Table of Contents

1. Introduction .................................................................................................................. 3
2. x86 - Basic Properties ................................................................................................... 3
   2.1. Basic Properties ...................................................................................................... 3
   2.2. Simultaneous Multithreading .................................................................................. 4
3. Programming Environment ............................................................................................. 5
   3.1. Modules ................................................................................................................... 5
   3.2. Compiling ................................................................................................................ 6
      3.2.1. Compilers .......................................................................................................... 6
      3.2.2. General Compiler Flags .................................................................................. 6
         3.2.2.1. GCC ............................................................................................................. 6
         3.2.2.2. Intel .............................................................................................................. 7
         3.2.2.3. Portland ....................................................................................................... 8
         3.2.2.4. PathScale .................................................................................................... 9
   3.3. MPI .......................................................................................................................... 9
      3.3.1. MVAPICH2 ...................................................................................................... 10
      3.3.2. Open MPI ........................................................................................................ 10
      3.3.3. Intel MPI ........................................................................................................ 12
      3.3.4. Platform MPI .................................................................................................. 13
   3.4. OpenMP .................................................................................................................. 14
      3.4.1. Compiler Flags ............................................................................................... 14
   3.5. Hybrid Programming .............................................................................................. 15
   3.6. Math Libraries ....................................................................................................... 16
      3.6.1. GSL (GNU Scientific Library) ........................................................................ 16
      3.6.2. MKL (Intel Math Kernel Library) ....................................................................... 18
      3.6.3. ACML (AMD Core Math Library) ................................................................... 19
      3.6.4. FFTW ............................................................................................................... 20
4. Tuning ............................................................................................................................. 20
   4.1. Optimisation ............................................................................................................ 20
      4.1.1. IEEE 754 Compliant Optimisation .................................................................... 20
      4.1.2. Compiler Optimisation Flags .......................................................................... 21
   4.2. Runtime Environment Settings ............................................................................. 25
      4.2.1. MPI Runtime Environment Variables .............................................................. 25
      4.2.2. OpenMP Runtime Environment Variables ....................................................... 28
5. Debugging ....................................................................................................................... 30
   5.1. Compiler Debug Flags and Environment Variables .............................................. 30
   5.2. Debuggers ............................................................................................................... 32
      5.2.1. GNU GDB ........................................................................................................ 32
      5.2.2. Totalview ......................................................................................................... 34
      5.2.3. DDT .................................................................................................................. 36
6. Performance Analysis ..................................................................................................... 40
   6.1. MPI Statistics ......................................................................................................... 40
   6.2. Tools ....................................................................................................................... 43
      6.2.1. GNU gprof ....................................................................................................... 43
      6.2.2. Oprofile .......................................................................................................... 43
      6.2.3. IPM .................................................................................................................. 47
      6.2.4. Scalasca .......................................................................................................... 48
      6.2.5. TAU ................................................................................................................. 50
      6.2.6. VTune ............................................................................................................. 52
7. Batch Systems ............................................................................................................... 54
   7.1. PBS Pro .................................................................................................................. 54
   7.2. Platform LSF ......................................................................................................... 57
   7.3. LoadLeveler ............................................................................................................ 59
1. Introduction

This guide provides an overview of best practices on using x86 HPC cluster systems. Topics discussed include programming environment, e.g. choice of compiler options and numerical libraries, runtime environment settings, debuggers, performance analysis tools and running jobs in batch systems.

The guide is not system-specific, i.e. it will not cover information that is targeted for one specific machine system but instead focus on topics that will be common to systems based on the x86 architecture. To get details about architectures and configurations for specific systems users are referred to system user guides, some of which can be found at http://www.prace-ri.eu/Best-Practice-Guides.

The guide is organised into these parts:

- Chapter 2 gives a general overview of the x86 architecture and its properties.
- Chapter 3 describes the environment setup through the modules system used when porting and developing codes. A selection of compilers and a list of common and useful options are described. The chapter discusses four of the available MPI implementations and their general usage when building parallel applications, and job launcher commands with options supported. Included is also information for compiling OpenMP codes, a section describing hybrid programming, and a coverage of math libraries used when building applications.
- Chapter 4 describes ways of improving application performance using compiler optimisation flags, and runtime tuning through MPI and OpenMP environment variables.
- Chapter 5 discusses the process of debugging codes using compiler debug flags and environment variables, and using command line and GUI debuggers.
- Chapter 6 describes how to analyse code behaviour through MPI summary statistics, and to find the source of performance problems using profilers and analysis tools.
- Chapter 7 covers some of the systems used for running batch jobs describing commands, job resources, and including sample batch scripts.

2. x86 - Basic Properties

2.1. Basic Properties

x86 is a instruction set architecture first introduced by Intel in 1978 in their 8086 CPU. The term x86 derives from the main CPU in the first succeeding processor generations that had names ending in "86". The instruction set length of the 8086 CPU was 16 bits. This was extended to 32 bits in the 80386 CPU by Intel in 1985, and later to 64 bits by AMD in their Opteron processor in 2003. The 64-bit extension architecture is also known as x86-64.

The x86 designs are superscalar allowing for instruction-level parallelism (ILP) where several independent instructions can be executed per clock cycle. The efficiency of superscalar CPUs can be additionally improved using simultaneous multithreading (SMT), see description below. x86 processors are also capable of out of order and speculative execution, i.e. instructions are executed in an order governed by the availability of input data rather than the original order in a program, and instructions can be carried out, if resources are available, before it is known if the work is needed. The x86 architecture has byte-addressing enabled and words are stored in memory with little-endian byte order.

Modern x86 processors include SIMD (Single Instruction, Multiple Data) units where instructions can operate in parallel on multiple data objects. These are implemented in the 3DNow! instruction set extension developed by AMD in 1997 and the Streaming SIMD Extensions (SSE) instruction sets first introduced by Intel in 1999. The SIMD registers allow for loading or storing 128 bits of memory data in one single instruction that can be operated on in parallel by vector instructions. Later extensions to SSE (introduced in SSE3) included thread-handling instructions to improve the performance of Intel's Hyper-Threading Technology (see below). With the introduction of Intel's SandyBridge processors registers were doubled to 256 bits, and extended functionality was added through the AVX (Advanced Vector Extensions) instruction set.¹

2.2. Simultaneous Multithreading

Simultaneous Multithreading (SMT) is a technique to increase the throughput of a conventional superscalar processor by multiplexing the execution of concurrent hardware threads. The key idea is to fill up the empty slots in the processor pipeline that arise when an application thread has low instruction-level parallelism (ILP) or suffers from long-latency stalls (e.g. cache misses, branch mispredictions), with instructions from an alternate thread that are ready to execute. Most resources in an SMT processor (e.g., caches, functional units, instruction queues) are either shared or partitioned between threads, while only those that are necessary to keep the architectural state for each thread (e.g. architectural registers) are replicated.

The hardware threads within an SMT processor, often referred to as logical processors, are exposed to the operating system as conventional processors. Hyper-Threading (HT) Technology is Intel's 2-way SMT implementation, which makes a single physical processor appear to the software as two logical processors. This is considered to be a cost-effective approach to provide hardware concurrency, since the additional cost in terms of hardware overhead and energy consumption is rather small (e.g., less than 5% in early HT implementations, as compared to their non-HT counterparts). However, due to the fact that hyper-threads operate mostly under an "all-shared", non-dedicated environment, there are performance implications that one needs to take into account when mapping a parallel application to the underlying architecture. In the following, we list the most important of them:

- The large degree of sharing in functional units often leads to contention between threads. This results to performance degradation as compared to the case when threads do not share any resources. In general, this slowdown depends on the profile of the application, but as a rule-of-thumb we can argue that the speedup boost that one can expect from HT ranges between 30%-50%.

- Resource contention, and hence efficiency of HT, highly depend on the heterogeneity of the application threads that are being co-scheduled on the hyper-threaded processor. Generally, it is decreased when threads have complementary needs for functional units, e.g. when we co-execute a memory-intensive thread with a computation-intensive thread, an integer-bound thread with a fp-bound thread, a high-ILP thread with a low-ILP one. Contention increases when threads compete for the same unit at the same time.

- To maximize the gain of a parallel application from HT, one can employ non-conventional parallelization techniques that target at increasing threads heterogeneity. Such techniques can be:
  - assigning communication operations in an MPI application to a separate thread, and pairing the computation and communication threads on the same HT processor.
  - assigning threads with imbalanced workloads (e.g. busy threads and threads with large idle periods) on the same HT processor
  - using prefetching helper threads that improve memory performance of the main computation thread on the same HT processor

To detect Hyper-Threading Technology and find which logical processors reside in the same hyper-threaded core (sibling threads), one can use the sys pseudo file system (sysfs) provided by the Linux kernel to publish machine topology. The directories and files that are important for extracting HT-related information are summarized in the following table.

Table 1. CPU Topology Info

<table>
<thead>
<tr>
<th>Directory / File</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>/sys/devices/system/cpu/cpuX/</td>
<td>Contains information regarding CPU with id &quot;X&quot; (the unique processor id as seen by the OS).</td>
</tr>
<tr>
<td>/sys/devices/system/cpu/cpuX/topology/</td>
<td>Details on X's topology.</td>
</tr>
<tr>
<td>/sys/devices/system/cpu/cpuX/topology/thread_siblings_list</td>
<td>In an architecture with multi-threaded cores (e.g. Intel Hyper-Threading), shows which other CPUs are in the same core with X, as different hardware threads.</td>
</tr>
</tbody>
</table>
3. Programming Environment

3.1. Modules

Software installations on computer systems span many applications, and many different version numbers of the same application. It is not possible (nor desirable) to use them all at the same time, since different versions of the same application may conflict with each other.

In order to simplify the control of which application versions are available in a specific session the Modules package provides for the dynamic modification of the user’s environment via modulefiles (or modules). Each modulefile contains the information needed to configure the environment for an application. The module command is used for interpreting modulefiles and will alter or set shell environment variables such as PATH, LIBRARY_PATH, etc.

The module command uses various sub-commands that perform different actions. A subset of available sub-commands is listed in the table below.

**Table 2. Selected module Sub-commands**

<table>
<thead>
<tr>
<th>Sub-command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>Print the list of available sub-commands.</td>
</tr>
<tr>
<td>avail</td>
<td>List available modules.</td>
</tr>
<tr>
<td>list</td>
<td>List loaded modules.</td>
</tr>
<tr>
<td>show &lt;module&gt;</td>
<td>Display the full path of &lt;module&gt; and the environment changes made if loaded.</td>
</tr>
<tr>
<td>load &lt;module&gt;</td>
<td>Load &lt;module&gt; into the shell environment.</td>
</tr>
<tr>
<td>unload &lt;module&gt;</td>
<td>Remove &lt;module&gt; from the shell environment.</td>
</tr>
<tr>
<td>switch &lt;module1&gt; &lt;module2&gt;</td>
<td>Unload &lt;module1&gt; and load &lt;module2&gt;.</td>
</tr>
<tr>
<td>purge</td>
<td>Unload all loaded modules.</td>
</tr>
</tbody>
</table>

For example, check available software packages installed (as modules) on the system:

```
$ module avail
adf/2012.01(default)          adf/2012.01a
ansys/13.0                    boost/1.46.1
fftw/2.1.5                    fftw/3.3(default)
```

When more than one version of a software package is installed one of the modules will be suffixed with "(default)". If no version number is specified for a given software this is the modulefile acted on by module. E.g.

```
$ module load adf
```

will load the adf/2012.01 version of the ADF package from the list in the example above.

To display information about a software package use e.g.

```
$ module show fftw/2.1.5
```

```
/path/to/modulefiles/fftw/2.1.5:
module-whatis    library for computing discrete the Fourier transform
prepend-path     CPATH /path/to/fftw/fftw-2.1.5/include
prepend-path     FPATH /path/to/fftw/fftw-2.1.5/include
prepend-path     LIBRARY_PATH /path/to/fftw/fftw-2.1.5/lib
prepend-path     LD_LIBRARY_PATH /path/to/fftw/fftw-2.1.5/lib
append-path      MANPATH /path/to/fftw/fftw-2.1.5/share/man
```

Loads FFTW version 2.1.5
3.2. Compiling

3.2.1. Compilers

Typically, several compilers will be available on most computer systems. The compilers will be accessed by loading the appropriate module file. This guide will cover the compiler suites available from GCC, Intel, Portland and PathScale for the Fortran and C/C++ languages. The table below summarizes invocation commands for the different compilers.

Table 3. Compilers

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Language</th>
<th>Invocation command</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>C</td>
<td>gcc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>g++</td>
</tr>
<tr>
<td></td>
<td>Fortran 77 / Fortran 95</td>
<td>gfortran</td>
</tr>
<tr>
<td>Intel</td>
<td>C</td>
<td>icc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>icpc</td>
</tr>
<tr>
<td></td>
<td>Fortran 77 / Fortran 95</td>
<td>ifort</td>
</tr>
<tr>
<td>Portland</td>
<td>C</td>
<td>pgcc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>pgCC</td>
</tr>
<tr>
<td></td>
<td>Fortran 77</td>
<td>pgf77</td>
</tr>
<tr>
<td></td>
<td>Fortran 95</td>
<td>pgf95</td>
</tr>
<tr>
<td>PathScale</td>
<td>C</td>
<td>pathcc</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>pathCC</td>
</tr>
<tr>
<td></td>
<td>Fortran 77 / Fortran 95</td>
<td>pathf95</td>
</tr>
</tbody>
</table>

3.2.2. General Compiler Flags

The following table shows options that are common to all compilers listed in Table 3, “Compilers”

Table 4. Common Compiler Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Compile or assemble the source files, but do not link.</td>
</tr>
<tr>
<td>-o filename</td>
<td>Name the output file filename.</td>
</tr>
<tr>
<td>-g</td>
<td>Produces symbolic debug information.</td>
</tr>
<tr>
<td>-pg</td>
<td>Generate extra code to write profile information suitable for the analysis program gprof (see Section 6.2.1, “GNU gprof”).</td>
</tr>
<tr>
<td>-Dname</td>
<td>Predefine name as a macro for the preprocessor, with definition 1.</td>
</tr>
<tr>
<td>-I dir</td>
<td>Specifies an additional directory dir to search for include files.</td>
</tr>
<tr>
<td>-l library</td>
<td>Search the library named library when linking.</td>
</tr>
<tr>
<td>-L dir</td>
<td>Search for libraries in a specified directory dir.</td>
</tr>
<tr>
<td>-fpic</td>
<td>Generate position-independent code.</td>
</tr>
<tr>
<td>-mcmode=medium</td>
<td>Generate code for models with global and static data larger than 2GB.</td>
</tr>
</tbody>
</table>

3.2.2.1. GCC

The GNU Compiler Collection [http://gcc.gnu.org/] (GCC) includes compilers for C, C++ and Fortran, and libraries for these languages on a variety of platforms including x86. GCC offers:
• Features from Fortran 2003 and Fortran 2008
• Partial support for ANSI/ISO C99
• Support for the ISO/ANSI C++ standard and partial C++11 compatibility
• OpenMP 3.1 support

In addition to the compiler options listed in Table 4, “Common Compiler Options” the table below lists a set of selected options available for the GCC compilers.

### Table 5. General GCC Compiler Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--version</td>
<td>Display the version number.</td>
</tr>
<tr>
<td>--help=&lt;category&gt;</td>
<td>List a description of options belonging to a specific &lt;category&gt;. E.g. --help=warnings displays options controlling warning messages.</td>
</tr>
<tr>
<td>-std=&lt;val&gt;</td>
<td>Conform to a specific language standard &lt;val&gt;. (C/C++ only)</td>
</tr>
<tr>
<td>-###</td>
<td>Print commands to run during stages of compilation without executing them.</td>
</tr>
<tr>
<td>-E</td>
<td>Preprocessed source files including #line directives are output to stdout.</td>
</tr>
<tr>
<td>-E -P</td>
<td>Inhibit generation of #line directives in the output from the preprocessing.</td>
</tr>
<tr>
<td>-J dir</td>
<td>Save/search for module files in directory dir. (Fortran only)</td>
</tr>
<tr>
<td>-static</td>
<td>Link libraries statically. Dynamic linking is the default.</td>
</tr>
<tr>
<td>-fdefault-integer-8</td>
<td>Set the default integer and logical types to 8 byte. (Fortran only)</td>
</tr>
<tr>
<td>-fdefault-real-8</td>
<td>Set the default real type to 8 byte. This will also promote the default width of &quot;DOUBLE PRECISION&quot; to 16 bytes. (Fortran only)</td>
</tr>
<tr>
<td>-fdefault-double-8</td>
<td>Set the &quot;DOUBLE PRECISION&quot; type to 8 byte. (Fortran only)</td>
</tr>
</tbody>
</table>

For optimising codes using the GCC compilers see GCC Optimisation Flags, and for debugging see Section 5.1, “Compiler Debug Flags and Environment Variables”.

### Further Information

• For more information using the GNU Compiler Collection see the GCC online documentation [http://gcc.gnu.org/onlinedocs](http://gcc.gnu.org/onlinedocs) page.

### 3.2.2.2. Intel


• Features from Fortran 2003 and Fortran 2008
• ANSI/ISO C99 and C++ standards
• OpenMP 3.1

In addition to the compiler options listed in Table 4, “Common Compiler Options” the table below lists a set of selected options available for the Intel compilers.

### Table 6. General Intel Compiler Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-v</td>
<td>Display the version number.</td>
</tr>
<tr>
<td>-help &lt;category&gt;</td>
<td>List a description of options belonging to a specific &lt;category&gt;. Specify -help help to see a list of available categories.</td>
</tr>
<tr>
<td>-std=&lt;val&gt;</td>
<td>Conform to a specific language standard &lt;val&gt;. (C/C++ only)</td>
</tr>
<tr>
<td>-dryrun</td>
<td>Driver tool commands will be shown but not executed.</td>
</tr>
</tbody>
</table>
Best Practice Guide - Generic x86

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-E</td>
<td>Preprocessed source files including #line directives are output to stdout.</td>
</tr>
<tr>
<td>-EP</td>
<td>Inhibit generation of #line directives in the output from the preprocessing.</td>
</tr>
<tr>
<td>-static</td>
<td>Link libraries statically. Dynamic linking is the default.</td>
</tr>
<tr>
<td>-integer-size &lt;size&gt;</td>
<td>Specifies the default KIND for integer and logical variables. The default &lt;size&gt; is 32 for 4 bytes long types. Specify 16 for 2 bytes or 64 for 8 bytes long types. (Fortran only)</td>
</tr>
<tr>
<td>-real-size &lt;size&gt;</td>
<td>Specifies the default KIND for real and complex declarations. The default &lt;size&gt; is 32 for 4 bytes long types. Specify 64 for 8 bytes or 128 for 16 bytes long types. This option will not effect the default size of DOUBLE PRECISION variables. (Fortran only)</td>
</tr>
<tr>
<td>-double-size &lt;size&gt;</td>
<td>Specifies the default KIND for DOUBLE PRECISION and DOUBLE COMPLEX variables. The default &lt;size&gt; is 64 for 8 bytes long types. Specify 128 for 16 bytes long types. (Fortran only)</td>
</tr>
<tr>
<td>-fp-model &lt;keyword&gt;</td>
<td>Controls the semantics of floating-point calculations. See the compiler man page for possible values of &lt;keyword&gt;.</td>
</tr>
</tbody>
</table>

For optimising codes using the Intel compilers see Intel Optimisation Flags, and for debugging see Section 5.1, “Compiler Debug Flags and Environment Variables”.

Further Information


3.2.2.3. Portland

The Portland Group [http://www.pgroup.com] (PGI) compiler suite is a commercial software product containing compilers for Fortran and C/C++. The PGI compilers offer full support for:

- Fortran 2003
- ANSI C99 with K&R extensions
- ISO/ANSI and GNU standards for C++
- OpenMP 3.0

In addition to the compiler options listed in Table 4, “Common Compiler Options” the table below lists a set of selected options available for the PGI compilers.

Table 7. General Portland Compiler Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-V</td>
<td>Display compiler version information.</td>
</tr>
<tr>
<td>-help=&lt;category&gt;</td>
<td>Display options belonging to &lt;category&gt;. Specify -help=groups to see a list of available categories.</td>
</tr>
<tr>
<td>-c89, -c99</td>
<td>Conform to the C89 or C99 language. (C/C++ only)</td>
</tr>
<tr>
<td>-dryrun</td>
<td>Print commands to run during stages of compilation without executing them.</td>
</tr>
<tr>
<td>-E</td>
<td>Preprocessed source files including #line directives are output to stdout.</td>
</tr>
<tr>
<td>-module dir</td>
<td>Save/search for module files in directory dir. (Fortran only)</td>
</tr>
<tr>
<td>-Bdynamic</td>
<td>Link libraries dynamically. This is default.</td>
</tr>
<tr>
<td>-Bstatic</td>
<td>Link libraries statically.</td>
</tr>
<tr>
<td>-i8</td>
<td>Specifies the default size for integer variables to be 8 bytes. (Fortran only)</td>
</tr>
</tbody>
</table>
Best Practice Guide - Generic x86

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-r8)</td>
<td>Specifies the default size for real variables to be 8 bytes. This option will not affect the default size of DOUBLE PRECISION variables. <em>(Fortran only)</em></td>
</tr>
</tbody>
</table>

For optimising codes using the PGI compilers see Portland Optimisation Flags, and for debugging see Section 5.1, “Compiler Debug Flags and Environment Variables”.

**Further Information**
- To access the full documentation and users guides of the PGI compilers see the PGI Documentation [http://www.pgroup.com/resources/docs.htm] page.

### 3.2.2.4. PathScale

The PathScale [http://www.pathscale.com] EKOPath compiler suite includes C, C++, and Fortran compilers for 64-bit Linux based systems offering support for:

- Features from Fortran 2003 and Fortran 2008
- ISO C99/C++ 2003, C++11 & GNU compatibility
- OpenMP 2.5

In addition to the compiler options listed in Table 4, “Common Compiler Options” the table below lists a set of selected options available for the PathScale compilers.

**Table 8. General PathScale Compiler Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-version)</td>
<td>Write compiler release version information to stdout.</td>
</tr>
<tr>
<td>(-std=&lt;val&gt;)</td>
<td>Conform to a specific language standard &lt;val&gt;. <em>(C/C++ only)</em></td>
</tr>
<tr>
<td>(-###)</td>
<td>Print commands to run during stages of compilation without executing them.</td>
</tr>
<tr>
<td>(-E)</td>
<td>Preprocessed source files including #line directives are output to stdout.</td>
</tr>
<tr>
<td>(-E -P)</td>
<td>Inhibit generation of #line directives in the output from the preprocessing.</td>
</tr>
<tr>
<td>(-module dir)</td>
<td>Save/search for module files in directory dir. <em>(Fortran only)</em></td>
</tr>
<tr>
<td>(-static)</td>
<td>Use static linking of libraries. Dynamic linking is the default.</td>
</tr>
<tr>
<td>(-i8)</td>
<td>Specifies the default size for integer variables to be 8 bytes. <em>(Fortran only)</em></td>
</tr>
<tr>
<td>(-r8)</td>
<td>Specifies the default size for real variables to be 8 bytes. <em>(Fortran only)</em></td>
</tr>
</tbody>
</table>

For optimising codes using the PathScale compilers see PathScale Optimisation Flags, and for debugging see Section 5.1, “Compiler Debug Flags and Environment Variables”.

**Further Information**
- A complete list of options for the EKOPath compilers can be found in the EKOPath man doc [http://www.pathscale.com/node/272].

### 3.3. MPI

MPI (Message-Passing Interface) is a widely used standard for the message-passing parallel programming model used on distributed memory systems. MPI is a library specification providing a set of routines for point-to-point and collective communication. In addition MPI includes functionality for other types of operations, e.g. remote-memory access and parallel I/O.

MPI is a specification, not an implementation. There are multiple implementations of MPI available, open source and vendor-specific. The next sections provide an overview of some of the more widely used implementations.
3.3.1. MVAPICH2

MVAPICH2 [http://mvapich.cse.ohio-state.edu/overview/mvapich2/] is an open-source MPI-3 implementation and is developed by the Network-Based Computing Laboratory (NBCL) at the Ohio State University.

Compiling and Running Applications

The compiler wrappers listed in the table below are used to compile and link programs providing the necessary libraries.

Table 9. MVAPICH2 Compiler Wrappers

<table>
<thead>
<tr>
<th>Language</th>
<th>Wrapper script</th>
<th>Environment variable</th>
<th>Command line</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>MPICH_CC</td>
<td>-cc=&lt;compiler&gt;</td>
</tr>
<tr>
<td>C++</td>
<td>mpicxx, mpic++</td>
<td>MPICH_CXX</td>
<td>-cxx=&lt;compiler&gt;</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpif77</td>
<td>MPICH_F77</td>
<td>-f77=&lt;compiler&gt;</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpif90</td>
<td>MPICH_F90</td>
<td>-f90=&lt;compiler&gt;</td>
</tr>
</tbody>
</table>

Specify option -show to see underlying compilers used, i.e. the compilers used to build the MVAPICH2 installation, and necessary paths and libraries without actually running the commands. It is possible to select a different compiler using one of the environment variables or command line options listed above. This should only be done if the compiler is compatible with the MVAPICH2 library.

To launch MVAPICH2 jobs use the mpirun_rsh command:

```
$ mpirun_rsh -np <N> -hostfile hfile ./mpi_prog
```

specifying the total number of processes <N> and the name of the hostfile hfile containing hosts, one per line.

Alternatively, MVAPICH2 also provides the Hydra process manager from MPICH2. Hydra can be used issuing the mpirun or mpiexec (they are both symbolic links to mpiexec.hydra) job launcher commands:

```
$ mpirun -np <N> -f hfile ./mpi_prog
```

When running under the PBS environment the Hydra mpirun/mpiexec commands will automatically use the PBS_NODEFILE environment variable to find the hosts to run on, i.e. specifying the -f hfile option is not necessary.

Add the -help flag to show available options to the job launcher commands.

MVAPICH2 supports architecture specific CPU mapping through the Portable Hardware Locality (hwloc) software package. CPU affinity is enabled by default. To see a description on how process placement and other features of MVAPICH2 can be tuned at runtime, see Section 4.2.1, “MPI Runtime Environment Variables”.

Further Information

- See the MVAPICH2 User Guide [http://mvapich.cse.ohio-state.edu/support].

3.3.2. Open MPI

Open MPI [http://www.open-mpi.org/] is an open source implementation of MPI and started as a merge of other implementations, FT-MPI from the University of Tennessee, LA-MPI from Los Alamos National Laboratory, LAM/MPI from Indiana University, and PACX-MPI from the University of Stuttgart. Open MPI includes all MPI 1.2-compliant and MPI 2-compliant routines.

Compiling and Running Applications

Load the proper module file to add the paths for Open MPI executables and libraries to your environment. The Open MPI’s wrapper compilers listed in the table below add in all the relevant compiler/linker flags and then invoke
the underlying compiler, i.e. the compiler the Open MPI installation was built with. Use the option `-showme` to see the underlying compiler, the compile and link flags, and the libraries that are linked.

### Table 10. Open MPI Compiler Wrappers

<table>
<thead>
<tr>
<th>Language</th>
<th>Wrapper script</th>
<th>Environment variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>OMPI_CC</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC, mpicxx, mpic++</td>
<td>OMPI_CXX</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpif77</td>
<td>OMPI_F77</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpif90</td>
<td>OMPI_FC</td>
</tr>
</tbody>
</table>

If necessary, it is possible to change the underlying compiler invoked when calling the compiler wrappers using the environment variables listed in the table. E.g. if you want link with an Intel compiler built Open MPI installation, combining Fortran and C code using `ifort` and `gcc` you can specify

```
$ OMPI_CC=gcc mpicc -c cfile.c
$ mpif90 cfile.o ffile.f90
```

Open MPI provides the commands `mpirun` and `mpiexec` to launch MPI jobs. The two commands are identical, they are symbolic links the back-end launcher command `orterun`. To see all options available to `mpirun` issue command

```
$ mpirun -h
```

The following lists some of the most important options:

- `-np <N>` Specify the number `<N>` of processes to run.
- `-hostfile <hfile>` Provide a hostfile to use. Notice, for Open MPI installations that include support for some batch systems, e.g. PBS Pro and LoadLeveler, there is no need to specify the `-hostfile` option. Open MPI automatically detect the list of hosts to run on.
- `-byslot` Assign processes round-robin by slot (the default).
- `-bynode` Assign processes round-robin by node.
- `-bysocket` Assign processes round-robin by socket.
- `-loadbalance` Uniform distribution of ranks across all nodes.
- `-bind-to-core` Bind processes to cores.
- `-bind-to-socket` Bind processes to processor sockets.
- `-report-bindings` Report any bindings for launched processes.
- `-stdin <rank>` The MPI rank that is to receive stdin. The default is to forward stdin to rank 0.
- `-x <VAR>` Export environment variable `<VAR>` to remote nodes.
- `-mca` Specify values to MCA parameters.

The default scheduling policy of Open MPI is by slot, i.e. processes are scheduled on a node until all of its default slots are exhausted before proceeding to the next node. This can be changed specifying one of the mapping options listed above.

For example, to run 8 mpi processes on two 2-socket 8-core nodes (`node0` and `node1`) mapping to sockets in a round-robin fashion and binding to cores, use the following command:

```
$ mpirun -np 8 -bysocket -bind-to-core -report-bindings ./mpi_prog
```
As can be seen from the output above using the \ `-report-bindings` option, the job will run 2 MPI processes on each socket binding to core id’s 0 and 1.

Runtime tuning can also be done setting global MCA parameters. They can be specified as options to the `mpirun` command or as environment variables with prefix `OMPI_MCA_`. For example, to show all MCA parameter values during MPI\_INIT specify use the \ `-mca` option:

\ $ mpirun -np 8 -mca mpi_show_mca_params all ./mpi_prog$

or using the environment variable:

\ $ export OMPI_MCA_mpi_show_mca_params=all$
\ $ mpirun -np 8 ./mpi_prog$

Command `ompi_info` displays information about the Open MPI installation. To see all MCA parameters specify

\ $ ompi_info -param all all$

See Section 4.2.1, “MPI Runtime Environment Variables” for a description of a subset of variables supported by Open MPI.

Further Information

• For more information using Open MPI see the online FAQ [http://www.open-mpi.org/faq] page.

3.3.3. Intel MPI

The Intel MPI [http://software.intel.com/en-us/intel-mpi-library] library is based on MPICH from Argonne National Laboratory and MVAPICH2 from Ohio State University. Intel MPI implements the MPI 2.2 specification on multiple fabrics.

Compiling and Running Applications

The following table shows available Intel MPI compiler commands, the underlying Intel and GNU compilers, and ways to override underlying compilers with environment variables or command line options:

<table>
<thead>
<tr>
<th>Table 11. Intel MPI Compiler Wrappers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>C++</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fortran 77 / Fortran 95</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Specify option \ `-show` with one of the compiler wrapper scripts to see the underlying compiler together with compiler options, link flags and libraries.
Use the `mpirun` command to launch programs linked with Intel MPI. The default underlying process manager using `mpirun` is Hydra. (Alternatively MPD can be selected as process manager, users are referred to the Intel documentation for more information on process managers.)

A list of selected options to `mpirun`:

- `-version`, `-V` Display the version of the Intel MPI Library.
- `-np <N>` Specify the number `<N>` of processes to run
- `-hostfile <hfile>` Specify a hostfile with hosts to run the job on. If a job is started within a batch system, e.g. PBS Pro, LSF, or LoadLeveler, `mpirun` will automatically extract the hostlist to run on, i.e. this option is not needed.
- `-l` Prefix lines written to standard output with the MPI process rank number.
- `-s <rank>` Specify the MPI rank to receive stdin. Default rank is 0.
- `-genv <VAR> <value>` Set the `<VAR>` environment variable to the given `<value>` for all MPI processes.
- `-genvall` Propagate all environment variables to all MPI processes.

See Section 4.2.1, “MPI Runtime Environment Variables” for a list of environment variables used to influence program behavior and performance at run time, and Section 6.1, “MPI Statistics” for variables used for collecting performance statistics of MPI communication operations.

**Further Information**


### 3.3.4. Platform MPI

IBM Platform MPI [http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/mpi] is based on MPICH from Argonne National Laboratory and LAM from the University of Notre Dame and Ohio Supercomputer Center. The implementation fully complies with the MPI 2.2 standard and includes ROMIO, a portable implementation of MPI I/O. It also ships with a prebuilt MPE (MPI Parallel Environment) profiling tool.

**Compiling and Running Applications**

Load the Platform MPI module file, make sure that the `MPI_ROOT` environment variable points to the location where Platform MPI is installed. Compiler wrappers are provided to set up the path for include and library directories, and libraries to link. The underlying compiler used is specified using environment variables or as command line options to the compile wrappers, see the table below.

**Table 12. Platform MPI Compiler Wrappers**

<table>
<thead>
<tr>
<th>Language</th>
<th>Wrapper script</th>
<th>Environment variable</th>
<th>Command line</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>mpicc</td>
<td>MPI_CC</td>
<td>-mpicc &lt;compiler&gt;</td>
</tr>
<tr>
<td>C++</td>
<td>mpiCC</td>
<td>MPI_CXX</td>
<td>-mpiCC &lt;compiler&gt;</td>
</tr>
<tr>
<td>Fortran 77</td>
<td>mpif77</td>
<td>MPI_F77</td>
<td>-mpif77 &lt;compiler&gt;</td>
</tr>
<tr>
<td>Fortran 90</td>
<td>mpif90</td>
<td>MPI_F90</td>
<td>-mpif90 &lt;compiler&gt;</td>
</tr>
</tbody>
</table>

For example, to compile a C program using the PGI compiler:

```
$ mpicc -mpicc pgcc prog.c
```

or alternatively specify the compiler environment variable first:
$ export MPI_CC=pgcc
$ mpicc prog.c

To compile code and link against Platform MPI without utilizing the wrapper tools specify the MPI library using the `-l` option when linking. E.g. using the PGI compiler:

$ pgcc prog.c -lpcmpi64

The above command assumes that the proper paths for Platform MPI include and library directories have been added to the environment.

Use the supplied Platform MPI launch utility `mpirun` when running applications. Listed below is a description of some of the options to `mpirun`:

- `-help` Prints usage information for `mpirun`.
- `-np <N>` Specifies the number `<N>` of processes to run.
- `-hostfile <hfile>` Launches an executable across multiple hosts listed in file `<hfile>`.
- `-stdio=p` Prepends the global rank of the originating process to `stdout` and `stderr` output.
- `-stdio=i` Broadcasts standard input to all MPI processes.
- `-intra=mix` Use shared memory for small messages, the default size is 256 KB. For larger messages, the interconnect is used for better bandwidth. The default is to use shared memory (`-intra=shm`) for all intrahost data transfers.
- `-T` Prints user and system times for each MPI rank.
- `-e var[=val]` Sets the environment variable `var` for the program and gives it the value `val` if provided.
- `-d` Debug mode. Prints additional information about application launch.

Platform MPI includes a set of environment variables that affect the behavior at run time. See Section 4.2.1, “MPI Runtime Environment Variables” for a description of a subset of these variables. For a complete list of environment variables see the `man mpienv` page.

### 3.4. OpenMP

The OpenMP API supports shared-memory parallel programming in C/C++ and Fortran. OpenMP comprises a set of compiler directives, library routines, and environment variables used for writing parallel programs that are portable across different shared-memory architectures. A number of compilers, including the ones covered in this guide, implement the OpenMP API.

#### 3.4.1. Compiler Flags

To enable the compiler to generate multithreaded code based on the OpenMP directives in the code add the option listed in the table below

**Table 13. OpenMP Compiler Flags**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td><code>-fopenmp</code></td>
</tr>
<tr>
<td>Intel</td>
<td><code>-openmp</code></td>
</tr>
<tr>
<td>Portland</td>
<td><code>-mp</code></td>
</tr>
<tr>
<td>PathScale</td>
<td><code>-mp</code>, <code>-openmp</code></td>
</tr>
</tbody>
</table>

The OpenMP specification defines several environment variables that control the execution of OpenMP programs, see Section 4.2.2, “OpenMP Runtime Environment Variables” for a description.
### 3.5. Hybrid Programming

Combining MPI and OpenMP in a parallel application can help to utilize shared memory within a multicore node more efficiently, thus avoiding the need of costly intra-node communication. The typical scenario in this model is for MPI to handle communication operations among nodes, and OpenMP to manage concurrency within each node. This can help application performance in the following ways:

- The overhead due to intra-node communication (e.g., message packing/unpacking) is mitigated, since OpenMP threads use directly shared memory for data exchange.

- Global synchronization can be more efficient because it is performed in a group-wise and hierarchical fashion (i.e., first, OpenMP threads within each thread group, and then, MPI processes on behalf of each thread group).

- The total memory footprint of the original MPI application is reduced, due to the elimination of message buffers and other internal data structures. This enables fitting larger working sets in the same HPC system.

- Inter-node communication can be improved because there is less pressure put on the network. Since the number of MPI processes per node is reduced, the total number of communication channels and the number of messages are also reduced, eliminating large portion of the startup communication cost. In most cases, it is generally preferable to combine communication data into fewer but larger messages, than the opposite.

- Applications that suffer from load imbalance can benefit from OpenMP’s ability to schedule work in a dynamic fashion. MPI does not provide similar constructs that would enable dynamic work assignment in an easy way.

- When scheduled on cores that share some level of cache hierarchy (e.g. L2 or L3), threads that work on common or adjacent data can usually execute faster. This is because they exploit shared cache space in a constructive way, either by having effectively larger cache capacity at their disposal, or due to the fact that the data fetched by some threads might be reused by others in the near future.

- Having a moderate mixture of \( P \) MPI processes and \( T \) OpenMP threads, as opposed to having a single MPI process and \( C \) OpenMP threads (where \( C = P \times T \) is the total number of cores in a node), can help to take advantage of NUMA capabilities more efficiently. The reason is that the \( P \) separate address spaces can be naturally mapped to different NUMA domains, which will provide uncontended access to main memory between the MPI processes.

The typical execution scenario of a hybrid MPI-OpenMP application is the following:

- \( P \) MPI processes are spawned on each node in the system
- each process spawns \( T \) OpenMP threads
- at regular periods, MPI processes communicate with each other to exchange data
- threads in the team belonging to each process start processing the data until the next global synchronization point

This scenario is implemented in the following sample code:

```c
#include <omp.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int nproc, my_rank, curr_step, max_steps;
    float* buf;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    ...

    while ( curr_step < max_steps ) {
        MPI_Send(&buf, ...); //send data from previous step
        MPI_Recv(&buf, ...); //receive data for current step
    }
}
```
```c
#pragma omp parallel for schedule(runtime)
for ( i = 0; i < N; i++ ) {
    //process data on curr_buf
}

curr_step++;
}

MPI_Finalize();
return 0;
```

To run the application, we may use a bash script like the one that follows. It acts as a wrapper to the MPI program, and additionally initializes certain OpenMP runtime variables before invoking the executable. Therefore, the script itself can be given as argument to the MPI launcher (e.g. `mpirun`).

```bash
#!/bin/bash
export OMP_NUM_THREADS=T
export OMP_SCHEDULE="dynamic"
./mpi__omp_prog arg1 arg2 ...
```

To further tune MPI-OpenMP interaction, the programmer can use appropriate arguments in the `MPI_Init_thread` function in order to specify the desired level of thread support. These arguments are supported as of MPI-2.1, and are the following:

- `MPI_THREAD_FUNNELED`
  - the process may be multithreaded, but only the main thread will make MPI calls

- `MPI_THREAD_SERIALIZED`
  - the process may be multithreaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads

- `MPI_THREAD_MULTIPLE`
  - multiple threads may call MPI, with no restrictions

### 3.6. Math Libraries

#### 3.6.1. GSL (GNU Scientific Library)

The GNU Scientific Library [http://www.gnu.org/software/gsl/] (GSL) is a free numerical software package for C and C++ covering a range of subject areas including:

- BLAS (level 1, 2, and 3) and linear algebra routines
- Fast Fourier transform (FFT) functions
- Numerical integration
- Random number generation functions

To use the library make sure that the necessary paths to the GSL installation are added to the environment. The library header files are installed in a subdirectory `gsl` that must prefix preprocessing include statements, see e.g. `program test_blas.c` below. The header files define functions to have `extern "C"` linkage when included in C++ programs allowing functions to be called directly from C++. The GSL library is available in both shared and static versions.

To link against the library specify both the main GSL library and the supporting CBLAS library. For example, compile and link the sample `test_blas.c` program:

```c
/* test_blas.c : Example using BLAS matrix multiply code sgemm */
```
#include <stdio.h>

#ifdef GSL
#include <gsl/gsl_cblas.h>
#elif defined MKL
#include <mkl_cblas.h>
#elif defined ACML
#include <acml.h>
#endif

int main(void)
{
    const int lda=3, ldb=3, ldc=3;
    int m, n, k;
    float alpha, beta;

    float a[] = { 0.11, 0.21, 0.12,
                 0.15, 0.13, 0.17,
                 0.22, 0.13, 0.23 };

    float b[] = { 1011, 1021, 1018,
                 1031, 1012, 1021,
                 1022, 1032, 1015 };

    float c[] = { 0.00, 0.00, 0.00,
                 0.00, 0.00, 0.00,
                 0.00, 0.00, 0.00 };

    m = 3; n = 3; k = 3;
    alpha = 1.0; beta = 0.0;

#ifdef ACML
    sgemm ('N', 'N', m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
#endif

#ifdef MKL || defined GSL
    cblas_sgemm (CblasColMajor, CblasNoTrans, CblasNoTrans,
                 m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);
#endif

    printf ("[ %g, %g, %g
", c[0], c[1], c[2]);
    printf ("  %g, %g, %g
", c[3], c[4], c[5]);
    printf ("  %g, %g, %g 
", c[6], c[7], c[8]);
    return 0;
}

using the GNU C compiler, adding -DGSL to follow the GSL path in the code:

$ gcc -DGSL test_blas.c -lgsl -lgslcblas

It is possible to link with an alternative BLAS library that is conforming to the CBLAS standard (functions prefixed with cblas_).

Further Information

• For more information see the online GSL - Reference Manual [http://www.gnu.org/software/gsl/manual/html_node/].
3.6.2. MKL (Intel Math Kernel Library)

The Intel Math Kernel Library [http://software.intel.com/en-us/intel-mkl/] (Intel MKL) contains highly optimised, extensively multithreaded math routines for different areas of computation. The library includes:

- BLAS (level 1, 2, and 3) and LAPACK linear algebra routines
- ScaLAPACK distributed processing routines and BLACS routines for communication
- Fast Fourier transform (FFT) functions
- Vectorized math functions
- Random number generation functions


Intel MKL is optimised for the latest Intel processors, including processors with multiple cores. The cluster components of the library, ScaLAPACK and Cluster FFT, are built on top of MPI. The library comes with two interface layers, LP64 for 32-bit integers and ILP64 for 64-bit integers.

Dynamic Linking

Since the Intel Compiler version 11.1 the -mkl option is available providing users a quick way to set the environment settings and required libraries to link with MKL:

- mkl or -mkl=parallel # link with standard multithreaded MKL
- mkl=sequential # link with sequential version of MKL
- mkl=cluster # link with MKL cluster (sequential)
  # components that use Intel MPI

For example, compile and link test_blas.c, adding -DMKL to follow the MKL path in the code:

$ icc -DMKL -mkl test_blas.c

Using the -mkl option the compiler links the application using the LP64 (32-bit integer) interface.

Static Linking

In case of static linking load the proper MKL modulefile. Make sure that the environment variables

MKLROOT
MKLPATH=$MKLROOT/lib/intel64
MKLINCLUDE=$MKLROOT/include

are defined. When linking enclose the libraries in grouping symbols. For non-cluster components of MKL specify the following linker options

-IL$MKLPATH -I$MKLINCLUDE \
-1Wl,-start-group $MKLPATH)/libmkl_intel_lp64.a \
$MKLPATH)/<thread_lib> $MKLPATH)/libmkl_core.a -Wl,-end-group \
<thread_option> -lpthread

and for cluster components of MKL (i.e. ScaLAPACK and Cluster FFT) linking against Intel MPI specify:

-IL$MKLPATH -I$MKLINCLUDE \
-1Wl,-start-group $MKLPATH)/<cluster_lib> \
$MKLPATH)/libmkl_intel_lp64.a $MKLPATH)/<thread_lib> \
$MKLPATH)/libmkl_core.a \n$MKLPATH)/libmkl_blacs_intelmpi_lp64.a -Wl,-end-group \n<thread_option> -lpthread

where you replace above:

<table>
<thead>
<tr>
<th>sequential library</th>
<th>multithreaded library</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;thread_lib&gt;</td>
<td>libmkl_sequential.a</td>
</tr>
<tr>
<td>&lt;thread_option&gt;</td>
<td>n/a</td>
</tr>
</tbody>
</table>

18
The above specifications link with the LP64 (32-bit integer) interface to the library. In order to link with the ILP64 (64-bit integer) interface replace 1p64 with ilp64 in the library names.

For example, to link a Fortran program prog.f with multithreaded ScaLAPACK using the Intel mpiifort compiler wrapper to setup the MPI environment specify:

```
$ mpiifort -I$(MKLROOT)/include prog.f  \
-Wl,--start-group  \
$(MKLROOT)/lib/intel64/libmkl_scalapack_lp64.a  \
$(MKLROOT)/lib/intel64/libmkl_intel_lp64.a  \
$(MKLROOT)/lib/intel64/libmkl_intel_thread.a  \
$(MKLROOT)/lib/intel64/libmkl_core.a  \
$(MKLROOT)/lib/intel64/libmkl_blacs_intelmpi_lp64.a  \
-Wl,--end-group -openmp -lpthread
```

**Further Information**


### 3.6.3. ACML (AMD Core Math Library)

The AMD Core Math Library [http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml] (ACML) is a set of numerical routines tuned specifically for AMD64 platform processors. ACML supports a variety of compilers, including GCC, Intel, and PGI.

ACML consists of the following main components

- A full implementation of Level 1, 2 and 3 Basic Linear Algebra Subroutines (BLAS)
- A full suite of Linear Algebra (LAPACK) routines
- A comprehensive suite of Fast Fourier Transforms (FFTs) in both single-, double-, single-complex and double-complex data types
- Random Number Generators in both single- and double-precision
- Optimised versions of most libm functions, including scalar, vector, and array versions

The routines are available via both FORTRAN 77 and C interfaces and multithreaded versions are available for the Level 3 BLAS, many LAPACK routines, and the 2D and 3D FFTs. The library is supplied in both static and shared versions, libacml.a and libacml.so, respectively.

The ACML installation includes versions of the library for different compilers. To link with the library specify the path to the proper library on the link command line. Set the path to the ACML installation directory, e.g.

```
$ export ACMLDIR=/opt/acml5.1.0
```

This may already have been done when loading the ACML modulefile. Check your environment to see the actual name of variable pointing to the ACML installation directory. Add the library subdirectory with the -L option based upon the compiler used when linking the code

```
-L$ACMLDIR/<compiler>/lib
```

For example, to compile and link test_blas.c using the PGI C compiler, adding -DACML to follow the ACML path in the code:

```
$ pgcc -DACML test_blas.c -L$ACMLDIR/pgi64/lib -lacml
```

For example, using the GNU Fortran compiler, linking against the multithreaded version of ACML:
$ gfortran -fopenmp prog.f -L$ACMLDIR/gfortran64_mp/lib -lacml_mp

Further Information

• For more information see the online ACML User Guide [http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml/user-guide].

3.6.4. FFTW

FFTW [http://www.fftw.org/] is a library developed at Massachusetts Institute of Technology (MIT) for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data. The library supports serial transforms, multi-threaded transforms, and distributed-memory parallel transforms using MPI. FFTW include interfaces for both C and Fortran, and support single, double and long-double precision.

All programs using FFTW must include the header file:

```c
#include <fftw3.h>
```

This is the same header file regardless of the precision of FFTW used. If using a different version than the (default) double precision version, instances of lowercase 'fftw_' in the source code should be replaced with 'fftwf_' for single precision or 'fftwl_' for long-double precision.

Depending on the installation, FFTW will include serial and parallel libraries for different precisions listed in the table below:

<table>
<thead>
<tr>
<th>Precision</th>
<th>Serial</th>
<th>Threads (POSIX)</th>
<th>OpenMP</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>libfftw3f.a</td>
<td>libfftw3f_threads.a</td>
<td>libfftw3f_omp.a</td>
<td>libfftw3f_mpi.a</td>
</tr>
<tr>
<td>double</td>
<td>libfftw3.a</td>
<td>libfftw3_threads.a</td>
<td>libfftw3_omp.a</td>
<td>libfftw3_mpi.a</td>
</tr>
<tr>
<td>long</td>
<td>libfftw3l.a</td>
<td>libfftw3l_threads.a</td>
<td>libfftw3l_omp.a</td>
<td>libfftw3l_mpi.a</td>
</tr>
</tbody>
</table>

(Installations built with gcc may also include a quadruple-precision version of FFTW.)

The basic usage of FFTW is to link with the serial FFTW library, e.g. in double precision, -lfftw3. When using one of the parallel versions of FFTW in addition add the appropriate library from the table.

For example, to link a C program using OpenMP transforms with the double precision FFTW version using the Intel compiler specify:

```
$ icc -openmp test_fft.c -lfftw3_omp -lfftw3
```

Further Information

• For more information see the FFTW Documentation [http://www.fftw.org/#documentation].

4. Tuning

This chapter describes how to improve application performance using compiler optimisation flags for single processor tuning, and environment variables used to control MPI and OpenMP behavior at runtime.

4.1. Optimisation

4.1.1. IEEE 754 Compliant Optimisation

The IEEE 754 standard specifies formats and methods for floating-point computations. The standard specifies number formats, basic operations, conversions, and floating-point exceptions and their handling.

Adhering to the IEEE 754 standard means that an application will generate exactly the same data regardless of the target processor. Because compiler optimisations are performed at the possible expense of precision most compilers will not generate full IEEE 754 compliant code unless this is specified explicitly by compiler flags.
See the table below for compiler flags that will produce IEEE 754 compliance for floating-point arithmetic.

### Table 14. IEEE 754 Compliant Optimisation Flags

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC</td>
<td>-frounding-math</td>
</tr>
<tr>
<td>Intel</td>
<td>-fp-model source</td>
</tr>
<tr>
<td>Portland</td>
<td>-Kieee</td>
</tr>
<tr>
<td>PathScale</td>
<td>-OPT:IEEE_arithmetic=1</td>
</tr>
</tbody>
</table>

#### 4.1.2. Compiler Optimisation Flags

The following sections list a number of optimisation flags available for the different compilers, and discuss some general optimisation recommendations.

**GCC**

**Table 15. GCC Optimisation Flags**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--help=optimizers</code></td>
<td>Display all optimisation options.</td>
</tr>
<tr>
<td><code>-Q --help=optimizers</code></td>
<td>Adding <code>-Q</code> with the option above lists optimisations that are enabled at each level.</td>
</tr>
<tr>
<td><code>-mtune=&lt;cpu-type&gt;</code></td>
<td>Produce code optimised for the selected processor type. See the compiler man page for choices for cpu-type.</td>
</tr>
<tr>
<td><code>-march=&lt;cpu-type&gt;</code></td>
<td>Generate instructions for the selected processor type. Specifying <code>-march=native</code> will tune for the architecture of the compiling machine, and implies <code>-mtune=native</code>.</td>
</tr>
<tr>
<td><code>-m&lt;target&gt;</code></td>
<td>Enable the use of extended instruction sets, where <code>&lt;target&gt;</code> is e.g. avx, sse4.2, 3dnow, etc.</td>
</tr>
<tr>
<td><code>-O0,-O1,-O2,-O3</code></td>
<td>Specify the level of increasingly aggressive optimisations. Default: <code>-O0</code></td>
</tr>
<tr>
<td><code>-Ofast</code></td>
<td>Combines all <code>-O3</code> optimisations with options that disregard strict standards compliance but can result in better optimised code. Turns on <code>-ffast-math</code>, see below.</td>
</tr>
<tr>
<td><code>-Os</code></td>
<td>Optimise for size. Includes <code>-O2</code> optimisations that do not increase code size.</td>
</tr>
<tr>
<td><code>-ffast-math</code></td>
<td>Turns on optimisations that may violate IEEE standards but can give faster code. Enables flushing of denormal results to zero.</td>
</tr>
<tr>
<td><code>-finline-functions</code></td>
<td>Consider all functions for inlining.</td>
</tr>
<tr>
<td><code>-funroll-loops</code></td>
<td>Perform loop unrolling when iteration count is known.</td>
</tr>
</tbody>
</table>

Notice that the default optimisation level is `-O0`, i.e. most optimisations are disabled. It is recommended to start optimising with the `-O2` flag, since this will provide maximum optimisation without increasing the executable size, and in addition to add `-march` to produce architecture tuned code, i.e.

```
$ gcc -march=native -O2 prog.c
```

**Intel**

**Table 16. Intel Optimisation Flags**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-help opt</code></td>
<td>Display optimisation options.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-help advanced</td>
<td>Display advanced optimisation options that allow fine tuning of compilation.</td>
</tr>
<tr>
<td>-m&lt;code&gt;</td>
<td>Generate specialized code for processors specified by &lt;code&gt;. The executable generated can run on any compatible (Intel and non-Intel) processor with support for the corresponding instruction set.</td>
</tr>
<tr>
<td>-x&lt;code&gt;</td>
<td>Generate specialized code for processors specified by &lt;code&gt;. The executable generated can only run on Intel processors with support for the corresponding instruction set. Specifying -xHost will generate instructions for the highest instruction set available on the compiling machine.</td>
</tr>
<tr>
<td>-ax&lt;code&gt;</td>
<td>Generate specialized code with multiple code paths for Intel processors based on &lt;code&gt;. The most suitable code path for the processor the application is running on is chosen at execution time. The default optimised code path can be modified by the -m or -x switches described above.</td>
</tr>
<tr>
<td>-O0,-O1,-O2,-O3</td>
<td>Specify the level of increasingly aggressive optimisations. Default: -O2</td>
</tr>
<tr>
<td>-fast</td>
<td>Maximize speed across the entire program.</td>
</tr>
<tr>
<td>-Os</td>
<td>Enable optimisations that do not increase code size and produces smaller code size than O2.</td>
</tr>
<tr>
<td>-no-prec-div</td>
<td>Optimise floating-point division that may not be full IEEE compliant. This option is enabled with -fast. The default is -prec-div, i.e. fully precise IEEE division.</td>
</tr>
<tr>
<td>-unroll-aggressive</td>
<td>Use more aggressive unrolling for certain loops.</td>
</tr>
<tr>
<td>-ipo</td>
<td>Enable interprocedural optimisation between files.</td>
</tr>
<tr>
<td>-ftz</td>
<td>Flush denormal results to zero.</td>
</tr>
<tr>
<td>-parallel</td>
<td>Generate multithreaded code for loops that can be safely executed in parallel.</td>
</tr>
<tr>
<td>-mkl</td>
<td>Link to certain parts of MKL, see also Section 3.6.2, “MKL (Intel Math Kernel Library)”.</td>
</tr>
<tr>
<td>-fno-exceptions</td>
<td>Disable exception handling support. (C++ only)</td>
</tr>
</tbody>
</table>

O2 is the generally recommended optimisation level. The O3 option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets unless loop and memory access transformations take place, O3 optimisations may slow down code in some cases compared to O2 optimisations.

The recommended starting point for optimisation generating optimal instructions for the compiling machine is, e.g. Fortran code

```bash
$ ifort -xHost -O2 prog.f90
```

### Portland

Table 17. Portland Optimisation Flags

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-help=opt</td>
<td>Print help for optimisation flags.</td>
</tr>
<tr>
<td>-Minfo</td>
<td>List information about optimisation made during compilation. See the compiler man page for suboptions.</td>
</tr>
<tr>
<td>-tp &lt;target&gt;</td>
<td>Sets the target architecture. The default is to produce code targeted for the processor of the compiling system.</td>
</tr>
<tr>
<td>-O0,-O1,-O2,-O3,-O4</td>
<td>Specify the level of increasingly aggressive optimisations. Default: -O1</td>
</tr>
<tr>
<td>-fast</td>
<td>Choose generally optimal flags for the target platform, including SSE/ SSE2 instructions. (See description below)</td>
</tr>
</tbody>
</table>
Option | Description
--- | ---
-Mflushz | Flush denormal results from floating-point calculations to zero.
-Munroll | Invoke the loop unroller. See the compiler man page for suboptions.
-Mvect=simd[128|256] | Use SSE (128 bit) or AVX (256 bit) SIMD instructions. See the compiler man page for other suboptions to -Mvect
-Mipa | Choose generally optimal flags for InterProcedural Analysis (IPA) for the target platform.
-Minline | Pass options to the function inliner.
-Mconcur | Enable auto-parallelizing of loops. Use with -Minfo to see information about code being parallelized.
--no_exceptions | Disable exception handling support. (C++ only)

As suggested in the PGI Compiler User's Guide [http://www.pgroup.com/doc/pgiug.pdf] (PDF) when optimising it is recommended to start with the option -fast. This is an aggregate option that incorporates a generally optimal set of flags for targets that support SSE capability. The option combines optimisation options to enable use of vector streaming SIMD instructions for 64-bit targets. It enables vectorization with SSE instructions, cache alignment, and flushes to zero. Use the -help switch to see the flags included with the -fast option:

$ pgf90 -fast -help

The contents of the -fast option are host-dependent and typically include the following:

-O2 | Specifies a code optimisation level of 2.
-Munroll=c:1 | Unrolls loops, executing multiple instances of the loop during each iteration.
-Mnoframe | Indicates to not generate code to set up a stack frame.
-Mlre | Indicates loop-carried redundancy elimination.
-Mvect=sse | Generates SSE instructions.
-Mscalarsse | Generates scalar SSE code with xmm registers.
-Mcache_align | Aligns long objects on cache-line boundaries.
-Mflushz | Flush denormal results to zero.
-Mpre | Indicates partial redundancy elimination.

The recommended optimisation flags for Fortran and C programs are:

$ pgf95 -fast -Mipa=fast prog.f90
$ pgcc -fast -Mipa=fast prog.c

and for C++ programs:

$ pgCC -fast -Mipa=fast -Minline=levels:10 --no_exceptions prog.cc

**PathScale**

**Table 18. PathScale Optimisation Flags**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-LIST:options=ON</td>
<td>List options enabled at a given optimisation level written to a listing (.lst) file.</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-march=&lt;cpu-type&gt;</td>
<td>Compiler will optimise code for the selected CPU type. The default is auto which means to optimise for the host platform that the compiler is running on.</td>
</tr>
<tr>
<td>-m&lt;target&gt;</td>
<td>Enable the use of extended instruction sets, where &lt;target&gt; is e.g. avx, sse4_2, 3dnow, etc.</td>
</tr>
<tr>
<td>-OPT: ...</td>
<td>This option group controls miscellaneous optimisations. See the man eko page for a complete list of available suboptions.</td>
</tr>
<tr>
<td>-LNO: ...</td>
<td>This group specifies options and transformations performed on loop nests. See the man eko page for a complete list of available suboptions. The -LNO options are enabled only if the optimisation level of -O3 or higher is in effect.</td>
</tr>
<tr>
<td>-IPA ...</td>
<td>The InterProcedural Analysis option group. See the man eko page for a complete list of available suboptions.</td>
</tr>
<tr>
<td>-ipa</td>
<td>InterProcedural Analysis with default IPA suboptions used.</td>
</tr>
<tr>
<td>-O0,-O1,-O2,-O3</td>
<td>Specify the level of increasingly aggressive optimisations. Default: -O2</td>
</tr>
<tr>
<td>-Ofast</td>
<td>Use optimisations selected to maximize performance.</td>
</tr>
<tr>
<td>-inline</td>
<td>Request inline processing.</td>
</tr>
<tr>
<td>-OPT:IEEE_arithmetic=&lt;N&gt;</td>
<td>Specify the level of floating point roundoff/overflow behavior. Setting &lt;N&gt;=1 will adhere to IEEE accuracy (see Section 4.1.1, “IEEE 754 Compliant Optimisation”). Setting &lt;N&gt; equal to either 2 or 3 will enable flushing of denormal numbers to zero.</td>
</tr>
<tr>
<td>-apo</td>
<td>Enable auto-parallelizing of code.</td>
</tr>
<tr>
<td>-fno-exceptions</td>
<td>Disables exception handling. The default is to enable exception handling. (C++ only)</td>
</tr>
</tbody>
</table>

When tuning codes it is recommended in the EKOPath User Guide [http://www.pathscale.com/EKOPath-User-Guide] to try the following set of options:

- -O2
- -O3
- -O3 -OPT:Ofast
- -Ofast

listed in the order of increasing optimisation level. The -fast option is equivalent to

- -O3
  Turn on aggressive optimisation.
- -ipa
  Invoke inter-procedural analysis (IPA).
- -fno-math-errno
  Tells the compiler to assume that the program does not test errno after calls to math library functions.
- -ffast-math
  Improves execution speed by relaxing ANSI and IEEE rules.
- -OPT:ro=2
  Departure from source language floating-point, round-off, and overflow semantics. Allows extensive transformations, such as the reordering of reduction loops.
- -OPT:Olimit=0
  Optimise code regardless of program size.
- -OPT:div_split=on
  Enable changing x/y into x*(recip(y))
- -OPT:alias=typed
  Assume that the code adheres to the ANSI/ISO C standard which states that two pointers of different types cannot point to the same location in memory.
Although the optimisations listed are generally safe, they may affect floating point accuracy due to rearrangement of computations.

4.2. Runtime Environment Settings

4.2.1. MPI Runtime Environment Variables

This section describes environment variables that can be used to modify the behavior of MPI at run time. Default settings are based on the best performance for most applications but some codes may benefit from adjusting the values of variables.

Below is a list of environment variables for different MPI implementations.

- **MVAPICH2**

  MV2\_ENABLE\_AFFINITY=[1|0]
  
  Enable (1) or disable (0) CPU affinity. The default is CPU affinity enabled. For multi-threaded programs, e.g. MPI/OpenMP, having CPU affinity enabled all threads may be scheduled to the same CPU. To avoid this MV2\_ENABLE\_AFFINITY should be disabled, i.e. set to 0.

  MV2\_SHOW\_CPU\_BINDING
  
  Shows the current process to core bindings of all processes on the node which contains the mpi rank 0 process written to stderr.

  MV2\_CPU\_BINDING\_POLICY=[bunch|scatter]
  
  Set the mapping of process to core policy. Specifying bunch processes are assigned in a block fashion, e.g. on a 2-socket node processes will map to the first socket until all cores are utilized before moving to the second socket. If scatter is set processes are mapped in a round-robin fashion.

  MV2\_CPU\_MAPPING
  
  Specify a user defined mapping to bind processes to cores.
  
  For example, to run 4 processes on a 2-socket 8-core node binding to core id's 0,1 (on the first socket) and 8,9 (on the second socket):

  ```
  $ export MV2\_CPU\_MAPPING=0:1:8:9
  $ export MV2\_SHOW\_CPU\_BINDING=1
  $ mpirun -np 4 ./mpi\_prog
  
  -------------CPU AFFINITY--------------
  RANK:0  CPU SET:   0
  RANK:1  CPU SET:   1
  RANK:2  CPU SET:   8
  RANK:3  CPU SET:   9
  ----------------------------------------
  ```

  MV2\_CPU\_BINDING\_LEVEL=[core|socket|numanode]
  
  Specify the process to core binding level. The default (core) is to bind processes to specific cores.
  
  For example, running 4 processes on a 2-socket 8-core node binding to sockets in a round-robin fashion:

  ```
  $ export MV2\_CPU\_BINDING\_LEVEL=socket
  $ export MV2\_CPU\_BINDING\_POLICY=scatter
  $ export MV2\_SHOW\_CPU\_BINDING=1
  $ mpirun -np 4 ./mpi\_prog
  
  -------------CPU AFFINITY--------------
  RANK:0  CPU SET:   0  1  2  3  4  5  6  7
  RANK:1  CPU SET:   8  9 10 11 12 13 14 15
  RANK:2  CPU SET:   0  1  2  3  4  5  6  7
  RANK:3  CPU SET:   8  9 10 11 12 13 14 15
  ```
Processes can migrate between the 8 cores on the socket.

**MV2_IBA_EAGER_THRESHOLD**
Specify the message size for switching from the eager to the rendezvous protocol for point-to-point communication. For better performance, the value of **MV2_IBA_EAGER_THRESHOLD** should be set to the same as **MV2_VBUF_TOTAL_SIZE**.

**MV2_SMP_EAGERSIZE**
Specify the message size for switching from the eager to the rendezvous protocol for intra-node communication.

**MV2_VBUF_TOTAL_SIZE**
The size of the basic basic communication buffer of MVAPICH2. For better performance, the value of **MV2_IBA_EAGER_THRESHOLD** should be set the same as **MV2_VBUF_TOTAL_SIZE**.

• **Open MPI**
The specifications listed in this section can alternatively also be specified as options to the Open MPI **mpirun** job launcher. For additional runtime tuning, e.g. process binding, see the Open MPI section.

**OMPI_MCA_btl_sm_eager_limit**
Specify the size of messages that is sent using the eager protocol for intra-node communications.

**OMPI_MCA_btl_openib_eager_limit**
Specify the size of messages that is sent using the eager protocol for inter-node communications.

**OMPI_MCA_mpi_preconnect_mpi**
Setup a fully connected topology during **MPI_INIT**. This might be beneficial for communication intensive applications.

• **Intel MPI**

**I_MPI_PLATFORM=<platform>**
Set the intended optimisation platform. See the Intel MPI documentation for the list of possible <platform> specifications.

**I_MPI_PERHOST=<value>**
Specify the default placement of processes on nodes. Possible selections of <value>:

<table>
<thead>
<tr>
<th>N</th>
<th>Place N processes per node.</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>All logical CPUs on a node.</td>
</tr>
<tr>
<td>allcores</td>
<td>All cores (physical CPUs) on a node.</td>
</tr>
</tbody>
</table>

**I_MPI_PIN**
Turn on/off process pinning. The default is to enable process pinning.

**I_MPI_PIN_PROCESSOR_LIST**
Define the processor placement. See the Intel MPI documentation for a detailed description of available specifications. To print pinning information set the **I_MPI_DEBUG** environment variable equal to 4.

For example, to run 8 mpi processes on two 2-socket 8-core nodes (node0 and node1) pinning to core id's 0, 1 (on the first socket) and 8, 9 (on the second socket) on each node, specify:

```bash
$ export I_MPI_PIN_PROCESSOR_LIST='grain=2,shift=4'
$ export I_MPI_DEBUG=4
$ mpirun -n 8 ./mpi_prog
```

<table>
<thead>
<tr>
<th>[0] MPI startup(): Rank</th>
<th>Pid</th>
<th>Node name</th>
<th>Pin cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>21075</td>
<td>node0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>21076</td>
<td>node0</td>
<td>1</td>
</tr>
</tbody>
</table>
**I_MPI_PIN_DOMAIN**

Define a number of non-overlapping subsets (domains) of logical processors on a node and start 1 MPI process per domain. Specifying this variable is useful when running hybrid MPI/OpenMP applications. Threads started within a domain can freely migrate between logical processors in the domain.

For example, to run 4 MPI processes on a 16 core node with 4 OpenMP threads per MPI process, use:

```bash
$ export OMP_NUM_THREADS=4
$ export I_MPI_PIN_DOMAIN=omp
$ export I_MPI_DEBUG=4
$ mpirun -n 4 ./mpi_omp_prog
```

<table>
<thead>
<tr>
<th>Rank</th>
<th>Pid</th>
<th>Node name</th>
<th>Pin cpu</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>63703</td>
<td>node0</td>
<td>{0,1,2,3}</td>
</tr>
<tr>
<td>1</td>
<td>63704</td>
<td>node0</td>
<td>{4,5,6,7}</td>
</tr>
<tr>
<td>2</td>
<td>63705</td>
<td>node0</td>
<td>{8,9,10,11}</td>
</tr>
<tr>
<td>3</td>
<td>63706</td>
<td>node0</td>
<td>{12,13,14,15}</td>
</tr>
</tbody>
</table>

See the Intel MPI documentation for a description of available specifications for **I_MPI_PIN_DOMAIN**.

**I_MPI_DEBUG**

Print out debugging information when an MPI program starts running.

**I_MPI_OUTPUT_CHUNK_SIZE=<size>**

Set the size of stdout/stderr output buffer. The default size is 1 KB.

**I_MPI_EAGER_THRESHOLD=<size>**

Specify the eager/rendezvous protocol message size for point-to-point communication. Default size is 256 KB.

**I_MPI_INTRANODE_EAGER_THRESHOLD=<size>**

Specify the eager/rendezvous protocol message size for intra-node communication. Default size is 256 KB.

**I_MPI_INTRANODE_DIRECT_COPY**

Turn on/off the intranode direct copy communication mode. Messages shorter than the specified **I_MPI_INTRANODE_EAGER_THRESHOLD** size are transferred using shared memory, and messages larger are transferred through the direct process memory access.

**I_MPI_DYNAMIC_CONNECTION**

Turn on/off the dynamic connection establishment. If disabled a fully connected topology is established upfront. This is the default for less than 64 processes.

• **Platform MPI**

**MPIRUN_OPTIONS**

This variable can be used for specifying additional command-line arguments to the *mpirun* command.

**MPI_FLAGS=[flag]**

This variable modifies the general behavior of Platform MPI. The syntax is a comma-separated list of flags.

For example, turn on function parameter error checking, specify:

```bash
$ export MPI_FLAGS=Eon
```

For example, to enable zero-buffering mode, specify:

```bash
$ export MPI_FLAGS=z
```
This can be used to verify that the application is MPI-safe, i.e. not dependent on a particular implementation of MPI with specific default settings for buffer sizes.

**MPI_CPU_AFFINITY=[option]**
Bind ranks to logical processors according to a given binding strategy. E.g. schedule ranks on CPUs according to packed rank ID:

```bash
$ export MPI_CPU_AFFINITY=rank
```

**MPI_BIND_MAP**
Allows specification of the integer CPU numbers when binding ranks to CPUs. E.g. map 8 MPI processes according to the given logical numbering of CPUs:

```bash
$ export MPI_BIND_MAP=0,1,4,5,2,3,6,7
$ export MPI_CPU_AFFINITY=map_cpu
```

**MPI_RDMA_MSGSIZE**
Specifies message protocol length.

### 4.2.2. OpenMP Runtime Environment Variables

The OpenMP specification defines a set of environment variables that affect several parameters of application execution during runtime. These variables are:

**OMP_NUM_THREADS num**
Sets the maximum number of threads to use in parallel regions if no other value is specified in the application. As of version 3.1 of the OpenMP specification, the environment variable takes a comma-separated number list that specifies the number of threads at each nesting level.

**OMP_SCHEDULE type[,chunk_size]**
Sets the iteration scheduling policy for parallel loops to one of **static**, **dynamic**, **guided** or **auto** types, and optionally defines their chunk size.

**OMP_DYNAMIC dynamic**
Enables or disables the dynamic adjustment of threads the runtime system uses for parallel regions. Valid values for **dynamic** are **true** or **false**. When enabled, the runtime system adjusts the number of threads so that it makes the most efficient use of system resources. This is meaningful when the user has not specified the desired number of threads globally or on a per-region basis.

**OMP_NESTED nested**
Enables or disables nested parallelism, i.e. whether OpenMP threads are allowed to create new thread teams. Valid values for **nested** are **true** or **false**.

**OMP_MAX_ACTIVE_LEVELS levels**
Sets the maximum number of nested active parallel regions.

**OMP_THREAD_LIMIT limit**
Sets the maximum number of threads participating in the OpenMP program. When the number of threads in the program is not specified (e.g. with the **OMP_NUM_THREADS** variable), the number of threads used is the minimum between this variable and the total number of CPUs.

**OMP_WAIT_POLICY policy**
Defines the behavior of threads waiting on synchronization events. When **policy** is set to **ACTIVE**, threads consume processor cycles while waiting (“busy-waiting”). When it is set to **PASSIVE**, threads do not consume CPU power but may require more time to resume.

**OMP_STACKSIZE size[B|K|M|G]**
Sets the default thread stack size in kilobytes, unless the number is suffixed by B, K, M or G (bytes, KB, MB, GB, respectively).
OMP_PROC_BIND bind
Specifications whether threads may be moved between processors. If set to true, OpenMP threads should not be
moved. If set to false, the operating system may move them.

The following table shows the default values for the aforementioned environment variables for different compilers.

### Table 19. OpenMP Variables, Default Values

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>GCC</th>
<th>Intel</th>
<th>PathScale</th>
<th>Portland</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_NUM_THREADS</td>
<td>Number of processors visible to the operating system</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMP_SCHEDULE</td>
<td>dynamic, chunk size = 1</td>
<td>static, no chunk size specified</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMP_DYNAMIC</td>
<td>false</td>
<td>(not supported)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMP_NESTED</td>
<td>false</td>
<td>(not supported)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OMP_MAX_ACTIVE_LEVELS</td>
<td>unlimited</td>
<td>(undocumented)</td>
<td>(not supported)</td>
<td></td>
</tr>
<tr>
<td>OMP_THREAD_LIMIT</td>
<td>unlimited</td>
<td>(undocumented)</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>OMP_WAIT_POLICY</td>
<td>(undocumented)</td>
<td>PASSIVE</td>
<td>(undocumented)</td>
<td>ACTIVE</td>
</tr>
<tr>
<td>OMP_STACKSIZE</td>
<td>system dependent</td>
<td>4M</td>
<td>(undocumented)</td>
<td>2M</td>
</tr>
<tr>
<td>OMP_PROC_BIND</td>
<td>(undocumented)</td>
<td>false</td>
<td>(undocumented)</td>
<td></td>
</tr>
</tbody>
</table>

Besides the environment variables defined in the OpenMP standard, many implementations provide their own
environment variables. Below we present briefly the most important ones. The user is referred to the corresponding
manuals for a more detailed description of the variables and their possible values.

- GCC

  GOMP_CPU_AFFINITY
  Specifies a list of CPUs where OpenMP threads should be bound.

- Intel

  KMP_BLOCKTIME
  Sets the time (in milliseconds) that a thread should wait, after completing the execution of a parallel region,
  before sleeping.

  KMP_DETERMINISTIC_REDUCTIONS
  If enabled secure a specific ordering of the reduction operations for implementing the reduction clause for
  an OpenMP parallel region. This means, for a given number of threads, in a given parallel region, for a
given data set and reduction operation, a floating point reduction done for an OpenMP reduction clause
  will have a consistent floating point result from run to run, since round-off errors will be identical.

  KMP_DYNAMIC_MODE
  Selects the method used to determine the number of threads for a parallel region when OMP_DYNAMIC
  is set to true.

  KMP_LIBRARY
  Selects the OpenMP runtime library execution mode.

  KMP_AFFINITY
  Enables the runtime system to bind OpenMP threads to CPUs.

  For example, to run 16 threads on a node with two 8-core sockets and hyper-threading enabled (i.e. a total
  of 32 logical cores), scattering threads on physical cores, use:

  ```
  $ export OMP_NUM_THREADS=16
  $ export KMP_AFFINITY=scatter,verbose
  $ ./a.out
  ```
KMP_AFFINITY: Initial OS proc set respected:
(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,...,29,30,31)
KMP_AFFINITY: 2 packages x 8 cores/pkg x 2 threads/core (16 total cores)
...
KMP_AFFINITY: Threads may migrate across 1 innermost levels of machine
KMP_AFFINITY: Internal thread 0 bound to OS proc set {0,16}
KMP_AFFINITY: Internal thread 1 bound to OS proc set {8,24}
KMP_AFFINITY: Internal thread 2 bound to OS proc set {1,17}
...
KMP_AFFINITY: Internal thread 15 bound to OS proc set {15,31}

- Portland

MP_BIND
Set to yes or y to bind threads to physical processors. The default is to not bind processes to processors.

MP_BLIST
Specifying a list of CPUs for mapping threads to CPUs.

MPSTKZ
Specify the thread stack size.

- PathScale

PSC_OMP_AFFINITY
Enables the operating system's affinity mechanism to assign threads to CPUs. The default value is true.

PSC_OMP_AFFINITY_MAP
Specifying a list of CPUs for mapping threads to CPUs.

PSC_OMP_STACK_SIZE
Specify the thread stack size.

PSC_OMP_THREAD_SPIN
Sets the number of times that the spin loops will spin at user-level before falling back to OS schedule/reschedule mechanisms.

5. Debugging

This chapter shows how to help debugging applications using compiler options and runtime environment variables. Also in this chapter is a description on how to use debugging tools like GNU GDB and the parallel GUI debuggers Totalview and DDT.

5.1. Compiler Debug Flags and Environment Variables

The table below shows a list of debugging actions and the corresponding compiler options for different compilers.

Table 20. Compiler Debug Flags

<table>
<thead>
<tr>
<th>Action</th>
<th>Compiler</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate symbolic debugging inform-</td>
<td>GCC</td>
<td>-g</td>
</tr>
<tr>
<td>ation</td>
<td>Intel</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Portland</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PathScale</td>
<td></td>
</tr>
<tr>
<td>Add runtime array bounds checking</td>
<td>GCC</td>
<td>-fcheck=bounds (Fortran only)</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>-check bounds (Fortran only)</td>
</tr>
<tr>
<td>Action</td>
<td>Compiler</td>
<td>Option</td>
</tr>
<tr>
<td>--------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>Check for uninitialized variables</td>
<td>Portland</td>
<td>-Mbounds</td>
</tr>
<tr>
<td>PathScale</td>
<td>-ffortran-bounds-check <em>(Fortran only)</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td>GCC</td>
<td>-Wuninitialized</td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>-check uninit <em>(Fortran only)</em> &lt;br&gt; -check=uninit <em>(C/C++ only)</em></td>
</tr>
<tr>
<td></td>
<td>Portland</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>PathScale</td>
<td>-trapuv</td>
</tr>
<tr>
<td>Trap floating point exceptions:</td>
<td>GCC</td>
<td>-ffpe-trap=zero,invalid,overflow <em>(Fortran only)</em></td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>-fpe-all=0 <em>(Fortran only)</em> &lt;br&gt; -fp-trap-all=common <em>(C/C++ only)</em></td>
</tr>
<tr>
<td></td>
<td>Portland</td>
<td>-Ktrap=fp</td>
</tr>
<tr>
<td></td>
<td>PathScale</td>
<td>-TENV:simd_zmask=OFF <em>(zero-divide)</em> &lt;br&gt; -TENV:simd_imask=OFF <em>(invalid)</em> &lt;br&gt; -TENV:simd_omask=OFF <em>(overflow)</em></td>
</tr>
<tr>
<td>Add debug information for runtime traceback</td>
<td>GCC</td>
<td>-fbacktrace <em>(Fortran only)</em></td>
</tr>
<tr>
<td></td>
<td>Intel</td>
<td>-traceback</td>
</tr>
<tr>
<td></td>
<td>Portland</td>
<td>-Meh_frame a</td>
</tr>
<tr>
<td></td>
<td>PathScale</td>
<td>n/a</td>
</tr>
</tbody>
</table>

aTo be used with the PGI_TERM environment variable, see below. (The PGI compilers also has a -traceback option, however, this does not currently seem to work.)

**Environment Variables**

The following list describes a set of compiler environment variables used to control runtime debug information.

- Intel
  
  decfort_dump_flag
  
  Set this variable to y or Y to force a core dump for severe errors that do not usually cause a core file to be created. *(Fortran only)*

- Portland
  
  PGI_TERM
  
  Controls the stack traceback functionality. The runtime libraries use the value of PGI_TERM to determine what action to take when a program abnormally terminates.

  E.g. compile to trap floating point exceptions and link with the -Meh_frame option to create a stacktrace

  $ pgf95 -Ktrap=fp -Meh_frame prog.f90

  and set the PGI_TERM variable as follows:

  $ export PGI_TERM=trace

  before running the executable.

- PathScale
  
  F90_BOUNDS_CHECK_ABORT
  
  Set to YES to abort program execution on first bounds check violation when compiled with the -ffortran-bounds-check option. *(Fortran only)*
F90_DUMP_MAP
Dump memory mapping information when a segmentation fault occurs. (Fortran only)

5.2. Debuggers

5.2.1. GNU GDB

GDB [http://www.gnu.org/software/gdb/], the GNU Project debugger, is a free software debugger that supports several programming languages including C, C++ and Fortran. GDB has a command-line interface and do not contain its own graphical user interface (GUI).

GDB Commands

To begin a debug session compile the code with the option -g to add debugging information and start GDB by running the command gdb adding the executable program as argument:

$ gdb prog

Once inside the GDB environment, indicated by the (gdb) prompt, you can issue commands. A list of selected GDB commands:

help - display a list of named classes of commands
run - start the program
attach - attach to a running process outside GDB
step - go to the next source line, will step into a function/subroutine
next - go to the next source line, function/subroutine calls are executed without stepping into them
continue - continue executing
break - sets breakpoint
watch - set a watchpoint to stop execution when the value of a variable or an expression changes
list - display (default 10) lines of source surrounding the current line
print - print value of a variable
backtrace - displays a stack frame for each active subroutine
detach - detach from a process
quit - exit GDB

Commands can be abbreviated to one or the first few letters of the command name if that abbreviation is unambiguous or in some cases where a single letter is specifically defined for a command. E.g. to start a program:

(gdb) r
Starting program: /path/to/executable/prog

To execute shell commands during the debugging session issue shell in front of the command, e.g.

(gdb) shell ls -l

Attaching to running processes

GDB can attach to already running processes using the attach process-id command. After attaching to a process GDB will stop it from running. This allows you to prepare the debug session using GDB commands, e.g. setting breakpoints or watchpoints. Then use the continue command to let the process continue running.

Although GDB is a serial debugger you can examine parallel programs by attaching to individual processes of the program.

The listing below displays a sample debug session attaching to one of the processes of a running MPI job for examining data. Lines starting with # are comments:

$ gdb
(gdb) # List the processes of the MPI program
(gdb) shell ps -eo pid,comm | grep mpi_prog
14957   mpi_prog
14961   mpi_prog
14962   mpi_prog
...etc.

(gdb) # Attach to one of the MPI processes
(gdb) attach 14961
Attaching to process 14961
Reading symbols from /path/to/executable/mpi_prog...done.
...etc

(gdb) # Set a watchpoint to stop execution when the variable Uc is updated
(gdb) watch Uc
Hardware watchpoint 1: Uc

(gdb) # Continue the execution of the program
(gdb) continue
Continuing.

Hardware watchpoint 1: Uc
Old value = -3.33545399
New value = -2.11184907
POTTEMP::ptemp (ldiad=...etc) at ptemp1.f90:298
298     Vc= dsdx(2,1,ie2)*u0 + dsdx(2,2,ie2)*v0 + dsdx(2,3,ie2)*w0

(gdb) # Set the list command to display 16 lines...
(gdb) set listsize 16
(gdb) # ...and display the source backwards starting 2 lines below the current one
(gdb) list +2
284     do k= 1, 8
285       kp= lnode2(k,ie2)
286     u0= u0 + u12(kp)
287     v0= v0 + u22(kp)
288     w0= w0 + u32(kp)
289     vt= vt + vtef2(kp)
290   enddo
291
292   u0= 0.125*u0;  v0= 0.125*v0;  w0= 0.125*w0;  vt= 0.125*vt
293
294
295   !----    Contravariant velocity
296   !
297   Uc= dsdx(1,1,ie2)*u0 + dsdx(1,2,ie2)*v0 + dsdx(1,3,ie2)*w0
298   Vc= dsdx(2,1,ie2)*u0 + dsdx(2,2,ie2)*v0 + dsdx(2,3,ie2)*w0
299   Wc= dsdx(3,1,ie2)*u0 + dsdx(3,2,ie2)*v0 + dsdx(3,3,ie2)*w0

(gdb) # Print a 5 element slice of the variable u12
(gdb) print u12(3006:3010)
$1 = (0.0186802763, 0.0188683271, 0.0145201795, 0.00553302653, -0.00918145757)

(gdb) # Release the process from GDB control
(gdb) detach
Detaching from program: /path/to/executable/mpi_prog, process 14961

(gdb) quit

**Examining Core Files**

Core files can be examined specifying both an executable program and the core file:

$ gdb prog core
One can also produce a core file from within the GDB session to preserve a snapshot of a program's state using the command:

```
(gdb) generate-core-file
```

**Further Information**

- For more information see the GDB Documentation [http://www.gnu.org/software/gdb/documentation/] page.

### 5.2.2. Totalview

Totalview is a GUI-based source code debugger from Rogue Wave Software, Inc. [http://www.roguewave.com/] It allows for debugging of serial and parallel codes. Program execution is controlled by stepping line by line through the code, setting breakpoints, or by setting watchpoints on variables. It is also efficient for debugging of memory errors and leaks and diagnosis problems like deadlocks.

Totalview comes with the ReplayEngine tool embedded. When debugging a program crash ReplayEngine lets you start from the point of failure and work backward in time to find the error that caused it. Notice, ReplayEngine increases the amount of memory your program uses as it keeps history and state information in memory.

TotalView works with C, C++ and Fortran applications, and supports OpenMP and several MPI implementations including MVAPICH2, Open MPI, Intel MPI and Platform MPI.

**Starting Totalview**

After compiling your MPI code with the `-g` flag, start Totalview with your executable, e.g. `mpi_prog`, by issuing the command:

```
$ totalview mpi_prog
```

Three windows, the TotalView Root window, Startup Parameters Dialog Box and the Process Windows, will appear.

The Startup Parameters Dialog Box allows you to specify arguments (command line and environment variables), standard I/O files, MPI implementation and number of MPI tasks, and enabling of ReplayEngine and memory debugging.

**Figure 1. Totalview Startup Parameters Dialog Box**
Fill in the information that allows TotalView to load your program into the Process Window.

**Figure 2. Totalview Process Window**

![Process Window Screenshot]

You are now ready to start a debugging session doing different actions, e.g.:

- Click the **Step** or **Next** buttons to go through the code statement by statement. For function calls **Step** goes into the function while **Next** executes the function.
- Create a breakpoint by clicking the line number displayed to the left in the Process Window. Click the **Go** button to run to this line.
- Monitor a variable's value by creating a watchpoint, select **Action Points > Create Watchpoint**. A watchpoint stops execution when a variable’s data changes.
- Examine variables. Dive into a variable by clicking **View > Lookup Variable** or double-click the variable using your left mouse button. The Variable Window appears.
- Visualize variables across processes by diving into a variable and click **View > Show Across > Processes** in the Variable Window.
• Examine array data. Dive into an array variable. Display array subsections by editing the Slice field in an array’s Variable Window. Show statistics information about the array (or slice of array) by clicking Tools > Statistics in the Variable Window.

Figure 3. Examining Data

Examining Core Files

If a program encounters a serious error and a process dumps a core file you can look at this file starting Totalview with:

$ totalview mpi_prog core

The Process Window displays the core file, with the Stack Trace, Stack Frame, and Source Panes showing the state of the process when it dumped core. The title bar of the Process Window names the signal that caused the core dump. The state of all variables at the time the error occurred can be examined.

Further Information

• For more information see the TotalView Documentation [http://www.roguewave.com/support/product-documentation/totalview.aspx#totalview] page.

5.2.3. DDT

Allinea [http://www.allinea.com/] DDT is a scalable, graphical debugging tool for scalar, multi-threaded and large-scale MPI applications. DDT has included a memory debugging feature that is able to detect some errors before they have caused a program crash and it can show the current state of communication between processes in a program.

DDT supports C, C++, Fortran, OpenMP, CUDA, UPC, Fortran 2008 Coarrays and many MPI implementations, including those mentioned in this guide.

36
Starting DDT

After compiling your program with -g, start DDT by issuing a command like this:

```
$ ddt mpi_prog <input arguments>
```

Two windows will appear: DDT's Main Window and the Run Dialog Box. The Run Dialog Box allows you to specify program arguments, input file, working directory, MPI implementation, number of processes, OpenMP support, CUDA support, support for memory debugging and environment variables.

Figure 4. DDT Startup Window

Press the Run button at the bottom of the dialog box, to start debugging your program.

The Main Window

DDT will now show your code in the Main Window:
The most used buttons on the toolbar with shortcut keys, from left to right, are:

- **Play/Continue (F9)**: Start/continue your program
- **Pause (F10)**: Pause your program
- **Add Breakpoint**: Press this button and add the line number. Notice, breakpoints can also be added by double-clicking a line.
- **Step Into (F5), Step Over (F8), Step Out (F6)**: Step into, over, or out of function calls.
- **Run To Line**: Press the button and enter the line number you want to run to.
- **Down Stack Frame (Ctrl-D), Up Stack Frame (Ctrl-U), Bottom Stack Frame (Ctrl-B)**: After your program has stopped inside of some function, you can use buttons to step up and down in the program's call stack.

The bottom window has a number of pages that can be selected through the tabs on the top of them. There, you can:

- Watch the output that the program prints to standard output
- Watch the breakpoints that you have set
- Monitor variables' value by creating watchpoints. A watchpoint stops execution, when a variable's data changes.
• Examine the current call stack. Clicking on a line in the call stack will make DDT show the corresponding program code. The number to the left of each line is the number of processes that are in that place in the code.
• Examine and set tracepoints. Tracepoints allow you to see what lines of code that your program is executing without stopping it. DDT prints the file and line number, when a tracepoint is reached. You can also capture the value of any variable at that point.

The window to the right allows the examination of:

• All the local variables in the current function/subroutine.
• The variables and their current values that are used in the current line(s). A group of lines can be selected.
• The current call stack with the value of each argument for each function call

Examing Data

If you right-click on a variable in the code a menu appears showing the type of the variable and for scalars also the value. For arrays select the View Array (MDA) menu option. A window will appear showing the content of the array, or e.g. showing a slice of an array as in the figure below.

Figure 6. Examining Data

Examing Core Files

If a program encounters a serious error and a process dumps a core file this can be examined in DDT. Start DDT without arguments and select Open Core Files on the Welcome Screen. This allows for selecting an executable and a core file to debug.

Further Information

6. Performance Analysis

Understanding how an application behaves during execution is important to identify and possibly eliminate bottlenecks in the code to improve program performance. This chapter describes some of the utilities, profilers and performance analysis tools available to assist in code analysing and tuning.

6.1. MPI Statistics

Some MPI implementations include statistics gathering facilities that can be used to determine potential performance problems. Below is a set of environment variables used to control MPI statistics recording.

- Intel MPI

  \[ I\_MPI\_STATS=[n-]m \]

  Collect performance statistics with various level of output information:

<table>
<thead>
<tr>
<th>n, m</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amount of data sent by each process</td>
</tr>
<tr>
<td>2</td>
<td>Number of calls and amount of transferred data</td>
</tr>
<tr>
<td>3</td>
<td>Statistics combined according to the actual arguments</td>
</tr>
<tr>
<td>4</td>
<td>Statistics defined by a buckets list</td>
</tr>
<tr>
<td>10</td>
<td>Collective operation statistics for all communication contexts</td>
</tr>
</tbody>
</table>

  \[ I\_MPI\_STATS=ipm \]

  Intel MPI also supports integrated performance monitoring (IPM) summary as part of the statistics gathering mechanism by specifying ipm as the output format. This allows for gathering statistics in regions of source codes using the MPI_Pcontrol function.

  Sample Fortran program:

  ```fortran
  program main
  include 'mpif.h'
  INTEGER :: i, numProc, myrank, rank, tot, ierr
  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,numProc,ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,myrank,ierr)
  CALL MPI_PCONTROL(1,"region_a"//char(0))
  do i = 1, numProc-1
    if(myrank == 0) then
      CALL MPI_RECV(rank,1,MPI_INTEGER,i,i,MPI_COMM_WORLD,ierr,MPI_STATUS_IGNORE)
      tot = tot + rank
    elseif(myrank == i) then
      CALL MPI_SEND(myrank,1,MPI_INTEGER,0,myrank,MPI_COMM_WORLD,ierr)
    endif
  enddo
  CALL MPI_PCONTROL(-1,"regioan_a"//char(0))
  CALL MPI_FINALIZE(ierr)
  end program
  ```

  Compile the code and run:

  ```
  $ export I_MPI_STATS=ipm
  $ mpirun -n 8 ./mpi_prog
  ```
This will produce an output file `stats.ipm` that contains region summary:

```plaintext
...<snip>
# region : region_a  [ntasks] = 8
# [total]  <avg>  min  max
# entries          8  1  1  1
# wallclock       0.000180483  2.25604e-05  1.78814e-05  3.69549e-05
# user            0  0  0  0
# system          0  0  0  0
# mpi             9.2268e-05  1.15335e-05  4.76837e-06  2.31266e-05
# %comm             51.1229       26.6667       69.403
# gflop/sec     NA          NA          NA          NA
# time          <calls>  <%mpi>  <%wall>
# MPI_Send       5.38826e-05  7  58.40     29.85
# MPI_Finalize   3.24249e-05  8  35.14     17.97
# MPI_Recv       5.96046e-06  7  6.46      3.30
# MPI_TOTAL     9.2268e-05   22  100.00    51.12
...<snip>
```

I_MPI_STATS_SCOPE=<subsystem>[:<ops>]

Select operations to collect statistics for. The `subsystem` defines the type of operation (point-to-point, collective) and `ops` defines the specific MPI routine (e.g. `bcast` for MPI_Bcast)

For example, collect information on operations `MPI_Bcast` and `MPI_Reduce`:

```
$ export I_MPI_STATS=4
$ export I_MPI_STATS_SCOPE="coll:bcast,reduce"
$ mpirun -np 4 ./mpi_prog
```

the output may look like (for process 1):

```
Process 1 of 4 on node node0 lifetime = 427.01

Data Transfers
Src     Dst     Amount(MB)      Transfers
-----------------------------------------
001 --> 000     3.814697e-06    1
001 --> 001     0.000000e+00    0
001 --> 002     0.000000e+00    0
001 --> 003     0.000000e+00    0
-----------------------------------------
Totals     3.814697e-06    1

Communication Activity
Operation      Volume(MB)      Calls
-----------------------------------------
Bcast          3.814697e-06    1
Reduce         3.814697e-06    1
-----------------------------------------

Communication Activity by actual args
Collectives
Operation     Context  Algo  Comm size  Message size  Calls  Cost(%)  
-----------------------------------------
Bcast
```

41
I_MPI_STATS_FILE
Define the statistics output file. Default name for IPM statistics is stats.ipm, otherwise stats.txt

- **Platform MPI**

**MPI_INSTR**
Enables counter instrumentation for profiling Platform MPI applications. See the documentation for available options.

For example, specifying the api option:

```bash
$ export MPI_INSTR=<filename>:api
$ mpirun -np 4 ./mpi_prog
```

will collect and write to the instrumentation output file <filename>.instr detailed information about MPI routine calls for each rank. Sample output:

```
Application Summary by Rank (second):

<table>
<thead>
<tr>
<th>Rank</th>
<th>Proc Wall Time</th>
<th>User</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15.399824</td>
<td>12.705724 (82.51%)</td>
<td>2.694100 (17.49%)</td>
</tr>
<tr>
<td>1</td>
<td>15.314737</td>
<td>12.660289 (82.67%)</td>
<td>2.654449 (17.33%)</td>
</tr>
<tr>
<td>2</td>
<td>15.346766</td>
<td>12.641753 (82.37%)</td>
<td>2.705013 (17.63%)</td>
</tr>
<tr>
<td>3</td>
<td>15.322787</td>
<td>12.698256 (82.87%)</td>
<td>2.624531 (17.13%)</td>
</tr>
</tbody>
</table>
```

---

<table>
<thead>
<tr>
<th>Rank</th>
<th>MPI_Op</th>
<th>MPI_Datatype</th>
<th>Num Calls</th>
<th>...etc.</th>
<th>Message Sizes</th>
<th>Total Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>R: 0</td>
<td>sum</td>
<td>fortran real</td>
<td>66486</td>
<td>(8 - 11016)</td>
<td>61712104</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Num Calls</th>
<th>Message Sizes</th>
<th>Total Bytes</th>
<th>Time (ms)</th>
<th>Mb / sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>58815</td>
<td>[0..64]</td>
<td>716696</td>
<td>273</td>
<td>2.627</td>
</tr>
<tr>
<td>29</td>
<td>[65..256]</td>
<td>5720</td>
<td>2</td>
<td>2.406</td>
</tr>
<tr>
<td>18</td>
<td>[257..1024]</td>
<td>9744</td>
<td>27</td>
<td>0.359</td>
</tr>
<tr>
<td>3</td>
<td>[1025..4096]</td>
<td>8928</td>
<td>0</td>
<td>135.187</td>
</tr>
<tr>
<td>7621</td>
<td>[4097..16384]</td>
<td>60971016</td>
<td>164</td>
<td>371.720</td>
</tr>
</tbody>
</table>

---
6.2. Tools

6.2.1. GNU gprof

GNU gprof is a widely used profiling tool for Unix systems which produces an execution profile of C and Fortran programs. It can show the application call graph, which represents the calling relationships between functions in the program, and the percentage of total execution time spent in each function. The tool first instruments the application executable with information that will help to build the call graph. It then runs the application and samples periodically the execution to track what code is currently being executed. This process is known as statistical sampling. Instrumentation can accurately determine the number of times a function is called, while sampling approximates the time spent within each function.

Profiling an application with gprof includes the following steps:

1. Compile and link the program with profiling enabled.

   To compile a source file for profiling, specify the -pg option when you run the compiler. This option injects special code in the executable that will collect information about the call graph during runtime, and additionally links in versions of the library routines that are compiled for profiling. It is advisable to turn on symbol information with the -g3 option when using the gcc compiler, which is necessary if the user wishes the source code to be annotated with profiling information.

2. Execute the application to generate profile data.

   In this step, the user runs the instrumented application as usual, and a special output file is generated (by default, it is named gmon.out). Because of the bookkeeping overhead of instrumentation and the use of a special system call needed for sampling, it is likely that gprof will slow down the application execution.

3. Invoke gprof to analyse and display profiling results.

   The general syntax of the command is as follows:

   ```
   $ gprof options [executable-file] [profile-data-file]
   ```

   If the executable file name is omitted, the file a.out is used by default. If the profile data file is omitted, gmon.out is used. The most important options are the following:

   - `--flat-profile`: prints the total amount of time spent and the number of calls to each function
   - `--graph`: prints the call-graph analysis from the application execution
   - `--annotated-source`: prints profiling information next to the original source code

   The main advantage of gprof is that it offers a "quick-and-dirty" way to identify hot-spots in an application, so that the programmer can focus on the most time-consuming parts of it as candidates for optimisation or parallelization. Its drawback is that it can only profile the "user-level" part of an application, i.e. it cannot profile external shared libraries (e.g. libc) or system calls. A serious side-effect of this is that multithreaded applications (e.g. OpenMP) cannot be profiled. In such cases, profiling information is gathered only for the main application thread, which might not be particularly helpful.

6.2.2. Oprofile

Introduction

OProfile is a low overhead, system-wide profiler for Linux that uses performance monitoring hardware incorporated in most modern processors to spot locations and possible causes of performance bottlenecks. In contrast to gprof, OProfile collects system-wide profiling information rather than application specific, for all threads on all processors in the system, and regardless of whether they execute user-level, shared library or kernel code. Apart from counting CPU cycles to measure where execution time is spent, the tool can also measure low-level, architecture-specific microarchitectural events such as cache misses, branch mispredictions, memory references or different types of instructions.
OProfile utilizes the underlying processor's performance counters to measure such events. Like gprof, it uses statistical sampling to associate their occurrence with specific locations in the program. This means that it does not record for every event where it happened, but for every $S$ events. The lower the value of $S$, the more accurate the results are but the higher the overhead inserted is. Generally, by setting $S$ to some moderate value we can have low overhead and at the same time quite accurate results. A list of common x86-based processors supported by OProfile is given in the following table.

Table 21. Typical x86-based processor models supported by Oprofile

<table>
<thead>
<tr>
<th>Processor</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>Atom, Core, Core2, Core i7, Nehalem, Westmere, Sandybridge, Pentium 4 (HT/non-HT), Pentium M, Pentium II/III/Pro, Itanium, Itanium 2</td>
</tr>
<tr>
<td>AMD</td>
<td>Athlon, family 10, family 11h, family12h, family14h, family15h, AMD64 (Hammer)</td>
</tr>
</tbody>
</table>

Configuration and Use

The basic steps to use OProfile are the following:

1. Setup

   The first thing we need is to load the kernel module by running the following command as root:

   ```
   # opcontrol --init
   ```

   To verify that the module was successfully loaded, run the `lsmod` command and the module should appear as oprofile. Furthermore, a `/dev/oprofile` entry should be available, with the file `/dev/oprofile/cpu_type` containing the specific model of the CPU being monitored. To get a list of events supported by the current CPU, we can simply execute:

   ```
   $ opcontrol --list-events
   ```

   Each event usually provides a set of unit masks. These masks can be used to further qualify the event according to certain aspects of its occurrence. For example, in many Intel processors we can use unit masks for data cache events to select only those that concern a specific MESI state (i.e., modified, exclusive, shared, invalid) or a combination of states (by logically or-ing the unit masks). In general, it is advised to consult the processor manuals to get a detailed description of events and possible fallacies regarding their measurement and interpretation.

   The next step is to configure OProfile to collect data for a specific event. This is done with the following command line:

   ```
   $ opcontrol --setup
   ```

   The `event` argument specifies one or more events that will be sampled. Note that, depending on the event type and the processor model, it is possible that the specified events cannot be counted simultaneously. In this case, separate runs of the application are needed, one for each event. The field `name` refers to the event name, as reported with the `list-events` option. The field `count` specifies that the processor will be sampled every `count` occurrences of the event to track the current value of the program counter. For rare events a small value may be needed to get a sufficient number of samples. If this field is left empty, a default value will be used. `unitmask` specifies a unit mask for the event, otherwise a default one will be used (usually the logical or of all supported unit masks). The `kernel` and `user` fields specify whether OProfile should sample when the processor is running in kernel or user mode, respectively.

   If no event is specified at all, then a default one will be used. This is usually a time-based event (e.g. processor cycles) configured at a sample rate of 100,000 occurrences. It can be reported using the following command:

   ```
   $ ophelp --get-default-event
   ```

   The following table lists the time-based events for some common processor models.
Table 22. Time-based events for common processor models

<table>
<thead>
<tr>
<th>Processor model</th>
<th>Event name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core, Core2, Core i7, Nehalem, Westmere, Sandybridge</td>
<td>CPU_CLK_UNHALTED</td>
</tr>
<tr>
<td>AMD</td>
<td>CPU_CLK_UNHALTED</td>
</tr>
<tr>
<td>Intel Pentium 4</td>
<td>GLOBAL_POWER_EVENTS</td>
</tr>
</tbody>
</table>

The argument `no-vmlinux` specifies that we are interested in getting profile information for user-level code only. If we want to attribute profiling samples to specific kernel functions, then the `vmlinux` option can be used, along with an uncompressed and unstripped image of the current kernel given as argument. The argument `separate=cpu` specifies that samples should be separated for each CPU in the system. This is particularly useful in order to profile parallel applications whose threads are mapped to different processors.

2. Profile application

To profile the application, we need to perform the following steps, one after the other:

```
$ opcontrol --start
$ ./program
$ opcontrol --dump
```

With the `start` option, the profiling daemon is launched. To verify that it is actually running, we check that there is a process named `oprofiled`. Once launched, the daemon starts collecting statistics for all programs running on all processors of the system. Therefore we can immediately start the application that we want to profile. After it finishes, we should collect the profiling data. By giving the `dump` option, we ensure that all the current profiling data is flushed to some special sample files before they can be analysed.

Because OProfile automatically collects data after it has started on any program that runs, we must take care to minimize "noise" inserted by other applications that run concurrently with the application under test. For this reason, we should ensure that the intervals between `opcontrol --start` and application's launch, and between application's termination and `opcontrol --dump` are as short as possible. In any case, we should strive to use a dedicated and "empty" execution environment for the profiling data to be as accurate as possible.

The daemon can be stopped by executing the command:

```
$ opcontrol --shutdown
```

To save the profiling data for future reference we can use the command:

```
$ opcontrol --save=session_name
```

This allows us to keep data from multiple experiments separate. The `session_name` can be given later as argument to a reporting tool to display the profile for a specific session.

3. Results analysis and report

After dumping samples to disk we can use the reporting tools of OProfile to analyse and display profile results. We execute the following command:

```
$ opreport --threshold=1 session:session_name
```

We get a report for the specified session, containing overall system profile regarding the events that were monitored, for all executables that ran in the profile period and had 1% (`threshold`) or more of the samples. The report lists in descending order the number of events that occurred and the name of the executable. For example, for a simple multithreaded program (`mt_test`) executed on a dual-core system, the output produced is like the following:

```
CPU: Core 2, speed 800 MHz (estimated)
Counted INST_RETIRED_ANY_P events (number of instructions retired) with
a unit mask of 0x00 (No unit mask) count 10000
```
Samples on CPU 0
Samples on CPU 1

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu:0</td>
<td>%</td>
<td>cpu:1</td>
<td>%</td>
</tr>
<tr>
<td>samples</td>
<td></td>
<td>samples</td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>554786 77.9251</td>
<td>418087 69.4634</td>
<td>mt_test</td>
<td></td>
</tr>
<tr>
<td>94429 13.2635</td>
<td>88184 14.6514</td>
<td>no-vmlinux</td>
<td></td>
</tr>
<tr>
<td>15267  2.1444</td>
<td>12080  2.0070</td>
<td>libc-2.12.2.so</td>
<td></td>
</tr>
<tr>
<td>7151  1.0044</td>
<td>16694  2.7736</td>
<td>chrome</td>
<td></td>
</tr>
</tbody>
</table>

For this session we counted the number of instructions retired on each processor, with a sample rate of 10000 instructions. We can see that for our multithreaded application there were counted 554K and 418K samples on CPU 0 and CPU 1, respectively, which corresponded to 77.9% and 69.4% of all instructions retired during the profile period. The rest of the instructions were executed in the Linux kernel (no-vmlinux), in glibc and the chrome application. Other applications that gathered less than 1% of total instructions are not shown. Note that the output of opreport contains sample count and not event count. To find the actual number of events, we have to multiply the sample count by the sample rate (10000 in the example).

It is also possible to restrict opreport to output results for a certain executable using the imag: argument. Furthermore, using the --symbols option, we can see what functions called in the executable have the most samples attributed to them. The prerequisite to have access to such information is to turn on symbol information while compiling the application using the -g option of the compiler. In the previous example, to produce output specific for mt_test application we could simply execute:

```
$ opreport --threshold=1 session:session_name image:mt_test --symbols
```

and get the following results:

```
CPU: Core 2, speed 800 MHz (estimated)
Counted INST_RETIRED_ANY_P events (number of instructions retired) with
a unit mask of 0x00 (No unit mask) count 10000
Samples on CPU 0            Samples on CPU 1
samples  %                  samples  %        symbol name
554786  100.000  418087    100.000           thread_fn
```

This shows that 100% of the samples for the mt_test application in both processors were collected while executing the thread_fn function, with the samples being actually distributed unevenly to processors.

It is possible to further refine the profiling results by attributing samples to specific source code lines or assembly instructions. This can be done using the opannotate command of OProfile as follows:

```
$ opannotate --source session:session_name image:mt_test
```

The output we get is the original source code, annotated with the profiling results as follows:

```c
/*
 * Total samples for file : "/home/user/mt_test.c"
 * 554786 100.0 418087 100.0
 */
...  
  :void* thread_fn(void *args)
  :{ /* thread_fn total: */
  :  /* 554786 100.0 418087 100.0 */
  :  ...  
  :  :if ( ta->id == 0 ) { //thread0 code
  :  222654 40.1333 0 0 : for ( i = 0; i < iters0; i++ )
  :  332132 59.8667 0 0 : s.val1++;
  :  :  else if ( ta->id == 1 ) { //thread1 code
  :  0 153575 36.7328 0 : for ( i = 0; i < iters1; i++ )
  :  0 264512 63.2672 : s.val2++;
  :  :   }
```

46
With threads being mapped on different processors, we see how samples are attributed to the lines corresponding to each thread's source code.

OProfile is a very useful tool but there are circumstances under which it can provide inaccurate results. First, the processor does not always trigger a sample interrupt on the exact instruction that caused the event. This is due to the out-of-order instruction execution performed by most modern processors, that could cause a sample to be attributed to an instruction near the one that actually triggered the event. Second, another possible cause for weird results are the aggressive optimisations done for performance reasons by most compilers. These optimisations transform the program to an intermediate form that often has little similarity with the original source code, in terms of line-by-line correspondence. Therefore, a general advice for the user is not to interpret the results of OProfile literally, but to treat them mostly as a hint on which regions of source code to look for hot-spots.

Further Information

• For more information see the online OProfile manual [http://oprofile.sourceforge.net/doc/index.html].

6.2.3. IPM

IPM [http://ipm-hpc.sourceforge.net/] is a portable profiling tool for parallel codes. It provides a low-overhead profile of the performance and resource utilization in a parallel program, focusing on communication, computation, and IO.

IPM is easy to use, it requires no modifications of the code (unless you want more information), it adds little overhead, and results are presented in text and web reports.

Using IPM

1. If IPM is installed as a module package load the proper modulefile. This will set the environment for using the IPM library and tools.

   There are two ways to use IPM, either dynamic usage (requires no relinking of the code) by preloading the library:

   $ export LD_PRELOAD=/path/to/ipm/lib/libipm.so

   or static usage by relinking the code

   $ mpicc prog.c -lipm

   (given the path to the library is set when loading the IPM modulefile).

2. Run the program. IPM will generate an XML file looking like

   username.1235369687.131403.0

3. Generate a graphical webpage using the ipm_parse perl script. First specify

   $ export IPM_KEYFILE=/path/to/ipm/include/ipm_key

   if this is not already set in the IPM modulefile. Generate the HTML file:

   $ ipm_parse -html username.1235369687.131403.0

   This will generate a subdirectory with files. Open the index.html in a browser. See sample output in the figure below.
Using Hardware Performance Monitor Counters (HPM) with IPM

IPM has the ability to hook up to a system's hardware performance monitors to study e.g. cache misses. Using HPM, also means that IPM gets less overhead, and results are more accurate. The counters are accessed by setting the IPM_HPM environment variable, for example:

```bash
$ export IPM_HPM=PAPI_FP_OPS,PAPI_TOT_INS
```

Defining regions

More detailed information can be gained by partitioning the code into regions using the MPI_Pcontrol function. For example:

```c
int main(int argc, char **argv)

...  
    MPI_Pcontrol( 1, "region-a");
    function_a();
    MPI_Pcontrol(-1, "region-a");
    MPI_Pcontrol( 1, "region-b");
    function_b();
    MPI_Pcontrol(-1, "region-b");
    ...  
    MPI_Finalize();
```

IPM will then generate separate output for each defined region.

6.2.4. Scalasca

Scalasca [http://www.scalasca.org] is a performance analysis toolset developed by the Jülich Supercomputing Centre and the German Research School for Simulation Sciences (Aachen). Scalasca supports measurement and analysis of MPI, OpenMP and hybrid MPI/OpenMP programs written in C, C++ and Fortran.
The analysis identifies potential performance bottlenecks, e.g. concerning communication and synchronization. Among the features of Scalasca is the ability to identify wait states that occur, for example, as a result of unevenly distributed workloads.

To use Scalasca add the appropriate action flag to the `scalasca` command. For an explanation of the usage, type:

```
$ scalasca -help
```

This will load the "Quick Reference" document (PDF) that contains detailed instructions on how to use Scalasca.

**Instrumenting and analysing**

In order to use Scalasca, the source code of the program to be profiled must be instrumented. This is done by prepending the `scalasca` command with the instrumentation flag when compiling the code. E.g. in a Makefile, replace

```make
FC = mpif90
LD = mpif90
```

with

```make
FC = scalasca -instrument mpif90
LD = scalasca -instrument mpif90
```

This will insert calls to the Scalasca measurement system into the code. Note that the instrumenter must also be used at the link stage.

Run the instrumented code by inserting `scalasca -analyze` in front of the job launching command, e.g. in a batch job script:

```
scalasca -analyze mpirun -np 160 ./mpi_prog
```

A unique directory is made, based on the name of the executable and the number of processes, containing log files and analysis reports with aggregate performance metrics for individual function call paths in the code. E.g. from the above example

```
epik_mpi_prog_160_sum
```

**Report examination**

The analysis reports can be processed and examined using the `scalasca -examine` command specifying the experiment reports directory:

```
$ scalasca -examine epik_mpi_prog_160_sum
```

This will launch the report viewer, see Figure 8, “Scalasca GUI”. The window is split into three panels showing:

- left panel: performance properties
- middle panel: call-tree or a flat profile of the source code
- right panel: topological view of the application’s processes

The bottom horizontal bar shows color coding according to severity, increasing from left to right.

After selecting a performance property, the middle panel shows its distribution across the call tree. Choosing one of the functions, the panel below shows the absolute value for that function and the fraction of the total (shown below the metric panel). The right panel shows the distribution across processes for that function.
Clicking the right mouse button in different parts of the panels will show additional information. For example, right-clicking on:

- a metric in the left panel choosing *Online description* will show information about that performance property
- a function name choosing *Called region > Source code* will show the source code of the corresponding function
- one of the boxes representing processes in the right panel will show the MPI process rank, the coordinate in the process grid, and values (for the metric chosen) for that process.

In addition to the automatic compiler-based instrumentation described above, Scalasca provides other ways to instrument codes. For example, instrumentation can be done manually inserting EPIK API calls (a measurement library used by Scalasca) directly into the code. Refer to the Scalasca documentation for details.

**Further Information**


### 6.2.5. TAU

TAU [http://www.cs.uoregon.edu/Research/tau/home.php](http://www.cs.uoregon.edu/Research/tau/home.php) is a performance evaluation tool that supports parallel profiling and tracing. TAU provides a suite of static and dynamic tools that provide graphical user interaction and inter-operation to form an integrated analysis environment for parallel Fortran, C++, C, Java, and Python applications. Profiling shows how much (total) time was spent in each routine, and tracing shows when the events take place in each process along a timeline. Profiling and tracing can measure time as well as hardware performance counters from the CPU.

**Instrument and run your program**

1. Set the proper environment by loading the TAU modulefile, e.g.

   ```sh
g $ module load tau
   ```

2. TAU provides three scripts, `tau_f90.sh`, `tau_cc.sh`, and `tau_cxx.sh` to instrument and compile Fortran, C, and C++ programs respectively. E.g. edit your Makefile as follows

   ```sh
   CC = tau_cc.sh
   ```
and compile your program. This will result in an instrumented executable that automatically calls the Tau library to record the values of performance registers during program execution.

Run the program.

3. At job completion you should have a set of profile.* files. These files can be packed into a single file with the command:

   $ paraprof --pack app.ppk

Analysis

1. Launch the paraprof visualization tool together with the profile data file:

   $ paraprof app.ppk

   Two windows will appear, the ParaProf Manager Window and the Main Data Window, see the figure below.

   **Figure 9. ParaProf Manager and Main Data Windows**

   ![ParaProf Manager and Main Data Windows](image)

2. The Main Data Window shows where the program used most time. The longer the bar, the more time was spent in this part of the code.
3. Right-click on the longest bar and select *Show Source Code*. This will show a file where one of the functions is highlighted in blue. This function is where your program spends most of its time.

4. Get more detailed information of the function found in the previous step.

   Exit *paraprof* and create a file named *select.tau* in the source code directory with the following content:

   ```
   BEGIN_INSTRUMENT_SECTION
   loops routine="#"
   END_INSTRUMENT_SECTION
   ```

5. Type at the command line:

   ```
   $ export TAU_OPTIONS="-optTauSelectFile=select.tau -optVerbose"
   ```

   Delete the object file (*.o) of the source code file containing the function found earlier. Recompile the code and rerun the executable.

6. Viewing the new profile data file with *paraprof* as before, in the source code only the single loop where most time is spent is highlighted, instead of the entire function.

### Using Hardware Counters for Measurement

1. For example, to analyse the code for floating-point operations versus cache misses ratio, set the environment variable *TAU_METRICS*:

   ```
   $ export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM
   ```

   Delete the profile.* files and rerun the program. This will result in 3 MULTI___* directories.

2. Pack and view the results as before.

3. In the ParaProf Manager Window choose *Options -> Show derived Metric Panel*

4. In the Expression panel click to make the derived metric "PAPI_FP_INS"/"TIME". Click apply.

5. Double-click this new metric listed. In the window that appears find the text *node 0* and single-click it.

   Another window appears listing values for the floating-point operations versus cache misses ratio.

   You can now consider how to change your program, e.g. changing the order of the loops, in order to make the ratio between floating point operations and cache misses go up. Once you have made a change, recompile your program and repeat the process listed above.

### Further Information

- For more information see the online TAU Documentation [http://www.cs.uoregon.edu/Research/tau/docs.php].

### 6.2.6. VTune

Intel's VTune [http://software.intel.com/en-us/intel-vtune-amplifier-xe] is a profiling tool for scalar, multi-threaded and MPI applications running on Intel CPUs. Although VTune can be used with MPI programs, its functionality is aimed at analysing multi-threaded programs. VTune can find lines in the source code where a program spends the most time executing and show how much of the execution time each thread was actually working.

#### Running Analysis

In order to analyse a program using VTune compile the code with the `-g` debug flag. Collecting analysis data can be done in the VTune GUI using the `amplxe-gui` command or on the command line using the `amplxe-cl` command.

The VTune GUI provides a complete environment for collecting and examining data about programs. Start the program
and create a new project by clicking the button *File > New > Project*. In the next window specify the target type:

- **Launch Application**
- **Attach to Process**
- **Profile System**

The first option can be used for running non-MPI programs while the second option is useful for attaching to one of the processes of a already running MPI program. The third option is used for performing a system wide lightweight analysis of the CPI (Cycles Per Instruction) value of all running programs and is more likely only interesting for system administrators. Once the target is selected, start the analysis *File > New > Analysis* and chose the analysis type, e.g. **Hotspots**. This will identify the most time-consuming part of the source code. Click the **Start** button to collect analysis data.

Use the VTune command-line mode for remote analysis, e.g. running in a batch system. Specify:

```
$ amplxe-cl -help
```

To see general help and add one of the available actions to see detailed help. For example:

```
$ amplxe-cl -help collect
```

will list info about data collection options.

To perform hotspots collection on the given application `prog`, specify:

```
$ amplxe-cl -collect hotspots ./prog
```

Collected data will be stored in directory `r000hs` where `r`="result", `000`=the result number and `hs`="hotspots".

**Examining results**

When the analysis run is complete the results can be viewed specifying

```
$ amplxe-gui r000hs/r000hs.amplxe
```

This will load the window below showing a summary of the collected performance data including a list of the top hotspots functions in the application.

**Figure 10. Summary Window**
Clicking the *Bottom-up* button will load the main analysis window showing the call stack function list and the sum of the time threads spent in each function. The bottom pane of the window shows a time-line for each thread with brown color showing thread activity and green idle time. Double-clicking on a function name loads the source code browser:

**Figure 11. Source Window**

The horizontal blue bar besides the shown source code lines show the sum of the time that the threads have spent on each of the lines. In the simple example above most of the time is spent updating the `sum` reduction variable in the given function.

**Further Information**

- For more information see the online Intel VTune Documentation [http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-documentation#lin].

### 7. Batch Systems

On large scale computers users must share available resources. To manage this process most systems use a workload management system or *batch system* to avoid system overload and enforce a fair share of the available resources. Users will need to specify resources they need for their job, the batch system will then queue up jobs until they are ready to run. The process of selecting which jobs to run is done according to a predetermined site policy.

#### 7.1. PBS Pro

Portable Batch System Professional (PBS Pro) is the commercial version of the PBS software offered by Altair Engineering [http://wwwpbsworks.com/Product.aspx?id=1] and originally developed for NASA.

**PBS Commands**

PBS Pro supplies a set of user commands that are used to submit, monitor, track, delete, and manipulate jobs. The table below lists a subset of these commands.
Table 23. PBS User Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsub</td>
<td>Submit a job.</td>
</tr>
<tr>
<td>qdel</td>
<td>Delete a job.</td>
</tr>
<tr>
<td>qalter</td>
<td>Alter a job's attributes.</td>
</tr>
<tr>
<td>qmove</td>
<td>Move a job to a different queue.</td>
</tr>
<tr>
<td>qhold</td>
<td>Place a hold on a queued job to prevent it from running.</td>
</tr>
<tr>
<td>qrls</td>
<td>Release hold on job.</td>
</tr>
<tr>
<td>qmsg</td>
<td>Send a message to be printed in the job output.</td>
</tr>
<tr>
<td>qstat</td>
<td>Request the status of jobs and queues.</td>
</tr>
<tr>
<td>tracejob</td>
<td>Report job history for a given jobid.</td>
</tr>
<tr>
<td>xpbs</td>
<td>Graphical User Interface to PBS commands.</td>
</tr>
</tbody>
</table>

The qsub command is used to submit a batch job to PBS. Submitting a job specifies a task, requests resources and sets job attributes. The qsub command can read from a job script, from standard input, or from the command line. The following table shows the most important options of the qsub.

Table 24. qsub Submission Options

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-N job_name</td>
<td>Specify the job name.</td>
</tr>
<tr>
<td>-A account_string</td>
<td>Specify the account number to charge for the job.</td>
</tr>
<tr>
<td>-o path</td>
<td>Specify path for the stdout file.</td>
</tr>
<tr>
<td>-e path</td>
<td>Specify path for the stderr file.</td>
</tr>
<tr>
<td>-l resource_list</td>
<td>Specify the set of resources requested for the job.</td>
</tr>
<tr>
<td>-q destination</td>
<td>Specify the job queue.</td>
</tr>
<tr>
<td>-m {a,b,e}</td>
<td>Send mail when the job begins (b), ends (e), or if it aborts (a).</td>
</tr>
<tr>
<td>-M user_list</td>
<td>List of users to whom mail about the job is sent.</td>
</tr>
<tr>
<td>-V</td>
<td>All environment variables in the qsub command’s environment are exported to the batch job.</td>
</tr>
<tr>
<td>-J range</td>
<td>Specify that the job is an array job.</td>
</tr>
</tbody>
</table>

PBS Pro also provides a set of environment variables that can be used in batch jobs. Some of these are:

PBS_JOBNAME
The job name supplied by the user using the -N option.

PBSJOBID
The job identifier assigned to the job.

PBS_O_WORKDIR
The absolute path of directory where command qsub was executed.

PBS_NODEFILE
The filename containing a list of nodes assigned to the job.

PBS_ARRAY_ID
Identifier for job arrays.
PBS Resources

Resources can be available on the server level and on the node (host) level and are specified using the `-l` option to command `qsub`. Resources made available by defining them at the server level are only used as job-wide resources. These resources (e.g. `walltime`) are requested as follows

```
-l RESOURCE=VALUE
```

A `chunk` is the smallest set of resources that will be allocated to a job. Requesting resources (e.g. number of processors) at node level is done using the `select` specification statement followed by the number of chunks `#` and a set of resources requested for each chunk separated by colons

```
-l select=</#:RESOURCE=VALUE:RESOURCE=VALUE:...
```

See the table below for a description of some of the resources provided by PBS Pro

**Table 25. PBS Resources**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>walltime</code></td>
<td>The wall clock time limit of the job.</td>
</tr>
<tr>
<td><code>cput</code></td>
<td>Amount of CPU time used by all processes for the job.</td>
</tr>
<tr>
<td><code>mem</code></td>
<td>Amount of physical memory allocated to the job.</td>
</tr>
<tr>
<td><code>ncpus</code></td>
<td>Number of processors requested for each chunk.</td>
</tr>
<tr>
<td><code>mpiprocs</code></td>
<td>Number of MPI processes to run in each chunk.</td>
</tr>
<tr>
<td></td>
<td>Defaults to 1 if <code>ncpus&gt;0</code>, 0 otherwise.</td>
</tr>
<tr>
<td><code>ompthreads</code></td>
<td>Number of threads per process in a chunk.</td>
</tr>
</tbody>
</table>

**Sample PBS Pro Job Script**

Although `qsub` can read input e.g. from the command line it is advised that users create batch job scripts that request resources for a job and contain the commands to be executed. Since such a job file is a shell script, the first line of the file specifies which shell to use to execute the script. Next follows the PBS directives requesting resources beginning with the default string “#PBS”.

For example, the following script will run a 1024 processes MPI job on a 8-processor node system. Selecting `ncpus=8` defines the `chunk size` to be equal to the size of one node. The job is run in a working subdirectory in the users home directory.

```bash
#!/bin/bash
#PBS -N test
#PBS -q normal
#PBS -l walltime=24:00:00
#PBS -l select=128:ncpus=8:mpiprocs=8

cd $PBS_O_WORKDIR

# Create the working directory
wdir=$HOME/work/$PBS_JOBNAME/$PBS_JOBID
if [ ! -d $wdir ]; then mkdir -p $wdir; fi

# Copy inputfile and move to working directory
cp case.inp $wdir
cd $wdir

# Run my MPI job
mpirun -np 1024 /path/to/executable/mpi_prog
```

Save the script, e.g. as `myjob.pbs` and submit it to the queueing system using the command `qsub`:

```
$ qsub myjob.pbs
```
When the job has submitted, PBS returns the job identifier for that job specifying a sequence_number and the servername, 384472.svcs1 in the example. The PBS_JOBID environment variable used in the script above is translated to this number.

Display the status of batch jobs:

```
$ qstat
```

<table>
<thead>
<tr>
<th>Job id</th>
<th>Name</th>
<th>User</th>
<th>Time Use</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>38432.svcs1</td>
<td>myjob</td>
<td>user1</td>
<td>287:39:0</td>
<td>R normal</td>
</tr>
<tr>
<td>38434.svcs1</td>
<td>case1</td>
<td>user1</td>
<td>262:32:5</td>
<td>R bigmem</td>
</tr>
<tr>
<td>38447.svcs1</td>
<td>test</td>
<td>user2</td>
<td>219:59:4</td>
<td>R normal</td>
</tr>
<tr>
<td>38448.svcs1</td>
<td>run.sh</td>
<td>user3</td>
<td>0</td>
<td>Q normal</td>
</tr>
</tbody>
</table>

Add the `-f` option to display full job status information for a selected job:

```
$ qstat -f 38447.svcs1
```

The job identifier is used when deleting a job:

```
$ qdel 38447.svcs1
```

### Further Information


## 7.2. Platform LSF

The IBM Platform LSF [http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf] batch system uses built-in and configured resources to track resource availability and usage. Jobs are scheduled according to the resources available on individual hosts.

Jobs submitted through the LSF system will have the resources they consume monitored while they are running. This information is used to enforce resource limits and load thresholds as well as fairshare scheduling.

### LSF Commands

The following table shows some of the most frequently used LSF batch commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsub</td>
<td>Submit job to batch subsystem.</td>
</tr>
<tr>
<td>bmod</td>
<td>Modifies job submission options of a job.</td>
</tr>
<tr>
<td>bpeek</td>
<td>Displays standard output and standard error output of unfinished jobs.</td>
</tr>
<tr>
<td>bstop</td>
<td>Suspend jobs.</td>
</tr>
<tr>
<td>bresume</td>
<td>Resume jobs.</td>
</tr>
<tr>
<td>bkill</td>
<td>Kill a job.</td>
</tr>
<tr>
<td>bhist</td>
<td>Displays historical information about jobs.</td>
</tr>
<tr>
<td>bacct</td>
<td>Display accounting statistics about finished jobs.</td>
</tr>
<tr>
<td>bqueues</td>
<td>Displays information about queues.</td>
</tr>
<tr>
<td>bjobs</td>
<td>List jobs in the batch subsystem.</td>
</tr>
<tr>
<td>bhosts</td>
<td>Display information about available hosts.</td>
</tr>
</tbody>
</table>
The `bsub` command submits a job to a host that satisfies all requirements of the job. If LSF cannot run all jobs immediately, scheduling policies determine the order of dispatch.

The most used options of the `bsub` command are listed in the table below.

### Table 27. `bsub` Submission Options

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-J job_name</code></td>
<td>Specify the job name.</td>
</tr>
<tr>
<td><code>-J job_name[index_list]</code></td>
<td>Specify the name of a job array and the index_list.</td>
</tr>
<tr>
<td><code>-P project_name</code></td>
<td>Specifies the project name to charge for the job.</td>
</tr>
<tr>
<td><code>-o path</code></td>
<td>Specify a file path for the standard output of the job.</td>
</tr>
<tr>
<td><code>-e path</code></td>
<td>Specify a file path for the standard error output of the job.</td>
</tr>
<tr>
<td><code>-W [hour:]minute</code></td>
<td>Specify the wall clock time limit of the job.</td>
</tr>
<tr>
<td><code>-c [hour:]minute</code></td>
<td>Limit the total CPU time the job.</td>
</tr>
<tr>
<td><code>-n proc</code></td>
<td>Specify the number of processors required to run the job.</td>
</tr>
<tr>
<td><code>-q queue_name</code></td>
<td>Specify the job queue.</td>
</tr>
<tr>
<td><code>-N</code></td>
<td>Send the job report by mail when the job finishes.</td>
</tr>
<tr>
<td><code>-u mail_user</code></td>
<td>Send mail to the specified email destination.</td>
</tr>
</tbody>
</table>

LSF transfers most environment variables between the submission and the execution hosts. In addition to environment variables inherited from the user environment, LSF also sets several other environment variables for batch jobs, some of these are:

- `LSB_JOBNAME`  
  Name of the job.

- `LSB_JOBID`  
  Job ID assigned by LSF.

- `LS_SUBCWD`  
  The directory from where the job was submitted.

- `LSB_HOSTS`  
  The list of hosts selected to run the job.

### Sample LSF Job Script

Build job scripts starting with the shell to use to execute the script and followed by the `bsub` directives, each line starting with "#BSUB". The last part of the script contains the commands to be executed.

For example, the following script will run a 1024 processes MPI job. The job is run in a working subdirectory in the user's home directory.

```bash
#!/bin/bash
#BSUB -J test
#BSUB -q normal
#BSUB -W 24:00
#BSUB -n 1024

cd $LS_SUBCWD

# Create the working directory
wdir=$HOME/work/$LSB_JOBNAME/$LSB_JOBID
if [ ! -d $wdir ]; then mkdir -p $wdir; fi
```

58
# Copy inputfile and move to working directory

cp case.inp $wdir

cd $wdir

# Run my MPI job

mpirun -np 1024 /path/to/executable/mpi_prog

# Report the job statistics

bjobs -l $LSB_JOBID

Save the script, e.g., as myjob.lsf and submit it to the queueing system using the command bsub:

$ bsub < myjob.lsf

Job <38447> submitted to queue <normal>

Notice the less-than sign “<”, i.e., the script is redirected as input to the bsub command. This means the script is spooled and can be modified right after submission. Leaving out the “<” means the script is not spooled and later changes to myjob.lsf before the job completes may affect this job.

Display the status of batch jobs:

$ bjobs

<table>
<thead>
<tr>
<th>JOBID</th>
<th>USER</th>
<th>STAT</th>
<th>QUEUE</th>
<th>FROM_HOST</th>
<th>EXEC_HOST</th>
<th>JOB_NAME</th>
<th>SUBMIT_TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>38432</td>
<td>user1</td>
<td>RUN</td>
<td>normal</td>
<td>svcs1</td>
<td>node16</td>
<td>myjob</td>
<td>May 13 07:34</td>
</tr>
<tr>
<td>38434</td>
<td>user1</td>
<td>RUN</td>
<td>bigmem</td>
<td>svcs1</td>
<td>node18</td>
<td>case1</td>
<td>May 14 08:48</td>
</tr>
<tr>
<td>38447</td>
<td>user2</td>
<td>RUN</td>
<td>normal</td>
<td>svcs1</td>
<td>node21</td>
<td>test</td>
<td>May 16 03:38</td>
</tr>
<tr>
<td>38448</td>
<td>user3</td>
<td>PEND</td>
<td>normal</td>
<td>svcs1</td>
<td></td>
<td>run.sh</td>
<td>May 18 14:31</td>
</tr>
</tbody>
</table>

7.3. LoadLeveler

The IBM Tivoli Workload Scheduler LoadLeveler [http://www-01.ibm.com/software/tivoli/products/scheduler-loadleveler/] is a parallel job scheduling system that allows users to run more jobs in an efficient way by maximizing resource utilization.

LoadLeveler Commands

A set of useful LoadLeveler commands is listed in the table below.

**Table 28. LoadLeveler Commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>llsubmit</td>
<td>Submit a job.</td>
</tr>
<tr>
<td>llhold</td>
<td>Place a hold on a job. Release the job by adding the (--r) switch.</td>
</tr>
<tr>
<td>llcancel</td>
<td>Cancel job from the queue.</td>
</tr>
<tr>
<td>llclass</td>
<td>Returns information about classes.</td>
</tr>
<tr>
<td>llq</td>
<td>Query information about jobs in the queues.</td>
</tr>
<tr>
<td>llstatus</td>
<td>Returns status information about nodes in the cluster.</td>
</tr>
</tbody>
</table>

Job submission to LoadLeveler is done using a job command file. A job command file is a shell script containing LoadLevler keywords. These keywords inform LoadLeveler of the job’s environment and the resources required for the job to run. A list of the most used keywords is shown in the table below.

**Table 29. LoadLeveler Keywords**

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job_name</td>
<td>Specify the name of the job.</td>
</tr>
<tr>
<td>account_no</td>
<td>Specify the account number to charge for the job.</td>
</tr>
<tr>
<td>Keyword</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>class</td>
<td>Specify the name of the class were the job will be run. Use command <code>llclass</code> to find available job classes.</td>
</tr>
<tr>
<td>job_type</td>
<td>Specify <code>serial</code> for serial and SMP (OpenMP) programs, <code>parallel</code> for IBM POE jobs, <code>MPICH</code> for programs running other MPI implementations.</td>
</tr>
<tr>
<td>node</td>
<td>Specify the number of nodes requested by a job step.</td>
</tr>
<tr>
<td>tasks_per_node</td>
<td>Specify the number of tasks per node.</td>
</tr>
<tr>
<td>task_geometry</td>
<td>Group tasks of a parallel job step to run together on the same node.</td>
</tr>
<tr>
<td>node_usage</td>
<td>Specify whether the job shares nodes with other jobs.</td>
</tr>
<tr>
<td>resources</td>
<td>Specify the resources consumed by each task of a job step.</td>
</tr>
<tr>
<td>dependency</td>
<td>Set dependencies between job steps.</td>
</tr>
<tr>
<td>output</td>
<td>Specify the name of the file to use as standard output.</td>
</tr>
<tr>
<td>error</td>
<td>Specify the name of the file to use as standard error.</td>
</tr>
<tr>
<td>wall_clock_limit</td>
<td>Set the wall clock time limit for the job.</td>
</tr>
<tr>
<td>environment</td>
<td>Specify login initial environment variables set by LoadLeveler. Set <code>COPY_ALL</code> to copy all environment variables from your shell.</td>
</tr>
<tr>
<td>env_copy</td>
<td>Specify <code>all</code> to copy environment variables to all nodes.</td>
</tr>
<tr>
<td>notification</td>
<td>Specify when to notify user by mail. Default is mail sent when job ends.</td>
</tr>
<tr>
<td>queue</td>
<td>Place the job step in the queue.</td>
</tr>
</tbody>
</table>

A group of environment variables are set by LoadLeveler for all jobs, including the following:

**LOADL_JOB_NAME**
Name of the job.

**LOADL_STEP_ID**
The job step ID.

**LOADL_STEP_INITDIR**
The initial working directory.

**LOADL_HOSTFILE**
The file that contains the host names assigned to all the tasks of the step.

**LOADL_TOTAL_TASKS**
Specifies the total number of tasks of the job step. This variable is available only when the job_type is set to MPICH.

### Sample LoadLeveler Job Script

When building a LoadLeveler job script the keywords are listed at the top of the script embedded in comments beginning with “# @” followed by the command statement section.

For example, the following script will run a 1024 processes MPI job on a 16-processor node system. The job is run in a working subdirectory in the users home directory.

```bash
#!/bin/bash
# @ job_name         = test
# @ job_type         = MPICH
# @ class            = normal
# @ wall_clock_limit = 24:00:00
# @ node             = 64
# @ tasks_per_node   = 16
```
# @ queue

cd $LOADL_STEP_INITDIR

# Create the working directory
wdir=$HOME/work/$LOADL_JOB_NAME/$LOADL_STEP_ID
if [ ! -d $wdir ]; then mkdir -p $wdir; fi

# Copy inputfile and move to working directory
cp case.inp $wdir
cd $wdir

# Run my MPI job
mpirun -np 1024 /path/to/executable/mpi_prog

# Report the job statistics
llq -w $LOADL_STEP_ID

Save the script, e.g. as myjob.ll and submit it to the queueing system using the command llsubmit:

$ llsubmit myjob.ll
llsubmit: The job "svcs1.38447" has been submitted

Display the status of batch jobs:

$ llq

<table>
<thead>
<tr>
<th>Id</th>
<th>Owner</th>
<th>Submitted</th>
<th>ST</th>
<th>PRI</th>
<th>Class</th>
<th>Running On</th>
</tr>
</thead>
<tbody>
<tr>
<td>svcs1.38432.0</td>
<td>user1</td>
<td>5/13 07:34</td>
<td>R</td>
<td>50</td>
<td>normal</td>
<td>node16</td>
</tr>
<tr>
<td>svcs1.38434.0</td>
<td>user1</td>
<td>5/14 08:48</td>
<td>R</td>
<td>50</td>
<td>bigmem</td>
<td>node18</td>
</tr>
<tr>
<td>svcs1.38447.0</td>
<td>user2</td>
<td>5/16 03:30</td>
<td>R</td>
<td>50</td>
<td>normal</td>
<td>node21</td>
</tr>
<tr>
<td>svcs1.38448.0</td>
<td>user3</td>
<td>5/18 14:31</td>
<td>I</td>
<td>50</td>
<td>normal</td>
<td></td>
</tr>
</tbody>
</table>

Further Information