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

007–5646–010

New Features

This revision includes the following updates:

- Miscellaneous editorial and technical corrections.

Record of Revision

Version	Description
001	November 2010 Original publication.
002	February 2011 Supports the SGI Performance Suite 1.1 release.
003	November 2011 Supports the SGI Performance Suite 1.3 release.
004	May 2012 Supports the SGI Performance Suite 1.4 release.
005	November 2013 Supports the SGI Performance Suite 1.7 release.
006	November 2013 Supports the SGI Performance Suite 1.7 release and includes a correction to the PerfSocket installation documentation.
007	May 2014 Supports the SGI Performance Suite 1.8 release.
008	May 2015 Supports the SGI Performance Suite 1.10 release.
009	November 2015 Supports the SGI Performance Suite 1.11 release.
010	May 2016 Supports the SGI Performance Suite 1.12 release.

Contents

About This Guide

This publication explains how to tune C and Fortran application programs compiled with an Intel compiler on SGI® UV™ series systems, SGI® ICE™ clusters, and SGI® Rackable™ clusters.

This guide is written for experienced programmers who are familiar with Linux commands and with either C or Fortran programming. 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 SGI Publications

The SGI Foundation Software release notes and the SGI Performance Suite release notes contain information about the specific software packages provided in those products. The release notes also list SGI publications that provide information about the products. The release notes are available in the following locations:

- Online at the SGI customer portal. After you log into the SGI customer portal, you can access the release notes.

  The SGI Foundation Software release notes are posted to the following website:

  https://support1-sgi.custhelp.com/app/answers/detail/a_id/4983

  The SGI Performance Suite release notes are posted to the following website:

  https://support1-sgi.custhelp.com/app/answers/detail/a_id/6093

  Note: You must sign into the SGI customer portal, at https://support.sgi.com/login, in order for the preceding links to work.

- On the product media. The release notes reside in a text file in the `/docs` directory on the product media. For example, `/docs/SGI-MPI-1.`*x*`-readme.txt`.
- On the system. After installation, the release notes and other product documentation reside in the `/usr/share/doc/packages/`*product* directory.

All SGI publications are available on the SGI support portal. The following software publications provide information about Linux implementations on SGI systems:

- *SGI Foundation Software (SFS) User Guide*

  This manual explains how to configure and use SFS, includes information about basic configuration for features such as the hardware event tracker (HET), CPU frequency scaling and partitioning.

- *SGI Cpuset Software Guide*

  Explains how to use cpusets within your application program. Cpusets restrict processes within a program to specific processors or memory nodes.

- *SGI MPI and SGI SHMEM User Guide*

  Describes the industry-standard message passing protocol optimized for SGI computers. This manual describes how to tune the run-time environment to improve the performance of an MPI message passing application on SGI computers. The tuning methods do not involve application code changes.

- *MPInside Reference Guide*

  Documents the SGI MPInside MPI profiling tool.

SGI creates hardware manuals that are specific to each product line. The hardware documentation typically includes a system architecture overview and describes the major components. It also provides the standard procedures for powering on and powering off the system, basic troubleshooting information, and important safety and regulatory specifications.

Related Publications From Other Sources

Compilers and performance tool information for software that runs on SGI Linux systems is available from a variety of sources. The following additional links might be useful to you:

- The GNU hosts *Debugging with GDB* and the *GDB User Manual* at the GNU Project Debugger website. This website is as follows:

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

- Intel provides documentation for all the Intel Parallel Studio XE products at the following website:

https://software.intel.com/en-us/intel-parallel-studio-xe-support/documentation

- Intel provides detailed application tuning information in the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, which is available from the following website:

  http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html?wapkw=

- Information about the Intel Xeon Processor E5 family of processors is at the following website:

  http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e5-family.html

- The Intel Software Network page includes information specific to the Intel VTune Performance Amplifier. This website is as follows:

  https://software.intel.com/en-us/intel-vtune-amplifier-xe

- Information about the OpenMP Standard and the OpenMP API specification for parallel programming can be found at the following website:

  http://openmp.org/wp/

Obtaining Publications

All SGI publications are available on the SGI customer portal at http://support.sgi.com. Select the following:

Support by Product > *productname* > Documentation

If you do not find what you are looking for, search for document-title keywords by selecting **Search Knowledgebase** and using the category **Documentation**.

You can view man pages by typing `man` *title* on a command line.

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.

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 either 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:

  http://www.sgi.com/support/supportcenters.html

SGI values your comments and will respond to them promptly.

The SGI Compiling Environment

This chapter includes the following topics:

About the Compiling Environment

This chapter provides an overview of the SGI compiling environment on the SGI family of servers.

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.

Virtual memory, also known as virtual addressing, divides 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 by a process and the process size can be determined by using the `pmap`(1) command or by examining the contents of the `proc`/*pid*/`status` directory.

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 memory 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). Performance is seriously affected when a system runs out of memory and uses swap-to-disk while running a program. Swapping becomes

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.

Compiler Overview

You can obtain an Intel Fortran compiler or an Intel C/C++ compiler from Intel Corporation or from SGI. For more information, see one of the following links:

- http://software.intel.com/en-us/intel-sdp-home
- https://software.intel.com/en-us/intel-parallel-studio-xe

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

For example, the following is the general format for the Fortran compiler command line:

```
% 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`.
- `-I`*directory_name*: looks for `include` files in *directory_name*.
- `-c`: compiles without invoking the linker. This option 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-flush 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. You can use the `time`(1) command to determine the amount of system time consumed. 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 the location of floating-point errors.

Environment Modules

A *module* modifies a user's environment dynamically. A user can load a module, rather than 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 retrieve a list of Modules that are available on your system, use the following command:

```
% module avail
```

To load Modules into your environment, use the following commands:

```
% module load intel-compilers-latest mpt/2.12
```

Note: The preceding commands are an example only. The actual release numbers 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 following command displays a list of all arguments accepted:

```
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 simplify local software development by hiding compilation details. Libraries are sometimes also called *archives.*

The SGI compiling environment contains several types of libraries. The following topics provide an overview about each library:

- "Static Libraries" on page 4
- "Dynamic Libraries" on page 5
- "C/C++ Libraries" on page 5
- "MPI and SHMEM Libraries" on page 6

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 `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, 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. When you load dynamic libraries into memory, they are available for access 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 `-L` *path* and `-l` *library_shortname* compiler options 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 Intel compiler provides the following C/C++ libraries:

- `libguide.a` and `libguide.so`, which support OpenMP-based programs.
- `libsvml.a`, which is the short vector math library.
- `libirc.a`, which includes Intel support for Profile-Guided Optimizations (PGO) and CPU dispatch.
- `libimf.a` and `libimf.so`, which are Intel's math libraries.
- `libcprts.a` and `libcprts.so`, which are the Dinkumware C++ libraries.
- `libunwind.a` and `libunwind.so`, which are the Unwinder libraries.
- `libcxa.a` and `libcxa.so`, which provide Intel run-time support for C++ features.

MPI and SHMEM Libraries

The SGI MPI software package facilitates parallel programming on large computer systems and on computer system clusters. SGI MPI supports both the Message Passing Interface (MPI) standard and the OpenSHMEM standard. SGI's support for MPI and OpenSHMEM is built on top of the SGI Message Passing Toolkit (MPT). SGI MPT is a high-performance communications middleware software product that runs on SGI's shared memory and cluster supercomputers. On some of these machines, SGI MPT uses SGI Array Services to launch applications. SGI MPT is optimized for all SGI hardware platforms.

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. You can use the SHMEM API alone or in combination with MPI routines in the same parallel program.

For more information, see the following:

- The `intro_shmem`(3) man page.
- The *SGI MPI and SGI SHMEM User Guide.*

Performance Analysis and Debugging

This chapter contains the following topics:

About 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, which is the time accumulated by a user process when it is attached to a CPU and is running.
- Elapsed (wall-clock) time, which is the amount of time that passes between the start and the termination of a process.
- System time, which is the amount of time spent on performing kernel functions such as 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 the problem in the code
3. Applying an optimization technique

This topics in this chapter describe how to analyze your code to determine performance bottlenecks. For information about how to tune your application for a single processor system and then tune it for parallel processing, see the following:

Chapter 5, "Performance Tuning" on page 53

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 might or might not provide good results.

The topology(1) command displays general information about SGI systems, with a focus on node information. This can include 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(1) command is part of the SGI Foundation Software package.

The following is example output from two topology(1) commands:

```
uv-sys:~ # topology
System type: UV2000
System name: harp34-sys
Serial number: UV2-00000034
Partition number: 0
       8 Blades
     256 CPUs
      16 Nodes
  235.82 GB Memory Total
   15.00 GB Max Memory on any Node
       1 BASE I/O Riser
       2 Network Controllers
       2 Storage Controllers
       2 USB Controllers
       1 VGA GPU

uv-sys:~ # topology --summary --nodes --cpus
System type: UV2000
System name: harp34-sys
Serial number: UV2-00000034
Partition number: 0
       8 Blades
     256 CPUs
      16 Nodes
  235.82 GB Memory Total
   15.00 GB Max Memory on any Node
       1 BASE I/O Riser
       2 Network Controllers
```

```
       2 Storage Controllers
       2 USB Controllers
       1 VGA GPU

Index      ID        NASID CPUS     Memory
-----------------------------------------
    0 r001i11b00h0      0   16      15316 MB
    1 r001i11b00h1      2   16      15344 MB
    2 r001i11b01h0      4   16      15344 MB
    3 r001i11b01h1      6   16      15344 MB
    4 r001i11b02h0      8   16      15344 MB
    5 r001i11b02h1     10   16      15344 MB
    6 r001i11b03h0     12   16      15344 MB
    7 r001i11b03h1     14   16      15344 MB
    8 r001i11b04h0     16   16      15344 MB
    9 r001i11b04h1     18   16      15344 MB
   10 r001i11b05h0     20   16      15344 MB
   11 r001i11b05h1     22   16      15344 MB
   12 r001i11b06h0     24   16      15344 MB
   13 r001i11b06h1     26   16      15344 MB
   14 r001i11b07h0     28   16      15344 MB
   15 r001i11b07h1     30   16      15344 MB

CPU        Blade PhysID CoreID APIC-ID Family Model Speed L1(KiB) L2(KiB) L3(KiB)
-------------------------------------------------------------------------------
  0 r001i11b00h0     00     00       0      6    45  2599 32d/32i     256   20480
  1 r001i11b00h0     00     01       2      6    45  2599 32d/32i     256   20480
  2 r001i11b00h0     00     02       4      6    45  2599 32d/32i     256   20480
  3 r001i11b00h0     00     03       6      6    45  2599 32d/32i     256   20480
  4 r001i11b00h0     00     04       8      6    45  2599 32d/32i     256   20480
  5 r001i11b00h0     00     05      10      6    45  2599 32d/32i     256   20480
  6 r001i11b00h0     00     06      12      6    45  2599 32d/32i     256   20480
  7 r001i11b00h0     00     07      14      6    45  2599 32d/32i     256   20480
  8 r001i11b00h1     01     00      32      6    45  2599 32d/32i     256   20480
  9 r001i11b00h1     01     01      34      6    45  2599 32d/32i     256   20480
 10 r001i11b00h1     01     02      36      6    45  2599 32d/32i     256   20480
 11 r001i11b00h1     01     03      38      6    45  2599 32d/32i     256   20480
...
```

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(1) command gets its information from /proc/cpuinfo, the /sys/devices/system directory structure, and /proc/sgi_uv/topology. When creating cpusets, the numbers reported in the output section called Processor Numbering on Node(s) correspond to the mems argument you use to define 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 *SGI Cpuset Software Guide*.

The following is example output:

```
uv# cpumap
Thu Sep 19 10:17:21 CDT 2013
harp34-sys.americas.sgi.com

This is an SGI UV
model name           : Genuine Intel(R) CPU @ 2.60GHz
Architecture         : x86_64
cpu MHz              : 2599.946
cache size           : 20480 KB (Last Level)

Total Number of Sockets                : 16
Total Number of Cores                  : 128   (8 per socket)
Hyperthreading                         : ON
Total Number of Physical Processors    : 128
Total Number of Logical Processors     : 256   (2 per Phys Processor)

UV Information
 HUB Version:                           UVHub  3.0
 Number of Hubs:                        16
 Number of connected Hubs:              16
 Number of connected NUMAlink ports:    128
================================================================================


Hub-Processor Mapping

  Hub Location      Processor Numbers -- HyperThreads in ()
  --- ----------    ---------------------------------------
    0 r001i11b00h0     0    1    2    3    4    5    6    7
                   (  128  129  130  131  132  133  134  135 )
    1 r001i11b00h1     8    9   10   11   12   13   14   15
```

```
                (  136  137  138  139  140  141  142  143 )
   2 r001i11b01h0    16   17   18   19   20   21   22   23
                (  144  145  146  147  148  149  150  151 )
   3 r001i11b01h1    24   25   26   27   28   29   30   31
                (  152  153  154  155  156  157  158  159 )
   4 r001i11b02h0    32   33   34   35   36   37   38   39
                (  160  161  162  163  164  165  166  167 )
   5 r001i11b02h1    40   41   42   43   44   45   46   47
                (  168  169  170  171  172  173  174  175 )
   6 r001i11b03h0    48   49   50   51   52   53   54   55
                (  176  177  178  179  180  181  182  183 )
   7 r001i11b03h1    56   57   58   59   60   61   62   63
                (  184  185  186  187  188  189  190  191 )
   8 r001i11b04h0    64   65   66   67   68   69   70   71
                (  192  193  194  195  196  197  198  199 )
   9 r001i11b04h1    72   73   74   75   76   77   78   79
                (  200  201  202  203  204  205  206  207 )
  10 r001i11b05h0    80   81   82   83   84   85   86   87
                (  208  209  210  211  212  213  214  215 )
  11 r001i11b05h1    88   89   90   91   92   93   94   95
                (  216  217  218  219  220  221  222  223 )
  12 r001i11b06h0    96   97   98   99  100  101  102  103
                (  224  225  226  227  228  229  230  231 )
  13 r001i11b06h1   104  105  106  107  108  109  110  111
                (  232  233  234  235  236  237  238  239 )
  14 r001i11b07h0   112  113  114  115  116  117  118  119
                (  240  241  242  243  244  245  246  247 )
  15 r001i11b07h1   120  121  122  123  124  125  126  127
                (  248  249  250  251  252  253  254  255 )

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

Processor Numbering on Node(s)

   Node    (Logical) Processors
  ------    -------------------------
     0      0    1    2    3    4    5    6    7  128  129  130  131  132  133  134  135
     1      8    9   10   11   12   13   14   15  136  137  138  139  140  141  142  143
     2     16   17   18   19   20   21   22   23  144  145  146  147  148  149  150  151
     3     24   25   26   27   28   29   30   31  152  153  154  155  156  157  158  159
     4     32   33   34   35   36   37   38   39  160  161  162  163  164  165  166  167
```

```
     5     40   41   42   43   44   45   46   47  168  169  170  171  172  173  174  175
     6     48   49   50   51   52   53   54   55  176  177  178  179  180  181  182  183
     7     56   57   58   59   60   61   62   63  184  185  186  187  188  189  190  191
     8     64   65   66   67   68   69   70   71  192  193  194  195  196  197  198  199
     9     72   73   74   75   76   77   78   79  200  201  202  203  204  205  206  207
    10     80   81   82   83   84   85   86   87  208  209  210  211  212  213  214  215
    11     88   89   90   91   92   93   94   95  216  217  218  219  220  221  222  223
    12     96   97   98   99  100  101  102  103  224  225  226  227  228  229  230  231
    13    104  105  106  107  108  109  110  111  232  233  234  235  236  237  238  239
    14    112  113  114  115  116  117  118  119  240  241  242  243  244  245  246  247
    15    120  121  122  123  124  125  126  127  248  249  250  251  252  253  254  255

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

Sharing of Last Level (3) Caches

  Socket    (Logical) Processors
  ------    ------------------------
     0      0    1    2    3    4    5    6    7  128  129  130  131  132  133  134  135
     1      8    9   10   11   12   13   14   15  136  137  138  139  140  141  142  143
     2     16   17   18   19   20   21   22   23  144  145  146  147  148  149  150  151
     3     24   25   26   27   28   29   30   31  152  153  154  155  156  157  158  159
     4     32   33   34   35   36   37   38   39  160  161  162  163  164  165  166  167
     5     40   41   42   43   44   45   46   47  168  169  170  171  172  173  174  175
     6     48   49   50   51   52   53   54   55  176  177  178  179  180  181  182  183
     7     56   57   58   59   60   61   62   63  184  185  186  187  188  189  190  191
     8     64   65   66   67   68   69   70   71  192  193  194  195  196  197  198  199
     9     72   73   74   75   76   77   78   79  200  201  202  203  204  205  206  207
    10     80   81   82   83   84   85   86   87  208  209  210  211  212  213  214  215
    11     88   89   90   91   92   93   94   95  216  217  218  219  220  221  222  223
    12     96   97   98   99  100  101  102  103  224  225  226  227  228  229  230  231
    13    104  105  106  107  108  109  110  111  232  233  234  235  236  237  238  239
    14    112  113  114  115  116  117  118  119  240  241  242  243  244  245  246  247
    15    120  121  122  123  124  125  126  127  248  249  250  251  252  253  254  255

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

HyperThreading

  Shared Processors
  -----------------
```

```
(    0,  128) (    1,  129) (    2,  130) (    3,  131)
(    4,  132) (    5,  133) (    6,  134) (    7,  135)
(    8,  136) (    9,  137) (   10,  138) (   11,  139)
(   12,  140) (   13,  141) (   14,  142) (   15,  143)
(   16,  144) (   17,  145) (   18,  146) (   19,  147)
(   20,  148) (   21,  149) (   22,  150) (   23,  151)
(   24,  152) (   25,  153) (   26,  154) (   27,  155)
(   28,  156) (   29,  157) (   30,  158) (   31,  159)
(   32,  160) (   33,  161) (   34,  162) (   35,  163)
(   36,  164) (   37,  165) (   38,  166) (   39,  167)
(   40,  168) (   41,  169) (   42,  170) (   43,  171)
(   44,  172) (   45,  173) (   46,  174) (   47,  175)
(   48,  176) (   49,  177) (   50,  178) (   51,  179)
(   52,  180) (   53,  181) (   54,  182) (   55,  183)
(   56,  184) (   57,  185) (   58,  186) (   59,  187)
(   60,  188) (   61,  189) (   62,  190) (   63,  191)
(   64,  192) (   65,  193) (   66,  194) (   67,  195)
(   68,  196) (   69,  197) (   70,  198) (   71,  199)
(   72,  200) (   73,  201) (   74,  202) (   75,  203)
(   76,  204) (   77,  205) (   78,  206) (   79,  207)
(   80,  208) (   81,  209) (   82,  210) (   83,  211)
(   84,  212) (   85,  213) (   86,  214) (   87,  215)
(   88,  216) (   89,  217) (   90,  218) (   91,  219)
(   92,  220) (   93,  221) (   94,  222) (   95,  223)
(   96,  224) (   97,  225) (   98,  226) (   99,  227)
(  100,  228) (  101,  229) (  102,  230) (  103,  231)
(  104,  232) (  105,  233) (  106,  234) (  107,  235)
(  108,  236) (  109,  237) (  110,  238) (  111,  239)
(  112,  240) (  113,  241) (  114,  242) (  115,  243)
(  116,  244) (  117,  245) (  118,  246) (  119,  247)
(  120,  248) (  121,  249) (  122,  250) (  123,  251)
(  124,  252) (  125,  253) (  126,  254) (  127,  255)
```

The `x86info`(1) command displays x86 CPU diagnostics information. Type one of the following commands to load the `x86info`(1) command if the command is not already installed:

- On Red Hat Enterprise Linux (RHEL) systems, type the following:

  # **yum install x86info.x86_64**

- On SLES systems, type the following:

  # **zypper install x86info**

The following is an example of x86info(1) command output:

```
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 GNU/Linux
```

For more system information, change to the `/sys/devices/system/node/node0/cpu0/cache` directory and list the contents. 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
```

Note: The preceding example output was truncated at the right for inclusion in this manual.

Sources of Performance Problems

The following three processes types typically cause program execution performance slowdowns:

- CPU-bound processes, whch are processes that perform slow operations, such as `sqrt` or floating-point divides, or nonpipelined operations, such as switching between add and multiply operations.
- Memory-bound processes, which consist of code that uses poor memory strides, occurrences of page thrashing or cache misses, or poor data placement in NUMA systems.
- I/O-bound processes, which are processes that wait on synchronous I/O or formatted I/O. These are also processes that wait when there is library-level or system-level buffering.

The following topics describe some of the tools that can help pinpoint performance slowdowns:

Profiling with `perf`

Linux Perf Events provides a performance analysis framework. It includes hardware-level CPU performance monitoring unit (PMU) features, software counters, and tracepoints. The `perf` RPM comes with the operating system, includes man pages, and is not an SGI product.

For more information, see the following man pages:

- `perf`(1)
- `perf-stat`(1)
- `perf-top`(1)
- `perf-record`(1)
- `perf-report`(1)
- `perf-list`(1)

Profiling with `PerfSuite`

PerfSuite is a set of tools, utilities, and libraries that you can use to analyze application software performance on Linux systems. You can use the PerfSuite tools to perform performance-related activities, ranging from assistance with compiler optimization reports to hardware performance counting, profiling, and MPI usage summarization. PerfSuite is Open Source software. It is approved for licensing under the University of Illinois / NCSA Open Source License (OSI-approved).

For more information, see one of the following websites:

- http://perfsuite.ncsa.uiuc.edu/
- http://perfsuite.sourceforge.net/
- http://perfsuite.ncsa.illinois.edu

  This website hosts NCSA-specific information about using PerfSuite tools.

PerfSuite includes the `psrun` utility, which gathers hardware performance information on an unmodified executable. For more information, see http://perfsuite.ncsa.uiuc.edu/psrun/.

Other Performance Analysis Tools

The following tools might be useful to you when you try to optimize your code:

- The Intel® VTune™ Amplifier XE, which is a performance and thread profiler. This tool can perform both local sampling and remote sampling experiments. In the remote samplling case, the VTune data collector runs on the Linux system, and an accompanying graphical user interface (GUI) runs on a Windows system, which is used for analyzing the results. The VTune profiler allows you to perform interactive experiments while connected to the host through its GUI.

  For information about Intel VTune Amplifier XE, see the following URL:

  http://software.intel.com/en-us/intel-vtune-amplifier-xe#pid-3773–760

- Intel Inspector XE, which is a memory and thread debugger. For information about Intel Inspector XE, see the following:

  http://software.intel.com/en-us/intel-inspector-xe/

- Intel Advisor XE, which is a threading design and prototyping tool. For information about Intel Advisor XE, see the following:

  http://software.intel.com/en-us/intel-advisor-xe

About Debugging

The following debuggers are available on SGI platforms:

- The Intel Debugger and GDB. You can start the Intel Debugger and GDB with the `ddd` command. The `ddd` command starts the Data Display Debugger, a GNU product that provides a graphical debugging interface.

- Totalview. TotalView is graphical debugger that you can use with MPI programs. For information about TotalView, including its licensing, see the following:

  http://www.roguewave.com

- Allinea DDT. The Allinea DDT debugger is a graphical debugger from Allinea. For information about Allinea DDT, see the following:

  www.allinea.com/products/ddt

The following topics provide more information about some of the debuggers available on SGI systems:

- "Using the Intel Debugger" on page 18
- "Using the GNU Data Display Debugger (GNU DDD)" on page 19

Using the Intel Debugger

The Intel Debugger for Linux is the Intel symbolic debugger. The Intel Debugger is part of Intel Parallel Studio XE Professional Edition and above. This debugger is based on the Eclipse GUI. This debugger works with the Intel® C and C++ compilers, the Intel® Fortran compilers, and the GNU compilers. This product is available if your system is licensed for the Intel compilers. You are asked during the installation if you want to install it or not. The `idb` command starts the GUI. The `idbc` command starts the command line interface. If you specify the `-gdb` option on the `idb` command, the shell command line provides user commands and debugger output similar to the GNU debugger.

You can use the Intel Debugger for Linux with single-threaded applications, multithreaded applications, serial code, and parallel code.

Figure 2-1 on page 19 shows the GUI.

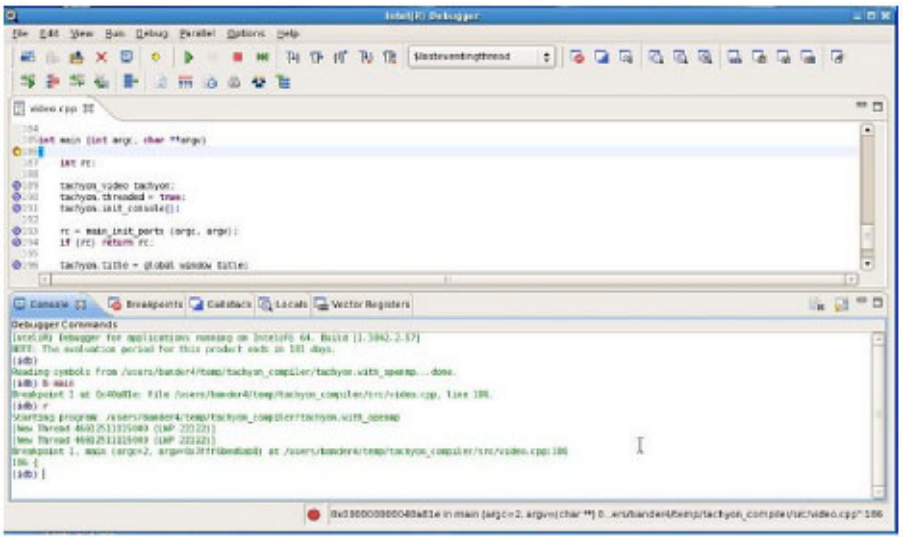

Figure 2-1 Intel® Debugger GUI

For more information, see the following:

http://software.intel.com/en-us/articles/idb-linux/

Using the GNU Data Display Debugger (GNU DDD)

GDB is the GNU debugger. The GDB debugger supports C, C++, Fortran, and Modula-2 programs. The following information pertains to these compilers:

- When compiling with C and C++, include the `-g` option on the compiler command line. The `-g` option produces the `dwarf2` symbols database that GDB uses.
- 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 standard GDB debugger does not support Fortran 95 programs. To debug Fortran 95 programs, download and install the `gdbf95` patch from the following website:

  http://sourceforge.net/project/showfiles.php?group_id=56720

  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 2012 Free Software Foundation, Inc.
```

The Data Display Debugger provides a graphical debugging interface for the GDB debugger and other command line debuggers. To use GDB through a GUI, use the `ddd` command. Specify the `--debugger` option to specify the debugger you want to use. For example, specify `--debugger "idb"` to specify the Intel Debugger. Use the `gdb` command to start GDB's command line interface.

When the debugger loads, the Data Display Debugger screen appears divided into panes that show the following information:

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

From the **View** menu, you can switch these panes on and off.

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

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

For a complete list of GDB commands, use the `help` option or see the documentation at the following website:

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

Note: The 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 the Intel Debugger for Linux.

Monitoring Commands

This chapter includes the following topics:

About the Operating System Monitoring Commands

You can use operating system commands to understand the usage and limits of your system. These commands allow you to observe both overall system performance and single-performance execution characteristics.

The topics in this chapter describe the commands that are included in your SGI system's operating system. The following are additional commands and utilities that are available:

- SGI Foundation Software utilities. SFS is included by default on your SGI system. For information about these utilities, see the following:

  SGI Foundation Software (SFS) User Guide

- Performance Co-Pilot. For documentation about this open source toolset, see the following website:

  http://www.pcp.io/documentation.html

Operating System Monitoring Commands

The following topics show several operating system commands you can use to determine user load, system usage, and active processes:

Using the w(1) command

To obtain a high-level view of system usage that includes information about who is logged into the system, 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 w command's output shows who is on the system, the duration of user sessions, processor usage by user, and currently executing user commands. The output consists of two parts:

- The first output line shows 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 in the last one, five, and 15 minutes.
- The rest of the output from the w command shows who is logged into the system, the duration of each user session, processor usage by user, and each user's current process command line.

Using the ps(1) Command

To determine active processes, use the ps(1) command, which displays a snapshot of the process table.

The ps -A r command example that follows returns all the processes currently running on a system:

```
[user@profit user]# ps -A r
   PID TTY      STAT   TIME COMMAND
211116 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211117 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
```

```
211118 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211119 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211120 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211121 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211122 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211123 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211124 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211125 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211126 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211127 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211128 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211129 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211130 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
211131 pts/0    R+     4:08 /usr/diags/bin/olconft RUNTIME=5
...
```

Using the `top`(1) Command

To monitor running processes, use the `top`(1) command. This command displays a sorted list of top CPU utilization processes.

Using the `vmstat`(8) Command

The `vmstat`(8) command reports virtual memory statistics. It reports information about processes, memory, paging, block IO, traps, and CPU activity. For more information, see the `vmstat`(8) man page.

In the following `vmstat`(8) command, the `10` specifies a 10–second delay between updates.

```
uv44-sys:~ # vmstat 10
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0      0 235984032 418748 8649568    0    0     0     0    0    0  0  0 100  0  0
 1  0      0 236054400 418748 8645216    0    0     0  4809 256729 3401  0  0 100  0  0
 1  0      0 236188016 418748 8649904    0    0     0   448 256200  631  0  0 100  0  0
 2  0      0 236202976 418748 8645104    0    0     0   341 256201 1117  0  0 100  0  0
 1  0      0 236088720 418748 8592616    0    0     0   847 257104 6152  0  0 100  0  0
 1  0      0 235990944 418748 8648460    0    0     0   240 257085 5960  0  0 100  0  0
 1  0      0 236049568 418748 8645100    0    0     0  4849 256749 3604  0  0 100  0  0
```

Without the *delay* parameter, which is `10` in this example, the output returns 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.

Using the `iostat`(1) command

The `iostat`(1) command monitors system input/output device loading by observing the time the devices are active, relative to their average transfer rates. You can use information from the `iostat` command to change system configuration information to better balance the input/output load between physical disks. For more information, see the `iostat`(1) man page.

In the following `iostat`(1) command, the `10` specifies a 10–second interval between updates:

```
# iostat 10
Linux 2.6.32-430.el6.x86_64 (harp34-sys)      02/21/2014    _x86_64_  (256 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          46.24    0.01    0.67    0.01    0.00   53.08

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda              53.66     23711.65     23791.93 21795308343 21869098736
sdb               0.01         0.02         0.00      17795          0

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          99.96    0.00    0.04    0.00    0.00    0.00

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda             321.20    149312.00    150423.20    1493120    1504232
sdb               0.00         0.00         0.00          0          0

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          99.95    0.00    0.05    0.00    0.00    0.00

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda             305.19    146746.05    148453.95    1468928    1486024
sdb               0.00         0.00         0.00          0          0
...
```

Using the `sar`(1) command

The `sar`(1) command returns the content of selected, cumulative activity counters in the operating system. Based on the values in the *count* and *interval* parameters, the command writes information *count* times spaced at the specified *interval*, which is in seconds. For more information, see the `sar`(1) man page. The following example shows the `sar`(1) command with a request for information about CPU 1, a count of 10, and an interval of 10:

```
uv44-sys:~ # sar -P 1 10 10
Linux 2.6.32-416.el6.x86_64 (harp34-sys)      09/19/2013   _x86_64_   (256 CPU)

11:24:54 AM     CPU     %user     %nice   %system   %iowait    %steal     %idle
11:25:04 AM       1      0.20      0.00      0.10      0.00      0.00     99.70
11:25:14 AM       1     10.10      0.00      0.30      0.00      0.00     89.60
11:25:24 AM       1     99.70      0.00      0.30      0.00      0.00      0.00
11:25:34 AM       1     99.70      0.00      0.30      0.00      0.00      0.00
11:25:44 AM       1      8.99      0.00      0.60      0.00      0.00     90.41
11:25:54 AM       1      0.10      0.00      0.20      0.00      0.00     99.70
11:26:04 AM       1     38.70      0.00      0.10      0.00      0.00     61.20
11:26:14 AM       1     99.80      0.00      0.10      0.00      0.00      0.10
11:26:24 AM       1     80.42      0.00      0.70      0.00      0.00     18.88
11:26:34 AM       1      0.10      0.00      0.20      0.00      0.00     99.70
Average:          1     43.78      0.00      0.29      0.00      0.00     55.93
```

Data Process and Placement Tools

This chapter contains the following topics:

About Nonuniform Memory Access (NUMA) Computers

On symmetric multiprocessor (SMP) computers, all data is visible from all processors. Each processor is functionally identical and has equal time access to every memory address. That is, all processors have equally fast (symmetric) access to memory. These types of systems are easy to assemble but have limited scalability due to memory access times.

In contrast, a NUMA system has a shared address space, but the access time to memory varies over physical address ranges and between processing elements. Each processor has its own memory and can address the memory attached to another processor through the Quick Path Interconnect (QPI).

In both cases, there is a single shared memory space and a single operating system instance.

There are two levels of NUMA: *intranode*, managed by the Intel QPI, and *internode*, managed through the SGI HUB ASIC and SGI NUMAlink technology.

The following topics explain other aspects of the SGI NUMA computers:

Distributed Shared Memory (DSM)

Scalability is the measure of how the work done on a computing system changes as you add CPUs, memory, network bandwidth, I/O capacity, and other resources. Many factors, for example, memory latency across the system, can affect scalability.

In the SGI UV series systems, memory is physically distributed both within and among the IRU enclosures, which consist of the compute, memory, and I/O blades. However, memory is accessible to and shared by all devices, connected by NUMAlink, within the single-system image (SSI). In other words, all components connected by NUMAlink share a single Linux operating system, and they 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.

The following are the terms used to refer to the types of memory within a system:

- If a processor accesses memory that is on a compute node blade, that memory is referred to as the node's *local memory*.
- If processors access memory located on other blade nodes within the IRU or within other NUMAlink IRUs, the memory is referred to as *remote memory*.
- The total memory within the NUMAlink system is referred to as *global memory*.

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 the following topics describe:

- "Cache Coherency" on page 30
- "Non-uniform Memory Access (NUMA)" on page 31

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 64–byte block of memory 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 and 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*. When more than one processor has a copy of the block, it is in a *shared state*. A bit vector indicates the caches that 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 *nonuniform*. For example, it takes less time for a processor blade to reference its locally installed memory than to reference remote memory.

About the Data and Process Placement Tools

For cc-NUMA systems, like the SGI UV systems, performance degrades when the application accesses remote memory, versus local memory. Because the Linux operating system has a tendency to migrate processes, SGI recommends that you use the data and process placement tools.

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, gets pages from single node.

  Perform initialization in parallel, such that each processor initializes data that it is likely to access later for calculation.
- In the parallel loop, multiple processors access that one memory page.

Figure 4-1 on page 32, shows how to code to get good data placement.

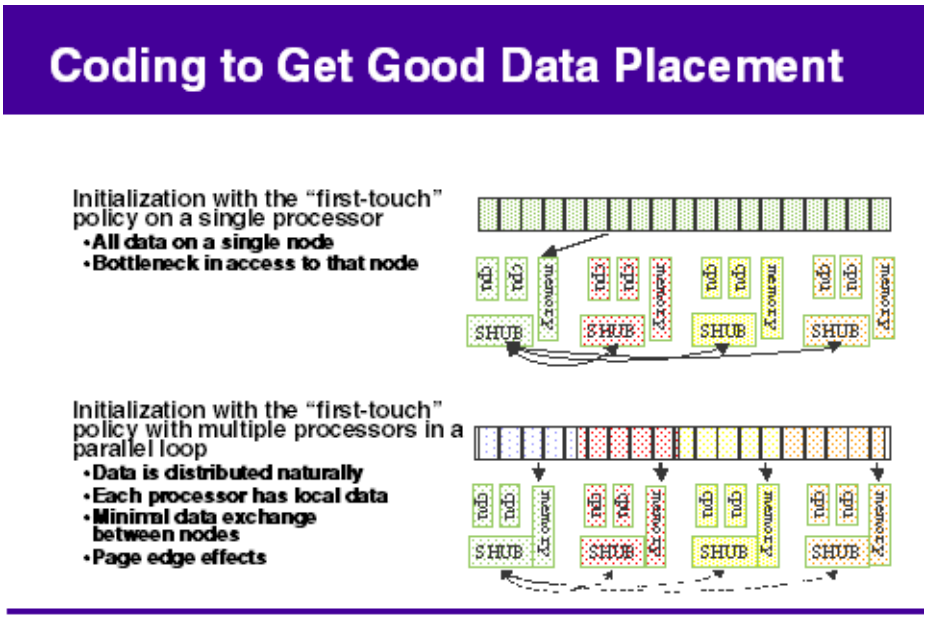

Figure 4-1 Coding to Get Good Data Placement

The `dplace`(1) tool, the `taskset`(1) command, and the cpuset tools are built upon the cpusets API. These tools enable your applications to avoid poor data locality caused by process or thread drift from CPU to CPU. The `omplace`(1) tool works like the `dplace`(1) tool and is designed for use with OpenMP applications. The differences among these tools are as follows:

- The `taskset`(1) command restricts execution to the listed set of CPUs when you specify the `-c` or `--cpu-list` option. The process is free to move among the CPUs that you specify.
- The `dplace`(1) tool differs from `taskset`(1) in that `dplace`(1) binds processes to specified CPUs in round-robin fashion. After a process is pinned, it does not migrate, so you can use this for increasing the performance and reproducibility of parallel codes.

- Cpusets are named subsets of system cpus/memories and are used extensively in batch environments. For more information about cpusets, see the *SGI Cpuset Software Guide*.

The following topics provide more information about the data and process placement utilities:

- "About cpusets and Control Groups (cgroups)" on page 33
- "`dplace` Command" on page 35
- "`omplace` Command" on page 41
- "`taskset` Command" on page 42
- "`numactl` Command" on page 44
- "`dlook` Command" on page 45

About cpusets and Control Groups (cgroups)

SGI systems support both cgroups and cpusets. The cpusets are a subsystem of cgroups.

For information about cpusets and cgroups, see the following:

- "Using cpusets" on page 33
- "Using cgroups" on page 34

Using cpusets

The cpuset facility is a workload manager tool that permits a system administrator to restrict the number of processor and memory resources that a process or set of processes can use. A *cpuset* defines a list of CPUs and memory nodes. A process contained in a cpuset can execute only on the CPUs in that cpuset and can only allocate memory on the memory nodes in that cpuset. Essentially, a cpuset provides you with a CPU and a memory container or a *soft partition* 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 run-time repeatability.

Be aware that when placed in a cpuset, certain kernel threads can exhibit undesirable behavior. In general, kernel threads are not confined to a cpuset, but when a

bootcpuset is created, all the kernel threads that are able to be placed in a cpuset become confined to the bootcpuset. In the case of the khugepaged daemon, this is undesirable because khugepaged becomes unable to allocate memory for processes that are on nodes outside of its cpuset. As a workaround, remove khugepaged from the bootcpuset after the machine is up and running. The following procedure explains how to implement the workaround.

Procedure 4-1 To remove khugepaged from the bootcpuset

1. Type the following command to retrieve the process ID of the khugepaged daemon:

   ```
   ps -ef | grep khugepaged | grep -v grep
   ```

   For example, in the following output, 1054 is the process ID:

   ```
   # ps -ef | grep khugepaged | grep -v grep
   root     1054     2  0 Mar04 ?      00:00:02 [khugepaged]
   ```

2. Use the echo(1) command, in the following format, to remove khugepaged from the bootcpuset:

   ```
   echo khugepaged_pid > /dev/cpuset/tasks
   ```

   For *pid*, specify the process ID for the khugepaged daemon.

3. (Optional) Script the preceding lines and run the script at boot time to ensure that the khugepaged thread is always removed from the bootcpuset.

For general information about cpusets, see the following:

- *SGI Cpuset Software Guide*
- https://www.kernel.org/doc/Documentation/cgroups/cpusets.txt

Using cgroups

If you use cgroups, you can exert finer control over memory than is possible with cpusets. If you use cgroups, be aware that their use can result in a 1–5% memory overhead penalty. If you use a batch scheduler, verify that it supports cgroups before you configure cgroups.

For general information about cgroups, see the following:

https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

`dplace` Command

You can use the `dplace`(1) command to improve the performance of processes running on your SGI nonuniform memory access (NUMA) machine.

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 is running, a higher percentage of memory references are made to remote nodes. Because remote accesses typically have higher access times, performance can degrade. CPU instruction pipelines also have to be reloaded.

The `dplace`(1) command specifies scheduling and memory placement policies for the process. 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 because a higher percentage of memory accesses are made to local nodes.

Processes always execute within a cpuset. The cpuset specifies the CPUs on which a process can run. By default, processes usually execute in a cpuset that contains all the CPUs in the system. For information about cpusets, see the *SGI Cpuset Software Guide*.

The `dplace`(1) command creates a placement container that includes all the CPUs, 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 run the command.

The command runs within this placement container and remains bound to the first CPU of the container. As the command forks child processes, the child processes 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 a parent or a child, is bound to CPU 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 about `dplace`(1), see the `dplace`(1) man page. The `dplace`(1) man page also includes examples of how to use the command.

Example 4-1 Using the dplace(1) command with MPI Programs

The following command improves the placement of MPI programs on NUMA systems and verifies placement of certain data structures of a long-running MPI program:

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

The -s1 parameter causes dplace(1) to start placing processes with the second process, p1. The first process, p0, is not placed because it is associated with the job launch, not with the job itself. The -c 0-63 parameter causes dplace(1) to use processors 0–63.

You can then use the dlook(1) command to verify placement of the data structures in another window on one of the slave thread PIDs. For more information about the dlook command, see "dlook Command" on page 45 and the dlook(1) man page.

Example 4-2 Using the dplace(1) command with OpenMP Programs

The following command runs an OpenMP program on logical CPUs 4 through 7 within the current cpuset:

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

Example 4-3 Using the dplace(1) command with OpenMP Programs

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

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

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

Example 4-4 Using the dplace(1) command with Linux commands

The following examples assume that you run the dplace commands from a shell that runs 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.

Example 4-5 Using the `dplace` command and a debugger for verification

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 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
```

You can also 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
```

These are placed as specified:

```
>> oncpus e00002343c528000 e000013817540000 e000013473aa8000
>> e000013817c68000 e0
00234704f0000 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
```

Example 4-6 Using the dplace(1) command for compute thread placement troubleshooting

Sometimes compute threads do not end up on unique processors when using commands such a dplace(1) or profile.pl. For information about Perfsuite, see the following:

"Profiling with PerfSuite" on page 16

In this example, assume that the dplace -s1 -c0-15 command bound 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
```

Even if an application starts some threads executing for a very short time, 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. If 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 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 CPUs 0-3, with `dplace -e -C0-16` and you see that threads are always placed on CPU 0, 1, 5, and 6, then one of the following commands should place your threads correctly:

  ```
  dplace -e -c0,1,x,x,x,2,3
  ```

  or

  ```
  dplace -x24 -c0-3  # x24 =11000, place the 2 first and skip 3 before placing
  ```

`omplace` Command

The `omplace`(1) command controls the placement of MPI processes and OpenMP threads. This command is a wrapper script for `dplace`(1). Use `omplace`(1), rather than `dplace`(1), if your application uses MPI, OpenMP, pthreads, or hybrid MPI/OpenMP and MPI/pthreads codes. The `omplace`(1) command generates the proper `dplace`(1) placement file syntax automatically. It also supports some unique options, such as block-strided CPU lists.

The `omplace`(1) 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. The CPU placement is performed by

dynamically generating a placement file and invoking dplace(1) with the MPI job launch.

For example, to run two MPI processes with four threads per process, and to display the generated placement file, type a command similar to the following:

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

The preceding command places the threads 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 the *SGI MPI and SGI SHMEM User Guide*.

taskset Command

You can use the taskset(1) command to perform the following tasks:

- Restricting execution to a list of CPUs. Use the -c parameter and the --cpu-list parameter.
- Retrieving or setting the CPU affinity of a process. Use the following parameters:

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

- Launching a new command with a specified CPU affinity.

CPU affinity is a scheduler property that bonds a process to a given set of CPUs on the system. The Linux scheduler honors the given CPU affinity and runs the process only on the specified CPUs. The process does not run on any other CPUs. Note that the scheduler also supports natural CPU affinity in which the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. 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. The *mask* parameter can specify more CPUs than are present. In other words, it might be true that not all CPUs specified in the *mask* exist on a given system. A retrieved mask reflects only the bits that correspond to CPUs physically on the system. If the mask does not correspond to any valid CPUs on the system, the mask is invalid, and the system returns an error. The masks are typically specified in hexadecimal notation. For example:

***mask* specification**	**CPUs specified**
`0x00000001`	Processor #0
`0x00000003`	Processors #0 and #1
`0xFFFFFFFF`	All processors (#0 through #31)

When `taskset`(1) returns, it is guaranteed that the given program has been scheduled to a valid CPU.

The `taskset`(1) command does not pin a task to a specific CPU. Rather, it restricts a task so that it does not run on any CPU that is not in the CPU list. For example, if you use `taskset`(1) to launch an application that forks multiple tasks, it is possible that the scheduler initially assigns multiple tasks to the same CPU, even though there are idle CPUs that are in the CPU list. Scheduler load balancing software eventually distributes 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, a task can jump to a different CPU. This outcome can affect memory latency as pages that were node-local before the jump can be remote after the jump.

If you are running an MPI application, SGI recommends that you do not use the `taskset`(1) command because the `taskset`(1) command can pin the MPI shepherd process, which wastes a CPU, and then put the remaining working MPI rank on one of the CPUs that already had some other rank running on it. Instead of `taskset`(1), SGI recommends that you use the `dplace`(1) command or the environment variable `MPI_DSM_CPULIST`. For more information, see "`dplace` Command" on page 35.

If you are using a batch scheduler that creates and destroys cpusets dynamically, SGI recommends that you use the `MPI_DSM_DISTRIBUTE` environment variable instead of either the `MPI_DSM_CPULIST` environment variable or the `dplace`(1) command.

Example 1. The following example shows how to run an MPI program on eight CPUs:

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

Example 2. The following example sets the MPI_DSM_CPULIST variable:

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

Example 3. The following example runs an executable on CPU 1. The *mask* for CPU 1 is 0x2, so type the following:

```
# taskset 0x2 executable_name
```

Example 4. The following example moves PID 14057 to CPU 0. The *mask* for CPU 0 is 0x1, so type the following:

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

Example 5. The following example runs an MPI Abaqus/Standard job on an SGI UV system with eight CPUs. Standard input is redirected to /dev/null to avoid a SIGTTIN signal for MPT applications. Type the following:

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

Example 6. The following example uses the taskset(1) command to lock a given process to a particular CPU, CPU5, and then uses the profile(1) command to profile it. The second command moves the process to another CPU, CPU3. Type the following:

```
# 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
```

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

numactl Command

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

`dlook` Command

You can use the `dlook`(1) command 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. The command allows you to display the memory map and CPU usage for a specified process. For each page in the virtual address space of the process, `dlook`(1) generates 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)

Example 4-7 Using `dlook`(1) with a PID

To specify a PID as a parameter to the `dlook`(1) command, you must be the owner of the process, or you must be logged in as the root user. The following `dlook`(1) command example shows output for the `sleep` process, with a PID of 191155:

```
$ dlook 191155
```

```
_____________________________________________________________________________
Peek:  sleep
Pid: 191155 Fri Sep 27 17:14:01 2013


Process memory map:
  00400000-00406000 r-xp 00000000 08:08 262250 /bin/sleep
 [0000000000400000-0000000000401000]        1 page   on node
4  MEMORY|SHARED
 [0000000000401000-0000000000402000]        1 page   on node
5  MEMORY|SHARED
 [0000000000403000-0000000000404000]        1 page   on node
7  MEMORY|SHARED
 [0000000000404000-0000000000405000]        1 page   on node
8  MEMORY|SHARED
```

```
  00605000-00606000 rw-p 00005000 08:08 262250 /bin/sleep
 [0000000000605000-0000000000606000]        1 page   on node
2  MEMORY|RW|DIRTY

  00606000-00627000 rw-p 00000000 00:00 0 [heap]
 [0000000000606000-0000000000608000]        2 pages  on node
2  MEMORY|RW|DIRTY

  7ffff7dd8000-7ffff7ddd000 rw-p 00000000 00:00 0
 [00007ffff7dd8000-00007ffff7dda000]        2 pages  on node
2  MEMORY|RW|DIRTY
 [00007ffff7ddc000-00007ffff7ddd000]        1 page   on node
2  MEMORY|RW|DIRTY

  7ffff7fde000-7ffff7fe1000 rw-p 00000000 00:00 0
 [00007ffff7fde000-00007ffff7fe1000]        3 pages  on node
2  MEMORY|RW|DIRTY

  7ffff7ffa000-7ffff7ffb000 rw-p 00000000 00:00 0
 [00007ffff7ffa000-00007ffff7ffb000]        1 page   on node
2  MEMORY|RW|DIRTY

  7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0 [vdso]
 [00007ffff7ffb000-00007ffff7ffc000]        1 page   on node
7  MEMORY|SHARED

  7ffff7ffe000-7ffff7fff000 rw-p 00000000 00:00 0
 [00007ffff7ffe000-00007ffff7fff000]        1 page   on node
2  MEMORY|RW|DIRTY

  7fffffffea000-7fffffffff000 rw-p 00000000 00:00 0 [stack]
 [00007fffffffd000-00007fffffffff000]        2 pages  on node
2  MEMORY|RW|DIRTY

  ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
 [ffffffffff600000-ffffffffff601000]        1 page   on node
0  MEMORY|DIRTY|RESERVED
```

The dlook(1) command generates the name of the process (Peek: sleep), the process ID, the time, and the date it was invoked. It provides total user and system CPU time in seconds for the process.

Under the Process memory map heading, the dlook(1) command generates 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, it shows the type of access, time of execution, the PID, and the object that owns the memory, which in this example is /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 shows the nodes on which the pages reside. A page is 16,384 bytes.

The node numbers reported by the dlook(1) command correspond to the numbers reported by the cpumap(1) command under the section Processor Numbering on Node(s). For more information, see the cpumap(1) command description in "Determining System Configuration" on page 8.

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

Example 4-8 Using dlook(1) with a command

When you pass a command as an argument to dlook(1), you specify the command and optional command arguments. The dlook(1) command issues an exec call on the command and passes the command arguments. When the process terminates, dlook(1) 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 fffffffffffc000 00:00 0
        [60000fffffff4000-60000fffffffc000]         2 pages on node   3  MEMORY|DIRTY
```

Example 4-9 Using the `dlook`(1) command with the `-s` *secs* option

If you use the `dlook`(1) command with the `-s` *secs* option, the information is sampled at regular internals. The example command and output are as follows:

```
$ dlook -s 5 sleep 50
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
```

Example 4-10 Using the `dlook`(1) command with the `mpirun`(1) command

You can run a Message Passing Interface (MPI) job using the `mpirun`(1) command and generate the memory map for each thread, or you can redirect the ouput to a file.

In the following example, the output has been abbreviated 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 about the dlook(1) command, see the dlook(1) man page.

Performance Tuning

This chapter includes the following topics:

About Performance Tuning

After you analyze 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.

Intel provides tuning information, including information about the Intel processors, at the following website:

http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

This chapter describes the process of tuning your application for a single processor system and then tuning it for parallel processing. It also addresses how to improve the performance of floating-point programs and MPI applications.

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 54.
- Use existing tuned code, such as that found in math libraries and scientific library packages. For details, see "Using Tuned Code" on page 55.
- Determine what needs tuning. For details, see "Determining Tuning Needs" on page 56.
- Use the compiler to do the work. For details, see "Using Compiler Options to Optimize Performance" on page 56.
- Consider tuning cache performance. For details, see "Tuning the Cache Performance" on page 61.
- Set environment variables to enable higher-performance memory management mode. For details, see "Managing Memory" on page 63.

Getting the Correct Results

One of the first steps in performance tuning is to verify that the correct answers are being obtained. After 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:

Option	Purpose
`-O`	Disables all optimization. The default is `-O2`.
`-g`	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`	Lets you specify compiler rules for the following: • Value safety • Floating-point (FP) expression evaluation

	• FPU environment access • Precise FP exceptions • FP contractions The default is `-fp-model fast=1`. Note that `-mp` is an old option and is replaced by `-fp-model`.
`-r`, `-i`	Sets default real, integer, and logical sizes to 8 bytes, which are useful for porting codes from Cray, Inc. systems. This option explicitly declares intrinsic and external library functions.

For information about debugging tools that you can use to verify that correct answers are being obtained, see the following:

"About Debugging" on page 17

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 in which 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 at the following website:

https://software.intel.com/en-us/intel-parallel-studio-xe-support/documentation

Determining Tuning Needs

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

Tool	Purpose
`time`(1)	Obtains an overview of user, system, and elapsed time.
`gprof`(1)	Obtains an execution profile of your program. This is a `pcsamp` profile. Use the `-p` compiler option to enable `gprof` use.
VTune	Monitors performance. 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`	Measures the performance of unmodified executables. This is a `PerfSuite` command-line utility. `psrun` takes as input a configuration XML document that describes the desired measurement. For more information, see the following website: http://perfsuite.ncsa.uiuc.edu/

For information about other performance analysis tools, see Chapter 2, "Performance Analysis and Debugging" on page 7.

Using Compiler Options to Optimize Performance

This topic describes several Intel compiler options that can optimize performance. In addition to the performance options and processor options that this topic describes, the following options might be useful to you:

- The `-help` option displays a short summary of the `ifort` or `icc` options.
- The `-dryrun` option displays the driver tool commands that `ifort` or `icc` generate. This option does not actually perform a compile.

For more information about the Intel compiler options, see the following:

https://software.intel.com/en-us/intel-parallel-studio-xe

Use the following options to help tune performance:

Option	Purpose
`-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 assumes 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 might generate incorrect code.
`-ip`	Generates single file, interprocedural optimization. A related option, `-ipo` generates multifile, interprocedural optimization. Most compiler optimizations work within a single procedure, such as 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 transformations* 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 by the `-opt_report_file` option.

`-prof_gen`, `-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`

Controls information specific to the vectorizer. Intel Xeon series processors support vectorization, which can provide a powerful performance boost.

`-Ofast`

Takes the place of, and is equivalent to specifying, the following options:

```
-ipo -O3 -no-prec-div -static -fp-model fast=2
-xHost
```

`-diag-enable`

Enables the Source Checker, which provides advanced diagnostics based on a detailed analysis of your source code. When enabled, the compiler performs static global analysis to find errors in software that the compiler does not typically detect. This general source code analysis tool is an additional diagnostic to help you debug your programs. You can use source code analysis options to detect the following types of potential errors in your compiled code:

- 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 run 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 can check for obscure cases. In fact, you do not need to run the program for Source Checker because the analysis is performed at compilation time. The only requirement is a successful compilation.

There are limitations to Source Checker analysis. Because the Source Checker does not fully interpret the analyzed program, it can generate so called false-positive messages. This is a fundamental difference between compiler errors and Source Checker errors. In the

case of the source checker, you decide whether the generated error is legitimate and needs to be fixed.

The Intel compilers support additional options that are specific to each processor model. To determine the processor used in your system, examine the contents of the `/proc/cpuinfo` file. For example:

```
# cat /proc/cpuinfo | grep "model name"
model name      : Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
model name      : Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
model name      : Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
.
.
.
```

Use the information in Table 5-1 on page 60 to determine the processor in your SGI system.

Table 5-1 SGI Systems and Intel Processors

SGI System	Intel Processor Model	Intel Processor Code Name
SGI UV 300	Xeon E7-8800 v4 series	Broadwell EX
	Xeon E7-8800 v2 series	Ivy Bridge EX
	Xeon E7-8800 v3 series	Haswell EX
SGI UV 3000	Xeon E5-4600 v3 series	Haswell EP 4S
SGI UV 2000	Xeon E5-4600 series	Sandy Bridge EP 4S
	Xeon E5-4600 v2 series	Ivy Bridge EP 4S
SGI UV 1000	Xeon 7500 series	Nehalem EX
	Xeon E7-8800 series	Westmere EX
SGI ICE or SGI Rackable	Xeon E5-2600 series	Sandy Bridge EP
	Xeon E5-2600 v2 series	Ivy Bridge EP
	Xeon E5-2600 v3 series	Haswell EP

The following list shows the processor-specific compiler options:

Processor Option	Purpose
`-xAVX`	Generates instructions for the Sandy Bridge processors and the Ivy Bridge processors, which support Intel Advanced Vector Extensions (AVX) instructions.
`-xCORE-AVX2`	Generates instructions for Haswell processors, which support Intel AVX2 instructions.
`-xHost`	Generates instructions for the highest instruction set available on the compilation host processor.
`-xSSE4.2`	Generates instructions for Nehalem processors and Westmere processors, which support Intel SSE4.2 instructions.

Tuning the Cache Performance

The processor cache stores recently-used information in a place where it can be accessed quickly. This topic uses the following terms to describe cache performance tuning:

- A *cache line* is the minimum unit of transfer from next-higher cache into this one.
- A *cache hit* is reference to a cache line that is present in the cache.
- A *cache miss* is reference to a cache line that is not present in this cache level and must be retrieved from a higher cache, from memory, or from swap space.
- The *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.
- A *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.

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

- Avoid large power-of-two (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.

For example, assume that a cache in the hierarchy has a size of 256 KB, which is 65536 four-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

The following topics pertain to memory management:

Memory Use Strategies

The following are some general memory use goals and guidelines:

- 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 the data and instructions do not fit in the cache, try to use what is in cache before using anything that is not in cache.
- Data locality. Try to access data that is nearby in memory before attempting to access data that is far away in memory.
- I/O efficiency. Perform a large amount of I/O operations all at once, rather than a little bit at a time. Do not mix calculations and I/O.

Memory Hierarchy Latencies

Memory is not arranged 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.

CPUs that are waiting for memory are not doing useful work. Software should be hierarchy-aware to achieve best performance, so observe the following guidelines:

- 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.

Tuning Multiprocessor Codes

The following topics explain multiprocessor tuning, which consists of the following major steps:

- Perform single processor tuning, which benefits multiprocessor codes also. For information, see "Single Processor Code Tuning" on page 54.
- Determine the parts of your code that can be parallelized. For information, see "Data Decomposition" on page 64.
- Choose the parallelization methodology for your code. For information, see "Measuring Parallelization and Parallelizing Your Code" on page 66.
- Analyze your code to make sure it is parallelizing properly. For information, see Chapter 2, "Performance Analysis and Debugging" on page 7.
- Determine if false sharing exists. False sharing refers to OpenMP, not MPI. For information, see "Fixing False Sharing" on page 69.
- Tune for data placement. For information, see Chapter 4, "Data Process and Placement Tools" on page 29.
- Use environment variables to assist with tuning. For information, see "Environment Variables for Performance Tuning" on page 69.

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. Decompose the data.

     Data decomposition is breaking up the data and mapping data to processors. You, the programmer, can break up the data explicitly by using message passing (with MPI) and data passing (using the SHMEM library routines). Alternatively, you can employ 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. Divide the work among processors.

Measuring Parallelization and Parallelizing Your Code

When tuning for performance, first assess the amount of code that is parallelized in your program. 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.

You can use one of the following to display CPU activity:

- The top(1) command.
- The vmstat(8) command.
- The open source Performance Co-Pilot tools. For example, pmval(1) (pmval kernel.percpu.cpu.user) or the visualization command pmchart(1).

Next, focus on using one of the following parallelization methodologies:

- "Using SGI MPI" on page 66
- "Using OpenMP" on page 67
- "Identifying OpenMP Nested Parallelism" on page 67
- "Using Compiler Options" on page 68
- "Identifying Opportunities for Loop Parallelism in Existing Code" on page 68

Using SGI MPI

The SGI Performance Suite includes the SGI Message Passing Toolkit (SGI MPT). SGI MPT includes both the SGI Message Passing Interface (SGI MPI) and SGI SHMEM. SGI MPI is optimized and more scalable for SGI UV series systems than the generic MPI libraries. SGI MPI takes advantage of the SGI UV architecture and SGI nonuniform memory access (NUMA) features.

Use the -lmpi compiler option to use MPI. For a list of environment variables that are supported, see the mpi(1) man page.

The MPIO_DIRECT_READ and MPIO_DIRECT_WRITE environment variables 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. The mpi_io(3) man page describes these environment variables.

For information about performance tuning for MPI applications, see the following:

- *SGI MPI and SGI SHMEM User Guide*
- *MPInside Reference Guide*

Using 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. OpenMP is included with the C, C++, and Fortran compilers.

To use OpenMP directives, specify the ifort -openmp or icc -openmp compiler options. These options use the OpenMP front-end that is built into the Intel compilers. The latest Intel compiler OpenMP run-time library name is libiomp5.so. The latest Intel compiler also supports the GNU OpenMP library as an either/or option, in other words, do not mix-and-match the GNU library with the Intel version.

For more information, see the OpenMP standard at the following website:

http://www.openmp.org/wp/openmp-specifications

Identifying OpenMP Nested Parallelism

The following Open MP nested parallelism output shows two primary threads and four secondary threads, called master/nested:

```
% 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
```

For more information, see the dplace(1) man page.

Using Compiler Options

You can use compiler options to invoke automatic parallelization. Use the -parallel or -par_report options to the ifort or icc compiler commands. 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 the -parallel option 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 does it analyze while or do while loops. The compiler also does not check for blocks of code that can be run in parallel.

Identifying Opportunities for Loop Parallelism 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.

If you suspect false sharing, take one of the following actions:

- Use the information in the following manual to determine the appropriate hardware performance counter names relevant to false sharing:

  http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

- On SGI UV systems, you can use the `hubstats`(1) command in the SGI Foundation Software suite to verify whether false sharing is occurring.

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 shows the location of the problem.
- Revise data structures or algorithms.
- Check shared data, static variables, common blocks, private variables, and public variables in shared objects.
- Use critical regions to identify the part of the code that has the problem.

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 MPI behavior, 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 environment 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 generates a segmentation fault immediately upon execution, you might 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

You can use multiple CPUs in the following ways:

- Take a conventional program in C, C++, or Fortran, and have the compiler find the parallelism that is implicit in the code.
- Write your source code to use explicit parallelism. In in the source code, specify the parts of the program that you want to execute asynchronously and how the parts are to coordinate with each other.

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

This section covers the following topics:

Adding CPUs to Shorten Execution Time

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

Suppose there is one loop in which the program spends 50% of the execution time. If you can divide the iterations of this loop so that half of them are done in one CPU while the other half are done at the same time in a different CPU, the whole loop can be finished in half the time. The result is 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 are diminishing benefits 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 coding to accomplish the following:

- 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 loop are performed on one CPU, and the other half run at the same time on a second CPU, the whole loop should complete in half the time. For example, consider the typical C loop in Example 5-1.

Example 5-1 Typical C Loop

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

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

The speedup gained from applying *n* CPUs, *Speedup(n)*, is the ratio of the single-CPU execution time to the *n*-CPU execution time: *Speedup*(*n*) = `T`*(1)* ÷ `T`*(n)*. If you measure the single-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 runs faster than the single CPU can. 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 loop. Some amount of code is devoted to setting up the loop and 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.

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

The potential speedup for a program is proportional to *p* divided by the CPUs you can apply, plus the remaining serial part, 1-*p*. As an equation, this appears as Example 5-2.

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

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

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

The reward for parallelization is small unless *p* is substantial (at least 0.8). 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 that includes communication overhead. This version shows that if the program has no serial part and you increase the number of cores, the following occurs:

- The amount of computations per core diminishes.
- The communication overhead increases (unless there is no communication and there is trivial parallelization).
- The efficiency of the code and the speedup diminishes.

The equation is as follows:

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

The preceding equation uses the following variables:

- *n* is the number of processes.
- *a* is the fraction of the given task not dividable into concurrent subtasks.
- *ts* is the time to execute the task in a single processor.
- *tc* is the communication overhead.

If *a*=0 and *tc*=0, there is no serial part and no communication. In this case, as in a trivial parallelization program, you see a 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 5-3.

Example 5-3 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. When this occurs, the formula in Example 5-3 returns a value of *p* greater than 1.0, which is clearly not useful. In this case, you need to calculate *p* from two other more realistic timings, for example T(2) and T(3). The general formula for *p* is shown in Example 5-4, where *n* and *m* are the two CPU counts whose speedups are known, *n*>*m*.

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

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

For more information about superlinear speedups, see the following:

"Understanding Superlinear Speedup" on page 72

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 effect 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 effect 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.

Floating-point Program Performance

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

These slowdowns occur when the hardware fails to complete a floating-point operation and requests help (emulation) from software. This happens, for instance, with denormalized numbers.

The symptoms are a slower than normal execution and an FPSWA message in the system log. Use the `dmesg`(1) to display the message. The average cost of an 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.

About MPI Application Tuning

When you design your MPI application, make sure to include the following in your design:

- The pinning of MPI processes to CPUs
- The isolating of multiple MPI jobs onto different sets of sockets and Hubs

You can achieve this design 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`. For more information about these programming practices, see the following:

- The `MPI_DSM_CPULIST` discussion in the *SGI MPI and SGI SHMEM User Guide.*
- The `omplace`(1) and `dplace`(1) man pages.
- The *SGI Cpuset Software Guide.*

MPI Application Communication on SGI Hardware

On an SGI UV system, the following two transfer methods facilitate MPI communication between processes:

- Shared memory
- The global reference unit (GRU), which is part of the SGI UV Hub ASIC

  The SGI UV series systems use a scalable nonuniform memory access (NUMA) architecture to allow the use of thousands of processors and terabytes of RAM in a single Linux operating system instance. As in other large, shared-memory systems, memory is distributed to processor sockets and accesses to memory are cache coherent. Unlike other systems, SGI UV systems use a network of Hub ASICs connected over NUMALink to scale to more sockets than any other x86-64 system, with excellent performance out of the box for most applications.

  When running on SGI UV systems with SGI's Message Passing Toolkit (MPT), applications can attain higher bandwidth and lower latency for MPI calls than when running on more conventional distributed memory clusters. However, knowing your SGI UV system's NUMA topology and the performance constraints that it imposes can still help you extract peak performance. For more information

about the SGI UV hub, SGI UV compute blades, Intel QPI, and SGI NUMALink, see your SGI UV hardware system user guide.

The MPI library chooses the transfer 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 allocates buffer space and control blocks between the logical processors being used by the MPI job.

MPI Job Problems and Application Design

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, but user tuning might be needed to improve performance. The following are some application performance problems and some ways that you might be able to improve MPI performance:

- Primary HyperThreads are idle.

  Most high-performance computing MPI programs run best when they use only one HyperThread per core. When an SGI UV system has multiple HyperThreads per core, logical CPUs are numbered such that primary HyperThreads are the high half of the logical CPU numbers. Therefore, the task of scheduling only on the additional HyperThreads may be accomplished by scheduling MPI jobs as if only half the full number exists, 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 command's output includes the following:

  – The number of physical and logical processors

  – Whether HyperThreading is enabled

  – How shared processors are paired

  If an MPI job uses only half of the available logical CPUs, set `GRU_RESOURCE_FACTOR` to 2 so that the MPI processes can use 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.

- Message bandwidth is inadequate.

  Use either huge pages or transparent huge pages (THP) to ensure that your application obtains optimal message bandwidth.

  To specify the use of hugepages, use the `MPI_HUGEPAGE_HEAP_SPACE` environment variable. The `MPI_HUGEPAGE_HEAP_SPACE` environment variable defines the minimum amount of heap space that each MPI process can allocate using huge pages. For information about this environment variable, see the `MPI`(1) man page.

  To use THPs, see "Using Transparent Huge Pages (THPs) in MPI and SHMEM Applications" on page 80.

  Some programs transfer large messages via the `MPI_Send` function. To enable unbuffered, single-copy transport in these cases, you can set `MPI_BUFFER_MAX` to `0`. For information about the `MPI_BUFFER_MAX` environment variable, see the `MPI`(1) man page.

- 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`. For more information, see the `MPI`(1) man page.

- Memory allocations are nonlocal.

  MPI application processes normally perform best if their local memory is allocated near the socket assigned to use 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 `bcfree` commands in the job prologue.

  For information about PBS Professional, including the availablilty of scripts, see the PBS Professional documentation and the `bcfree`(1) man page.

MPI Performance Tools

SGI supports several MPI performance tools. You can use the following tools to enhance or troubleshoot MPI program performance:

- MPInside. MPInside is an MPI profiling tool that can help you optimize your MPI application. The tool provides information about data transferred between ranks, both in terms of speed and quantity.
- 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's large, shared-memory systems that can take advantage of the SGI UV Hub.
- SGI PerfCatcher. SGI PerfCatcher uses a wrapper library to return MPI and SHMEM function profiling information. The information returned includes percent CPU time, total time spent per function, message sizes, and load imbalances. For more information, see the following:

For more information about the MPI performance tools, see the following:

- The `MPInside`(1) man page.
- The `perfcatch`(1) man page
- *MPInside Reference Guide*
- *SGI MPI and SGI SHMEM User Guide*

Using Transparent Huge Pages (THPs) in MPI and SHMEM Applications

On SGI UV systems, THP is important because it contributes to attaining the best GRU-based data transfer bandwidth in Message Passing Interface (MPI) and SHMEM programs. On newer kernels, the THP feature is enabled by default. If THP is disabled on your SGI UV system, see "Enabling Huge Pages in MPI and SHMEM Applications on Systems Without THP" on page 82.

On SGI ICE systems, if you use a workload manager, such as PBS Professional, your site configuration might let you enable or disable THP on a per-job basis.

The THP feature can affect the performance of some OpenMP threaded applications in a negative way. For certain OpenMP applications, some threads in some shared data structures might be forced to make more nonlocal references because the application assumes a smaller, 4–KB page size.

The THP feature affects users in the following ways:

- Administrators:

  To activate the THP feature on a system-wide basis, write the keyword `always` to the following file:

  `/sys/kernel/mm/transparent_hugepage/enabled`

  To create an environment in which individual applications can use THP if memory is allocated accordingly within the application itself, type the following:

  ```
  # echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  To disable THP, type the following command:

  ```
  # echo never > /sys/kernel/mm/transparent_hugepage/enabled
  ```

  If the `khugepaged` daemon is taking a lot of time when a job is running, then defragmentation of THP might be causing performance problems. You can type the following command to disable defragmentation:

  ```
  # echo never > /sys/kernel/mm/transparent_hugepage/defrag
  ```

  If you suspect that defragmentation of THP is causing performance problems, but you do not want to disable defragmentation, you can tune the `khugepaged` daemon by editing the values in `/sys/kernel/mm/transparent_hugepage/khugepaged`.

- MPI application programmers:

  To determine whether THP is enabled on your system, type the following command and note the output:

  ```
  % cat /sys/kernel/mm/transparent_hugepage/enabled
  ```

  The output is as follows on a system for which THP is enabled:

  ```
  [always] madvise never
  ```

  In the output, the bracket characters (`[ ]`)appear around the keyword that is in effect.

Enabling Huge Pages in MPI and SHMEM Applications on Systems Without THP

If the THP capability is disabled on your SGI UV system, you can use the `MPI_HUGEPAGE_HEAP_SPACE` environment variable and the `MPT_HUGEPAGE_CONFIG` command to create huge pages.

The `MPT_HUGEPAGE_CONFIG` command configures the system to allow huge pages to be available to MPT's memory allocation interceptors. The `MPI_HUGEPAGE_HEAP_SPACE` environment variable enables an application to use the huge pages reserved by the `MPT_HUGEPAGE_CONFIG` command.

For more information, see the `MPI_HUGEPAGE_HEAP_SPACE` environment variable on the `MPI`(1) man page, or see the `mpt_hugepage_config`(1) man page.

Flexible File I/O

This chapter covers the following topics:

About FFIO

Flexible File I/O (FFIO) can improve the file I/O performance of existing applications without having to resort to source code changes. 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.

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 such as open, read, and write, and replaces them with FFIO equivalent routines. These routines route I/O requests through the FFIO subsystem, which uses 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 bypass 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 (for example, 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

To use FFIO, set one of the following environment variables: `LD_PRELOAD` or `FF_IO_OPTS`.

In order to enable FFIO to trap standard I/O calls, set the `LD_PRELOAD` environment variable, as follows:

```
# 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 that replace the standard I/O calls. An application that is not dynamically linked with the `glibc` library cannot work with FFIO because the standard I/O calls cannot be intercepted. To disable FFIO, type 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 format 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.

Example 1. Assume that you want a shared buffer cache of 128 pages. Each page is to be 16 megabytes (that is, 4096*4k). The cache has a lead of six pages and uses a stride of one. The command is 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. For example, file names of `/tmp/test16` and `/var/tmp/testit` both match.

Example 2. This more complicated usage of `FF_IO_OPTS` builds upon the previous example. Multiple types of file names can share the same cache, as the following example shows:

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

Example 3. You can specify multiple caches 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. The command, which uses the backslash (\) continuation character, is 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. For examples of this diagnostic output, see the following:

"Simple Examples" on page 85

Simple Examples

This topic includes some simple FFIO examples. 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)'
```

It can be difficult to tell what FFIO might or might not be doing even with a simple program. The examples in this topic use a small C program called `fio` that reads

4–megabyte chunks from a file for 100 iterations. When the program runs, it produces the following output:

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

Example 1. You can direct a simple FFIO operations summary to standard output by making the following simple addition to FF_IO_OPTS:

```
% 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 runs:

```
% ./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. In addition, there were 29 asynchronous reads (reada) issued, for a total of 464 megabytes.

Example 2. You can generate additional diagnostic information by specifying the `.diag` modifier. The following is an example of the diagnostic output generated when the `.diag` modifier is used:

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

The preceding output lists information for both the file and the cache. In the `mbytes transferred` information, the lines in **bold** are for write and read operations, respectively. Only for very simple I/O patterns can the difference between (`parent --> eie`) and (`eie --> child`) read statistics 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`.

You can generate additional diagnostic information by specifying the `.diag` modifier and the `.event.summary` modifier. The two modifiers operate independently from one another. The following specification uses both modifiers:

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

Multithreading Considerations

FFIO works 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 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 you 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 as shared, all files within the same ranking thread can 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. To accomplished this, set one, and only one, of the following environment variables:

- `setenv SGI_MPI /usr/lib`
- `setenv LAM_MPI`
- `setenv HP_MPI`

Note: LAM MPI and HP MPI are usually distributed via a third party application. The precise paths to the LAM and the HP MPI libraries are application dependent. See the application installation guide to find the correct path.

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 generate a no match with FFIO. This means that none of the rank 2 files are cached by FFIO. In this case, the software does not default to FF_IO_OPTS.

Fortran and C/C++ applications that use the pthreads interface 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 high-performance computing 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 as follows:

```
% 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

If you specify the `.trace` option as part of the `event` parameter, you can enable the event tracing feature in FFIO.

For example:

```
% 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`. To chang this destination, set 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 such as I/O sizes and offsets.

System Information and Issues

The FFIO subsystem supports applications written in C, C++, and Fortran. C and C++ applications can be built with either the Intel or gcc compiler. Only Fortran codes built with the Intel compiler work with FFIO.

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 for I/O on a socket.
- Do not link your application with the `librt` asynchronous I/O library.
- FFIO does not intercept calls that operate on files in `/proc`, `/etc`, and `/dev`.
- FFIO does not intercept calls that operate on `stdin`, `stdout`, and `stderr`.
- FFIO is not intended for generic I/O applications such as `vi`, `cp`, or `mv`, and so on.

I/O Tuning

This chapter contains the following topics:

About 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, assume 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
```

To run an OpenGL graphics program, such as glxgears(1), on the third graphics processing unit using numactl(8), type 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.

The -N parameter specifies to run the command on node 14. The -m parameter specifies to allocate memory only from node 14.

You could also use the dplace(1) command to place the application.

For information about the dplace(1) command, see the following:

"dplace Command" on page 35

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 request completion waits for all of the striped pieces to complete.

The latency spikes' impact on throughput can be to stall all the I/O requests or to delay a few I/O requests while others continue. It depends on how the I/O request is striped across the devices. If the volumes are constructed as stripes to span all devices and the I/O requests are sized to be full stripes, the I/O requests stall because every I/O request has to touch every device. If the I/O requests 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/O requests 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/O requests 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. These are as follows:

1. Transient disk delays - one disk pauses
2. Slow disks
3. 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 includes the following topics:

Determining Process Placement

This topic describes methods that you can use 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. Note that all examples use the C shell.

The following procedure explains how to set up the computing environment.

Procedure 8-1 To create the computing environment

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

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

   The `pu` command alias shows current processes.

2. Create the `.toprc` preferences file in your login directory to set the appropriate `top`(1) options.

   If you prefer to use the `top`(1) defaults, delete the `.toprc` file.

   ```
   % cat <<EOF>> $HOME/.toprc
   YEAbcDgHIjklMnoTP|qrsuzV{FWX
   2mlt
   EOF
   ```

3. Inspect all processes, determine which CPU is in use, and create an alias file for this procedure.

The CPU number appears in the first column of the top(1) 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
```

The following topics present examples:

- "Example Using pthreads" on page 96
- "Example Using OpenMP" on page 98

Example Using `pthreads`

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

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

Use the `pu` alias and the `top`(1) command to see the output, as follows:

```
% 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, which is as follows:

1. Place 1, skip 2 of them, place 3 more `[ 0 1 1 0 0 0 ]`.
2. Reverse the bit order and create the `dplace`(1) `-x` mask:

   ```
   [ 0 0 0 1 1 0 ]  -->  [ 0x06 ]  --> decimal 6
   ```

   The `dplace`(1) command does not currently process hexadecimal 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     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
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

% 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
```

Resetting System Limits

To regulate these limits on a per-user basis for applications that do not rely on `limit.h`, you can modify the `limits.conf` file. The system limits that you can modify include maximum file size, maximum number of open files, maximum stack size, and so on. To view this file, type the following:

```
[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 information about how to change these limits, see "Resetting the File Limit Resource Default" on page 101.

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 might 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`, you can modify the `limits.conf` file. 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.

The following procedure explains how to change the `/etc/pam.d/login` file.

Procedure 8-2 To change the file limist resource default

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 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 of 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 do 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 101 and add the following lines to the `/etc/security/limits.conf` file:

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

These lines set 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 increase the soft limit, you can modify your shell's startup script. For example, if your login shell is `bash`, add a line similar to the following to your `.bashrc` file:

```
% ulimit -s 300000
```

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

For more information on default settings, see "Resetting the File Limit Resource Default" on page 101.

Avoiding Segmentation Faults

The default stack size in the Linux operating system is 8MB (8192 kbytes). This value often needs to be increased to avoid segmentation fault errors. If your application fails to run immediately, check the stack size.

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, use 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, increase the KMP_STACKSIZE to a larger size. The default size in Intel Compilers is 4MB.

For example, to increase it to 64MB, use the following commands:

- In the C shell, set the environment variable as follows:

  ```
  setenv KMP_STACKSIZE 64M
  ```

- In the Bash shell, set the environment variable as follows:

  ```
  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 using the Bash shell, use commands such as the following when setting this limit:

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

If you are using the C shell, use commands such as the following when setting this limit:

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

For example. The following MPI program fails with a memory-mapping error because of a virtual memory parameter, `vmemoryuse`, value that is 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
```

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 that erroneously detect memory quota violations.

The `get_weighted_memory_size` function weighs shared memory regions by the number of processes using the regions. Thus, if 100 processes each share 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 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);
```

The call returns the weighted memory (RSS) size for a *pid*, in bytes. This call weights the size of the shared regions by the number of processes accessing the region. Returns -1 when an error occurs and sets 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`	Cannot 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 SHMEM

You can specify the maximum number of queue pairs (QPs) for SHMEM applications when run on large clusters over an OFED fabric, such as InfiniBand. If the log_num_qp parameter is set to a number that is too low, the system generates the following message:

```
MPT Warning: IB failed to create a QP
```

SHMEM codes use the InfiniBand RC protocol for communication between all pairs of processes in the parallel job, which requires a large number of QPs. The log_num_qp parameter defines the $\log_2$ of the number of QPs. The following procedure explains how to specify the log_num_qp parameter.

Procedure 8-3 To specify the log_num_qp parameter

1. Log into one of the hosts upon which you installed the MPT software as the root user.
2. Use a text editor to open file /etc/modprobe.d/libmlx4.conf.
3. Add a line similar to the following to file /etc/modprobe.d/libmlx4.conf:

   ```
   options mlx4_core log_num_qp=21
   ```

   By default, the maximum number of queue pairs is 2^{18} (262144). This is true across all platforms (RHEL 7.1, RHEL 6.6, SLES 12, and SLES 11SP3).
4. Save and close the file.
5. Repeat the preceding steps on other hosts.

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 GC threads to start, but the number is generally 0.5 to 1 times the number of CPUs, which is appropriate on a 1– or 2–socket system.

However, this strategy does not scale well to systems with a larger core count. Java command line options let you control the number of GC threads that the Java virtual machine (JVM) will use. In many cases, a single GC thread is sufficient. In other cases, a larger number might 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 example:

- For Oracle Java:

  ```
  -XX:ParallelGCThreads
  ```

- For IBM Java:

  ```
  -Xgcthreads
  ```

An example command line option:

```
java -XX:+UseParallelGC -XX:ParallelGCThreads=1
```

As an administrator, you might choose 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. The environment variable settings are as follows:

- For Oracle Java:

  ```
  JAVA_OPTIONS="-XX:ParallelGCThreads=1"
  ```

- For IBM Java:

  ```
  IBM_JAVA_OPTIONS="-Xgcthreads1"
  ```

Index