile [modulefile ...]]
        + use [-a|--append]     dir [dir ...]
        + unuse                 dir [dir ...]
        + update
        + purge
        + list
        + clear
        + help                  [modulefile [modulefile ...]]
        + whatis                [modulefile [modulefile ...]]
        + apropos|keyword       string
        + initadd               modulefile [modulefile ...]
        + initprepend           modulefile [modulefile ...]
        + initrm                modulefile [modulefile ...]
        + initswitch            modulefile1 modulefile2
        + initlist
        + initclear

--------
```

For details about using modules, see the `module`(1) 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 build an appplication to have all static versions of standard libraries in the application binary, use the `-static` option on the compiler command line.

### Dynamic Libraries

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

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

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

### C/C++ Libraries

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

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

## SHMEM Message Passing Libraries

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

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

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

SHMEM routines support remote data transfer through put operations, which transfer data to a different PE, get operations, which transfer data from a different PE, and remote pointers, which allow direct references to data objects owned by another PE. Other operations supported are collective broadcast and reduction, barrier synchronization, and atomic memory operations. An atomic memory operation is an atomic read-and-update operation, such as a fetch-and-increment, on a remote or local data object.

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

## Other Compiling Environment Features

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

- `idb`: the Intel debugger (available if your system is licensed for the Intel compilers). This is a fully symbolic debugger and supports Fortran, C, and C++ debugging. The Intel® Debugger for Linux is a fully fledged Eclipse graphical user interface based debug solution. To use the command line debugger (`gdb` like), you need to use the `idbc` command. It works with the following compilers: Intel® C++ Compilers (now called Intel® C++ Composer XE) and Intel® Fortran Compilers gcc, g++, and g77 compilers (now called Intel® Visual Fortran Composer XE and Intel® Visual Fortran Composer XE with IMSL). The Intel Debugger can debug both single and multi-threaded applications, serial and parallel code. For more information, see http://software.intel.com/en-us/articles/idb-linux/.
- `gdb`: the GNU project debugger, which supports C, C++ and Modula-2. It also supports Fortran 95 debugging when the gdbf95 patch is installed.
- `ddd`: a graphical user interface to `gdb` and the other debuggers.
- TotalView: a licensed graphical debugger useful in an MPI environment (see http://www.roguewave.com/products/totalview-family/totalview.aspx)

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 three categories of measured time:

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

Any application tuning process involves the following steps:

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

This chapter describes the process of analyzing your code to determine performance bottlenecks. See Chapter 6, "Performance Tuning" on page 73, 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.

The `topology`(1) command displays general information about SGI systems, with a focus on node information. This includes node counts for blades, node IDs, NASIDs, memory per node, system serial number, partition number, UV Hub versions, CPU to node mappings, and general CPU information. The `topology` command is installed by the `pcp-sgi` RPM package.

Example output is, as follows:

```
uv-sys:~ # topology
Serial number: UV-00000044
Partition number: 0
4 Blades
64 CPUs
125.97 Gb Memory Total

Blade         ID       asic  NASID         Memory
-------------------------------------------------
    0 r001i01b00  UVHub 2.0      0    16757488 kB
    1 r001i01b01  UVHub 2.0      2    16777216 kB
    2 r001i01b02  UVHub 2.0      4    16777216 kB
    3 r001i01b03  UVHub 2.0      6    16760832 kB

CPU      Blade PhysID CoreID APIC-ID Family Model Speed L1(KiB) L2(KiB) L3(KiB)
-------------------------------------------------------------------------------
  0 r001i01b00     00     00       0      6    46  1866 32d/32i     256   18432
  1 r001i01b00     00     03       6      6    46  1866 32d/32i     256   18432
  2 r001i01b00     00     08      16      6    46  1866 32d/32i     256   18432
  3 r001i01b00     00     11      22      6    46  1866 32d/32i     256   18432
  4 r001i01b00     01     00      32      6    46  1866 32d/32i     256   18432
  5 r001i01b00     01     03      38      6    46  1866 32d/32i     256   18432
  6 r001i01b00     01     08      48      6    46  1866 32d/32i     256   18432
  7 r001i01b00     01     11      54      6    46  1866 32d/32i     256   18432
  8 r001i01b01     02     00      64      6    46  1866 32d/32i     256   18432
  9 r001i01b01     02     03      70      6    46  1866 32d/32i     256   18432
 10 r001i01b01     02     08      80      6    46  1866 32d/32i     256   18432
 11 r001i01b01     02     11      86      6    46  1866 32d/32i     256   18432
 12 r001i01b01     03     00      96      6    46  1866 32d/32i     256   18432
 13 r001i01b01     03     03     102      6    46  1866 32d/32i     256   18432
 14 r001i01b01     03     08     112      6    46  1866 32d/32i     256   18432
 15 r001i01b01     03     11     118      6    46  1866 32d/32i     256   18432
 16 r001i01b02     04     00     128      6    46  1866 32d/32i     256   18432
                                  ...
 63 r001i01b03     07     11     247      6    46  1866 32d/32i     256   18432
```

The `cpumap`(1) command displays logical CPUs and shows relationships between them in a human-readable format. Aspects displayed include hyperthread relationships, last level cache sharing, and topological placement. The `cpumap` command gets its information from `/proc/cpuinfo`, the `/sys/devices/system` directory structure, and /proc/sgi_uv/topology. When creating cpusets, the Socket numbers reported in the output section Processor Numbering on Socket(s) corresponds to the mems argument you would use in the definition of a cpuset. The cpuset mems argument is the list of memory nodes that tasks in the cpuset are allowed to use. For more information, see the Using “Cpusets at the Shell Prompt” section in Chapter 2 “Cpusets on Linux” in the *Linux Resource Administration Guide* available at http://docs.sgi.com.

Example output is, as follows:

```
uv-sys:~ # cpumap
Mon Oct 18 13:40:26 CDT 2010
uv-sys.mycompany.com

This an SGI UV
model name           : Intel(R) Xeon(R) CPU E7520 @ 1.87GHz
Architecture         : x86_64
cpu MHz              : 1866.557
cache size           : 18432 KB (Last Level)

Total Number of Sockets                 : 8
Total Number of Cores                   : 32    (4 per socket)
Hyperthreading                          : ON
Total Number of Physical Processors     : 32
Total Number of Logical Processors      : 64    (2 per Phys Processor)

UV Information
 HUB Version:                            UVHub  2.0
 Number of Hubs (Blades):                4
 Number of connected NUMAlink ports:    12
=============================================================================

Hub-Processor Mapping

  Hub Location      Processor Numbers -- HyperThreads in ()
  --- ----------    ---------------------------------------
    0 r001i01b00       0    1    2    3    4    5    6    7
                  (   32   33   34   35   36   37   38   39 )
    1 r001i01b01       8    9   10   11   12   13   14   15
                  (   40   41   42   43   44   45   46   47 )
    2 r001i01b02      16   17   18   19   20   21   22   23
                  (   48   49   50   51   52   53   54   55 )
    3 r001i01b03      24   25   26   27   28   29   30   31
                  (   56   57   58   59   60   61   62   63 )

=============================================================================

Processor Numbering on Socket(s)

  Socket    (Logical) Processors
  ------    -------------------------
     0      0    1    2    3   32   33   34   35
     1      4    5    6    7   36   37   38   39
     2      8    9   10   11   40   41   42   43
     3     12   13   14   15   44   45   46   47
     4     16   17   18   19   48   49   50   51
     5     20   21   22   23   52   53   54   55
     6     24   25   26   27   56   57   58   59
     7     28   29   30   31   60   61   62   63

=============================================================================

Sharing of Last Level (3) Caches

  Socket    (Logical) Processors
  ------    -------------------------
     0      0    1    2    3   32   33   34   35
     1      4    5    6    7   36   37   38   39
     2      8    9   10   11   40   41   42   43
     3     12   13   14   15   44   45   46   47
     4     16   17   18   19   48   49   50   51
     5     20   21   22   23   52   53   54   55
     6     24   25   26   27   56   57   58   59
     7     28   29   30   31   60   61   62   63

=============================================================================

HyperThreading

  Shared Processors
  -----------------
 (    0,   32) (    1,   33) (    2,   34) (    3,   35)
 (    4,   36) (    5,   37) (    6,   38) (    7,   39)
 (    8,   40) (    9,   41) (   10,   42) (   11,   43)
 (   12,   44) (   13,   45) (   14,   46) (   15,   47)
 (   16,   48) (   17,   49) (   18,   50) (   19,   51)
 (   20,   52) (   21,   53) (   22,   54) (   23,   55)
 (   24,   56) (   25,   57) (   26,   58) (   27,   59)
 (   28,   60) (   29,   61) (   30,   62) (   31,   63)
```

Use the `x86info`(1) command to display x86 CPU diagnostics information, as follows:

```
uv44-sys:~ # x86info
x86info v1.25.  Dave Jones 2001-2009
Feedback to .

Found 64 CPUs
--------------------------------------------------------------------------
CPU #1
EFamily: 0 EModel: 2 Family: 6 Model: 46 Stepping: 6
CPU Model: Unknown model.
Processor name string: Intel(R) Xeon(R) CPU           E7520  @ 1.87GHz
Type: 0 (Original OEM)  Brand: 0 (Unsupported)
Number of cores per physical package=16
Number of logical processors per socket=32
Number of logical processors per core=2
APIC ID: 0x0    Package: 0  Core: 0   SMT ID 0
--------------------------------------------------------------------------
CPU #2
EFamily: 0 EModel: 2 Family: 6 Model: 46 Stepping: 6
CPU Model: Unknown model.
Processor name string: Intel(R) Xeon(R) CPU           E7520  @ 1.87GHz
Type: 0 (Original OEM)  Brand: 0 (Unsupported)
Number of cores per physical package=16
Number of logical processors per socket=32
Number of logical processors per core=2
APIC ID: 0x6    Package: 0  Core: 0   SMT ID 6
--------------------------------------------------------------------------
CPU #3
EFamily: 0 EModel: 2 Family: 6 Model: 46 Stepping: 6
CPU Model: Unknown model.
Processor name string: Intel(R) Xeon(R) CPU           E7520  @ 1.87GHz
Type: 0 (Original OEM)  Brand: 0 (Unsupported)
Number of cores per physical package=16
Number of logical processors per socket=32
Number of logical processors per core=2
APIC ID: 0x10   Package: 0  Core: 0   SMT ID 16
-------------------------------------------------------------------------
                       ...
```

You can also use the `uname` command, which returns the kernel version and other machine information. For example:

```
uv44-sys:~ # uname -a
Linux uv44-sys 2.6.32.13-0.4.1.1559.0.PTF-default #1 SMP 2010-06-15 12:47:25 +0200 x86_64 x86_64 x86_64
```

For more system information, change directory (cd) to the `/sys/devices/system/node/node0/cpu0/cache` directory.

For example:

```
uv44-sys:/sys/devices/system/node/node0/cpu0/cache # ls
index0  index1  index2  index3
```

Change directory to index0 and list the contents, as follows:

```
uv44-sys:/sys/devices/system/node/node0/cpu0/cache/index0 # ls
coherency_line_size  level  number_of_sets  physical_line_partition  shared_cpu_list  shared_cpu_map  size  type  way
```

## Sources of Performance Problems

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

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

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

### Profiling with `perf`

The `perf`(1) software provides the performance analysis tools for Linux. Performance counters for Linux are a kernel-based subsystem that provide a framework for all things performance analysis. It covers hardware level CPU/performance monitoring unit (PMU) features and software features, such as, software counters and tracepoints, as well. To use the `perf` profiling tools, you need to make sure the `perf` RPM is installed. The `perf` RPM comes with the SLES operating system, it is not an SGI product. . For more information, see the following man pages: `perf-stat`(1), `perf-top`(1), `perf-record`(1), `perf-report`(1), `perf-list`(1).

### Profiling with `PerfSuite`

“PerfSuite is an easy-to-use collection of tools, utilities, and libraries to support application software performance analysis on Linux-based systems. It includes components to assist with a wide variety of performance-related tasks, ranging from assistance with compiler optimization reports to hardware performance counting, profiling, and MPI usage summarization. PerfSuite is Open Source software, approved for licensing under the University of Illinois/NCSA Open Source License (OSI-approved). You can find out more about PerfSuite at the project web sites, located at: http://perfsuite.ncsa.uiuc.edu/ or http://perfsuite.sourceforge.net/.”

For NCSA specific information about using PerfSuite tools, see http://www.ncsa.illinois.edu/UserInfo/Resources/Software/Tools/PerfSuite/.

“psrun is a PerfSuite command-line utility that can be used to gather hardware performance information on an unmodified executable. It’s a convenient and flexible way to do quick performance monitoring/measurement.” For more information, see http://perfsuite.ncsa.uiuc.edu/psrun/.

## Using VTune for Remote Sampling

The Intel® VTune™ Performance Analyzer (now called Intel® VTune™ Amplifier XE) 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. VTune allows you to perform interactive experiments while connected to the host through its GUI. PTU (Performance Tuning Utility) is another tool which requires theIntel VTune license.

For details about using VTune, see the following URL:

http://software.intel.com/en-us/articles/intel-vtune-performance-analyzer-for-linux-documentation/

## 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.

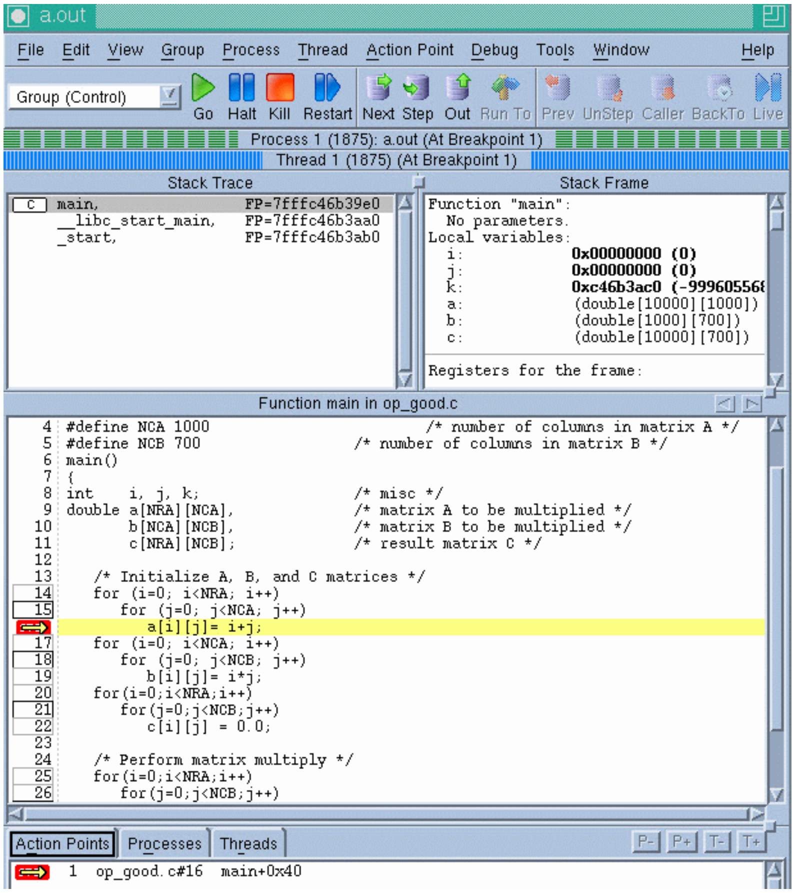
- `idb`: the Intel debugger. This is a fully symbolic debugger for the Linux platform. The debugger provides extensive support for debugging programs written in C, C++, FORTRAN 77, and Fortran 90. `idb` includes a GUI and it supports both Intel and GNU compilers.

  Running `idb` with the `-gdb` option on the shell command line provides `gdb`(1)-like user commands and debugger output.

- `ddd`: a GUI to a command line debugger. It supports `gdb` and `idb`. For details about usage, see the following subsection.

- TotalView: a licensed graphical debugger useful in an MPI environment (see http://www.totalviewtech.com/)

Figure 3-1 on page 20 shows a TotalView sesssion.


a.out
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Prev UnStep Caller BackTo Live
Process 1 (1875): a.out (At Breakpoint 1)
Thread 1 (1875) (At Breakpoint 1)
Stack Trace
Stack Frame
Function main in op_good.c
Action Points Processes Threads
P- P+ T- T+

```
C  main,              FP=7fffc46b39e0
   __libc_start_main, FP=7fffc46b3aa0
   _start,            FP=7fffc46b3ab0
```

```
Function "main":
  No parameters.
Local variables:
  i:              0x00000000 (0)
  j:              0x00000000 (0)
  k:              0xc46b3ac0 (-99960556
  a:              (double[10000][1000])
  b:              (double[1000][700])
  c:              (double[10000][700])

Registers for the frame:
```

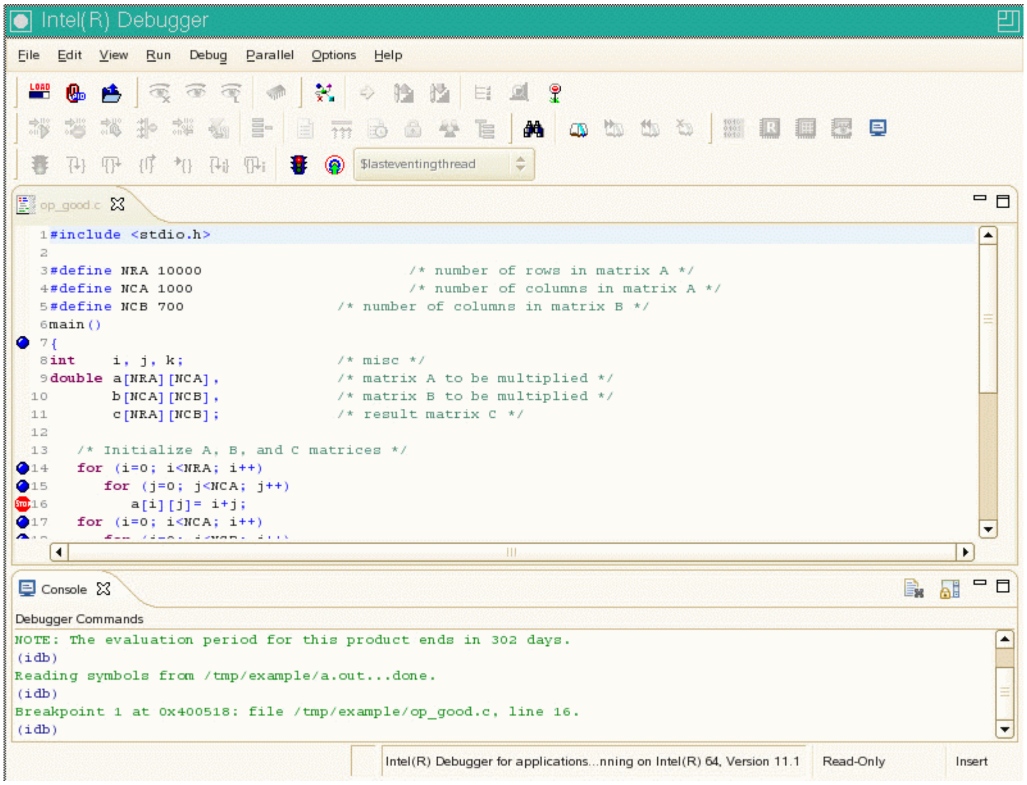
```
#define NCA 1000                    /* number of columns in matrix A */
#define NCB 700            /* number of columns in matrix B */
main()
{
int    i, j, k;            /* misc */
double a[NRA][NCA],        /* matrix A to be multiplied */
       b[NCA][NCB],        /* matrix B to be multiplied */
       c[NRA][NCB];        /* result matrix C */

   /* Initialize A, B, and C matrices */
   for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
         a[i][j]= i+j;
   for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
         b[i][j]= i*j;
   for(i=0;i<NRA;i++)
      for(j=0;j<NCB;j++)
         c[i][j] = 0.0;

   /* Perform matrix multiply */
   for(i=0;i<NRA;i++)
      for(j=0;j<NCB;j++)
```

```
1  op_good.c#16  main+0x40
```

**Figure 3-1** TotalView Session

## Using the Intel Debugger `idb`

`idb` is part of the Intel® Compiler suite, both Fortran and C/C++ (now called Intel® Composer XE). You are asked during the installation if you want to install it or not. When running `idb` you get the GUI interface. When you invoke the `idbc` command, you get the command line interface.


Intel(R) Debugger
File Edit View Run Debug Parallel Options Help
$lasteventingthread
op_good.c
Console
Debugger Commands
Intel(R) Debugger for applications...nning on Intel(R) 64, Version 11.1
Read-Only
Insert

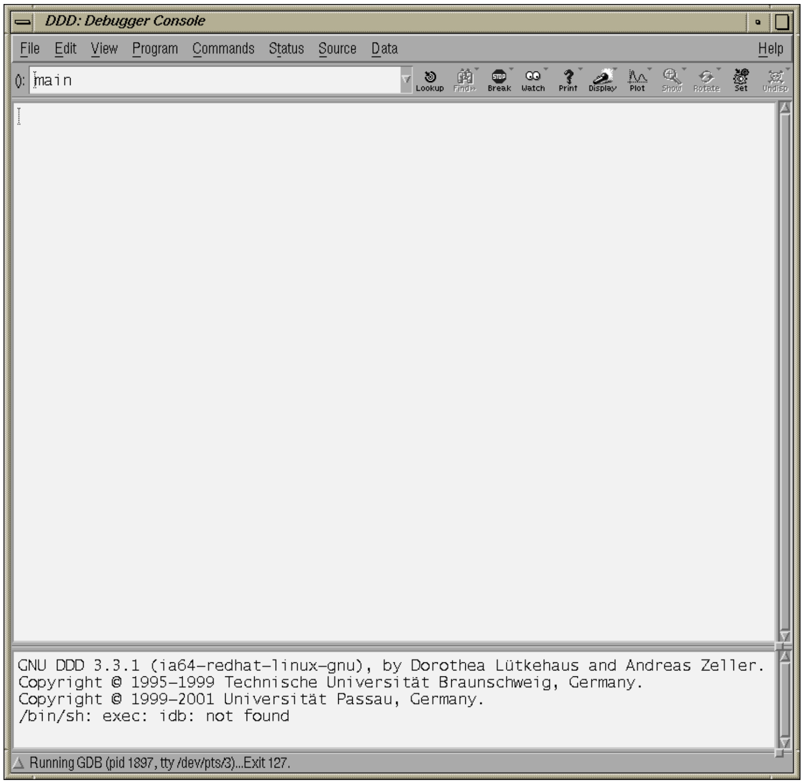
```
#include <stdio.h>

#define NRA 10000                           /* number of rows in matrix A */
#define NCA 1000                            /* number of columns in matrix A */
#define NCB 700                     /* number of columns in matrix B */
main()
{
int     i, j, k;                    /* misc */
double  a[NRA][NCA],                /* matrix A to be multiplied */
        b[NCA][NCB],                /* matrix B to be multiplied */
        c[NRA][NCB];                /* result matrix C */

   /* Initialize A, B, and C matrices */
   for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
         a[i][j]= i+j;
   for (i=0; i<NCA; i++)
```

```
NOTE: The evaluation period for this product ends in 302 days.
(idb)
Reading symbols from /tmp/example/a.out...done.
(idb)
Breakpoint 1 at 0x400518: file /tmp/example/op_good.c, line 16.
(idb)
```

**Figure 3-2** **Intel® Debugger** GUI

## Using ddd

The DataDisplayDebugger `ddd`(1) tool is a GUI to an arbitrary command line debugger as shown in Figure 3-3 on page 22. When starting `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 3-3** DataDisplayDebugger(`ddd`)(1)

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

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

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

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

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

*Chapter 4*

# Monitoring Tools

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

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

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

## System Monitoring Tools

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

- "Hardware Inventory and Usage Commands" on page 25
- "Performance Co-Pilot Monitoring Tools" on page 30
- "System Usage Commands" on page 33
- "Memory Statistics and `nodeinfo` Command" on page 37

### Hardware Inventory and Usage Commands

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

- "`hwinfo`(1) Command" on page 26
- "`topology`(1) Command" on page 26
- "`gtopology`(1) Command" on page 27

### **hwinfo**(1) Command

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

```
% rpm -q hwinfo
hwinfo-15.33-0.4.26
```

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

### **topology**(1) Command

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

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

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

```
uv44-sys:~ # topology
Serial number: UV-00000044
Partition number: 0
4 Blades
64 CPUs
125.97 Gb Memory Total

Blade          ID       asic  NASID         Memory
-------------------------------------------------
    0 r001i01b00  UVHub 2.0      0    16757488 kB
    1 r001i01b01  UVHub 2.0      2    16777216 kB
    2 r001i01b02  UVHub 2.0      4    16777216 kB
    3 r001i01b03  UVHub 2.0      6    16760832 kB

CPU      Blade PhysID CoreID APIC-ID Family Model Speed L1(KiB) L2(KiB) L3(KiB)
-------------------------------------------------------------------------------
  0 r001i01b00     00     00       0      6    46  1866 32d/32i     256   18432
  1 r001i01b00     00     03       6      6    46  1866 32d/32i     256   18432
  2 r001i01b00     00     08      16      6    46  1866 32d/32i     256   18432
  3 r001i01b00     00     11      22      6    46  1866 32d/32i     256   18432
  4 r001i01b00     01     00      32      6    46  1866 32d/32i     256   18432
```

```
 5 r001i01b00     01     03      38      6    46  1866 32d/32i     256   18432
 6 r001i01b00     01     08      48      6    46  1866 32d/32i     256   18432
 7 r001i01b00     01     11      54      6    46  1866 32d/32i     256   18432
 8 r001i01b01     02     00      64      6    46  1866 32d/32i     256   18432
 9 r001i01b01     02     03      70      6    46  1866 32d/32i     256   18432
10 r001i01b01     02     08      80      6    46  1866 32d/32i     256   18432
11 r001i01b01     02     11      86      6    46  1866 32d/32i     256   18432
12 r001i01b01     03     00      96      6    46  1866 32d/32i     256   18432
13 r001i01b01     03     03     102      6    46  1866 32d/32i     256   18432
14 r001i01b01     03     08     112      6    46  1866 32d/32i     256   18432
15 r001i01b01     03     11     118      6    46  1866 32d/32i     256   18432
16 r001i01b02     04     00     128      6    46  1866 32d/32i     256   18432
                                      ...
62 r001i01b03     07     08     241      6    46  1866 32d/32i     256   18432
63 r001i01b03     07     11     247      6    46  1866 32d/32i     256   18432
```

## `gtopology`(1) Command

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

Figure 4-1 on page 28, shows the ring topology (the eight nodes are shown in pink, the NUMAlink connections in cyan) of an SGI system with 16 CPUs.

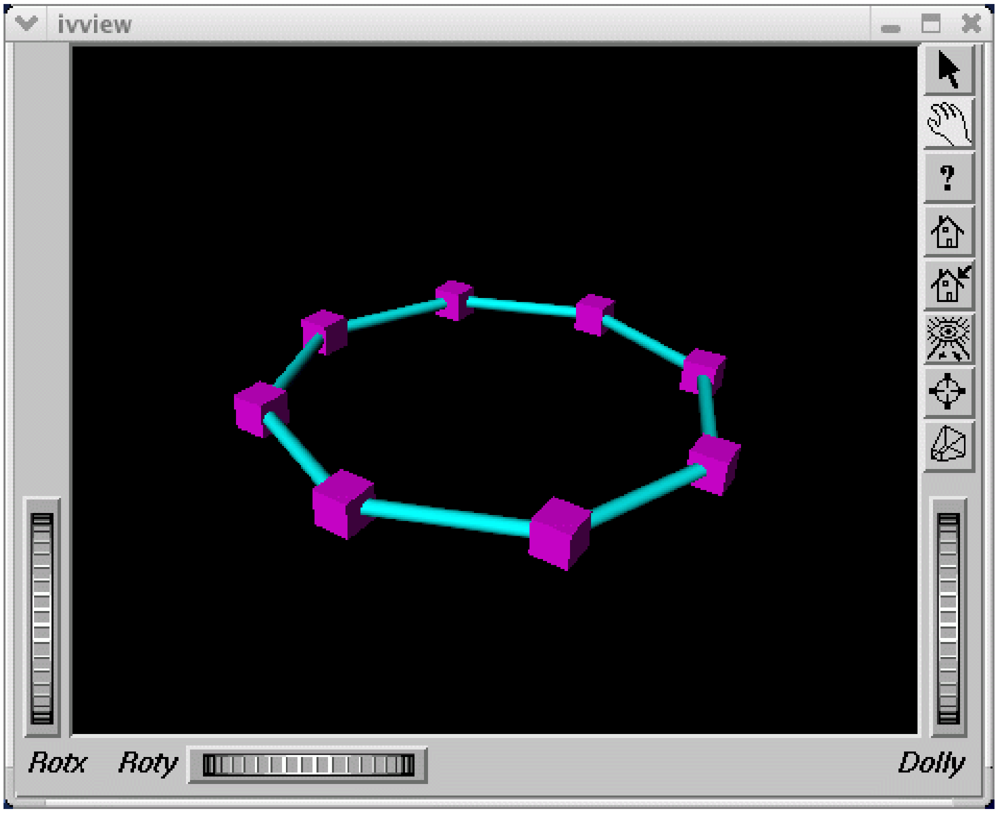


**Figure 4-1** Ring Topology of an System with 16 CPUs

Figure 4-2 on page 29, shows the fat-tree topology of an SGI system with 32 CPUs. Again, nodes are the pink cubes. Routers are shown as blue spheres (if all ports are used) otherwise, yellow.

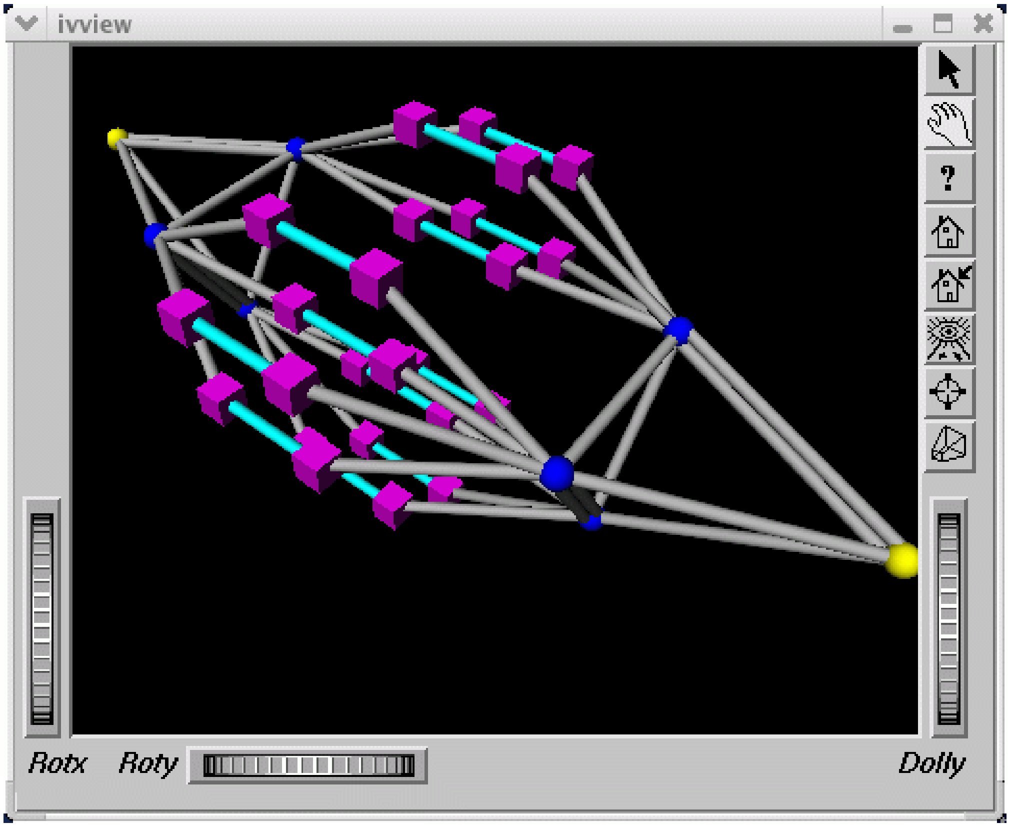


**Figure 4-2** An SGI System with 32 CPUs Fat-tree Topology

Figure 4-3 on page 30, shows an SGI system with 512 CPUs. The dual planes of the fat-tree topology are clearly visible.

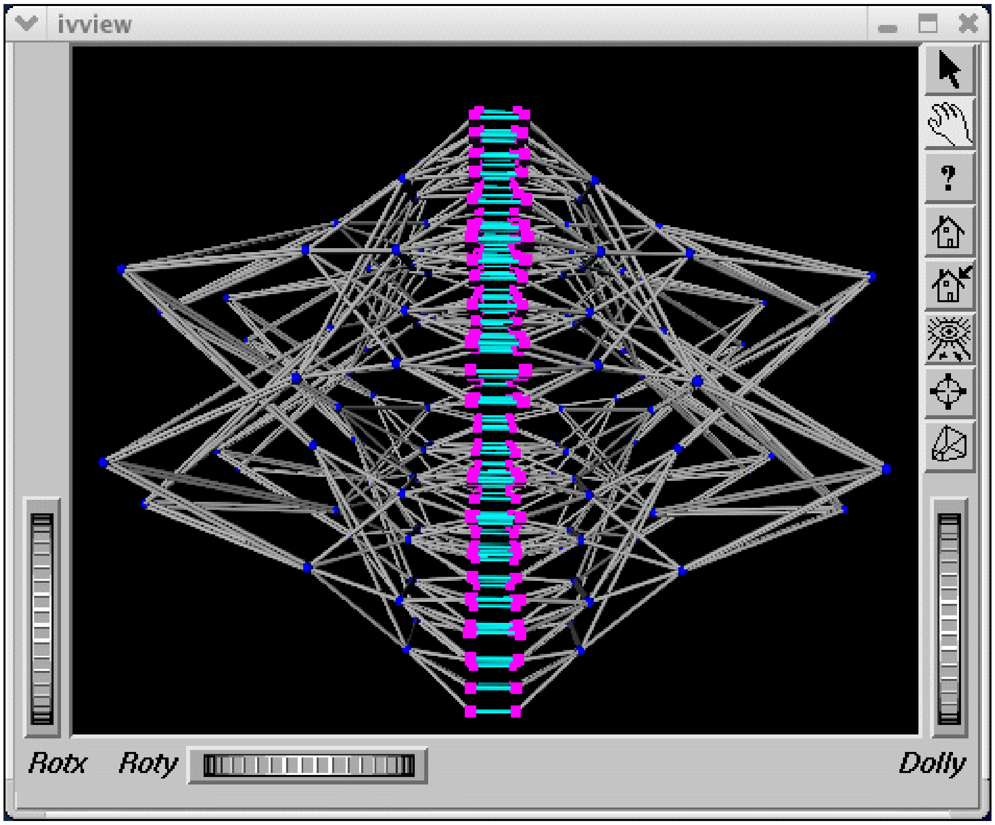


**Figure 4-3** An SGI System with 512 CPUs

## Performance Co-Pilot Monitoring Tools

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

- "`hubstats`(1) Command" on page 31
- "`linkstat-uv`(1) Command" on page 31
- "Other Performance Co-Pilot Monitoring Tools" on page 31

### hubstats(1) Command

The hubstats(1) command is a command line tool for monitoring NUMAlink traffic, directory cache operations and global reference unit (GRU) traffic statistics on SGI UV systems. It will not work on any other platform. For more information, see the hubstats(1) man page.

### linkstat-uv(1) Command

The linkstat-uv(1) command is a command-line tool for monitoring NUMAlink traffic and error rates on SGI UV systems. It will not work on any other platform.This tool shows packets and Mbytes sent/received on each NUMAlink in the system, as well as error rates. It is useful as a performance monitoring tool, as well as, a tool for helping you to diagnose and identify faulty hardware. For more information, see the linkstat-uv(1) man page.

### Other Performance Co-Pilot Monitoring Tools

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

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

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

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

- pmcd(1) – the Performance Co-Pilot metrics collector daemon
- PCPIntro(1) - introduction to Performance Co-Pilot monitoring tools, generic command line usage and environment variables
- PMAPI(3) - introduction to the Performance Co-Pilot API libraries for developing new performance monitoring tools
- PMDA(3) - introduction to the Performance Co-Pilot Metrics Domain Agent API, for developing new Performance Co-Pilot agents
- topology(1) – displays general information about SGI UV systems, with a focus on node information. This includes counts of the cpus, nodes, routers and memory, as well as, various I/O devices. More detailed information is available for node IDs, NASIDs, memory per node, system serial number, partition number, UV Hub versions, CPU to node mappings, I/O device descriptions, and general CPU and I/O information.

## 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:

```
uv44-sys:~ # uptime
  3:48pm  up   2:50,  5 users,  load average: 0.12, 0.25, 0.40
```

The output displays the current time, the length of time the system has been up, the number of users on the system, and the average number of jobs in the run queue over the last one, five, and 15 minutes.

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

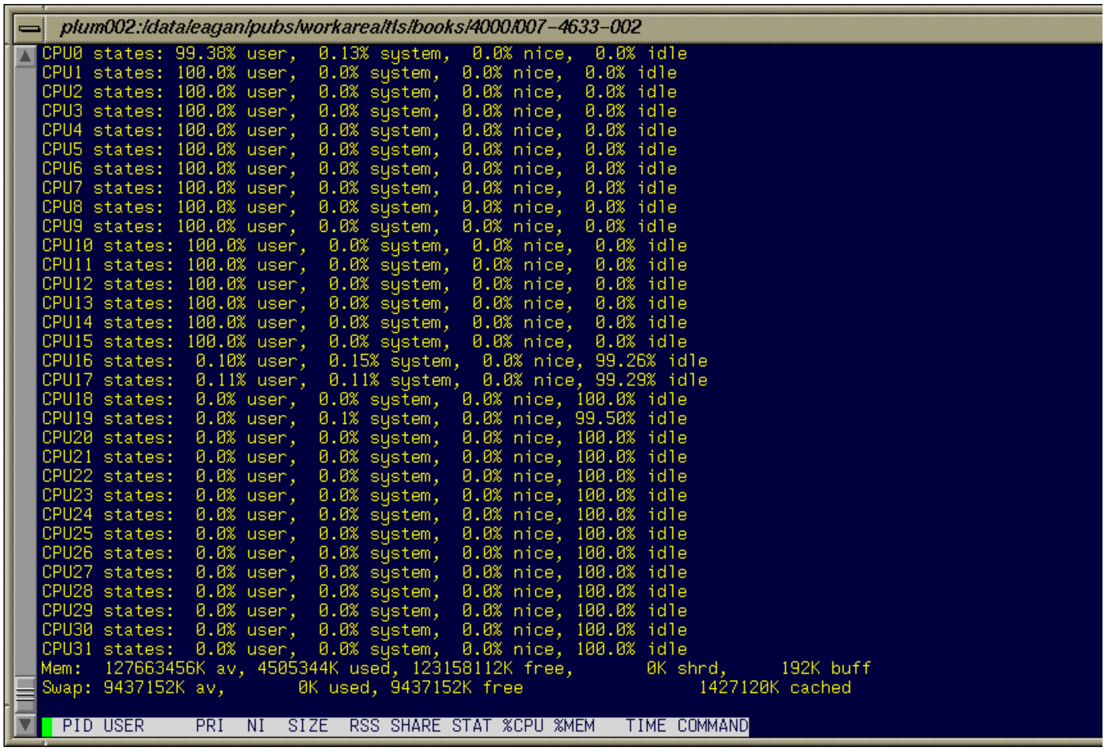
```
uv44-sys:~ # w
 15:47:48 up  2:49,  5 users,  load average: 0.04, 0.27, 0.42
USER     TTY        LOGIN@   IDLE   JCPU   PCPU WHAT
root     pts/0     13:10    1:41m  0.07s  0.07s -bash
root     pts/2     13:31    0.00s  0.14s  0.02s w
boetcher pts/4     14:30    2:13   0.73s  0.73s -csh
root     pts/5     14:32    1:14m  0.04s  0.04s -bash
root     pts/6     15:09   31:25   0.08s  0.08s -bash
```

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 4-4 on page 35.



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

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

```
uv44-sys:~ # vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 130301028  79868 1287576    0    0     1     0    8    7  0  0 100  0  0
```

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

The `iostat`(1) command is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates.

The iostat command generates reports that can be used to change system configuration to better balance the input/output load between physical disks. For more information, see the iostat(1) man page.

```
uv44-sys:~ # iostat
Linux 2.6.32.13-0.4.1.1559.0.PTF-default (uv44-sys) 10/18/2010  _x86_64_

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.18    0.00    0.04    0.02    0.00   99.77

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               3.02        72.80        16.28     722432     161576
sdb               0.01         0.34         0.00       3419          0
```

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

```
uv44-sys:~ # sar
Linux 2.6.32.13-0.4.1.1559.0.PTF-default (uv44-sys)     10/18/2010      _x86_64_

12:58:47 PM       LINUX RESTART

01:00:01 PM     CPU     %user     %nice   %system   %iowait    %steal     %idle
01:10:01 PM     all      0.00      0.00      0.01      0.01      0.00     99.99
01:20:01 PM     all      0.00      0.00      0.00      0.00      0.00     99.99
01:30:01 PM     all      0.00      0.00      0.01      0.00      0.00     99.99
01:40:01 PM     all      0.02      0.00      0.01      0.01      0.00     99.96
01:50:01 PM     all      0.03      0.00      0.01      0.02      0.00     99.94
02:00:01 PM     all      0.12      0.00      0.01      0.01      0.00     99.86
02:10:01 PM     all      0.01      0.00      0.01      0.00      0.00     99.98
02:20:01 PM     all      0.76      0.00      0.04      0.00      0.00     99.20
02:30:01 PM     all      0.28      0.00      0.01      0.00      0.00     99.71
02:40:01 PM     all      0.27      0.00      0.14      0.05      0.00     99.54
02:50:01 PM     all      0.00      0.00      0.01      0.00      0.00     99.99
03:00:01 PM     all      0.00      0.00      0.01      0.03      0.00     99.96
03:10:01 PM     all      0.20      0.00      0.03      0.01      0.00     99.76
03:20:01 PM     all      0.02      0.00      0.01      0.01      0.00     99.96
```

```
03:30:01 PM     all      0.15      0.00      0.08      0.00      0.00     99.77
03:40:01 PM     all      1.03      0.00      0.10      0.08      0.00     98.79
Average:        all      0.18      0.00      0.03      0.02      0.00     99.77
```

## Memory Statistics and nodeinfo Command

nodeinfo(1) is a tool for monitoring per-node NUMA memory statistics on SGI UV systems. The nodeinfo tool reads /sys/devices/system/node/*/meminfo and /sys/devices/system/node/*/numastat on the local system to gather NUMA memory statistics.

Sample memory statistic from the nodeinfo(1) command are, as follows:

```
uv44-sys:~ # nodeinfo
Memory Statistics  Tue Oct 26 12:01:58 2010
uv44-sys
      ------------------------- Per Node KB --------------------------------  ------ Preferred Alloc -------  -- Loc/
node       Total        Free        Used       Dirty        Anon        Slab     hit    miss foreign interlv   local
   0    16757488    16277084      480404          52       34284       36288   20724       0       0       0   20720
   1    16777216    16433988      343228          68        6772       17708    4477       0       0       0    3381
   2    16777216    16438568      338648          76        6908       12620    1804       0       0       0     709
   3    16760832    16429844      330988          56        2820       16836    1802       0       0       0     708
   4    16777216    16444408      332808          88       10124       13588    1517       0       0       0     417
   5    16760832    16430300      330532          72        1956       17304    4546       0       0       0    3453
   6    16777216    16430788      346428          36        3236       15292    3961       0       0       0    2864
   7    16760832    16435532      325300          44        1220       14800    3971       0       0       0    2877
 TOT   134148848   131320512     2828336         492       67320      144436   42802       0       0       0   35129
 Press "h" for help
```

From an interactive nodeinfo session, enter **h** for a help statement:

```
Display memory statistics by node.
        q   quit
        +   Increase starting node number. Used only if more nodes than will
            fit in the current window.
        -   Decrease starting node number. Used only if more nodes than will
            fit in the current window.
        b   Start output with node 0.
        e   Show highest node number.
        k   show sizes in KB.
        m   show sizes in MB.
        p   show sizes in pages.
        t   Change refresh rate.
        A   Show/Hide memory policy stats.
        H   Show/Hide hugepage info.
        L   Show/Hide LRU Queue stats.

Field definitions:
        hit - page was allocated on the preferred node
        miss - preferred node was full. Allocation occurred on THIS node
               by a process running on another node that was full

        foreign - Preferred node was full. Had to allocate somewhere
                  else.

        interlv - allocation was for interleaved policy

        local - page allocated on THIS node by a process running on THIS node
        remote - page allocated on THIS node by a process running on ANOTHER node


  (press any key to exit from help screen)
```

For more information on using `nodeinfo` to view memory consumption on the nodes assigned to your job, see "Other ccNUMA Performance Issues" on page 99.

*Chapter 5*

# Data Placement Tools

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

- "Data Placement Tools Overview" on page 39
- "`taskset` Command" on page 44
- "`dplace` Command" on page 46
- "`dlook` Command" on page 55
- "`omplace` Command" on page 61
- "`numactl` Command" on page 62
- "Installing NUMA Tools" on page 62
- "An Overview of the Advantages Gained by Using Cpusets" on page 63

## Data Placement Tools Overview

On an symmetric multiprocessor (SMP) machine, all data is visible from all processors. NonUniform Memory Access (NUMA) machines also have a shared address space. In both cases, there is a single shared memory space and a single operating system instance. However, in an SMP machine, each processor is functionally identical and has equal time access to every memory address. In contrast, a NUMA system has a shared address space, but the access time to memory vary over physical address ranges and between processing elements. The Intel Xeon 7500 series processor (Nehalem i7 architecture) is an example of NUMA architecture. Each processor has its own memory and can address the memory attached to another processor through the Quick Path Interconnect (QPI).

The SGI UV 1000 series is a family of multiprocessor distributed shared memory (DSM) computer systems that initially scale from 32 to 2560 Intel processor cores, as a cache-coherent single system image (SSI). The SGI UV 100 series is a family of multiprocessor distributed shared memory (DSM) computer systems that initially scale from 16 to 768 Intel processor cores as a cache-coherent SSI. The SGI UV 2000 series scales from 32 to 4,096 Intel processor cores, as a cache-coherent SSI.

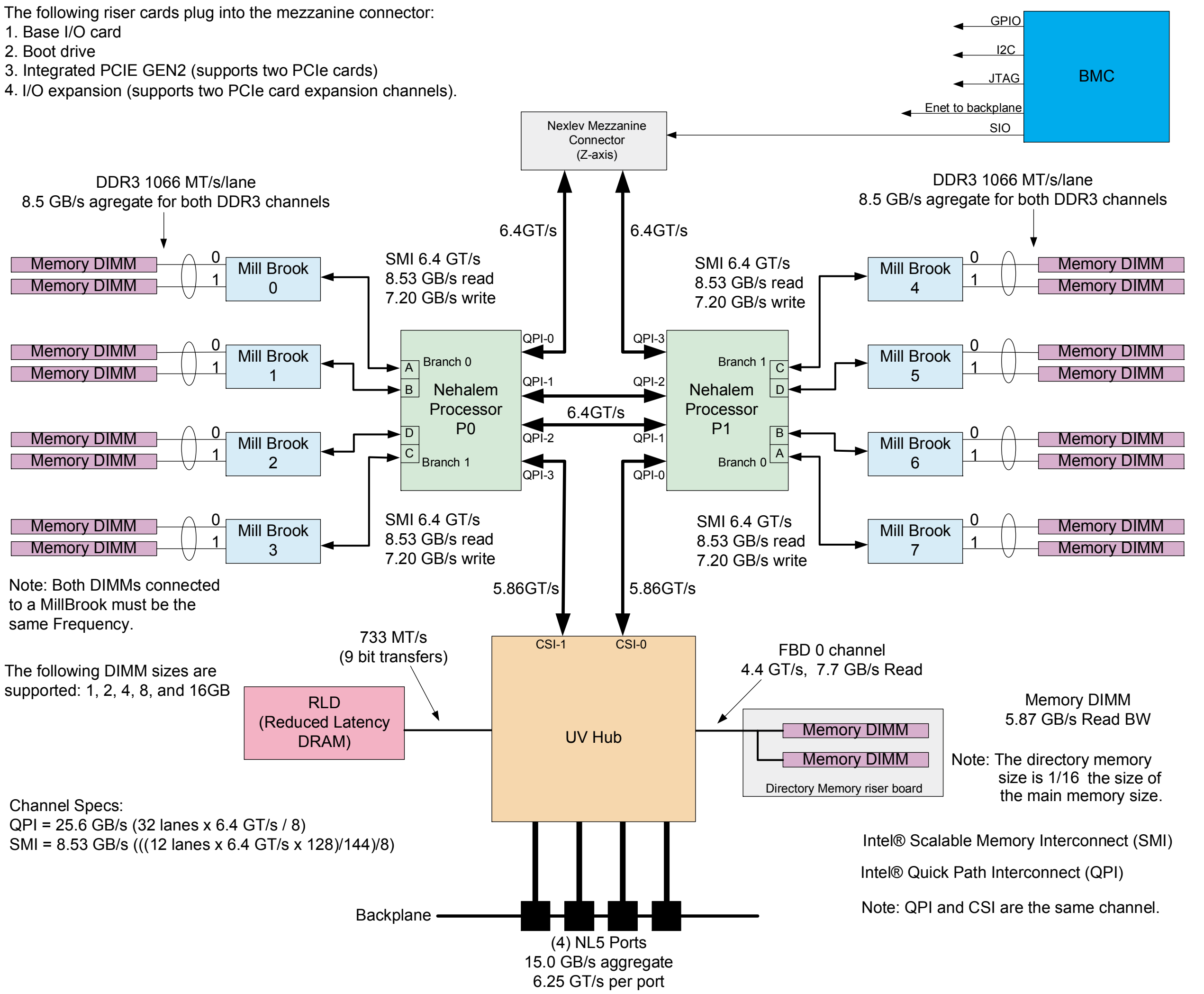
In both UV series systems, there are two levels of NUMA: intranode managed by the QPI and internode managed through the HUB ASIC and NUMAlink 5 or NUMAlink 6. This section covers the following topics:

- "Distributed Shared Memory (DSM)" on page 40
- "ccNUMA Architecture" on page 41
- "Cache Coherency" on page 42
- "Non-uniform Memory Access (NUMA)" on page 42

## Distributed Shared Memory (DSM)

In the SGI UV 100, SGI UV 1000 system or SGI UV 2000 series system, memory is physically distributed both within and among the IRU enclosures (compute/memory/I/O blades); however, it is accessible to and shared by all devices connected by NUMAlink within the single-system image (SSI). This is to say that all components connected by NUMAlink sharing a single Linux operating system, operate and share the memory "fabric" of the system. Memory latency is the amount of time required for a processor to retrieve data from memory. Memory latency is lowest when a processor accesses local memory. Note the following sub-types of memory within a system:

- If a processor accesses memory that is on a compute node blade, the memory is referred to as the node's local memory. Figure 5-1 on page 41 shows a conceptual block diagram of the blade's memory, compute and I/O pathways.
- If processors access memory located in other blade nodes within the IRU, (or other NUMAlink IRUs) the memory is referred to as remote memory.
- The total memory within the NUMAlink system is referred to as global memory.



The following riser cards plug into the mezzanine connector:
1. Base I/O card
2. Boot drive
3. Integrated PCIE GEN2 (supports two PCIe cards)
4. I/O expansion (supports two PCIe card expansion channels).

Note: Both DIMMs connected to a MillBrook must be the same Frequency.

The following DIMM sizes are supported: 1, 2, 4, 8, and 16GB

Channel Specs:
QPI = 25.6 GB/s (32 lanes x 6.4 GT/s / 8)
SMI = 8.53 GB/s (((12 lanes x 6.4 GT/s x 128)/144)/8)

Note: The directory memory size is 1/16 the size of the main memory size.

Intel® Scalable Memory Interconnect (SMI)

Intel® Quick Path Interconnect (QPI)

Note: QPI and CSI are the same channel.

**Figure 5-1** Blade Node Block Diagram

## ccNUMA Architecture

As the name implies, the cache-coherent non-uniform memory access (ccNUMA) architecture has two parts, cache coherency and nonuniform memory access, which are discussed in the sections that follow.

### Cache Coherency

The SGI UV systems use caches to reduce memory latency. Although data exists in local or remote memory, copies of the data can exist in various processor caches throughout the system. Cache coherency keeps the cached copies consistent.

To keep the copies consistent, the ccNUMA architecture uses directory-based coherence protocol. In directory-based coherence protocol, each block of memory (128 bytes) has an entry in a table that is referred to as a directory. Like the blocks of memory that they represent, the directories are distributed among the compute/memory blade nodes. A block of memory is also referred to as a cache line.

Each directory entry indicates the state of the memory block that it represents. For example, when the block is not cached, it is in an unowned state. When only one processor has a copy of the memory block, it is in an exclusive state. And when more than one processor has a copy of the block, it is in a shared state; a bit vector indicates which caches may contain a copy.

When a processor modifies a block of data, the processors that have the same block of data in their caches must be notified of the modification. The SGI UV systems use an invalidation method to maintain cache coherence. The invalidation method purges all unmodified copies of the block of data, and the processor that wants to modify the block receives exclusive ownership of the block.

### Non-uniform Memory Access (NUMA)

In DSM systems, memory is physically located at various distances from the processors. As a result, memory access times (latencies) are different or "non-uniform." For example, it takes less time for a processor blade to reference its locally installed memory than to reference remote memory.

## Data Placement Practices

For cc-NUMA systems like the SGI UV systems, there is a performance penalty to access remote memory versus local memory. Because the Linux operating system has a tendency to migrate processes, the importance of using a placement tool becomes more apparent. Various data placement tools are described in this section.

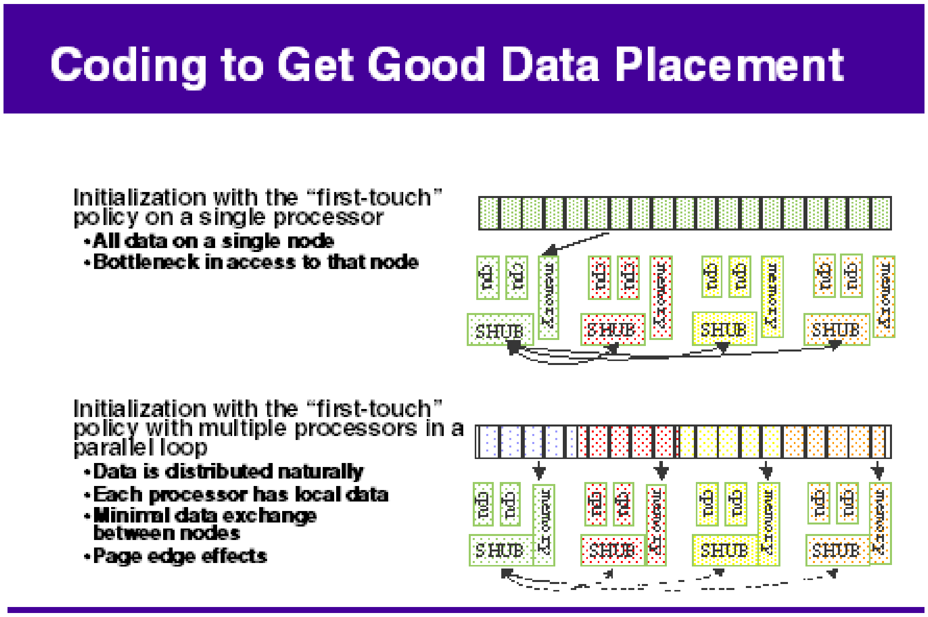
For a detailed overview of SGI UV system architecture, see the *SGI Altix UV 100 System User's Guide*, *SGI Altix UV 1000 System User's Guide* or *SGI UV 2000 System User Guide*.

Special optimization applies to SGI UV systems to exploit multiple paths to memory, as follows:

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

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

Figure 5-2 on page 43, shows how to code to get good data placement.



Coding to Get Good Data Placement

Initialization with the “first-touch” policy on a single processor

- All data on a single node
- Bottleneck in access to that node

Initialization with the “first-touch” policy with multiple processors in a parallel loop

- Data is distributed naturally
- Each processor has local data
- Minimal data exchange between nodes
- Page edge effects

**Figure 5-2** Coding to Get Good Data Placement

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

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

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

## taskset Command

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

```
taskset [options] mask command [arg]...
taskset [options] -p [mask] pid
```

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

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

| | |
|---|---|
| 0x00000001 | is processor #0 |

| | |
|---|---|
| `0x00000003` | is processors #0 and #1 |
| `0xFFFFFFFF` | is all processors (#0 through #31) |

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

The `taskset` command does not pin a task to a specific CPU. It only restricts a task so that it does not run on any CPU that is not in the `cpulist`. For example, if you use `taskset` to launch an application that forks multiple tasks, it is possible that multiple tasks will initially be assigned to the same CPU even though there are idle CPUs that are in the `cpulist`. Scheduler load balancing software will eventually distribute the tasks so that CPU bound tasks run on different CPUs. However, the exact placement is not predictable and can vary from run-to-run. After the tasks are evenly distributed (assuming that happens), nothing prevents tasks from jumping to different CPUs. This can affect memory latency since pages that were node-local before the jump may be remote after the jump.

If you are running an MPI application, SGI recommends that you do not use the `taskset` command. The `taskset` command can pin the MPI shepherd process (which is a waste of a CPU) and then putting the remaining working MPI rank on one of the CPUs that already had some other rank running on it. Instead of `taskset`, SGI recommends using the `dplace`(1) (see "`dplace` Command" on page 46) or the environment variable `MPI_DSM_CPULIST`. The following example assumes a job running on eight CPUs. For example:

```
# mpirun -np 8 dplace -s1 -c10,11,16-21 myMPIapplication ...
```

To set `MPI_DSM_CPULIST` variable, perform a command similar to the following:

```
setenv MPI_DSM_CPULIST 10,11,16-21 mpirun -np 8 myMPIapplication ...
```

If they are using a batch scheduler that creates and destroys cpusets dynamically, you should use `MPI_DSM_DISTRIBUTE` environment variable instead of either `MPI_DSM_CPULIST` environment variable or the `dplace` command.

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

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

```
# taskset 0x2 executable name
```

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

```
# taskset -p 0x1 14057
```

To run an MPI Abaqus/Std job on a UV system with eight CPUs, perform the following:

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

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

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

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


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

# `dplace` Command

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

## Using the `dplace` Command

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

```
dplace [-e] [-c cpu_numbers] [-s skip_count] [-n process_name] \
             [-x skip_mask] [-r [l|b|t]] [-o log_file] [-v 1|2] \
             command [command-args]
dplace [-p placement_file] [-o log_file] command [command-args]
dplace [-q] [-qq] [-qqq]
```

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

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

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

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

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

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

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

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

The dplace(1) command accepts the following options:

- -c *cpu_numbers*: Specified as a list of cpus, optionally strided cpu ranges, or a striding pattern. Example: "-c 1", "-c 2-4", "-c 1,4-8,3", "-c 2-8:3", "-c CS", "-c BT". The specification "-c 2-4" is equivalent to "-c 2,3,4" and "-c 2-8:3" is equivalent to 2,5,8. Ranges may also be specified in reverse order: "-c 12-8" is equivalent to 12,11,10,9,8. CPU numbers are **NOT** physical cpu numbers. They are logical cpu number that are relative to the cpus that are in the set of allowed cpus as specified by the current cpuset. A cpu value of "`x`" (or "`*`"), in the argument list for `-c` option, indicates that binding should not be done for that process."x" should be used only if the `-e` option is also used. Cpu numbers start at 0. For striding patterns any subset of the characters (B)lade, (S)ocket, (C)ore, (T)hread may be used and their ordering specifies the nesting of the iteration. For example "SC" means to iterate all the cores in a socket before moving to the next CPU socket, while "CB" means to pin to the first core of each blade, then the second core of every blade, etc. For best results, use the `-e` option when using stride patterns. If the `-c` option is not specified, all cpus of the current cpuset are available. The command itself (which is `exec`'d by `dplace`) is the first process to be placed by the `-c` *cpu_numbers*.

- `-e`: Exact placement. As processes are created, they are bound to cpus in the exact order that the cpus are specified in the cpu list. Cpu numbers may appear multiple times in the list. A cpu value of "`x`" indicates that binding should not be done for that process. If the end of the list is reached, binding starts over at the beginning of the list.

- `-o`: Write a trace file to Write a trace file to <*log file*> that decribes the placement actions that were made for each `fork`, `exec`, and so on. Each line contains a timestamp, process id:thread number, cpu that task was executing on, taskname | placement action. Works with version 2 only.

- `-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.

- `-r`: Specifies that text should be replicated on the node or nodes where the application is running. In some cases, replication will improve performance by reducing the need to make offnode memory references for code. The replication option applies to all programs placed by the dplace command. See the `dplace`(5) man page for additional information on text replication. The replication options are a string of one or more of the following characters:

| | |
|---|---|
| `l` | Replicate library text |
| `b` | Replicate binary (a.out) text |

| | |
|---|---|
| `t` | Thread round-robin option |

- `-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 should be placed using the `-x` option with a `skip_mask` of `2` (`-x2`). This could change in future versions of OpenMP. For applications compiled using the Native Posix Thread Library (NPTL), use the `-x2` option.

- `-v`: Provides the ability to run in version 1 or version 2 compatibility mode if the kernel support is available. If not specified, version 2 compatibility is selected. See COMPATIBILITY section of the `dplace`(1) man page for more details. Note: version 1 requires kernel support for PAGG.
- `-p` *placement_file*: Specifies a placement file that contains additional directives that are used to control process placement.
- `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. If specified twice, lists the current `dplace` jobs that are running. If specified three times, lists the current `dplace` jobs and the tasks that are in each job.

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

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

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

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

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

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

```
%efc -o prog -openmp -O3 program.f
%setenv OMP_NUM_THREADS 4
%dplace -x6 -c4-7 ./prog
```

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

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

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

**Example 5-3** Using the dplace command with Linux commands

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

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

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

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

#  mpirun is placed on cpu 0 in this example #  the root mpiapp is
placed on cpu 1 in this example

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


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


exit
```

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

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

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

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

Placed as specified:

```
>> oncpus e00002343c528000 e000013817540000 e000013473aa8000
>> e000013817c68000 e0
000234704f0000 e000023466ed8000 e00002384cce0000 e00002342c448000
task: 0xe00002343c528000  mpiapp cpus_allowed: 4
task: 0xe000013817540000  mpiapp cpus_allowed: 5
task: 0xe000013473aa8000  mpiapp cpus_allowed: 6
task: 0xe000013817c68000  mpiapp cpus_allowed: 7
task: 0xe0000234704f0000  mpiapp cpus_allowed: 8
task: 0xe000023466ed8000  mpiapp cpus_allowed: 9
task: 0xe00002384cce0000  mpiapp cpus_allowed: 10
task: 0xe00002342c448000  mpiapp cpus_allowed: 11
```

## dplace for Compute Thread Placement Troubleshooting Case Study

This section describes common reasons why compute threads do not end up on unique processors when using commands such a dplace(1) or profile.pl (see "Profiling with PerfSuite" on page 17).

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

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

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

             cpu01   90.1%    0.0%    0.0%   0.0%     9.7%    0.0%    0.0%
             cpu02   99.9%    0.0%    0.0%   0.0%     0.0%    0.0%    0.0%

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
7655 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   5 mocassin
7658 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   8 mocassin
7659 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08   9 mocassin
7660 ccao      25   0  115G 586M  114G R    99.8  0.9   0:08  10 mocassin
7662 ccao      25   0  115G 586M  114G R    99.7  0.9   0:08  12 mocassin
7657 ccao      25   0  115G 586M  114G R    88.5  0.9   0:07   7 mocassin
7661 ccao      25   0  115G 586M  114G R    88.3  0.9   0:07  11 mocassin
7649 ccao      25   0  115G 586M  114G R    55.2  0.9   0:04   2 mocassin
7651 ccao      25   0  115G 586M  114G R    54.1  0.9   0:03   1 mocassin
7650 ccao      25   0  115G 586M  114G R    50.0  0.9   0:04   0 mocassin
7647 ccao      25   0  115G 586M  114G R    49.8  0.9   0:03   0 mocassin
7652 ccao      25   0  115G 586M  114G R    44.7  0.9   0:04   2 mocassin
7648 ccao      25   0  115G 586M  114G R    35.9  0.9   0:03   1 mocassin
```

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

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

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

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

# `dlook` Command

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

## Using the dlook Command

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

```
dlook [-a] [-p] [-h] [-l] [-n] [-o outfile] [-s secs] command [command-args]
dlook [-a] [-p] [-h] [-l] [-n] [-o outfile] [-s secs] pid
```

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

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

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

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

- `-a`: Shows the physical addresses of each page in the address space.
- `-h`: Explicitly lists holes in the address space.
- `-l`: Shows libraries.
- `-p`: Show raw hardware page table entries.
- `-o`: Outputs to file name (*outfile*). If not specified, output is written to stdout.
- `-s`: Specifies a sample interval in seconds. Information about the process is displayed every second (`secs`) of CPU usage by the process.

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

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

```
dlook 4702

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

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

Process memory map:
  2000000000000000-2000000000030000 r-xp 0000000000000000 04:03 4479 /lib/ld-2.2.4.so
        [2000000000000000-200000000002c000]        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. A page is 16, 384 bytes.

The node numbers reported by the dlook command correspond to Socket numbers reported by the cpumap(1) command under the section Processor Numbering on Socket(s). For more information, see the cpumap(1) command description in "Determining System Configuration" on page 11.

*Dirty memory* means that the memory has been modified by a user.

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

**dlook** ***date***

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

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


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

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

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

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

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

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

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

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

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

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


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

  2000000000134000-2000000000140000 rw-p 0000000000000000 00:00 0

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

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

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

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

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

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

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

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

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

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

Contents of dlook.out:

Exit:  ft.C.8
Pid: 2306       Fri Aug 30 14:33:37 2002


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

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

Exit:  ft.C.8
Pid: 2310       Fri Aug 30 14:33:37 2002


Process memory map:
  2000000000030000-200000000003c000 rw-p 0000000000000000 00:00 0
        [2000000000030000-2000000000034000]            1 page  on node  25  MEMORY|DIRTY
        [2000000000034000-200000000003c000]            2 pages on node  12  MEMORY|DIRTY|SHARED

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

                                           ...

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

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

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


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

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

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

## omplace Command

The omplace(1) command is a tool for controlling the placement of MPI processes and OpenMP threads.

The omplace command causes the successive threads in a hybrid MPI/OpenMP job to be placed on unique CPUs. The CPUs are assigned in order from the effective CPU list within the containing cpuset.

This command is a wrapper script for dplace(1) that can be used with MPI, OpenMP, pthreads, and hybrid MPI/OpenMP and MPI/pthreads codes. It generates the proper `dplace` placement file syntax automatically. It also supports some unique options like block-strided CPU lists.

The CPU placement is performed by dynamically generating a placement file and invoking `dplace` with the MPI job launch. For example, To run two MPI processes with four threads per process, and to display the generated placement file run a command similar to the following:

```
# mpirun -np 2 omplace -nt 4 -vv ./a.out
```

The threads would be place as follows:

```
rank 0 thread 0 on CPU 0
rank 0 thread 1 on CPU 1
rank 0 thread 2 on CPU 2
rank 0 thread 3 on CPU 3
rank 1 thread 0 on CPU 4
rank 1 thread 1 on CPU 5
rank 1 thread 2 on CPU 6
rank 1 thread 3 on CPU 7
```

For more information, see the `omplace`(1) man page and “Run-Time Tuning” chapter in the *Message Passing Toolkit (MPT) User’s Guide*.

## `numactl` Command

The `numactl`(8) command runs processes with a specific NUMA scheduling or memory placement policy. The policy is set for command and inherited by all of its children. In addition it can set persistent policy for shared memory segments or files. For more information, see the `numactl`(8) man page.

## Installing NUMA Tools

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

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

   ```
   chkconfig --add numatools
   ```

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

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

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

The following steps are required to disable numatools:

1. To turn off numatools, enter the following:

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

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

   ```
   chkconfig --del numatools
   ```

## An Overview of the Advantages Gained by Using Cpusets

The cpuset facility is primarily a workload manager tool permitting a system administrator to restrict the number of processor and memory resources that a process or set of processes may use. A *cpuset* defines a list of CPUs and memory nodes. A process contained in a cpuset may only execute on the CPUs in that cpuset and may only allocate memory on the memory nodes in that cpuset. Essentially, cpusets provide you with a CPU and memory containers or “soft partitions” within which you can run sets of related tasks. Using cpusets on an SGI UV system improves cache locality and memory access times and can substantially improve an application’s performance and runtime repeatability. Restraining all other jobs from using any of the CPUs or memory resources assigned to a critical job minimizes interference from other jobs on the system. For example, Message Passing Interface (MPI) jobs frequently consist of a number of threads that communicate using message passing interfaces. All threads need to be executing at the same time. If a single thread loses a CPU, all threads stop making forward progress and spin at a barrier.

Cpusets can eliminate the need for a gang scheduler, provide isolation of one such job from other tasks on a system, and facilitate providing equal resources to each thread in a job. This results in both optimum and repeatable performance.

In addition to their traditional use to control the placement of jobs on the CPUs and memory nodes of a system, cpusets also provide a convenient mechanism to control the use of Hyper-Threading Technology.

Cpusets are represented in a hierarchical virtual file system. Cpusets can be nested and they have file-like permissions.

The `sched_setaffinity`, `mbind`, and `set_mempolicy` system calls allow you to specify the CPU and memory placement for individual tasks. On smaller or limited-use systems, these calls may be sufficient.

The kernel cpuset facility provides additional support for system-wide management of CPU and memory resources by related sets of tasks. It provides a hierarchical structure to the resources, with filesystem-like namespace and permissions, and support for guaranteed exclusive use of resources.

You can have a *boot* cpuset running the traditional daemon and server tasks and a second cpuset to hold interactive `telnet`, `rlogin` and/or secure shell (SSH) user sessions called the *user* cpuset.

Creating a user cpuset provides additional isolation between interactive user login sessions and essential system tasks. For example, a user process in the user cpuset consuming excessive CPU or memory resources will not seriously impact essential system services in the boot cpuset.

This section covers the following topics:

- "Linux 2.6 Kernel Support for Cpusets" on page 64
- "Cpuset Facility Capabilities" on page 65
- "Initializing Cpusets" on page 66
- "How to Determine if Cpusets are Installed" on page 66
- "Fine-grained Control within Cpusets" on page 67
- "Cpuset Interaction with Other Placement Mechanism" on page 67
- "Cpusets and Thread Placement" on page 69
- "Safe Job Migration and Cpusets" on page 70

## Linux 2.6 Kernel Support for Cpusets

The Linux 2.6 kernel provides the following support for cpusets:

- Each task has a link to a cpuset structure that specifies the CPUs and memory nodes available for its use.

- Hooks in the `sched_setaffinity` system call, used for CPU placement, and in the `mbind` system call, used for memory placement, ensure that any requested CPU or memory node is available in that task's cpuset.
- All tasks sharing the same placement constraints reference the same cpuset.
- Kernel cpusets are arranged in a hierarchical virtual file system, reflecting the possible nesting of "soft partitions".
- The kernel task scheduler is constrained to only schedule a task on the CPUs in that task's cpuset.
- The kernel memory allocation mechanism is constrained to only allocate physical memory to a task from the memory nodes in that task's cpuset.
- The kernel memory allocation mechanism provides an economical, per-cpuset metric of the aggregate memory pressure of the tasks in a cpuset. *Memory pressure* is defined as the frequency of requests for a free memory page that is not easily satisfied by an available free page.
- The kernel memory allocation mechanism provides an option that allows you to request that memory pages used for file I/O (the kernel page cache) and associated kernel data structures for file inodes and directories be evenly spread across all the memory nodes in a cpuset. Otherwise, they are preferentially allocated on whatever memory node that the task first accessed the memory page.
- You can control the memory migration facility in the kernel using per-cpuset files. When the memory nodes allowed to a task by cpusets changes, any memory pages no longer allowed on that node may be migrated to nodes now allowed. For more information, see "Safe Job Migration and Cpusets" on page 70.

## Cpuset Facility Capabilities

A cpuset constrains the jobs (set of related tasks) running in it to a subset of the system's memory and CPUs. The cpuset facility allows you and your system service software to do the following:

- Create and delete named cpusets.
- Decide which CPUs and memory nodes are available to a cpuset.
- Attach a task to a particular cpuset.

- Identify all tasks sharing the same cpuset.
- Exclude any other cpuset from overlapping a given cpuset, thereby, giving the tasks running in that cpuset exclusive use of those CPUs and memory nodes.
- Perform bulk operations on all tasks associated with a cpuset, such as varying the resources available to that cpuset or hibernating those tasks in temporary favor of some other job.
- Perform sub-partitioning of system resources using hierarchical permissions and resource management.

## Initializing Cpusets

The kernel, at system boot time, initializes one cpuset, the root cpuset, containing the entire system's CPUs and memory nodes. Subsequent user space operations can create additional cpusets.

Mounting the cpuset virtual file system (VFS) at /dev/cpuset exposes the kernel mechanism to user space. This VFS allows for nested resource allocations and the associated hierarchical permission model.

You can initialize and perform other cpuset operations, using any of the these three mechanisms, as follows:

- You can create, change, or query cpusets by using shell commands on /dev/cpuset, such as echo(1), cat(1), mkdir(1), or ls(1).
- You can use the cpuset(1) command line utility to create or destroy cpusets or to retrieve information about existing cpusets and to attach processes to existing cpusets.
- You can use the libcpuset C programming application programming interface (API) functions to query or change them from within your application. You can find information about libcpuset at /usr/share/doc/packages/libcpuset/libcpuset.html.

## How to Determine if Cpusets are Installed

You can issue several commands to determine whether cpusets are installed on your system, as follows:

1. Use the grep(1) command to search the/proc/filesystems for cpusets, as follows:

   ```
   % grep cpuset /proc/filesystems
   nodev   cpuset
   ```

2. Determine if cpuset tasks file is present on your system by changing directory to /dev/cpuset and listing the content of the directory, as follows:

```
% cd /dev/cpuset
Directory: /dev/cpuset

% ls
cpu_exclusive  cpus  mem_exclusive  mems  notify_on_release
pagecache_list  pagecache_local  slabcache_local  tasks
```

3. If the /dev/cpuset/tasks file is not present on your system, it means the cpuset file system is not mounted (usually, it is automatically mounted when the system was booted). As root, you can mount the cpuset file system, as follows:

   ```
   % mount -t cpuset cpuset /dev/cpuset
   ```

## Fine-grained Control within Cpusets

Within a single cpuset, use facilities such as taskset(1), dplace(1), first-touch memory placement, pthreads, sched_setaffinity and mbind to manage processor and memory placement to a more fine-grained level.

The user–level bitmask library supports convenient manipulation of multiword bitmasks useful for CPUs and memory nodes. This bitmask library is required by and designed to work with the cpuset library. You can find information on the bitmask library on your system at /usr/share/doc/packages/libbitmask/libbitmask.html.

## Cpuset Interaction with Other Placement Mechanism

The Linux 2.6 kernel supports additional processor and memory placement mechanisms, as follows:

**Note:** Use the uname(1) command to print out system information to make sure you are running the Linux 2.6.x sn2 kernel, as follows:

```
% uname -r -s
Linux 2.6.27.19-5-default
```

- The sched_setaffinity(2) and sched_getaffinity(2) system calls set and get the CPU affinity mask of a process. This determines the set of CPUs on which the process is eligible to run. The taskset(1) command provides a command line utility for manipulating the CPU affinity mask of a process using these system calls. For more information, see the appropriate man page.
- The set_mempolicy system call sets the NUMA memory policy of the current process to *policy*. A NUMA machine has different memory controllers with different distances to specific CPUs. The memory *policy* defines in which node memory is allocated for the process.

  The get_mempolicy(2) system retrieves the NUMA policy of the calling process or of a memory address, depending on the setting of *flags*. The numactl(8) command provides a command line utility for manipulating the NUMA memory policy of a process using these system calls.
- The mbind(2) system call sets the NUMA memory policy for the pages in a specific range of a task’s virtual address space.

Cpusets are designed to interact cleanly with other placement mechanisms. For example, a batch manager can use cpusets to control the CPU and memory placement of various jobs; while within each job, these other kernel mechanisms are used to manage placement in more detail. It is possible for a batch manager to change a job’s cpuset placement while preserving the internal CPU affinity and NUMA memory placement policy, without requiring any special coding or awareness by the affected job.

Most jobs initialize their placement early in their timeslot, and jobs are rarely migrated until they have been running for a while. As long a batch manager does **not** try to migrate a job at the same time as it is adjusting its own CPU or memory placement, there is little risk of interaction between cpusets and other kernel placement mechanisms.

The CPU and memory node placement constraints imposed by cpusets always override those of these other mechanisms.

Calls to the sched_setaffinity(2) system call automatically mask off CPUs that are not allowed by the affected task's cpuset. If a request results in all the CPUs being masked off, the call fails with errno set to EINVAL. If some of the requested CPUs are allowed by the task's cpuset, the call proceeds as if only the allowed CPUs were requested. The unallowed CPUs are silently ignored. If a task is moved to a different cpuset, or if the CPUs of a cpuset are changed, the CPU affinity of the affected task or tasks is lost. If a batch manager needs to preserve the CPU affinity of the tasks in a job that is being moved, it should use the sched_setaffinity(2) and sched_getaffinity(2) calls to save and restore each affected task's CPU affinity across the move, relative to the cpuset. The cpu_set_t mask data type supported by the C library for use with the CPU affinity calls is different from the libbitmask bitmasks used by libcpuset, so some coding will be required to convert between the two, in order to calculate and preserve cpuset relative CPU affinity.

Similar to CPU affinity, calls to modify a task's NUMA memory policy silently mask off requested memory nodes outside the task's allowed cpuset, and will fail if that results in requested an empty set of memory nodes. Unlike CPU affinity, the NUMA memory policy system calls to not support one task querying or modifying another task's policy. So the kernel automatically handles preserving cpuset relative NUMA memory policy when either a task is attached to a different cpuset, or a cpusets mems value setting is changed. If the old and new mems value sets have the same size, the cpuset relative offset of affected NUMA memory policies is preserved. If the new mems value is smaller, the old mems value relative offsets are folded onto the new mems value, modulo the size of the new mems. If the new mems value is larger, then just the first N nodes are used, where N is the size of the old mems value.

## Cpusets and Thread Placement

If your job uses the placement mechanisms described in "Cpuset Interaction with Other Placement Mechanism" on page 67 and operates under the control of a batch manager, you **cannot** guarantee that a migration will preserve placement done using the mechanisms. These placement mechanisms use system wide numbering of CPUs and memory nodes, not cpuset relative numbering and the job might be migrated without its knowledge while it is trying to adjust its placement. That is, between the point where an application computes the CPU or memory node on which it wants to place a thread and the point where it issues the sched_setaffinity(2), mbind(2) or set_mempolicy(2) call to direct such a placement, the thread might be migrated to a different cpuset, or its cpuset changed to different CPUs or memory nodes, invalidating the CPU or memory node number it just computed.

The libcpuset library provides the following mechanisms to support cpuset relative thread placement that is robust even if the job is being migrated using a batch scheduler.

If your job needs to pin a thread to a single CPU, you can use the convenient cpuset_pin function. This is the most common case.

If your job needs to implement some other variation of placement, such as to specific memory nodes, or to more than one CPU, you can use the following functions to safely guard such code from placement changes caused by job migration, as follows:

- cpuset_get_placement
- cpuset_equal_placement
- cpuset_free_placement

## Safe Job Migration and Cpusets

Jobs that make use of cpuset aware thread pinning described in "Cpusets and Thread Placement" on page 69 can be safely migrated to a different cpuset or have the CPUs or memory nodes of the cpuset safely changed without destroying the per-thread placement done within the job.

**Procedure 5-1** Safe Job Migration Between Cpusets

To safely migrate a job to a different cpuset, perform the following steps:

1. Suspend the tasks in the job by sending their process group a SIGSTOP signal.
2. Use the cpuset_init_pidlist function and related pidlist functions to determine the list of tasks in the job.
3. Use sched_getaffinity(2) to query the CPU affinity of each task in the job.
4. Create a new cpuset, under a temporary name, with the new desired CPU and memory placement.
5. Invoke cpuset_migrate_all function to move the job’s tasks from the old cpuset to the new cpuset.
6. Use cpuset_delete to delete the old cpuset.
7. Use rename(2) on the /dev/cpuset based path of the new temporary cpuset to rename that cpuset to the to the old cpuset name.

8. Convert the results of the previous `sched_getaffinity`(2) calls to the new cpuset placement, preserving cpuset relative offset by using the `cpuset_c_rel_to_sys_cpu` and related functions.

9. Use `sched_setaffinity`(2) to reestablish the per-task CPU binding of each thread in the job.

10. Resume the tasks in the job by sending their process group a `SIGCONT` signal.

The `sched_getaffinity`(2) and `sched_setaffinity`(2) C library calls are limited by C library internals to systems with 1024 CPUs or less. To write code that will work on larger systems, you should use the `syscall`(2) indirect system call wrapper to directly invoke the underlying system call, bypassing the C library API for these calls.

The suspend and resume operation are required in order to keep tasks in the job from changing their per thread CPU placement between steps three and six. The kernel automatically migrates the per-thread memory node placement during step four. This is necessary, because there is no way for one task to modify the NUMA memory placement policy of another task. The kernel does not automatically migrate the per-thread CPU placement, as this can be handled by the user level process doing the migration.

Migrating a job from a larger cpuset (more CPUs or nodes) to a smaller cpuset will lose placement information and subsequently moving that cpuset back to a larger cpuset will **not** recover that information. This loss of CPU affinity can be avoided as described above, using `sched_getaffinity`(2) and `sched_setaffinity`(2) to save and restore the placement (affinity) across such a pair of moves. This loss of NUMA memory placement information cannot be avoided because one task (the one doing the migration) cannot save nor restore the NUMA memory placement policy of another. So if a batch manager wants to migrate jobs without causing them to lose their `mbind`(2) or `set_mempolicy`(2) placement, it should only migrate to cpusets with at least as many memory nodes as the original cpuset.

For detailed information about using cpusets, see The “Cpusets on Linux” chapter in the *Linux Resource Administration Guide* at at http://docs.sgi.com.

## Application Performance on Large SGI UV Systems

This section describes cpuset settings you should pay particular attention to when running applications on large SGI UV 1000 or SGI UV 2000 series systems.

**mem_exclusive**

Flag (0 or 1). If set (1), the cpuset has exclusive use of its memory nodes (no sibling or cousin may overlap). Also if set (1), the cpuset is a hardwall cpuset. See "Hardwall" section in the `cpuset`(7) man page for more information. By default, this is off (0). Newly created cpusets also initially default this to off (0).

**mem_spreadpage**

Flag (0 or 1). If set (1), pages in the kernel page cache (file-system buffers) are uniformly spread across the cpuset. By default, this is off (0) in the top cpuset, and inherited from the parent cpuset in newly created cpusets. See the "Memory Spread" section in the `cpuset`(7) man page for more information.

**mem_hardwall**

Flag (0 or 1). If set (1), the cpuset is a Hardwall cpuset. See "Hardwall" section in the `cpuset`(7) man page for more information. Unlike `mem_exclusive`, there is no constraint on whether cpusets marked `mem_hardwall` may have overlapping memory nodes with sibling or cousin cpusets. By default, this is off (0). Newly created cpusets also initially default this to off (0).

*Chapter 6*

# 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 82

It also describes how to improve the performance of floating-point programs and MPI applications on SGI UV series systems.

- "Floating-point Programs Performance" on page 94
- "MPInside Profiling Tool" on page 95
- "SGI PerfBoost" on page 96
- "Perfcatcher" on page 97
- "Performance Tuning Running MPI on SGI UV Systems" on page 97

Intel provides detailed application tuning information including the Intel Xeon processor 5500 at this location http://developer.intel.com/Assets/PDF/manual/248966.pdf and specific tuning information tutorial for Nehalem (Intel Xeon 5500) at http://software.intel.com/sites/webinar/tuning-your-application-for-nehalem/.

## 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 74.

- Use existing tuned code, such as that found in math libraries and scientific library packages. For details, see "Using Tuned Code" on page 75.
- Determine what needs tuning. For details, see "Determining Tuning Needs" on page 76.
- Use the compiler to do the work. For details, see "Using Compiler Options Where Possible" on page 76.
- Consider tuning cache performance. For details, see "Tuning the Cache Performance" on page 79.
- Set environment variables to enable higher-performance memory management mode. For details, see "Managing Memory" on page 80.

## 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. In the past, using `-g` automatically put down the optimization level. In Intel compiler today, you can use `-O3` with `-g`.
- `-fp-model`: the `-fp-model` option lets you specify the compiler rules for:
  - Value safety
  - Floating-point (FP) expression evaluation
  - FPU environment access
  - Precise FP exceptions
  - FP contractions

Default is `-fp-model fast=1`. Note that `-mp` option is an old flag replaced by `-fp-model`.

- `-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 18 for more details.

## Managing Heap Corruption Problems

You can use environment variables to check for heap corruption problems in programs that use `glibc malloc`/`free` dynamic memory management routines.

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

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.

## 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:

- 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 is an Intel performance monitoring tool. You can run it directly on your SGI UV system. The Linux server/Windows client is useful when you are working on a remote system.
- `psrun` is a `PerfSuite` (see http://perfsuite.ncsa.uiuc.edu/) command-line utility that allows you to take performance measurements of unmodified executables. `psrun` takes as input a configuration XML document that describes the desired measurement.

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 `ifort` or `icc` options, use the `-help` option on the compiler command line. Use the `-dryrun` option to show the driver tool commands that `ifort` or `icc` generate. This option does not actually compile.

Use the following options to help tune performance:

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

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

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

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

- `-O3`: Enables `-O2` optimizations plus more aggressive optimizations, including loop transformation and prefetching. *Loop transformation* are found in a transformation file created by the compiler; you can examine this file to see what suggested changes have been made to loops. *Prefetch instructions* allow data to be moved into the cache before their use. A prefetch instruction is similar to a load instruction.

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

- `-opt_report`: Generates an optimization report and places it in the file specified in `-opt_report_file`.

- `-override_limits`: This is an undocumented option that sometimes allows the compiler to continue optimizing when it has hit an internal limit.

- `-prof_gen` and `-prof_use`: Generates and uses profiling information. These options require a three-step compilation process:

  1. Compile with proper instrumentation using `-prof_gen`.
  2. Run the program on one or more training datasets.
  3. Compile with `-prof_use`, which uses the profile information from the training run.

- `-S`: Compiles and generates an assembly listing in the `.s` files and does not link. The assembly listing can be used in conjunction with the output generated by the `-opt_report` option to try to determine how well the compiler is optimizing loops.

- `-vec-report`: For information specific to the vectorizer. Intel Xeon 7500 series processors can perform short vector operations which provides a powerful performance boost.

- `-fast`: equivalent to writing: `-ipo -O3 -no-prec-div -static -xHos`

- `-xHost`: Can generate instructions for the highest instruction set and processor available on the compilation host.
- Specific processor architecture to compile for: `-xSSE4.2` for Nehalem EP/EX, for example. Useful if compiling in a different system than an SGI UV.
- `-xSSE4.2`: Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family.

  Another important feature of new Intel compilers is the Source Checker, which is enabled using the flag `-diag -enable` + options. The source checker is a compiler feature that provides advanced diagnostics based on detailed analysis of source code. It performs static global analysis to find errors in software that go undetected by the compiler itself. It is a general source code analysis tool that provides an additional diagnostic capability to help you debug your programs. You can use source code analysis options to detect potential errors in your compiled code.

Specific processor architecture to compile for: `-xSSE4.2` for Nehalem EP/EX Useful if compiling in a different system than an SGI UV.

`-xSSE4.2`: Can generate Intel® SSE4 Efficient Accelerated String and Text Processing instructions supported by Intel® Core i7 processors. Can generate Intel® SSE4 Vectorizing Compiler and Media Accelerator, Intel® SSSE3, SSE3, SSE2, and SSE instructions and it can optimize for the Intel® Core™ processor family. Another important feature of new Intel compilers is the Source Checker, which is enabled using the flag `-diag-enable` + options The source checker is a compiler feature that provides advanced diagnostics based on detailed analysis of source code. It performs static global analysis to find errors in software that go undetected by the compiler itself. general source code analysis tool that provides an additional diagnostic capability to help you debug your programs. You can use source code analysis options to detect potential errors in your compiled code including the following:

- Incorrect usage of OpenMP directives
- Inconsistent object declarations in different program units
- Boundary violations
- Uninitialized memory
- Memory corruptions

- Memory Leaks
- Incorrect usage of pointers and allocatable arrays
- Dead code and redundant executions
- Typographical errors or uninitialized variables
- Dangerous usage of unchecked input

Source checker analysis performs a general overview check of a program for all possible values simultaneously. This is in contrast to run-time checking tools that execute a program with a fixed set of values for input variables; such checking tools cannot easily check all edge effects. By not using a fixed set of input values, the source checker analysis can check for all possible corner cases. In fact, you do not need to run the program for Source Checker, the analysis is performed at compilation time. Only requirement is a successful compilation. Important caveat: Limitations of Source Checker Analysis: Since the source checker does not perform full interpretation of analyzed programs, it can generate so called false-positive messages. This is a fundamental difference between the compiler and source checker generated errors; in the case of the source checker, you decide whether the generated error is legitimate and needs to be fixed.

## Tuning the Cache Performance

The processor cache stores recently-used information in a place where it can be accessed extremely fast. For more information, see "Cache Terminology" on page 81.

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

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

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

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

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

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

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

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

- Cache bank conflicts can occur if there are two accesses to the same 16-byte-wide bank at the same time.

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

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

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

## Managing Memory

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

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

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

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

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

### Memory Use Strategies

This section describes some general memory use strategies, as follows:

- Register reuse: do a lot of work on the same data before working on new data
- Cache reuse: the program is much more efficient if all of the data and instructions fit in cache; if not, try to use what is in cache a lot before using anything that is not in cache.
- Data locality: try to access data that is near each other in memory before data that is far.
- I/O efficiency: do a bunch of I/O all at once rather than a little bit at a time; do not mix calculations and I/O.

### Cache Terminology

*Cache line* is the minimum unit of transfer from next-higher cache into this one. *Cache hit* is reference to a cache line which is present in the cache. *Cache miss* is reference to a cache line which is not present in this cache level and must be retrieved from a higher cache (or memory or swap space). *Hit time* is the time to access the upper level of the memory hierarchy, which includes the time needed to determine whether the access is a hit or a miss. *Miss penalty* is the time to replace a block in the upper level with the corresponding block from the lower level, plus the time to deliver this block to the processor. The time to access the next level in the hierarchy is the major component of the miss penalty.

### Memory Hierarchy Latencies

Programmers tend to think of memory as a flat, random access storage device. It is critical to understand that memory is a hierarchy to get good performance. Memory latency differs within the hierarchy. Performance is affected by where the data resides. Registers: 0 cycles latency (cycle = 1/freq) L1 cache: 1 cycle L2 cache: 5-6 cycles L3 cache: 12-17 cycles Main memory: 130-1000+ cycles. CPUs which are waiting for memory are not doing useful work. Software should be "hierarchy-aware" to achieve best performance:

- Perform as many operations as possible on data in registers
- Perform as many operations as possible on data in the cache(s)
- Keep data uses spatially and temporally local
- Consider temporal locality and spatial locality

  Memory hierarchies take advantage of temporal locality by keeping more recently accessed data items closer to the processor. Memory hierarchies take advantage of spatial locality by moving contiguous words in memory to upper levels of the hierarchy.

## 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 73.

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 83.
- Choose the parallelization methodology for your code. For details, see "Parallelizing Your Code" on page 84.

- 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. False sharing refers to OpenMP, not MPI. For details, see "Fixing False Sharing" on page 87.
- Tune for data placement. For details, see "Using `dplace` and `taskset`" on page 88.
- Use environment variables to assist with tuning. For details, see "Environment Variables for Performance Tuning" on page 88.

## 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 synchronization 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
- **Explicit decomposition advantages**: The programmer has full control over insertion of communication and synchronization calls; the source code is portable to other systems; code performance can be better than implicitly parallelized codes.
- **Explicit decomposition disadvantages**: Harder to program; the source code is harder to read and the code is longer (typically 40% more).

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

## Parallelizing Your Code

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

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

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

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

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

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

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

### Use MPT

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

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

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

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

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

### Use OpenMP

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

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

- `ifort -openmp` or `icc -openmp`: These options use the OpenMP front-end that is built into the Intel compilers. The latest Intel compiler OpenMP runtime name is `libiomp5.so`. The latest Intel compiler also supports the GNU OpenMP OpenMP library as an either/or option (not to be mixed-and-matched with the Intel version).

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

### OpenMP Nested Parallelism

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

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

```
% cat place_nested
firsttask cpu=0
thread name=a.out oncpu=0 cpu=4 noplace=1 exact onetime thread name=a.out oncpu=0
cpu=1-3 exact thread name=a.out oncpu=4 cpu=5-7 exact
```

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

### Use Compiler Options

Use the compiler to invoke automatic parallelization. Use the `-parallel` and `-par_report` option to the `ifort` or `icc` 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 with data dependencies because of temporary variables, reductions, nested loops, or function calls or subroutines.

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

## Fixing False Sharing

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

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

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

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

- Use the hardware counter to run a profile that monitors storage to shared cache lines. This will show the location of the problem.
- Revise data structures or algorithms.
- Check shared data, static variables, common blocks, and private and public variables in shared objects.
- Use critical regions to identify the part of the code that has the problem.

## Using dplace and taskset

The dplace command binds processes to specified CPUs in a round-robin fashion. Once bound to a process, they do not migrate. dplace numbering is done in the context of the current CPU memory set. See Chapter 4, "Monitoring Tools" on page 25 for details about dplace.

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

## Environment Variables for Performance Tuning

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

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

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

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

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

- If set to serial, single-processor execution is used.
- If set to throughput, CPUs yield to other processes when waiting for work. This is the default and is intended to provide good overall system performance in a multiuser environment.
- If set to turnaround, worker threads do not yield while waiting for work. Setting KMP_LIBRARY to turnaround may improve the performance of benchmarks run on dedicated systems, where multiple users are not contending for CPU resources.

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

## Understanding Parallel Speedup and Amdahl's Law

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

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

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

This section covers the following topics:

- "Adding CPUs to Shorten Execution Time" on page 90
- "Understanding Parallel Speedup" on page 90
- "Understanding Amdahl's Law" on page 91
- "Calculating the Parallel Fraction of a Program" on page 92
- "Predicting Execution Time with n CPUs" on page 93

## 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 UV architectures this means:

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

## Understanding Parallel Speedup

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

**Example 6-1** Typical C Loop

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

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

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

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

### Understanding Superlinear Speedup

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

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

## Understanding Amdahl’s Law

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

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

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

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

```
                    1
Speedup(n) = -----------
              (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.

There is a slightly more sophisticated version of Amdahl's law which includes communication overhead, showing also that if the program has no serial part that as we increase the number of cores the amount of computation per core diminishes and the communication overhead (unless there is not communication and we have trivial parallelization) increases, also diminishing the efficiency of the code and the speedup. The equation is: $Speedup(n) = n / (1+ a*(n-1) + n*(tc / ts))$ Where: $n$: number of processes $a$: the fraction of the given task not dividable into concurrent subtasks $ts$: time to execute the task in a single processor $tc$: communication overhead If $a$=0 and $tc$=0 (no serial part and no communications) like in a trivial parallelization program, you will get linear speedup.

## Calculating the Parallel Fraction of a Program

You do not have to guess at the value of $p$ for a given program. Measure the execution times T(1) and T(2) to calculate a measured *Speedup*(2) = T(1) / T(2). The

Amdahl's law equation can be rearranged to yield *p* when *Speedup* (2) is known, as in Example 6-3.

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

```
     2    SpeedUp(2) - 1
p = --- * --------------
     1      SpeedUp(2)
```

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

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

In some cases, the *Speedup*(2) = T(1)/T(2) is a value greater than 2; in other words, a superlinear speedup ("Understanding Superlinear Speedup" on page 91). When this occurs, the formula in Example 6-3 returns a value of *p* greater than 1.0, which is clearly not useful. In this case you need to calculate *p* from two other more realistic timings, for example T(2) and T(3). The general formula for *p* is shown in Example 6-4, where *n* and *m* are the two CPU counts whose speedups are known, *n*>*m*.

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

```
                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.

## Gustafson's Law

Gustafson's law proposes that programmers set the size of problems to use the available equipment to solve problems within a practical fixed time. Therefore, if faster (more parallel) equipment is available, larger problems can be solved in the same time. Amdahl's law is based on fixed workload or fixed problem size. It implies that the sequential part of a program does not change with respect to machine size (for example, the number of processors). However, the parallel part is evenly distributed by `n` processors. The impact of Gustafson's law was to shift research goals to select or reformulate problems so that solving a larger problem in the same amount of time would be possible. In particular, the law redefines efficiency as a need to minimize the sequential part of a program, even if it increases the total amount of computation. The bottom line is that by running larger problems, it is hoped that the bulk of the calculation will increase faster than the serial part of the program, allowing for better scaling. There is a slightly more sophisticated version of Amdahl's law which includes communication overhead, showing also that if the program has no serial part that as we increase the number of cores the amount of computation per core diminishes and the communication overhead (unless there is not communication and you have trivial parallelization) increases, also diminishing the efficiency of the code and the speedup. The equation is:

```
Speedup(n) = n/(1+ a*(n-1) + n*(tc/ts))
```

Where: `n`: number of processes `a`: the fraction of the given task not dividable into concurrent subtasks `ts`: time to execute the task in a single processor `tc`: communication overhead If `a`=0 and `tc`=0 (no serial part and no communications) like in a trivial parallelization program, you will get linear speedup.

## Floating-point Programs Performance

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

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

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

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

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

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

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

## MPInside Profiling Tool

`MPInside`(3) is an MPI profiling tool which provides valuable information on optimizing your MPI application. It helps you determine where the MPI Send/Receive pairs are not executed synchronously. With non-synchronized Send/Receive, the MPI communications can be very slow, independent of the power of the underlying MPI library/hardware engine. For most MPI applications, the MPI communication times are more accountable to the lack of synchronizations of these Send/Receive pair than to the MPI/hardware engine. MPInside, among other valuable functions, measures this non-synchronized time for all the MPI ranks involved in the application, for all the MPI function activated. It allows you to tell at what actual speed the MPI engine did such communications, for example, the ratio Bytes received / (time of the MPI function minus the synchronization time). It provides this latter information, accumulated per CPU a well as in a CPUxCPU matrix. In addition, MPInside automatically and precisely reports the timing described above on a branch basis. A branch is an MPI function with all its ancestors in the calling sequence. MPInside provides the routine name and the source file line number for all the routines defining a branch. All branches are put in relation with the other CPU branches that had a Send/receive partnership with them. For any CPU, any Received branch performed by that CPU has partners. A partner set is described by four numbers: the Sending rank number, the Sending CPU branch identification, the percentage of time accounted to this partnership in regard to the total execution wait time of this Received branch and the percentage of time this last was to account to lack of synchronization. Even if the MPI/hardware engine performance may also be accountable to the non synchronized communications, most of it is accountable to the application itself and therefore is the developer's responsibility. MPInside tells you where and how much such non synchronized communication occurred in the application.

To load the MPInside module into your environment (for any shell), use the following command:

```
% module load MPInside/3.1
```

For details about using software modules, see the `module`(1) man page.

For more information on using MPInside, see the following documents located in the MPInside software module:

- `mpinside_3.1_ref_manual.pdf`

  *MPInside 3.1 Reference Manual* describes how to use the MPI profiling tool.

- `MPInside_window_how_to.pdf`

  *HOW TO select a window of observation with MPInside* slide set

- `MPInside.3`

  The `MPInside`(3) man page describes how to use MPInside

You can also view an updated version of the *MPInside Reference Guide* at the SGI Technical Publications Library at http://docs.sgi.com.

## SGI PerfBoost

SGI PerfBoost uses a wrapper library to run applications compiled against other MPI implementations under the SGI Message Passing Toolkit (MPT) product on SGI platforms. The PerfBoost software allows you to run SGI MPT which is a version of MPI optimized for SGI large, shared-memory systems and can take advantage of the UV Hub. For more information, see "Performance Tuning Running MPI on SGI UV Systems" on page 97 and Chapter 6, “PerfBoost” in the *Message Passing Toolkit (MPT) User's Guide* available on the Tech Pubs Library at http://docs.sgi.com.

## Perfcatcher

The simple-to-use Perfcatcher tool uses a wrapper library to return MPI and SHMEM function profiling information. Some analysis is done, and information like percent CPU time, total time spent per function, message size, and load imbalance are reported. See the `perfcatch`(1) man page and Chapter 8, “MPI Performance Profiling” in the *Message Passing Toolkit (MPT) User’s Guide* available on the Tech Pubs Library at http://docs.sgi.com.

## Performance Tuning Running MPI on SGI UV Systems

The SGI UV 100, SGI UV 1000 series systems and SGI UV 2000 series systems are scalable nonuniform memory access (NUMA) systems that support a single Linux image of thousands of processors distributed over many sockets and SGI UV Hub application-specific integrated circuits (ASICs). The UV Hub is the heart of the SGI UV system compute blade. Each "processor" is a hyperthread on a particular core within a particular socket.

### UV 100 and UV 1000 Hub ASIC

For SGI UV 100 and UV 1000 systems, each SGI UV Hub normally connects to two sockets. All communication between the sockets and the UV Hub uses Intel QuickPath Interconnect (QPI) channels. The UV Hub has NUMAlink 5 ports that connect with the NUMAlink 5 interconnect fabric. The UV Hub acts as a crossbar between the processors, local SDRAM memory, and the network interface. The Hub ASIC enables any processor in the single-system image (SSI) to access the memory of all processors in the SSI. For more information on the SGI UV hub, SGI UV compute blades, QPI, and NUMAlink 5, see the *SGI Altix UV 1000 System User’s Guide* or the *SGI Altix UV 100 System User’s Guide*, respectively.

### UV 2000 Hub ASIC

For SGI UV 2000 systems, there is one SGI system architecture ASIC based board assembly per compute blade. The SGI system ASIC has two hubs. Each hub supports supports one 8.0 GT/s QPI channel to a processor socket. All communication between the sockets and the UV Hub uses Intel QuickPath Interconnect (QPI) channels. The UV Hub has NUMAlink 6 ports that connect with the NUMAlink 6 interconnect fabric. The UV Hub acts as a crossbar between the processors, local SDRAM memory, and the network interface. The Hub ASIC enables any processor in the single-system image (SSI) to access the memory of all processors in the SSI. For more information on the SGI UV hub, SGI UV compute blades, QPI, and NUMAlink 6, see the *SGI UV 2000 System User Guide*.

## MPI Communication between Processes

When MPI communicates between processes, two transfer methods are possible on an SGI UV system:

- By use of shared memory
- By use of the global reference unit (GRU), part of the SGI UV Hub ASIC

MPI chooses the method depending on internal heuristics, the type of MPI communication that is involved, and some user-tunable variables. When using the GRU to transfer data and messages, the MPI library uses the GRU resources it allocates via the GRU resource allocator, which divides up the available GRU resources. It fairly allocates buffer space and control blocks between the logical processors being used by the MPI job.

## General Considerations

Running MPI jobs optimally on SGI UV systems is not very difficult. It is best to pin MPI processes to CPUs and isolate multiple MPI jobs onto different sets of sockets and Hubs, and this is usually achieved by configuring a batch scheduler to create a cpuset for every MPI job. MPI pins its processes to the sequential list of logical processors within the containing cpuset by default, but you can control and alter the pinning pattern using `MPI_DSM_CPULIST`. See the `MPI_DSM_CPULIST` discussion in the *Message Passing Toolkit (MPT) User's Guide*, and the `omplace`(1) and `dplace`(1) man pages.

## Job Performance Types

The MPI library chooses buffer sizes and communication algorithms in an attempt to deliver the best performance automatically to a wide variety of MPI applications. However, applications have different performance profiles and bottlenecks, and so user tuning may be of help in improving performance. Here are some application performance types and ways that MPI performance may be improved for them:

- Odd HyperThreads are idle.

Most high performance computing MPI programs run best using only one HyperThread per core. When an SGI UV system has multiple HyperThreads per core, logical CPUs are numbered such that odd HyperThreads are the high half of the logical CPU numbers. Therefore, the task of scheduling only on the even HyperThreads may be accomplished by scheduling MPI jobs as if only half the full number exist, leaving the high logical CPUs idle. You can use the `cpumap`(1) command to determine if cores have multiple HyperThreads on your SGI UV system. The output tells the number of physical and logical processors and if **Hyperthreading** is **ON** or **OFF** and how shared processors are paired (towards the bottom of the command's output).

If an MPI job uses only half of the available logical CPUs, set `GRU_RESOURCE_FACTOR` to 2 so that the MPI processes can utilize all the available GRU resources on a Hub rather than reserving some of them for the idle HyperThreads. For more information about GRU resource tuning, see the `gru_resource`(3) man page.

- MPI large message bandwidth is important.

  MPI can improve the bandwidth of large messages if `MPI_GRU_CBS` is set to 0. This favors large message bandwidth at the cost of suppressing asynchronous MPI message delivery. In addition, some programs transfer large messages via the `MPI_Send` function. To switch on the use of unbuffered, single copy transport in these cases you can set `MPI_BUFFER_MAX` to 0. See the `MPI`(1) man page for more details.

- MPI small or near messages are very frequent.

  For small fabric hop counts, shared memory message delivery is faster than GRU messages. To deliver all messages within an SGI UV host via shared memory, set `MPI_SHARED_NEIGHBORHOOD` to "`host`". See the `MPI`(1) man page for more details.

## Other ccNUMA Performance Issues

MPI application processes normally perform best if their local memory is allocated on the socket assigned to execute it. This cannot happen if memory on that socket is exhausted by the application or by other system consumption, for example, file buffer cache. Use the `nodeinfo`(1) command to view memory consumption on the nodes assigned to your job and use `bcfree`(1) to clear out excessive file buffer cache. PBS Professional batch scheduler installations can be configured to issue bcfreecommands in the job prologue. For more information, see PBS Professional documentation and the bcfree(1) man page.

For detailed information on MPI application tuning, see the *Message Passing Toolkit (MPT) User's Guide*.

*Chapter 7*

# Flexible File I/O

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

- "FFIO Operation" on page 101
- "Environment Variables" on page 102
- "Simple Examples" on page 103
- "Multithreading Considerations" on page 106
- "Application Examples " on page 107
- "Event Tracing " on page 108
- "System Information and Issues " on page 108

## FFIO Operation

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

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

## Environment Variables

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

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

For SGI systems, perform the following:

```
# export LD_PRELOAD="/usr/lib64/libFFIO.so"
```

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

```
unset LD_PRELOAD
```

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

```
# export FF_IO_OPTS  ’<string>(eie.direct.mbytes:<size>:<num>:<lead>:<share>:<stride>:0)’
```

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

`<string>` Matches the names of files that can use the buffer cache.

`<size>` Number of 4k blocks in each page of the I/O buffer cache.

`<num>` Number of pages in the I/O buffer cache.

`<lead>` The maximum number of "read ahead" pages.

`<share>` A value of 1 means a shared cache, 0 means private

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

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

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

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

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

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

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

```
setenv FF_IO_OPTS ’output* test*(eie.direct.mbytes:4096:128:6:1:1:0) special42(eie.direct.mbytes:8192:256:6:0:1:0)’
```

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

## Simple Examples

This section walks you through some simple examples using FFIO.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The last two lines are for write and read operations, respectively. Only for very simple I/O patterns, the difference between (parent –> eie) and (eie –> child) read statistics can be explained by the number of read aheads. For random reads of a large file over a long period of time, this is not the case. All write operations count as async.

## Multithreading Considerations

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

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

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

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

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

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

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

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

## Application Examples

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

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

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

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

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

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

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

## Event Tracing

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

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

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

## System Information and Issues

Applications written in C, C++, and Fortran are supported. C and C++ applications can be built with either the Intel or gcc compiler. Only Fortran codes built with the Intel compiler will work.

The following restrictions on FFIO must also be observed:

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

# I/O Tuning

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

## Application Placement and I/O Resources

It is useful to place an application on the same node as its I/O resource. For graphics applications, for example, this can improve performance up to 30 percent.

For example, for an SGI UV system with the following devices:

```
# gfxtopology

Serial number: UV-00000021
Partition number: 0
8 Blades
248 CPUs
283.70 Gb Memory Total
5 I/O Risers

Blade Location    NASID  PCI Address    X Server Display   Device
----------------------------------------------------------------------
    0 r001i01b08      0  0000:05:00.0                  -   Matrox Pilot
    4 r001i01b12      8  0001:02:01.0                  -   SGI Scalable Graphics Capture
    6 r001i01b14     12  0003:07:00.0          Layout0.0   nVidia Quadro FX 5800
                         0003:08:00.0          Layout0.1   nVidia Quadro FX 5800
    7 r001i01b15     14  0004:03:00.0          Layout0.2   nVidia Quadro FX 5800
```

For example, to run an OpenGL graphics program, such as `glxgears`(1), on the third graphics processing unit using `numactl`(8), perform the following command:

```
% numactl -N 14 -m 14 /usr/bin/glxgears -display :0.2
```

This example assumes the X server was started with `:0 == Layout0`.

You could also use the `dplace`(1) command to place the application, see "`dplace` Command" on page 46.

## Layout of Filesystems and XVM for Multiple RAIDs

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

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

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

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

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

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

# Suggested Shortcuts and Workarounds

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

- "Determining Process Placement" on page 111
- "Resetting System Limits" on page 118
- "Linux Shared Memory Accounting" on page 124
- "OFED Tuning Requirements for UPC and SHMEM" on page 125
- "Setting Java Enviroment Variables" on page 126

## 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:

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

   The `pu` command shows current processes.

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

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

   YEAbcDgHIjklMnoTP|qrsuzV{FWX
   2mlt
   EOF
   ```

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

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

Use the following variation to produce output with column headings:

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

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

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

Use the following variation to produce output with column headings:

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

## Example Using pthreads

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

```
% th n
```

```
% th 4
% pu

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

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

LC %CPU   PID USER     PRI  NI  SIZE  RSS SHARE STAT %MEM   TIME COMMAND
 3  0.0 15072 guest1     16   0  3488 1536  3328 S     0.0   0:00 grep
 5  0.0 13785 guest1     15   0  5872 3664  4592 S     0.0   0:00 csh
 5  0.0 15062 guest1     16   0 15824 2080  4384 S     0.0   0:00 th
 5  0.0 15063 guest1     15   0 15824 2080  4384 S     0.0   0:00 th
 5 99.8 15064 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
 7  0.0 13826 guest1     18   0  5824 3552  5632 S     0.0   0:00 csh
10 99.9 15066 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
11 99.9 15067 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
13 99.9 15065 guest1     25   0 15824 2080  4384 R     0.0   0:14 th
15  0.0 13857 guest1     15   0  5840 3584  5648 S     0.0   0:00 csh
15  0.0 15071 guest1     16   0 70048 1600 69840 S     0.0   0:00 ort
15  1.5 15070 guest1     15   0  5056 2832  4288 R     0.0   0:00top
```

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

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

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

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

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

```
15  0.0 15094 guest1     16   0 70048 1600 69840 S     0.0   0:00 sort
15  1.6 15093 guest1     15   0  5056 2832  4288 R     0.0   0:00 top
```

## Example Using OpenMP

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

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

The following output is created:

```
% pu

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

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

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

```
14 98.7 22187 guest1     39   0 26432 2704  4272 R     0.0   0:11 md
```

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

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

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

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

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

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

## Combination Example (MPI and OpenMP)

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

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

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

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

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

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

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

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

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

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

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

   4         2          x,0,1,2,3,x,x,x,x,x,x,x,x,4,5,6,7
   4         3
x,0,1,2,3,x,x,x,x,x,x,x,x,4,5,6,7,8,9,10,11
```

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

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

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

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

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

% pu

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

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

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

LC %CPU   PID USER      PRI  NI  SIZE  RSS SHARE STAT %MEM   TIME COMMAND
 0  0.0 21551 guest1     15   0  5904 3712  4592 S     0.0   0:00 csh
 0  0.0 23394 guest1     15   0  883M 9456  882M S     0.1   0:00 hybrid
 4  0.0 21956 guest1     15   0  5856 3616  5664 S     0.0   0:00 csh
 4  0.0 23412 guest1     16   0 70048 1600 69840 S     0.0   0:00 sort
 4  1.6 23411 guest1     15   0  5056 2832  4288 R     0.0   0:00 top
 5  0.0 23413 guest1     16   0  3488 1536  3328 S     0.0   0:00 grep
 8  0.0 22005 guest1     15   0  5840 3584  5648 S     0.0   0:00 csh
 8  0.0 23404 guest1     15   0  894M  10M  889M S     0.1   0:00 hybrid
 8 99.9 23401 guest1     39   0  894M  10M  889M R     0.1   0:09 hybrid
 9  0.0 23403 guest1     15   0  894M  10M  894M S     0.1   0:00 hybrid
 9 99.9 23402 guest1     25   0  894M  10M  894M R     0.1   0:09 hybrid
10 99.9 23407 guest1     25   0  894M  10M  894M R     0.1   0:09 hybrid
11 99.9 23408 guest1     25   0  894M  10M  889M R     0.1   0:09 hybrid
12  0.0 23391 guest1     15   0  5072 2928  4400 S     0.0   0:00 mpirun
12  0.0 23406 guest1     15   0  894M  10M  894M S     0.1   0:00 hybrid
14  0.0 23405 guest1     15   0  894M  10M  889M S     0.1   0:00 hybrid
```

## Resetting System Limits

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

```
[user@machine user]# cat /etc/security/limits.conf
# /etc/security/limits.conf
#
#Each line describes a limit for a user in the form:
#
#            #
#Where:
# can be:
#        - an user name
#        - a group name, with @group syntax
#        - the wildcard *, for default entry
#
# can have the two values:
```

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

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

# End of file
```

For instructions on how to change these limits, see "Resetting the File Limit Resource Default" on page 119.

## 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 system hardware.

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

Follow this procedure to execute these changes:

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

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

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

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

The following command shows the new limit:

```
ulimit -H -n
```

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

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

The `ulimit -a` command displays all limits, as follows:

```
sys:~ # ulimit -a
core file size          (blocks, -c) 1
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 511876
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) 55709764
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
```

```
cpu time               (seconds, -t) unlimited
max user processes              (-u) 511876
virtual memory          (kbytes, -v) 68057680
file locks                      (-x) unlimited
```

## Resetting the Default Stack Size

Some applications will not run well on an SGI system with a small stack size. To set a higher stack limit, follow the instructions in "Resetting the File Limit Resource Default" on page 119 and add the following lines to the `/etc/security/limits.conf` file:

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

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

Another method that does not require root privilege relies on the fact that many MPI implementation use `ssh`, `rsh`, or some sort of login shell to start the MPI rank processes. If you merely need to bump up the soft limit, you can modify your shell's startup script. For example, if your login shell is `bash` then add something like the following to your `.bashrc` file:

```
% ulimit -s 300000
```

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

For more information on defaul settings, also see "Resetting the File Limit Resource Default" on page 119.

## Avoiding Segmentation Faults

The default stack size in the Linux operating system is 8MB (8192 kbytes). This value need to be increased to avoid "Segmentation Faults" errors.

You can use the `ulimit -a` command to view the stack size, as follows:

```
uv44-sys:~ # ulimit -a
core file size          (blocks, -c) unlimited
```

```
data seg size           (kbytes, -d) unlimited

file size               (blocks, -f) unlimited

pending signals                 (-i) 204800

max locked memory       (kbytes, -l) unlimited

max memory size         (kbytes, -m) unlimited

open files                      (-n) 16384

pipe size            (512 bytes, -p) 8

POSIX message queues     (bytes, -q) 819200

stack size              (kbytes, -s) 8192

cpu time               (seconds, -t) unlimited

max user processes              (-u) 204800

virtual memory          (kbytes, -v) unlimited

file locks                      (-x) unlimited
```

To change the value, perform a command similar to the following:

```
uv44-sys:~ # ulimit -s 300000
```

There is a similar variable for OpenMP programs. If you get a segmentation fault right away while running a program parallelized with OpenMP, a good idea is to increase the KMP_STACKSIZE to a larger size. The default size in Intel Compilers is 4MB.

For example, to increase it to 64MB in csh shell perform the following:

```
setenv KMP_STACKSIZE 64M
```

in bash:

```
export KMP_STACKSIZE=64M
```

## Resetting Virtual Memory Size

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

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

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

```
% limit vmemoryuse 7128960

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

bytes) Killed n
```

The program now succeeds when virtual memory is unlimited:

```
%  limit vmemoryuse unlimited


% mpirun -v -np 4 ./program
MPI: libxmpi.so ’SGI MPI 4.9 MPT 1.14  07/18/06 08:43:15’
MPI: libmpi.so  ’SGI MPI 4.9 MPT 1.14  07/18/06 08:41:05’
MPI: MPI_MSGS_MAX = 524288
MPI: MPI_BUFS_PER_PROC= 32

HELLO WORLD from Processor 0

HELLO WORLD from Processor 2

HELLO WORLD from Processor 1

HELLO WORLD from Processor 3
```

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

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

## Linux Shared Memory Accounting

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

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

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

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

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

The syntax of the memacct call is, as follows:

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

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

ESRCH Process pid was not found.

ENOSYS The function is not implemented. Check if numatools kernel package is up-to-date.

Normally, the following errors should not occur:

ENOENT Can not open /proc/numatools device file.

EPERM No read permission on /proc/numatools device file.

ENOTTY Inappropriate ioctl operation on /proc/numatools device file.

EFAULT Invalid arguments. The ioctl() operation performed by the function failed with invalid arguments.

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

## OFED Tuning Requirements for UPC and SHMEM

The SGI UPC run-time environment depends on the SGI Message Passing Toolkit (MPT) MPI and SHMEM libraries and the job launch, parallel job control, memory mapping, and synchronization functionality they provide. UPC jobs are launched like MPT MPI or SHMEM jobs, using the mpirun(1) or mpiexec_mpt(1) commands. UPC thread numbers correspond to SHMEM PE numbers and MPI rank numbers for MPI_COMM_WORLD (see the Unified Parallel C (UPC) User's Guide at http://docs.sgi.com.

This section describes how to tune the OpenFabrics Enterprise Distribution (OFED) maximum queue pair max_qp variable to be able to run SHMEM and UPC on large node count clusters over the OFED fabric running SUSE Linux Enterprise Server 11 Service Pack 1 (SLES 11 SP1), SLES 11 SP2, or Red Hat Enterprise Server 6.2 (RHEL 6.2).

**SLES11 SP1**

The mlx4_core module provided by the kernel-default package does **not** have the log_num_qp parameter.

The maximum number of queue pairs (QPs) will default to 2^17 (131072).

The mlx4_core provided by the ofed-kmp-default (from the PTF.693243 update only, not from the standard SLES11 SP1 release), **does** have the log_num_qp parameter.

It can be adjusted by making an addition to the /etc/modprobe.conf.local file, as follows:

```
options mlx4_core log_num_qp=21
```

### SLES11 SP2

The `mlx4_core` module provided by the `kernel-default` package **does** support the `log_num_qp` parameter.

The maximum number of QPs will default to 2^17 (131072).

It can be adjusted by making an addition to the `/etc/modprobe.conf.local` file, as follows:

```
options mlx4_core log_num_qp=21
```

### RHEL 6.2

The `mlx4_core` module provided by the kernel package **does** support the `log_num_qp` parameter.

The maximum number of QPs will default to 2^17 (131072).

You can adjust this with an entry in the `/etc/modprobe.conf` file, as follows:

```
options mlx4_core log_num_qp=21
```

## Setting Java Enviroment Variables

When Java software starts, it checks the environment in which it is running and configures itself to fit, assuming that it owns the entire environment. The default for some Java implementations (for example, IBM J9 1.4.2) is to start a garbage collection (GC) thread for every CPU it sees. Other Java implementations use other algorithms to decide the number of GCs to start, but the number will generally be 0.5 to 1 times the number of CPUs. On a 1 or 2 socket system, that works out well enough. This strategy does not scale well to large core-count systems, however.

Java command line options allow you to control the number of GC threads that the Java virtual machine (JVM) will use. In many cases, a single GC thread is sufficient, as set in the examples above. In other cases, a larger number may be appropriate and can be set with the applicable environment variable or command line option. Properly tuning the number of GC threads for an application is an exercise in performance optimization, but a reasonable starting point is to use one GC thread per active worker thread.

**For Sun Java (now Oracle Java):**

```
-XX:ParallelGCThreads
```

**For IBM Java:**

```
-Xgcthreads
```

An example command line option:

```
java -XX:+UseParallelGC -XX:ParallelGCThreads=1
```

The system administrator may choose to implement the option to limit the number of GC threads to a reasonable value with an environment variable set in the global profile, for example the `/etc/profile.local` file so casual Java users can avoid difficulties.

Environment variable settings, are as follows:

For **Sun Java (now Oracle Java):**

```
JAVA_OPTIONS="-XX:ParallelGCThreads=1"
```

**For IBM Java:**

```
IBM_JAVA_OPTIONS="-Xgcthreads1"
```

# Index