The NCIT Cluster Resources User's Guide

Version 3.0

Ruxandra CIOROMELA, Alexandra - Nicoleta FIRICA
{ruxandra.cioromela12|alexandra.firica}@gmail.com

Adrian LASCATEU, Cristina ILIE
{adrian.lascateu|cristina.ilie}@cti.pub.ro

Alexandru HERISANU, Sever APOSTU, Alexandru GAVRILA
{heri|sever|alexg}@hpc.pub.ro

Release date: 23 July 2010

Contents

1 Latest Changes

This is a document concerning the use of NCIT and CoLaborator cluster resources. It was developed during both the GridInitiative 2008 Summer School and the GridInitiative 2010 Summer School.

Several other versions will follow, please consider this simply as a rough sketch.

v.1.0	jul 2008	S.A, A.G	Initial release
v.1.1	2 nov 2008	H.A	Added examples, reformatted LaTex code
v.2.0	jul 2009	A.L, C.I.	Added chapters 7, 8 and 9 Updated chapters 3, 5 and 6

2 Introduction

The **NCIT** (National Center for Information Technology) of "Politehnica" University of Bucharest started back in 2001 with the creation of **CoLaborator**, a Research Base with Multiple Users (R.B.M.U) for High Performance Computing (HPC) , that took part of a World Bank project.
CoLaborator was designed as a path of communication between Universities, at a national level, using the national network infrastructure for education and research.

Back in 2006, **NCIT**'s infrastructure was enlarged with the creation of a second, more powerful, computing site, more commmonly referred to as the **NCIT Cluster**.

Both clusters are used in research and teaching purposes by teachers, PhD students, grad students and students alike. In the near future a single sign-on (SSO) scenario will be implemented, with the same users' credentials across both sites, using the already existing LDAP infrastructure behind the `http://curs.cs.pub.ro` project.

This document was created with the given purpose of serving as an introduction to the parallel computing paradigm and as a guide to using the clusters' specific resources. You will find within the next paragraphs descriptions of the various existing hardware architectures, operating and programming environments, as well as further (external) information as the given subjects.

2.1 The Cluster

Although the two computing sites are different administrative entities and have different locations, the approach we will use throughout this document will be that of a single cluster with various platforms. This approach is justified also by the future 10Gb Ethernet link between the two sites.

Given both the fact that the cluster was created over a rather large period of time (and it keeps changing, since new machines will continue to be added), and that there is a real need for software testing on multiple platforms, the clusters structure is a heterogenous one, in terms of hardware platforms and operating/programming environments.

There are, currently, four different platforms available on the cluster:

- Intel x86 Linux
- Intel x86_64 Linux
- Sun SPARC Solaris

- Sun Intel x86_64 Solaris

Not all of these platforms are currently given a frontend, the only frontends available at the moment being **fep.hpc.pub.ro** and **fep.grid.pub.ro** machines (**F**ront **E**nd **P**rocessors). The machines behind them are all running non-interactive, being available **only** to jobs sent **through** the frontends.

2.2 Software Overview

The tools used in the **NCIT and CoLaborator cluster** for software developing are Sun Studio, OpenMP and OpenMPI. For debugging we use the TotalView Debugger and Sun Studio and for profiling and performance analysis - Sun Studio Performance Tools.

Sun Studio and GNU compilers were used to compile our tools. The installation of all the tools needed was done using the repository available online at `http://storage.grid.pub.ro/GridInitRepo/GridInitRepo.repo`.

2.3 Further Information

The latest version of this document will be kept online at `http://gridinitiative.ncit.pub.ro/clusterguide`.

You may find the latest news on the **CoLaborator** teaching activities and ongoing projects on `http://hpc.pub.ro`.

For any questions or feedback on this document, please feel free to write us at **mail@hpc.pub.ro**.

3 Hardware

This section covers in detail the different hardware architectures available in the cluster.

3.1 Configuration

The following table contains the list of all the nodes available for general use. There are also various machines which are curently used for maintenance purposes. They will not be presented in the list below.

Model	Processor Type	Sockets/Cores	Memory	Hostname
IBM HS21 28 nodes	Intel Xeon E5405, 2 GHz	2/8	16 GByte	quad-wn
IBM LS22 14 nodes	AMD Opteron	2/12	16 GByte	opteron-wn
IBM QS22 4 nodes	BECell Broadband	2/4	8 GByte	cell-qs
Fujitsu Esprimo 66 nodes	Intel P4 3 GHz	1/1	2 GByte	p4-wn
Fujitsu Celsius 2 nodes	Intel Xeon 3 GHz	2/2	2 GByte	dual-wn

3.2 Intel Xeon Based Machines

3.2.1 The Intel Xeon Processor

The Intel Xeon Processor refers to many families of Intel's x86 multiprocessing CPUs for dual-processor (DP) and multi-processor (MP) configuration on a single motherboard targeted at non-consumer markets of server and workstation computers, and also at blade servers and embedded systems.The Xeon CPUs generally have more cache than their desktop counterparts in addition to multiprocessing capabilities.

The Intel Xeon Processor with 800MHz System Bus available at **NCIT Cluster** is a 90nm process technology processor, with a 3.00 Ghz clock frequency, a 16KB L1 Cache, 1024KB full speed L2 Cache with 8-way associativity and Error Correcting Code (ECC).

It provides binary compatibility with applications running on previous members of Intel's IA-32 architecture, hyper-threading technology, enhanced branch prediction and enables system support for up to 64 GB pf physical memory.

3.2.2 IBM eServer xSeries 336

The IBM eServer xSeries 336 servers available at **NCIT Cluster** are 1U rack-mountable corporate business servers, each with one Intel Xeon 3.0 GHz processor with Intel Extended Memory 64 Technology and upgrade possibility, Intel E7520 Chipset Type and a Data Bus Speed of 800MHz.

They come with 512MB DDR2 SDRAM ECC main memory working at 400 MHz (upgradable to a maximum of 16GB), one Ultra320 SCSI integrated controller and one UltraATA 100 integrated IDE controller. Network wise, they have two network interfaces, Ethernet 10Base-T/100Base-TX/1000BaseT (RJ-45).

Eight such machines are available at **NCIT Cluster**, one of which is the second frontend, **fep.grid.pub.ro**.

More information on the IBM eServer xSeries 336 can be found on IBM's support site, http://ibm.com/systems.

3.2.3 Fujitsu-SIEMENS PRIMERGY TX200 S3

The Fujitsu-SIEMENS PRIMERGY TX200 S3 servers available at **NCIT Cluster** have each two Intel Dual-Core Xeon 3.0Ghz with Intel Extended Memory 64 Technology and upgrade possibility, Intel 5000V Chipset Type and a Data Bus Speed of 1066MHz. These processors have 4096KB of L2 Cache, ECC.

They come with 1024MB DDR2 SDRAM ECC main memory, upgradable to a maximum of 16GB, 2-way interleaved, working at 400 MHz., one 8-port SAS variant controller, one Fast-IDE controller and a 6-port controller.

They have two network interfaces, Ethernet 10Base-T/100Base-TX/1000BaseT(RJ-45).

More information on the Fujitsu-SIEMENS PRIMERGY TX200 S3 can be found on Fujitsu-SIEMENS's site, site1 or on site2.

3.3 Fujitsu Esprimo Machines

Core machines. There are curently 66 Fujitsu Esprimo, model P5905, available. They each have an Intel Pentium 4 3.0Ghz CPU, with 2048KB L2 cache, 2048MB DDR2 man memory (upgradable to a maximum of 4GB) working at 533Mhz. Storage SATAII (300MB/s) 250 GB. More information can be found here.

3.4 IBM Blade Center H

There are three chassis and each can fit 14 blades in 9U. You can find general information about the model here. Currently there are two types of blades installed: 32 Intel based **HS21** blades and 4 Cell based **QS22** blades. In the near future, two AMD based **LS22** blades (AMD Opteron processor, 6 cores) will be added.

3.4.1 HS 21 blade

There are 32 H21 blades of which 28 are used for hte batch system and 4 are for developement. Each blade has an Intel Xeon quad-core processor at 2Ghz with 2x6MB L2 cache, 1333Mhz FSB and 16GB of memory. Full specifications here.

3.4.2 QS 22 blade

The Cell based QS22 blade features two dual core 3.2 GHz IBM PowerXCell 8i Processors, 512 KB L2 cache per IBM PowerXCell 8i Processor, plus 256 KB of local store memory for each eDP SPE. Memory capacity 8GB.

They have no local storage, ergo they boot over the network. For more features QS 22 features.

3.4.3 LS22 blade

There are 20 LS22 blades of which 16 are available for general use. Each blade has an Opteron six-core processor at 2,6Ghz. Full specifications here.

3.5 Storage System

The storage system is composed of the following DELL solutions: 2 PowerEgde 2900 and 2 PowerEdge 2950 servers, and 4 PowerVault MD1000 Storage Arrays.

Recently added the Lustre system which are a capacity of 14 TBytes.

3.6 Network Connections

The mainly used means of connecting is the Gigabit Ethernet.

However, there are 14 of the 16 available Opteron six-core machines which are connected through InfiniBand (40 Gbps).

4 Operating systems

There is only one operating system running in the **NCIT Cluster**: Linux.
It runs on the x86_64 and x86 platforms from Intel available at the **NCIT Cluster**.

4.1 Linux

Linux is the UNIX-like operating system. It's name comes from the Linux kernel, originally written in 1991 by Linus Torvalds. The system's utilities and libraries usually come from the GNU operating system, announced in 1983 by Richard Stallman.

The Linux release used at the **NCIT Cluster** is a RHEL (Red Hat Enterprise Linux) clone called Scientific Linux, co-developed by Fermi National Accelerator Laboratory and the European Organization for Nuclear Research (CERN).

The Linux kernel version is displayed by the command:

```
$ uname -r
```

whereas the distribution release is displayed by the command:

```
$ cat /etc/issue
```

4.2 Addressing Modes

Linux supports 64bit addressing, thus programs can be compiled and linked either in 32 or 64bit mode. This has no influence on the capacity or precision of floating point numbers (4 or 8 byte real numbers), affecting only memory addressing, the usage of 32 or 64bit pointers. Obviously, programs requiring more than 4GB of memory have to use the 64bit addressing mode.

5 Environment

5.1 Login

Logging into UNIX-like systems is done through the secure shell (**SSH**). Since usually the SSH daemon is installed by default both on Unix and Linux systems. You can log into each one of the cluster's frontends from your local UNIX machine, using the ssh command:

```
$ ssh username@fep.hpc.pub.ro
$ ssh username@fep.grid.pub.ro
$ ssh username@cell.grid.pub.ro (if access is given)
```

Usage example:

```
$ssh username@fep-53-3.grid.pub.ro
```

The username is the one already existent on curs.cs.pub.ro or the one specially provided for the potential user by the authorities in case.

Depending on your local configuration it may be necessary to use the -Y flag to enable the trusted forwarding of graphical programs.

Logging into one of the frontends from your local Windows or MAC OS X machine using only a command line is done with the use of a SSH client such as putty. There are SSH Clients for Windows or for MAC OS X that provide a graphical file manager for copying files to and from the cluster as well. Tools providing such a functionality:
Windows: WinSCP
MAC OS X: cyberduck
Both are free software.

5.2 File Management

Every user of the cluster has a home directory on a shared filesystem within the cluster. This is **$HOME**=/export/home/*username*.

Transfering files to the cluster from your local UNIX-like machine is done through the secure copy command **scp**, e.g:

```
$ scp localfile username@fep.hpc.pub.ro:
$ scp -r localdirectory username@fep.grid.pub.ro:
```

The default directory where scp copies the file is the home directory. If you want to specify a different path where to save the file, you should write the path after ":"

Example:

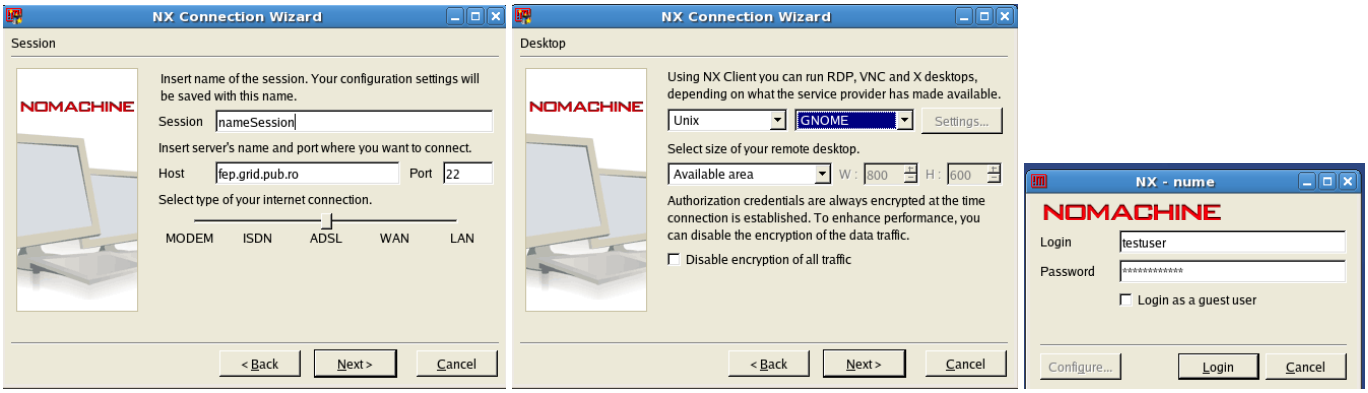
```
$ scp localfile username@fep.hpc.pub.ro:your/relative/path/to/home
$ scp localfile username@fep.hpc.pub.ro:/your/absolute/path
```

Transfering files from the cluster to your local UNIX-like machine is done through the secure copy command **scp**, e.g:

```
$ scp username@fep.hpc.pub.ro:path/to/file /path/to/destination/on/local/machine
```

WinSCP is a **scp** client for Windows that provides a graphical file manager for copying files to and from the cluster.

NX Client is a solution for secure remote access, desktop virtualization, and hosted desktop deployment. It is used both on Linux and Windows. There is a NX Server installed on fep.grid.pub.ro, which NX Clients can connect to. After installing the client, it is necessary to configure it so that the user can log in as shown in the example below. The data requested to log in is the same with the one used to begin a ssh session on the cluster .



If you want to copy an archive from a link in your current directory it is easier to use wget. You should use *Copy Link Location* (from your browser) and paste the link as parameter for wget.

Example: wget http://link/for/download

5.2.1 Sharing Files Using Subversion

Apache Subversion, more commonly known as Subversion (command name svn) is a version control system. It is mostly used in software development projects, where a team of people may alter the same files or folders. Changes are usually identified by a number (sometimes a letter) code, called the "revision number", "revision level", or simply "revision". For example, an initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and, with most types of files, merged.

First of all, a repository has to be created in order to host all the revisions. This is generally done using the create command as shown below. Note that any machine can host this type of repository, but in some cases such as our cluster you are required to have certain rights of access in order to create one.

```
$ svnadmin create /path/to/repository
```

Afterwards, the other users are provided with an address which hosts their files. Every user must install a svn version (e.g.: subversion-1.6.2) in order to have access to the svn commands and utilities. It is recommended that all the users involved in the same project use the same version.

Before getting started setting the default editor for the svn log messages is a good idea. Choose whichever editor you see fit. In the example below I chose vi.

```
$ export SVN_EDITOR=vim
```

Here are a few basic commands you should master in order to be able to use svn properly and efficiently:
- import - this command is used only once, when file sources are added for the first time; this part has been previously referred to as adding revision 1.

```
$ svn import /path/to/files/on/local/machine  /SVN_address/New_directory_for_your_project
```

- add - this command is used when you what to add a new file to the ones that are already existent. Be careful though - this phase itself does not commit the changes. It must be followed by an explicit commit command.

```
$ svn add /path/to/new_file_to_add
```

- commit - this command is used when adding a new file or when submitting the changes made to one of the files. Before the commit, the changes are only visible to the user how makes them and not to the entire team.

```
$ svn commit /path/to/file/on/local/machine -m "explicit message explaining your changes"
```

- checkout - this command is used when you want to retrieve the latest version of the project and bring it to your local machine.

```
$ svn checkout /SVN_address/Directory_of_your_project /path/to/files/on/local/machine
```

- rm - this command is used when you want to delete an existing file from your project. This change is visible to all of your team members.

```
$ svn rm /address/to/file_to_be_deleted -m message explaining your action
```

- merge - this command is used when you want to merge two or more revisions. M and N are the revision numbers you want to merge.

```
$ svn merge sourceURL1[@N] sourceURL2[@M] [WORKINGPATH]
```

OR:

```
$ svn merge -r M:HEAD /SVN_address/project_directory  /path/to/files/on/local/machine
```

The last variant merges the files from revision M with the last revision existent.

- update - this command is used when you want to update the version you have on your local machine to the latest revision. It is also an easy way to merge your file with the changes made by your team before you commit your own changes. Do not worry. Your changes will not be lost. If by any chance, both you and the other members have modified the same lines in a file, a conflict will be signaled and you will be given the opportunity to choose the final version of that line.

```
$ svn update
```

For further information and examples, here is where you can go:
- SVN Book Red Bean
- SVN Tutorial

5.3 Module Package

The **Module package** provides for the dynamic modification of the user's environment. Initialization scripts can be loaded and unloaded to alter or set shell environemnt variables such as `$PATH` or `$LD_LIBRARY_PATH`, to choose for example a specific compiler version or use software packages.

The advantage of the modules system is that environment changes can easily be undone by unloading a module. Dependencies and conflicts can be easily controlled. If, say, you need mpi with gcc then you'll just have to load both the gcc compiler and mpi-gcc modules. The module files will make all the necessary changes to the environment variables.

Note: The changes will remain active only for your current session. When you exit, they will revert back to the initial settings.

For working with modules, the **module** command is used. The most important options are explained in the following.

To get help about the module command you can either read the manual page (man module), or type

```
$ module help
```

To get the list of available modules type

```
$ module avail
--------------------- /opt/env-switcher/share/env-switcher ---------------------
batch-system/sge-6.2u5          mpi/openmpi-1.3.2_sunstudio12.1
compilers/sunstudio12.1

------------------------ /opt/modules/oscar-modulefiles ------------------------
switcher/1.0.13(default)

---------------------------- /opt/modules/version ----------------------------
3.2.5

-------------------- /opt/modules/Modules/3.2.5/modulefiles --------------------
dot         module-cvs  module-info modules     null        use.own

--------------------------- /opt/modules/modulefiles ---------------------------
apps/codesaturn-2.0.0RC1         compilers/pgi-7.0.7
apps/gaussian03                  compilers/sunstudio12.1
apps/hrm                         debuggers/totalview-8.4.1-7
apps/matlab                      debuggers/totalview-8.6.2-2
apps/uso09                       grid/gLite-UI-3.1.31-Prod
batch-system/sge-6.2u3           java/jdk1.6.0_13-32bit
batch-system/sge-6.2u5           java/jdk1.6.0_13-64bit
blas/atlas-9.11_gcc              mpi/openmpi-1.3.2_gcc-4.1.2
blas/atlas-9.11_sunstudio12.1    mpi/openmpi-1.3.2_gcc-4.4.0
cell/cell-sdk-3.1                mpi/openmpi-1.3.2_intel-11.0_081
compilers/gcc-4.1.2              mpi/openmpi-1.3.2_pgi-7.0.7
```

```
compilers/gcc-4.4.0              mpi/openmpi-1.3.2_sunstudio12.1
compilers/intel-11.0_081         oscar-modules/1.0.5(default)
```

An available module can be loaded with

```
$ module load [module name] -> $ module load compilers/gcc-4.1.2
```

A module which has been loaded before but is no longer needed can be removed using

```
$ module unload [module name]
```

If you want to use another version of a software (e.g. another compiler), we strongly recom- mend switching between modules.

```
$ module switch [oldfile] [newfile]
```

This will unload all modules from bottom up to the oldle , unload the oldle , load the newle and then reload all previously unload modules. Due to this procedure the order of the loaded modules is not changed and dependencies will be rechecked. Furthermore some modules adjust their environment variables to match previous loaded modules.

You will get a list of loaded modules with

```
$ module list
```

A short information about the software initialized by a module can be obtained by

```
$ module whatis [file]
e.g.: $ module whatis compilers/gcc-4.1.2
compilers/gcc-4.1.2  : Sets up the GCC 4.1.2 (RedHat 5.3) Environment.
```

You can add a directory with your own module files with

```
$ module use path
```

Note : If you loaded module files in order to compile a program, you probably have to load the same module files before running that program. Otherwise some necessary libraries may not be found at program start time. This is also true if using the batch system!

5.4 Batch System

A batch system controls the distribution of tasks (batch jobs) to the available machines or resources. It ensures that the machines are not overbooked, to provide optimal program execution. If no suitable machines have available resources, the batch job is queued and will be executed as soon as there are resources available. Compute jobs that are expected to run for a large period of time or use a lot of resources should use the batchsystem in order to reduce load on the frontend machines.

You may submit your jobs for execution on one of the available queues. Each of the queues has an associated environment.

To display queues summary:

```
$ qstat -g c [-q queue]
CLUSTER QUEUE                   CQLOAD   USED    RES  AVAIL  TOTAL aoACDS  cdsuE
--------------------------------------------------------------------------------
all.q                             0.00      0      0    192    200      0      8
gridinit.q                        0.02     24      0      0     24      0      0
uso09                             0.00      0      0      1      1      0      0
```

5.4.1 Sun Grid Engine

To submit a job for execution over a cluster, you have two options: either specify the command directly, or provide a script that will be executed. This behavior is controlled by the "-b y—n" parameter as follows: "y" means the command may be a binary or a script and "n" means it will be treated as a script.

Some examples of submitting jobs (both binaries and scripts).

```
$ qsub -q [queue]  -b y [executable] -> $ qsub -q queue_1 -b y /path/my_exec

$ qsub -pe [pe_name] [no_procs] -q [queue]  -b n [script]
e.g: $ qsub -pe pe_1 4 -q queue_1 -b n my_script.sh
```

To watch the evolution of the submited job, use **qstat**. Running it without any arguments shows information about the jobs submited by you alone.

To see the progress of all the jobs use the -f flag. You may also specify which queue jobs you are interested in by using the -q [queue] parameter, e.g:

```
$ qstat [-f] [-q queue]
```

Typing "watch qstat" will automatically run qstat every 2 sec. To exit type "Ctrl-C".
In order to delete a job that was previously submitted invoke the qdel command, e.g:

```
$ qdel [-f] [-u user_list] [job_range_list]
```

where:
-f - forces action for running jobs
-u - users whose jobs will be removed. To delete all the jobs for all users use -u "*".

An example of submitting a job with SGE looks like that:

```
$ cat script.sh
#!/bin/bash
‘pwd‘/script.sh
$ chmod +x script.sh
$ qsub -q queue_1 script.sh   (you may omit -b and it will behave like -b n)
```

To display the sumited jobs of all users(-u "*") or a specified user, use:

```
$ qstat [-q queue] [-u user]
```

To display extended information about some jobs, use:

```
$ qstat -t [-u  user]
```

To print detailed information about one job, use:

```
$ qstat -j job_id
```

6 Programming

This section covers in detail the programming tools available on the cluster, including compilers, debuggers and profiling tools.
On the Linux operating system the freely available GNU compilers are the somewhat "natural" choice. Code generated by the gcc C/C++ compiler performs accptably on the Opteron-based machines. Starting with version 4.2 of the gcc now offers support for shared memory parallelization with OpenMP. Code generated by the old g77 Fortran compiler typically does not perform well. Since version 4 of the GNU compiler suite a FORTRAN 95 compiler - gfortran - is available. Due to performance reasons, we recommend that Fortran programmers use the Intel or Sun compiler in 64-bit mode.
Caution: As there is an almost unlimited number of possible combinations of compilers and libraries and also the two addressing modes, 32- and 64-bit, we expect that there will be problems with incompatibilities, especially when mixing C++ compilers.

6.1 Compilers

6.1.1 General Compiling and Linker hints

To access non-default compilers you have to load the appropriate module le - using module avail to see the availables modules and the module load to load the modul (see *5.3 Module Package*). You canthen access the compilers by there original name, e.g. g++, gcc, gfortran, or by environment-variables $CXX, $CC or $FC. When, however, loading more than one compiler module, you have to be aware that environment variables point to the compiler loaded at last.
For convenient switching between compilers and platforms, we added environment variables for the most important compiler flags. These variables can be used to write a generic makefile which compiles on all our Unix like platforms:

- **FC, CC, CXX** -a variable containing the appropiate compiler name.
- **FLAGS_DEBUG** -enable debug information.
- **FLAGS_FAST** - include the options which usually offer good performance.For many compiler this will be the -fast option.
- **FLAGS_FAST_NO_FPOPT** - like fast, but disallow any floating point optimizations which will have an impact on rounding errors.
- **FLAGS_ARCH32** - build 32 bit executables or libraries.
- **FLAGS_ARCH64** - build 64 bit executables or libraries.

- **FLAGS_AUTOPAR** - enable autoparallelization, if the compiler supports it.

- **FLAGS_OPENMP** - enable OpenMP support, if supported by the compiler.

To produce debugging information in the operating systems native format use -g option at compile time.

In order to be able to mix different compilers all these variables exist also with the compiler name in the variable name, like GCC_CXX or FLAGS_GCC_FAST.

```
$ $PSRC/pex/520|| $CXX $FLAGS_FAST $FLAGS_ARCH64 $FLAGS_OPENMP $PSRC/cpop/pi.cpp
```

In general we recommend to use the same ags for compiling and for linking. Otherwise the program may not run correctly or linking may fail.

The order of the command line options while compiling and linking does matter. If you get unresolved symbols while linking, this may be caused by a wrong order of libraries. If a library xxx uses symbols out of the library yyy, the library yyy has to be right of xxx in the command line, e.g ld ... -lxxx -lyyy.

The search path for header les is extended with the -Idirectory option and the library search path with the -Ldirectory option.

The environment variable `ld_library_path` species the search path where the program loader looks for shared libraries. Some compile time linker, e.g. the Sun linker, also use this variable while linking, while the GNU linker does not.

6.1.2 Programming Hints

Generally, when developing a program, you want to make it tu run faster. In order to improve your way of coding, there are some pieces of advice, in order to use the hardware resources:

1. Turn on high optimization while compiling. The use of $FLAGS_FAST options which may be a good starting point. However keep in mind that optimization may change rounding errors of floating point calculations. You may want to use the variables supplied by the compiler modules. An optimized program runs typically 3 to 10 times faster than the non-optimized one.

2. Try another compiler. The ability of different compilers to generate efficient executables varies. The runtime differences are often between 10% to 30%.

3. Write efficient code, which can be optimized by the compiler. Look up for information regarding the compiler you want to use on the Internet.

4. Access memory continously in order to reduce cache and TLB misses. This especially effects multidimensional arrays and structures.

5. Use optimized libraries, e.g. the Sun Performance Library on the ACML library.

6. Use a profiling tool, like the Sun Collector and Analyzer, to find the compute or time intensive parts of your program, since thsese are th parts where you want to start optimizing.

7. Consider parallelization to reduce the runtime of your program.

6.1.3 GNU Compilers

The **GNU C/C++/Fortran** compilers are available by using the binaries gcc, g++, g77 and gfortran or by environment variables `$CXX, $CC or $FC`. If you cannot access them you have to load the appropiate module file as is described in section *5.3 Module Package*.

For further references the manual pages are available.

Some options to compile your program and increase their performance are:

- -m32 or -m64, to produce code with 32-bit or 64-bit addresing - as mentioned above, the default is platform dependant
- -march=opteron, to optimize for the Pentium processor (NCIT Cluster)
- -mcpu=ultrasparc optimize for the Ultrasparc I/II processors (CoLaborator)
- -O2 or -O3, for different levels of optimization
- -malign-double, for Pentium specific optimization
- -ffast-math, for floating point optimizations

GNU Compilers with versions above 4.2 support OpenMP by default. Use the -fopenmp flag to enable the OpenMP support.

6.1.4 GNU Make

Make is a tool which allows the automation and hence the efficient execution of tasks. In particular, it is used to auto-compile programs. In order to obtain an executable from more sources it is inefficient to compile every file each time and link-edit them after that. GNU Make compiles every files separately and once one of them is changed, only the modified one will be recompiled.

The tool Make uses a configuration file called Makefile. Such a file contains rules and commands of automation. Here is a very simple Makefile example which helps clarify the Make syntax.

Makefile_example1

```
all:
	gcc -Wall hello.c -o hello
clean:
	rm -f hello
```

For the execution of the example above the following commands are used:

```
$ make
   gcc -Wall hello.c -o hello

$ ./hello
hello world!
```

The example presented before contains two rules: all and clean. When run, the make command performs the first rule written in the Makefile (in this case all - the name is not particularly important).

The executed command is gcc - Wall hello.c -o hello. The user can choose explicitly what rule will be performed by giving it as a parameter to the make command.

```
$ make clean
rm -f hello

$ make all
gcc -Wall hello.c -o hello
```

In the above example, the clean rule is used in order to delete the executable hello and the make all command to obtain the executable again.

It can be seen that no other arguments are passed to the make command to specify what Makefile will be analyzed. By default, GNU Make searches, in order, for the following files: GNUmakefile, Makefile, makefile and analyzes them.

The syntax of a rule

Here is the syntax of a rule from a Makefile file:

```
target: prerequisites
<tab> command
```

* target is, usually, the file which will be obtained by performing the command "command". As we had seen from the previous examples, this can also be a virtual target, meaning that it has no file associated with it.

* prerequisites represents the dependencies needed to follow the rule. These are usually the various files needed for the obtaining of the target.

* <tab>represents the tab character and it MUST, by all means, be used before specifying the command.

* command - a list of one or more commands which are executed when the target id obtained.

Here is another example of Makefile:

Makefile_example2

```
all: hello
hello: hello.o
        gcc hello.o -o hello
hello.o: hello.c
```

```
        gcc -Wall -c hello.c
clean:
        rm -f *.o hello
```

Observation: The rule *all* is performed implicitly.
* all has a hello dependency and executes no commands.
* hello is dependent on hello.o; it makes the link-editing of the file hello.o.
* hello.o has a hello.c dependency; it makes the compiling and assembling of the hello.c file.

In order to obtain the executable, the following commands are used:

```
$ make -f Makefile_example2
gcc -Wall -c hello.c
gcc hello.o -o hello
```

The use of the variables

A Makefile file allows the use of variables. Here is an example:

Makefile_example3

```
CC = gcc
CFLAGS = -Wall -g
all: hello
hello: hello.o
$(CC) $^ -o $@
hello.o: hello.c
$(CC) $(CFLAGS) -c $<
.PHONY: clean
clean:
rm -f *.o hello
```

In the example above, the variables CC and CFLAGS were defined. CC stands for the compiler used, and CFLAGS for the options (flags) used for compiling. In this case, the options used show the warnings (-Wall) and compiling with debugging support (-g). The reference to a variable is done using the construction $(VAR_NAME). Therefore, $(CC) is replaced with gcc, and $(CFLAGS) is replaced with -Wall -g.

There are also some predefined useful variables:
* $@ expands to the name of the target;
* $^ expands to the list of requests;
* $< expands to the first request.

Ergo, the command $(CC) $^ -o $@ reads as:

```
gcc hello.o -o hello
```

and the command $(CC) $(CFLAGS) -c $< reads as:

```
gcc -Wall -g -c hello.c
```

The usage of implicit rules

Many times there is no need to specify the command that must be executed as it can be detected implicitly.This way, in case the following rule is specified :

```
main.o: main.c
```

the implicit command $(CC) $(CFLAGS) -c -o $@ $< is used.

Thus, the Makefile_example2 shown before can be simplified, using implicit rules, like this:

Makefile_example4

```
CC = gcc
CFLAGS = -Wall -g
all: hello
hello: hello.o
hello.o: hello.c
.PHONY: clean
clean:
	rm -f *.o *~ hello
```

A phony target is one that is not really the name of a file. It is just a name for some commands to be executed when you make an explicit request. There are two reasons to use a phony target: to avoid a conflict with a file of the same name, and to improve performance. If you write a rule whose commands will not create the target file, the commands will be executed every time the target comes up for remaking. Here is an example:

```
clean:
	rm *.o hello
```

Because the rm command does not create a file named "clean", probably no such file will ever exist. Therefore, the rm command will be executed every time the "make clean" command is run.

The phony target will cease to work if anything ever does create a file named "clean" in that directory. Since it has no dependencies, the file "clean" would inevitably be considered up to date, and its commands would not be executed. To avoid this problem, the explicit declaration of the target as phony, using the special target .PHONY is recommended.

```
.PHONY : clean
```

Once this is done, "make clean" will run the commands regardless of whether there is a file named "clean" or not.

Since the compiler knows that phony targets do not name actual files that could be remade from other files, it skips the implicit rule search for phony targets . This is why declaring a target phony is good for performance, even if you are not worried about the actual file existing. Thus, you first write the line that states that clean is a phony target, then you write the rule, like this:

```
.PHONY: clean
clean:
        rm *.o hello
```

It can be seen that in the Makefile_example4 implicit rules are used. The Makefile can be simplified even more, like in the example below:

Makefile_example5

```
CC = gcc
CFLAGS = -Wall -g
all: hello
hello: hello.o
.PHONY: clean
clean:
        rm -f *.o hello
```

In the above example, the rule hello.o:hello.c was deleted. Make sees that there is no file hello.o and it looks for the file C from which it can obtained. In order to do that, it creates an implicit rule and compiles the file hello.c:

```
$ make -f Makefile.ex5
gcc -Wall -g  -c -o hello.o hello.c
gcc   hello.o  -o hello
```

Generally, if we have only one source file, there is no need for a Makefile file to obtain the desired executable.

```
$ls
hello.c
$ make hello
cc hello.c -o hello
```

Here is a complete example of a Makefile using gcc. Gcc ca be easily replaced with other compilers. The structure of the Makefile remains the same.

Using all the facilites discussed up to this point, we can write a complete example using gcc (the most commonly used compiler), in order to obtain the executables from both a client and a server file.

Files used:

* the server executable depends on the C files server.c, sock.c, cli_handler.c, log.c and on the header files sock.h, cli_handler.h, log.h;
* the client executable depends on the C files client.c, sock.c, user.c, log.c and on the header files sock.h, user.h, log.h.

The structure of the Makefile file is presented below:

Makefile_example6

```
CC = gcc                # the used compiler
CFLAGS = -Wall -g       # the compiling options
LDLIBS = -lefence       # the linking options

#create the client and server executables
all: client server

#link the modules client.o user.o sock.o in the client executable
client: client.o user.o sock.o log.o

#link the modules server.o cli_handler.o sock.o in the server executable
server: server.o cli_handler.o sock.o log.o

#compile the file client.c in the object module client.o
client.o: client.c sock.h user.h log.h

#compile the file user.c in the object module user.o
user.o: user.c user.h

# compile the file sock.c in the module object sock.o
sock.o: sock.c sock.h
#compiles the file server.c in the object module server.o
server.o: server.c cli_handler.h sock.h log.h

#compile the file cli_handler.c in the object module cli_handler.o
cli_handler.o: cli_handler.c cli_handler.h

#compile the file log.c in the object module log.o
log.o: log.c log.h

.PHONY: clean
clean:
        rm -fr *.o server client
```

6.1.5 Sun Compilers

We use Sun Studio 6 on the Solaris machines (soon to be upgraded) and Sun Studio 12 on the Linux machines. Nevertheless, the use of these two versions of Sun Studio is pretty much the same.

The **Sun Studio** development tools include the Fortran95, C and C++ compilers. The best way to keep your applications free of bugs and at the actual performance level we recommend you to recompile your code with the latest production compiler. In order to check the version that your are currently using use the flag -V.

The commands that invoke the compilers are **cc, f77, f90, f95** and **CC**.

An important aspect about the Fortran 77 compiler is that from Sun Studio 7 is no longer available. Actually, the command **f77** invokes a script that is a wrapper and it is used to pass the necessary compatibility options, like **-f77**, to the f95 compiler. We recommend adding **-f77 -trap=common** in order to revert to f95 settings for error trapping. At the link step you may

want to use the **-xlang=f77** option(when linking to old f77 object binaries). Detailed information about compatibility issues between Fortran 77 and Fortran 95 can be found in http://docs.sun.com/source/816-2457/5_f77.html

For more information about the use of Sun Studio compilers you may use the man pages but you may also use the documentation found at http://developers.sun.com/sunstudio/documentation.

6.1.6 Intel Compilers

Use the module command to load the Intel compilers into your environment. The curent version of Intel Compiler is 11.1. The Intel C/C++ and Fortran77/Fortran90 compilers are invoked by the commands **icc** — **icpc** — **ifort** on Linux. The corresponding manual pages are available for further information. Some options to increase the performance of the produced code include -O3 high optimization -fp-model fast=2 enable oating point optimization -openmp turn on OpenMP -parallel turn on auto-parallelization In order to read or write big-endian binary data in Fortran programs, you can use the compiler option **-convert big_endian**.

6.1.7 PGI Compiler

PGI compilers are a set of commercially available Fortran, C and C++ compilers for High Performance Computing Systems from Portland Group.

PGI Compiler:

- PGF95 - for fortran
- PGCC - for c
- PGC++ - for c++

PGI Recommended Default Flags:

- -fast A generally optimal set of options including global optimization, SIMD vectorization, loop unrolling and cache optimizations.
- -Mipa=fast,inline Aggressive inter-procedural analysis and optimization, including automatic inlining.
- -Msmartalloc Use optimized memory allocation (Linux only).
- –zc_eh Generate low-overhead exception regions.

PGI Tuning Flags

- -Mconcur Enable auto-parallelization; for use with multi-core or multi-processor targets.
- -mp Enable OpenMP; enable user inserted parallel programming directives and pragmas.
- -Mprefetch Control generation of prefetch instructions to improve memory performance in compute-intensive loops.

- -Msafeptr Ignore potential data dependencies between C/C++ pointers.
- -Mfprelaxed Relax floating point precision; trade accuracy for speed.
- -tp x64 Create a PGI Unified Binary which functions correctly on and is optimized for both Intel and AMD processors.
- -Mpfi/-Mpfo Profile Feedback Optimization; requires two compilation passes and an interim execution to generate a profile.

6.2 OpenMPI

RPM's are available compiled both for 32and 64bit machines. It was compiled with both Sun Studio and GNU Compilers and the user may select which one to use depending on the task.

The compilers provided by OpenMPI are mpicc, mpiCC, mpic++, mpicxx, mpif77 and mpif90. Please note that mpiCC, mpic++ and mpicxx all invoke the same C++ compiler with the same options. Another aspect is that all of these commands are only wrappers that actually call opal_wrapper. Using the -show flag does not invoke the compiler, instead it prints the command that would be executed. To find out all the possible flags these commands may receive, use the -flags flag.

To compile your program with mpicc, use:

```
$ mpicc -c pr.c
```

To link your compiled program, use:

```
$ mpicc -o pr pr.o
```

To compile and link all at once, use:

```
$ mpicc -o pr pr.c
```

For the others compilers the commands are the same - you only have to replace the compiler's name with the proper one.

The mpirun command executes a program, like

```
$ mpirun [options] <program> [<args>]
```

The most used option specifies the number of cores to run the job: -n #. It is not necessary to specify the hosts on which the job would execute because this will be managed by Sun Grid Engine.

6.3 OpenMP

OpenMP is an Application Program Interface (API), jointly defined by a group of major computer hardware and software vendors. OpenMP provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on multiple architectures, including UNIX and Windows NT. This tutorial covers most of the major features of OpenMP, including its various constructs and directives for specifying parallel regions, work sharing, synchronisation and data environment. Runtime library functions and environment variables are also covered. This tutorial includes both C and Fortran example codes and an exercise.

An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism is comprised of three primary API components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

The API is specified for C/C++ and Fortran and most major platforms have been implemented including Unix/Linux platforms and Windows NT, thus making it portable. It is standardised: jointly defined and endorsed by a group of major computer hardware and software vendors and it is expected to become an ANSI standard.

6.3.1 What does OpenMP stand for?

Short answer: Open Multi-Processing

Long answer: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

OpenMP is not meant for distributed memory parallel systems (by itself) and it is not necessarily implemented identically by all vendors. It doesn't guarantee to make the most efficient use of shared memory and it doesn't require to check for data dependencies, data conflicts, race conditions or deadlocks. It doesn't require to check for code sequences that cause a program to be classified as non-conforming. It is also not meant to cover compiler-generated automatic parallel processing and directives to the compiler to assist it and the design won't guarantee that input or output to the same file is synchronous when executed in parallel. The programmer is responsible for the synchronising part.

6.3.2 OpenMP Programming Model

OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over the parallel processing. OpenMP uses the fork-join model of parallel execution. All OpenMP programs begin as a single process: the master thread. The master thread runs sequentially until the first parallel region construct is encountered.

FORK: the master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel amongst the various team threads.

JOIN: When the team threads complete, they synchronise and terminate, leaving only the master thread.

Most OpenMP parallelism is specified through the use of compiler directives which are embedded in C/C++ or Fortran source code. Nested Parallelism Support: the API provides for the placement of parallel constructs inside of other parallel constructs. Implementations may or may not support this feature.

Also, the API provides for dynamically altering the number of threads which may be used to execute different parallel regions. Implementations may or may not support this feature.

OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file. If every thread conducts I/O to a different file, the issue is

not significant. It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory, as the producers claim. In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time. When it is critical that all threads view a shared variable identically, the programmer is responsible for ensuring that the variable is FLUSHed by all threads as needed.

6.3.3 Environment Variables

OpenMP provides the following environment variables for controlling the execution of parallel code. **All environment variable names are uppercase.** The values assigned to them are not case sensitive.

OMP_SCHEDULE
Applies only to DO, PARALLEL DO (Fortran) and for, parallel for C/C++ directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors. For example:

```
setenv OMP_SCHEDULE "guided, 4"
setenv OMP_SCHEDULE "dynamic"
```

OMP_NUM_THREADS
Sets the maximum number of threads to use during execution. For example:

```
setenv OMP_NUM_THREADS 8
```

OMP_DYNAMIC
Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE. For example:

```
setenv OMP_DYNAMIC TRUE
```

OMP_NESTED
Enables or disables nested parallelism. Valid values are TRUE or FALSE. For example:

```
setenv OMP_NESTED TRUE
```

Implementation notes:
Your implementation may or may not support nested parallelism and/or dynamic threads. If nested parallelism is supported, it is often only nominal, meaning that a nested parallel region may only have one thread. Consult your implementation's documentation for details - or experiment and find out for yourself.

OMP_STACKSIZE
New feature available with OpenMP 3.0. Controls the size of the stack for created (non-Master) threads. Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
```

```
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

OMP_WAIT_POLICY
New feature available with OpenMP 3.0. Provides a hint to an OpenMP implementation about the desired behaviour of waiting threads. A compliant OpenMP implementation may or may not abide by the setting of the environment variable. Valid values are ACTIVE and PASSIVE. ACTIVE specifies that waiting threads should mostly be active, i.e. consume processor cycles, while waiting. PASSIVE specifies that waiting threads should mostly be passive, i.e. not consume processor cycles, while waiting. The details of the ACTIVE and PASSIVE behaviours are implementation defined. Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

OMP_MAX_ACTIVE_LEVELS
New feature available with OpenMP 3.0. Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. The behaviour of the program is implementation-defined if the requested value of OMP_MAX_ACTIVE_LEVELS is greater than the maximum number of nested active parallel levels an implementation can support or if the value is not a non-negative integer. Example:

```
setenv OMP_MAX_ACTIVE_LEVELS 2
```

OMP_THREAD_LIMIT
New feature available with OpenMP 3.0. Sets the number of OpenMP threads to use for the whole OpenMP program. The value of this environment variable must be a positive integer. The behaviour of the program is implementation-defined if the requested value of OMP_THREAD_LIMIT is greater than the number of threads an implementation can support or if the value is not a positive integer. Example:

```
setenv OMP_THREAD_LIMIT 8
```

6.3.4 Directives format

Fortran Directives Format

Format: (not case sensitive)

```
sentinel  directive-name  [clause ...]
```

All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend on the type of Fortran source. Possible sentinels are:

```
!$OMP
C$OMP
*$OMP
```

Example:

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
```

Fixed Form Source:

- !$OMP C$OMP *$OMP are accepted sentinels and must start in column 1.
- All Fortran fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line.
- Initial directive lines must have a space/zero in column 6.
- Continuation lines must have a non-space/zero in column 6.

Free Form Source:

- !$OMP is the only accepted sentinel. Can appear in any column, but must be preceded by white space only.
- All Fortran free form rules for line length, white space, continuation and comment columns apply for the entire directive line
- Initial directive lines must have a space after the sentinel.
- Continuation lines must have an ampersand as the last non-blank character in a line. The following line must begin with a sentinel and then the continuation directives.

General Rules:

* Comments can not appear on the same line as a directive.
* Only one directive name may be specified per directive.
* Fortran compilers which are OpenMP enabled generally include a command line option which instructs the compiler to activate and interpret all OpenMP directives.
* Several Fortran OpenMP directives come in pairs and have the form shown below. The "end" directive is optional but advised for readability.

```
!$OMP  directive
    [ structured block of code ]
!$OMP end  directive
```

C / C++ Directives Format

Format:

```
#pragma omp directive-name  [clause, ...] newline
```

A valid OpenMP directive must appear after the *pragma* and before any clauses. Clauses can be placed in any order, and repeated as necessary, unless otherwise restricted. It is required that that the *pragma* clause precedes the structured block which is enclosed by this directive.

Example:

```
#pragma omp parallel default(shared) private(beta,pi)
```

General Rules:

* Case sensitive
* Directives follow conventions of the C/C++ standards for compiler directives.
* Only one directive-name may be specified per directive.

* Each directive applies to at most one succeeding statement, which must be a structured block.

* Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

PARALLEL Region Construct

Purpose: A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.

Example:

Fortran

```
!$OMP PARALLEL [clause ...]
                     IF (scalar_logical_expression)
                     PRIVATE (list)
                     SHARED (list)
                     DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
                     FIRSTPRIVATE (list)
                     REDUCTION (operator: list)
                     COPYIN (list)
                     NUM_THREADS (scalar-integer-expression)
        block
      !$OMP END PARALLEL
```

C/C++

```
#pragma omp parallel [clause ...]  newline
                          if (scalar_expression)
                          private (list)
                          shared (list)
                          default (shared | none)
                          firstprivate (list)
                          reduction (operator: list)
                          copyin (list)
                          num_threads (integer-expression)
        structured_block
```

Notes:

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.

- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.

- There is an implicit barrier at the end of a parallel section. Only the master thread continues execution past this point.

- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

How Many Threads?

The number of threads in a parallel region is determined by the following factors, in order of precedence:

1. Evaluation of the IF clause
2. Setting of the NUM_THREADS clause
3. Use of the omp_set_num_threads() library function
4. Setting of the OMP_NUM_THREADS environment variable
5. Implementation default - usually the number of CPUs on a node, though it could be dynamic.

Threads are numbered from 0 (master thread) to N-1.

Dynamic Threads:

Use the omp_get_dynamic() library function to determine if dynamic threads are enabled. If supported, the two methods available for enabling dynamic threads are:

1. The omp_set_dynamic() library routine;
2. Setting of the OMP_DYNAMIC environment variable to TRUE.

Nested Parallel Regions:

Use the omp_get_nested() library function to determine if nested parallel regions are enabled. The two methods available for enabling nested parallel regions (if supported) are:

1. The omp_set_nested() library routine
2. Setting of the OMP_NESTED environment variable to TRUE

If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

Clauses:

IF clause: If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.

Restrictions:

A parallel region must be a structured block that does not span multiple routines or code files. It is illegal to branch into or out of a parallel region. Only a single IF clause is permitted. Only a single NUM_THREADS clause is permitted.

Example: Parallel Region - Simple "Hello World" program

- Every thread executes all code enclosed in the parallel section
- OpenMP library routines are used to obtain thread identifiers and total number of threads

Fortran - Parallel Region Example

```
INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
          +   OMP_GET_THREAD_NUM

           C     Fork a team of threads with each thread having a private TID variable
           !$OMP PARALLEL PRIVATE(TID)
```

```
      C     Obtain and print thread id
            TID = OMP_GET_THREAD_NUM()
            PRINT *, 'Hello World from thread = ', TID

      C     Only master thread does this
            IF (TID .EQ. 0) THEN
             NTHREADS = OMP_GET_NUM_THREADS()
           PRINT *, 'Number of threads = ', NTHREADS
           END IF

      C     All threads join master thread and disband
      !$OMP END PARALLEL

      END
```

C / C++ - Parallel Region Example

```
#include <omp.h>
main ()  {
int nthreads, tid;

/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master thread and terminate */
}
```

General rules of directives (for more details about these directives you can go to openMP Directives):

- They follow the standards and conventions of the C/C++ or Fortran compilers;
- They are case sensitive;
- In a directive, only one name can me specified;
- Any directive can be applied only to the statement following it, which must be a structured block.
- "Long" directives can be continued on the next lines by adding a \ at the end of the first line of the directive.

6.3.5 The OpenMP Directives

PARALLEL region: a block will be executed in parallel by OMP_NUM_THREADS number of threads. It is the fundamental construction in OpenMP.

Work-sharing structures:
DO/for - shares an iteration of a cycle over all threads (parallel data);
SECTIONS - splits the task in separated sections (functional parallel processing);
SINGLE - serialises a code section.
Synchronizing constructions:
MASTER - only the master thread will execute the region of code;
CRITICAL - that region of code will be executed only by one thread;
BARRIER - all threads from the pool synchronize;
ATOMIC - a certain region of memory will be updated in an atomic mode - a sort of critical section;

FLUSH - identifies a syncronization point in which the memory must be in a consistent mode;
ORDERED - the iterations of the cycle from this directive will be executed in the same order like the corresponding serial execution;
THREADPRIVATE - it is used to create from the global variables, local separated variables which will be executed on several parallel regions.

Clauses to set the context:
These are important for programming in a programming model with shared memory. It is used together with the PARALLEL, DO/for and SECTIONS directives.
PRIVATE - the variables from the list are private in every thread;
SHARED - the variables from the list are shared by the threads of the current team;
DEFAULT - it allows the user to set the default "PRIVATE", "SHARED" or "NONE" for all the variables from a parallel region;
FIRSTPRIVATE - it combines the functionality of the clause PRIVATE with the automated initialization of the variables from the list: the initialisation of the local variables is made using the previous value from the cycle;
LASTPRIVATE - it combines the functionality of the PRIVATE clause with a copy of the last iteration from the current section;
COPYIN - it offers the possibility to assign the same value to the variables THREADPRIVATE for all the threads in the pool;
REDUCTION - it makes a reduction on the variables that appear in the list (with a specific operation: + - * /,etc.).

6.3.6 Examples using OpenMP with C/C++

Here are some examples using OpenMP with C/C++:

```
/******************************************************************************
* OpenMP Example - Hello World - C/C++ Version
* FILE: omp_hello.c
* DESCRIPTION:
```

```
*   In this simple example, the master thread forks a parallel region.
*   All threads in the team obtain their unique thread number and print it.
*   The master thread only prints the total number of threads.  Two OpenMP
*   library routines are used to obtain the number of threads and each
*   thread's number.
* SOURCE: Blaise Barney  5/99
* LAST REVISED:
******************************************************************************/

#include <omp.h>
main ()  {
int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and disband */
}

/******************************************************************************
* OpenMP Example - Loop Work-sharing - C/C++ Version
* FILE: omp_workshare1.c
* DESCRIPTION:
*   In this example, the iterations of a loop are scheduled dynamically
*   across the team of threads.  A thread will perform CHUNK iterations
*   at a time before being scheduled for the next CHUNK of work.
* SOURCE: Blaise Barney  5/99
* LAST REVISED: 03/03/2002
******************************************************************************/

#include <omp.h>
#define CHUNKSIZE   10
#define N       100

main ()  {
```

```
int nthreads, tid, i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i,nthreads,tid)
  {
  tid = omp_get_thread_num();

  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i < N; i++)
    {
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d  c[i]= %f\n", tid,i,c[i]);
    }

  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* end of parallel section */
}

/******************************************************************************
* OpenMP Example - Sections Work-sharing - C/C++ Version
* FILE: omp_workshare2.c
* DESCRIPTION:
*   In this example, the iterations of a loop are split into two different
*   sections.  Each section will be executed by one thread.  Extra threads
*   will not participate in the sections code.
* SOURCE: Blaise Barney  5/99
* LAST REVISED: 03/03/2002
******************************************************************************/
#include <omp.h>
#define N     50

main ()
{
int i, nthreads, tid;
float a[N], b[N], c[N];

/* Some initializations */
```

```
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i,tid,nthreads)
  {
  tid = omp_get_thread_num();
  printf("Thread %d starting...\n",tid);

  #pragma omp sections nowait
    {

    #pragma omp section
      for (i=0; i < N/2; i++)
        {
        c[i] = a[i] + b[i];
        printf("tid= %d i= %d c[i]= %f\n",tid,i,c[i]);
        }

    #pragma omp section
      for (i=N/2; i < N; i++)
        {
        c[i] = a[i] + b[i];
        printf("tid= %d i= %d c[i]= %f\n",tid,i,c[i]);
        }
    }  /* end of sections */

  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  }  /* end of parallel section */
}

/******************************************************************************
* OpenMP Example - Combined Parallel Loop Reduction - C/C++ Version
* FILE: omp_reduction.c
* DESCRIPTION:
*   This example demonstrates a sum reduction within a combined parallel loop
*   construct.  Notice that default data element scoping is assumed - there
*   are no clauses specifying shared or private variables.  OpenMP will
*   automatically make loop index variables private within team threads, and
*   global variables shared.
* SOURCE: Blaise Barney  5/99
* LAST REVISED:
******************************************************************************/
```

```
#include <omp.h>

main ()  {
int   i, n;
float a[100], b[100], sum;

/* Some initializations */
n = 100;
for (i=0; i < n; i++)
  a[i] = b[i] = i * 1.0;
sum = 0.0;

#pragma omp parallel for reduction(+:sum)
  for (i=0; i < n; i++)
    sum = sum + (a[i] * b[i]);
printf("   Sum = %f\n",sum);
}
```

6.3.7 Running OpenMP

On Linux machines. GNU C Compiler now provides integrated support for OpenMP. To compile your programs to use "#pragma omp" directives use the -fopenmp flag in addition to the gcc command.

In order to compile on a local station, for a C/C++ program the command used is:

- `gcc -fopenmp my_program.c`

In order to compile on fep.grid.pub.ro , the following command can be used (gcc):

- `gcc -fopenmp -xopenmp -xO3 file_name.c -o binary_name`

For defining the number of threads use a structure similar to the following one:

```
#define NUM_THREADS 2
```

combined with the function omp_set_num_threads(NUM_THREADS). Similarily,

```
export OMP_NUM_THREADS=4
```

can be used in the command line in order to create a 4 thread-example.

6.3.8 OpenMP Debugging - C/C++

In this section, there are some C and Fortran programs examples using OpenMP that have bugs. We will show you how to fix these programs, and we will shortly present a debugging tool, called TotalView, that will be explained later in the documentation.

```
/******************************************************************************
* OpenMP Example - Combined Parallel Loop Work-sharing - C/C++ Version
* FILE: omp_workshare3.c
```

```
* DESCRIPTION:
*   This example attempts to show use of the parallel for construct.  However
*   it will generate errors at compile time.  Try to determine what is causing
*   the error.  See omp_workshare4.c for a corrected version.
* SOURCE: Blaise Barney  5/99
* LAST REVISED: 03/03/2002
******************************************************************************/

#include <omp.h>
#define N       50
#define CHUNKSIZE   5

main ()  {
int i, chunk, tid;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel for     \
  shared(a,b,c,chunk)            \
  private(i,tid)             \
  schedule(static,chunk)
  {
  tid = omp_get_thread_num();
  for (i=0; i < N; i++)
    {
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
  }  /* end of parallel for construct */
}
```

The output of the gcc command, is:

```
[testuser@fep-53-2 ~]$ gcc -fopenmp test_openmp.c -o opens
test_openmp.c: In function \u2018main\u2019:
test_openmp.c:19: error: for statement expected before \u2018{\u2019 token
test_openmp.c:24: warning: incompatible implicit declaration
              of built-in function \u2018printf\u2019
```

The cause of these errors is the form of the code that follows the pragma declaration. It is not allowed to include code between the parallel for and the for loop. Also, it is not allowed to include the code that follows the pragma declaration between parenthesis (e.g.: {}).

The revised, correct form of the program above, is the following:

```
/******************************************************************************
* OpenMP Example - Combined Parallel Loop Work-sharing - C/C++ Version
* FILE: omp_workshare4.c
* DESCRIPTION:
*   This is a corrected version of the omp_workshare3.c example. Corrections
*   include removing all statements between the parallel for construct and
*   the actual for loop, and introducing logic to preserve the ability to
*   query a thread's id and print it from inside the for loop.
* SOURCE: Blaise Barney  5/99
* LAST REVISED: 03/03/2002
******************************************************************************/

#include <omp.h>
#define N       50
#define CHUNKSIZE   5

main ()  {
int i, chunk, tid;
float a[N], b[N], c[N];
char first_time;

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
first_time = 'y';

#pragma omp parallel for     \
  shared(a,b,c,chunk)            \
  private(i,tid)             \
  schedule(static,chunk)     \
  firstprivate(first_time)

  for (i=0; i < N; i++)
    {
    if (first_time == 'y')
      {
      tid = omp_get_thread_num();
      first_time = 'n';
      }
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
}
```

If we easily detected the error above only by taking into consideration various syntax matters, things won't work as simply every time. There are errors that cannot be detected on compiling. In

this case, a specialized debugger, called TotalView, is used. More details about these debuggers you can find at the "Debuggers" section.

```
/******************************************************************************
* FILE: omp_bug2.c
* DESCRIPTION:
*   Another OpenMP program with a bug.
******************************************************************************/
#include <omp.h>

main ()  {
int nthreads, i, tid;
float total;

/*** Spawn parallel region ***/
#pragma omp parallel
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  /* Only master thread does this */
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d is starting...\n",tid);

  #pragma omp barrier

  /* do some work */
  total = 0.0;
  #pragma omp for schedule(dynamic,10)
  for (i=0; i<1000000; i++)
     total = total + i*1.0;
  printf ("Thread %d is done! Total= %f\n",tid,total);
  } /*** End of parallel region ***/
}
```

The bugs in this case are caused by neglecting to scope the TID and TOTAL variables as PRIVATE. By default, most OpenMP variables are scoped as SHARED. These variables need to be unique for each thread. It is also necessary to include stdio.h in order to have no warnings.

The repaired form of the program, is the following:

```
#include <omp.h>
#include <stdio.h>

main ()  {
int nthreads, i, tid;
float total;
```

```
/*** Spawn parallel region ***/
#pragma omp parallel private(tid,total)
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  /* Only master thread does this */
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d is starting...\n",tid);

  #pragma omp barrier

  /* do some work */
  total = 0.0;
  #pragma omp for schedule(dynamic,10)
  for (i=0; i<1000000; i++)
     total = total + i*1.0;
  printf ("Thread %d is done! Total= %f\n",tid,total);
  } /*** End of parallel region ***/
}

/******************************************************************************
* FILE: omp_bug3.c
* DESCRIPTION:
*   Run time error
* AUTHOR: Blaise Barney  01/09/04
* LAST REVISED: 06/28/05
******************************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N     50

int main (int argc, char *argv[])
{
int i, nthreads, tid, section;
float a[N], b[N], c[N];
void print_results(float array[N], int tid, int section);

/* Some initializations */
for (i=0; i<N; i++)
  a[i] = b[i] = i * 1.0;

#pragma omp parallel private(c,i,tid,section)
```

```
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  /*** Use barriers for clean output ***/
  #pragma omp barrier
  printf("Thread %d starting...\n",tid);
  #pragma omp barrier

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      section = 1;
      for (i=0; i<N; i++)
        c[i] = a[i] * b[i];
      print_results(c, tid, section);
      }

    #pragma omp section
      {
      section = 2;
      for (i=0; i<N; i++)
        c[i] = a[i] + b[i];
      print_results(c, tid, section);
      }

    }  /* end of sections */

  /*** Use barrier for clean output ***/
  #pragma omp barrier
  printf("Thread %d exiting...\n",tid);

  }  /* end of parallel section */
}

void print_results(float array[N], int tid, int section)
{
  int i,j;

  j = 1;
  /*** use critical for clean output ***/
```

```
  #pragma omp critical
  {
  printf("\nThread %d did section %d. The results are:\n", tid, section);
  for (i=0; i<N; i++) {
    printf("%e  ",array[i]);
    j++;
    if (j == 6) {
      printf("\n");
      j = 1;
      }
    }
    printf("\n");
  } /*** end of critical ***/
  #pragma omp barrier
  printf("Thread %d done and synchronized.\n", tid);
}
```

Solving the problem:

The run time error is caused by by the OMP BARRIER directive in the PRINT_RESULTS subroutine. By definition, an OMP BARRIER can not be nested outside the static extent of a SECTIONS directive. In this case it is orphaned outside the calling SECTIONS block. If you delete the line with "#pragma omp barrier" from the print_results function, the program won't hang anymore.

```
/******************************************************************************
* FILE: omp_bug4.c
* DESCRIPTION:
*   This very simple program causes a segmentation fault.
* AUTHOR: Blaise Barney  01/09/04
* LAST REVISED: 04/06/05
******************************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1048

int main (int argc, char *argv[])
{
int nthreads, tid, i, j;
double a[N][N];

/* Fork a team of threads with explicit variable scoping */
#pragma omp parallel shared(nthreads) private(i,j,tid,a)
  {

  /* Obtain/print thread info */
  tid = omp_get_thread_num();
```

```
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n", tid);

  /* Each thread works on its own private copy of the array */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i][j] = tid + i + j;

  /* For confirmation */
  printf("Thread %d done. Last element= %f\n",tid,a[N-1][N-1]);
  }  /* All threads join master thread and disband */
}
```

If you run the program, you can see that it causes segmentation fault. OpenMP thread stack size is an implementation dependent resource. In this case, the array is too large to fit into the thread stack space and causes the segmentation fault. You have to modify the environment variable, for Linux, KMP_STACKSIZE with the value 20000000.

```
******************************************************************************
* FILE: omp_bug5.c
* DESCRIPTION:
*   Using SECTIONS, two threads initialize their own array and then add
*   it to the other's array, however a deadlock occurs.
* AUTHOR: Blaise Barney  01/29/04
* LAST REVISED: 04/06/05
******************************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
#define PI 3.1415926535
#define DELTA .01415926535

int main (int argc, char *argv[])
{
int nthreads, tid, i;
float a[N], b[N];
omp_lock_t locka, lockb;

/* Initialize the locks */
omp_init_lock(&locka);
omp_init_lock(&lockb);
```

```
/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel shared(a, b, nthreads, locka, lockb) private(tid)
  {

  /* Obtain thread number and number of threads */
  tid = omp_get_thread_num();
  #pragma omp master
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n", tid);
  #pragma omp barrier

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      printf("Thread %d initializing a[]\n",tid);
      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        a[i] = i * DELTA;
      omp_set_lock(&lockb);
      printf("Thread %d adding a[] to b[]\n",tid);
      for (i=0; i<N; i++)
        b[i] += a[i];
      omp_unset_lock(&lockb);
      omp_unset_lock(&locka);
      }

    #pragma omp section
      {
      printf("Thread %d initializing b[]\n",tid);
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        b[i] = i * PI;
      omp_set_lock(&locka);
      printf("Thread %d adding b[] to a[]\n",tid);
      for (i=0; i<N; i++)
        a[i] += b[i];
      omp_unset_lock(&locka);
      omp_unset_lock(&lockb);
      }
    }  /* end of sections */
  }  /* end of parallel region */
}
```

EXPLANATION:

The problem in omp_bug5 is that the first thread acquires locka and then tries to get lockb before releasing locka. Meanwhile, the second thread has acquired lockb and then tries to get locka before releasing lockb. The solution overcomes the deadlock by using locks correctly.

```
/******************************************************************************
* FILE: omp_bug5fix.c
* AUTHOR: Blaise Barney  01/29/04
* LAST REVISED: 04/06/05
******************************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 1000000
#define PI 3.1415926535
#define DELTA .01415926535

int main (int argc, char *argv[])
{
int nthreads, tid, i;
float a[N], b[N];
omp_lock_t locka, lockb;

/* Initialize the locks */
omp_init_lock(&locka);
omp_init_lock(&lockb);

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel shared(a, b, nthreads, locka, lockb) private(tid)
  {

  /* Obtain thread number and number of threads */
  tid = omp_get_thread_num();
  #pragma omp master
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n", tid);
  #pragma omp barrier

  #pragma omp sections nowait
    {
    #pragma omp section
      {
      printf("Thread %d initializing a[]\n",tid);
      omp_set_lock(&locka);
```

```
      for (i=0; i<N; i++)
        a[i] = i * DELTA;
      omp_unset_lock(&locka);
      omp_set_lock(&lockb);
      printf("Thread %d adding a[] to b[]\n",tid);
      for (i=0; i<N; i++)
        b[i] += a[i];
      omp_unset_lock(&lockb);
      }

    #pragma omp section
      {
      printf("Thread %d initializing b[]\n",tid);
      omp_set_lock(&lockb);
      for (i=0; i<N; i++)
        b[i] = i * PI;
      omp_unset_lock(&lockb);
      omp_set_lock(&locka);
      printf("Thread %d adding b[] to a[]\n",tid);
      for (i=0; i<N; i++)
        a[i] += b[i];
      omp_unset_lock(&locka);
      }
    }  /* end of sections */
  }  /* end of parallel region */
}
```

6.3.9 OpenMP Debugging - FORTRAN

```
C******************************************************************************
C FILE: omp_bug1.f
C DESCRIPTION:
C   This example attempts to show use of the PARALLEL DO construct.  However
C   it will generate errors at compile time.  Try to determine what is causing
C   the error.  See omp_bug1fix.f for a corrected version.
C AUTHOR: Blaise Barney  5/99
C LAST REVISED:
C******************************************************************************
      PROGRAM WORKSHARE3

      INTEGER TID, OMP_GET_THREAD_NUM, N, I, CHUNKSIZE, CHUNK
      PARAMETER (N=50)
      PARAMETER (CHUNKSIZE=5)
      REAL A(N), B(N), C(N)

!     Some initializations
      DO I = 1, N
```

```
        A(I) = I * 1.0
        B(I) = A(I)
      ENDDO
      CHUNK = CHUNKSIZE

!$OMP  PARALLEL DO SHARED(A,B,C,CHUNK)
!$OMP& PRIVATE(I,TID)
!$OMP& SCHEDULE(STATIC,CHUNK)

      TID = OMP_GET_THREAD_NUM()
      DO I = 1, N
         C(I) = A(I) + B(I)
         PRINT *,'TID= ',TID,'I= ',I,'C(I)= ',C(I)
      ENDDO

!$OMP  END PARALLEL DO
      END
```

EXPLANATION:

This example illustrates the use of the combined PARALLEL for-DO directive. It fails because the loop does not come immediately after the directive. Corrections include removing all statements between the PARALLEL for-DO directive and the actual loop. Also, logic is added to preserve the ability to query the thread id and print it from inside the loop. Notice the use of the FIRSTPRIVATE clause to intialise the flag.

```
C******************************************************************************
C FILE: omp_bug1fix.f
C DESCRIPTION:
C   This is a corrected version of the omp_bug1fix.f example. Corrections
C   include removing all statements between the PARALLEL DO construct and
C   the actual DO loop, and introducing logic to preserve the ability to
C   query a thread's id and print it from inside the DO loop.
C AUTHOR: Blaise Barney  5/99
C LAST REVISED:
C******************************************************************************

      PROGRAM WORKSHARE4

      INTEGER TID, OMP_GET_THREAD_NUM, N, I, CHUNKSIZE, CHUNK
      PARAMETER (N=50)
      PARAMETER (CHUNKSIZE=5)
      REAL A(N), B(N), C(N)
      CHARACTER FIRST_TIME

!     Some initializations
      DO I = 1, N
        A(I) = I * 1.0
```

```
        B(I) = A(I)
      ENDDO
      CHUNK = CHUNKSIZE
      FIRST_TIME = 'Y'

!$OMP  PARALLEL DO SHARED(A,B,C,CHUNK)
!$OMP& PRIVATE(I,TID)
!$OMP& SCHEDULE(STATIC,CHUNK)
!$OMP& FIRSTPRIVATE(FIRST_TIME)

      DO I = 1, N
         IF (FIRST_TIME .EQ. 'Y') THEN
            TID = OMP_GET_THREAD_NUM()
            FIRST_TIME = 'N'
         ENDIF
         C(I) = A(I) + B(I)
         PRINT *,'TID= ',TID,'I= ',I,'C(I)= ',C(I)
      ENDDO
!$OMP  END PARALLEL DO

      END
C******************************************************************************
C FILE: omp_bug5.f
C DESCRIPTION:
C   Using SECTIONS, two threads initialize their own array and then add
C   it to the other's array, however a deadlock occurs.
C AUTHOR: Blaise Barney  01/09/04
C LAST REVISED:
C******************************************************************************

PROGRAM BUG5

      INTEGER*8 LOCKA, LOCKB
      INTEGER NTHREADS, TID, I,
     +        OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
      PARAMETER (N=1000000)
      REAL A(N), B(N), PI, DELTA
      PARAMETER (PI=3.1415926535)
      PARAMETER (DELTA=.01415926535)

C     Initialize the locks
      CALL OMP_INIT_LOCK(LOCKA)
      CALL OMP_INIT_LOCK(LOCKB)

C     Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL SHARED(A, B, NTHREADS, LOCKA, LOCKB) PRIVATE(TID)
```

```
C     Obtain thread number and number of threads
      TID = OMP_GET_THREAD_NUM()
!$OMP MASTER
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Number of threads = ', NTHREADS
!$OMP END MASTER
      PRINT *, 'Thread', TID, 'starting...'
!$OMP BARRIER

!$OMP SECTIONS

!$OMP SECTION
      PRINT *, 'Thread',TID,' initializing A()'
      CALL OMP_SET_LOCK(LOCKA)
      DO I = 1, N
         A(I) = I * DELTA
      ENDDO
      CALL OMP_SET_LOCK(LOCKB)
      PRINT *, 'Thread',TID,' adding A() to B()'
      DO I = 1, N
         B(I) = B(I) + A(I)
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKB)
      CALL OMP_UNSET_LOCK(LOCKA)

!$OMP SECTION
      PRINT *, 'Thread',TID,' initializing B()'
      CALL OMP_SET_LOCK(LOCKB)
      DO I = 1, N
         B(I) = I * PI
      ENDDO
      CALL OMP_SET_LOCK(LOCKA)
      PRINT *, 'Thread',TID,' adding B() to A()'
      DO I = 1, N
         A(I) = A(I) + B(I)
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKA)
      CALL OMP_UNSET_LOCK(LOCKB)

!$OMP END SECTIONS NOWAIT

      PRINT *, 'Thread',TID,' done.'

!$OMP END PARALLEL
```

```
      END

C******************************************************************************
C FILE: omp_bug5fix.f
C DESCRIPTION:
C   The problem in omp_bug5.f is that the first thread acquires locka and then
C   tries to get lockb before releasing locka. Meanwhile, the second thread
C   has acquired lockb and then tries to get locka before releasing lockb.
C   This solution overcomes the deadlock by using locks correctly.
C AUTHOR: Blaise Barney  01/09/04
C LAST REVISED:
C******************************************************************************

      PROGRAM BUG5

      INTEGER*8 LOCKA, LOCKB
      INTEGER NTHREADS, TID, I,
     +        OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
      PARAMETER (N=1000000)
      REAL A(N), B(N), PI, DELTA
      PARAMETER (PI=3.1415926535)
      PARAMETER (DELTA=.01415926535)

C     Initialize the locks
      CALL OMP_INIT_LOCK(LOCKA)
      CALL OMP_INIT_LOCK(LOCKB)

C     Fork a team of threads giving them their own copies of variables
!$OMP PARALLEL SHARED(A, B, NTHREADS, LOCKA, LOCKB) PRIVATE(TID)

C     Obtain thread number and number of threads
      TID = OMP_GET_THREAD_NUM()
!$OMP MASTER
      NTHREADS = OMP_GET_NUM_THREADS()
      PRINT *, 'Number of threads = ', NTHREADS
!$OMP END MASTER
      PRINT *, 'Thread', TID, 'starting...'
!$OMP BARRIER

!$OMP SECTIONS

!$OMP SECTION
      PRINT *, 'Thread',TID,' initializing A()'
      CALL OMP_SET_LOCK(LOCKA)
      DO I = 1, N
         A(I) = I * DELTA
```

```
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKA)
      CALL OMP_SET_LOCK(LOCKB)
      PRINT *, 'Thread',TID,' adding A() to B()'
      DO I = 1, N
         B(I) = B(I) + A(I)
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKB)

!$OMP SECTION
      PRINT *, 'Thread',TID,' initializing B()'
      CALL OMP_SET_LOCK(LOCKB)
      DO I = 1, N
         B(I) = I * PI
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKB)
      CALL OMP_SET_LOCK(LOCKA)
      PRINT *, 'Thread',TID,' adding B() to A()'
      DO I = 1, N
         A(I) = A(I) + B(I)
      ENDDO
      CALL OMP_UNSET_LOCK(LOCKA)

!$OMP END SECTIONS NOWAIT

      PRINT *, 'Thread',TID,' done.'

!$OMP END PARALLEL

      END
```

6.4 Debuggers

6.4.1 Sun Studio Integrated Debugger

Sun Studio includes a debugger for serial and multithreaded programs. You will find more information on how to use this environment online. You may start debugging your program from *Run - Debugg executabile*.

There is a series of actions that are available from the *Run* menu but which may be specified from the command line when starting Sun Studio. In order to just start a debuggind session, you can attach to a running program by:

```
$ sunstudio -A pid[:program_name]
```

or from *Run - Attach Debugger*. To analyse a core dump, use:

```
$ sunstudio -C core[:program_name]
```

or *Run - Debug core file*

6.4.2 TotalView

TotalView is a sophisticated software debugger product of TotalView Technologies that has been selected as the Department of Energy's Advanced Simulation and Computing (ASC) program's debugger. It is used for debugging and analyzing both serial and parallel programs and it is especially designed for use with complex, multi-process and/or multi-threaded applications. It is designed to handle most types of HPC parallel coding and it is supported on most HPC platforms. It provides both a GUI and command line interface and it can be used to debug programs, running processes, and core files, including also memory debugging features. It provides graphical visualization of array data and icludes a comprehensive built-in help system.

Supported Platforms and Languages

Supported languages include the usual HPC application languages:
o C/C++
o Fortran77/90
o Mixed C/C++ and Fortran
o Assembler

Compiling Your Program

-g:
Like many UNIX debuggers, you will need to compile your program with the appropriate flag to enable generation of symbolic debug information. For most compilers, the *-g* option is used for this.
TotalView will allow you to debug executables which were not compiled with the *-g* option. However, only the assembler code can be viewed.

Beyond -g:
Don't compile your program with optimization flags while you are debugging it. Compiler optimizations can "rewrite" your program and produce machine code that doesn't necessarily match your source code.
Parallel programs may require additional compiler flags.

Overview

TotalView is a full-featured, source-level, graphical debugger for C, C++, and Fortran (77 and 90), assembler, and mixed source/assembler codes based on the X Window System from Etnus. TotalView supports MPI, PVM and HPF.
Information on TotalView is available in the release notes and user guide at the Etnus Online Documentation page. Also see "man totalview" for command syntax and options.
Note: In order to use TotalView, you must be using a terminal or workstation capable of displaying X Windows. See Using the X Window System for more information.

TotalView on Linux Clusters

TotalView is available on NCSA's Linux Clusters. On Abe there is a 384 token TotalView license and you only checkout the number of licenses you need . We do not currently have a way to guarantee

you will get a license when your job starts if you run in batch.

GNU and Intel compilers are supported.

Important: For both compilers you need to compile and link your code with *-g* to enable source code listing within TotalView.

TotalView is also supported on Abe.

Starting TotalView on the cluster:

To use TotalView over a cluster there are a few steps to follow. It is very important to have the $HOME directory shared over the network between nodes.

11 steps that show you how to run TotalView to debug your program:

1. Download an NX client on your local station (more details about how you need to configure it you can find at the Environment chapter, File Management subsection from the present Cluster-guide);

2. Connect with the NX client to the cluster;

3. Write or upload your document in the home folder;

4. Compile it using the proper compiler and add *-g* at the compiling symbols. It produces debugging information in the system's native format.

```
gcc -fopenmp -O3 -g -o app_lab4_gcc openmp_stack_quicksort.c
```

5. Find the value for $DISPLAY using the command:

```
$ echo $DISPLAY
```

You also need to find out the port the X11 connection is forwarded on. For example, if DISPLAY is host:14.0 the connection port will be 6014.

```
setenv DISPLAY fep.grid.pub.ro:14.0
```

6. Type the command xhost +, in order to disable the access control

```
$ xhost +
```

Now access control is disabled and clients can connect from any host (X11 connections are allowed).

7. Currently, the only available queue is ibm-quad.q and the version of TotalView is totalview-8.6.2-2. To find out what queues are available, use the command:

```
qconf -sql
```

8. Considering the ibm-quad.q queue available, write after the following command:

```
qsub -q ibm-quad.q -cwd
```

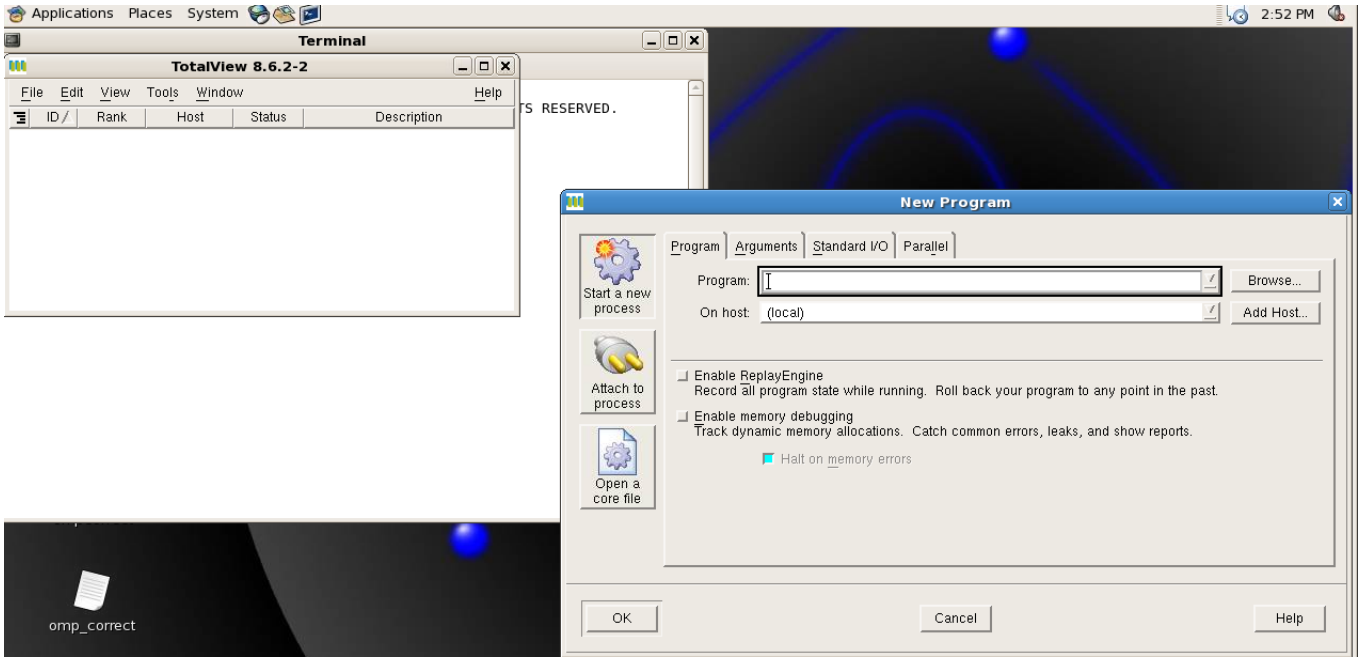
9. After running the command above you have to write the following lines (:1000.0 can be replaced by the value that you obtained after typing the command echo $DISPLAY):

```
module load debuggers/totalview-8.4.1-7
setenv DISPLAY fep-53-2.grid.pub.ro:1000.0
totalview
```

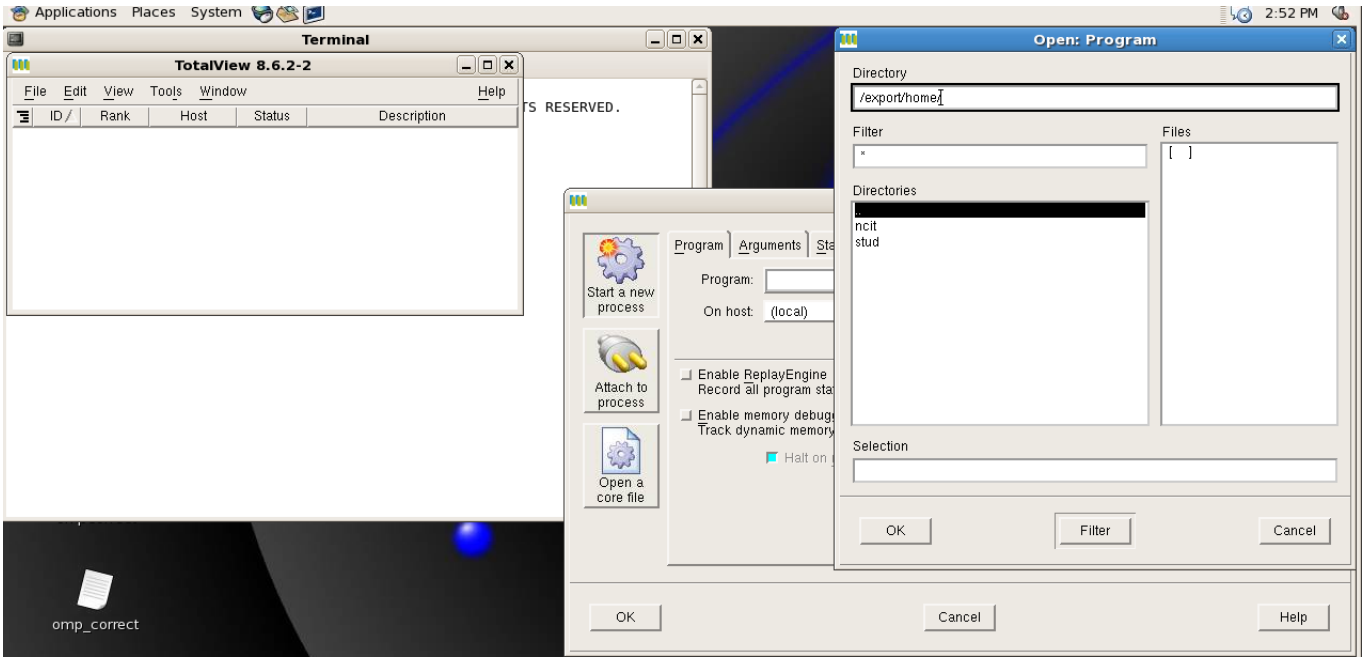
10. After that, press Ctrl+D in order to submit your request to the queue.

When the xterm window will appear, we will launch TotalView. If the window does not appear check the job output. This should give you some clue and why your script failed; maybe you misspelled a command or maybe the port for the X11 forwarding is closed from the firewall. You should check these two things first.

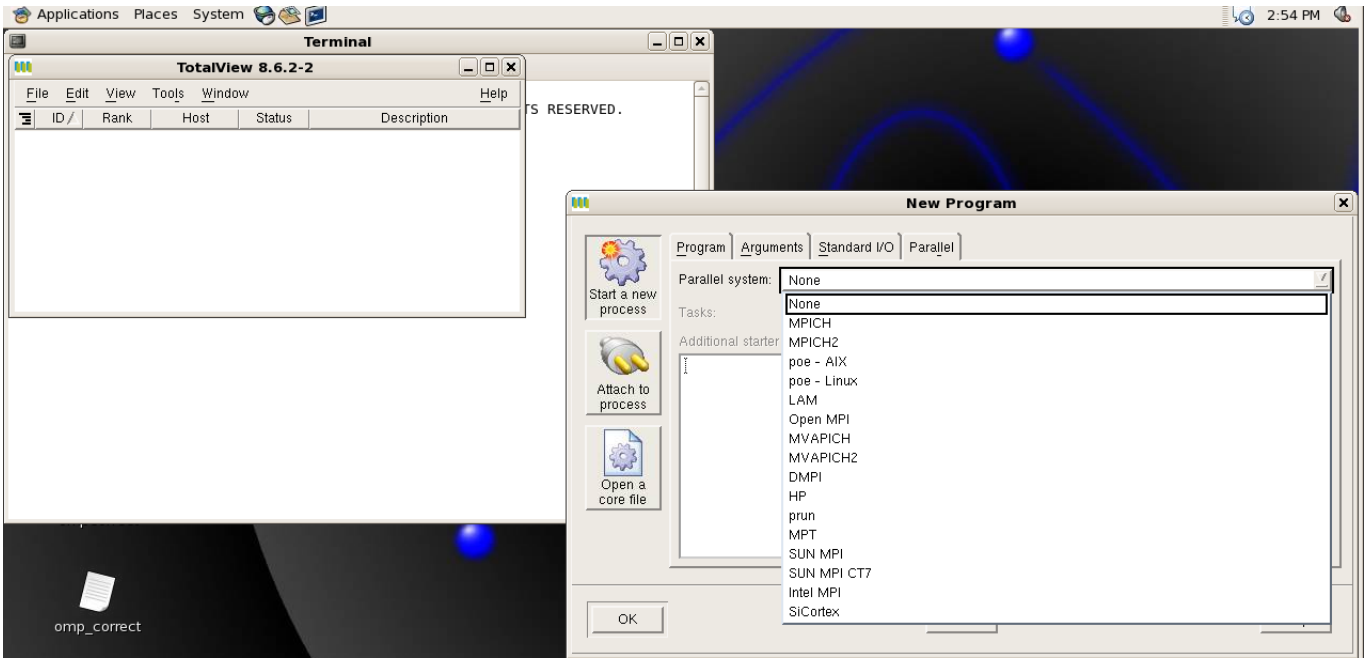
11. Open /opt/totalview/toolworks/totalview.8.6.2-2/bin and run the totalview



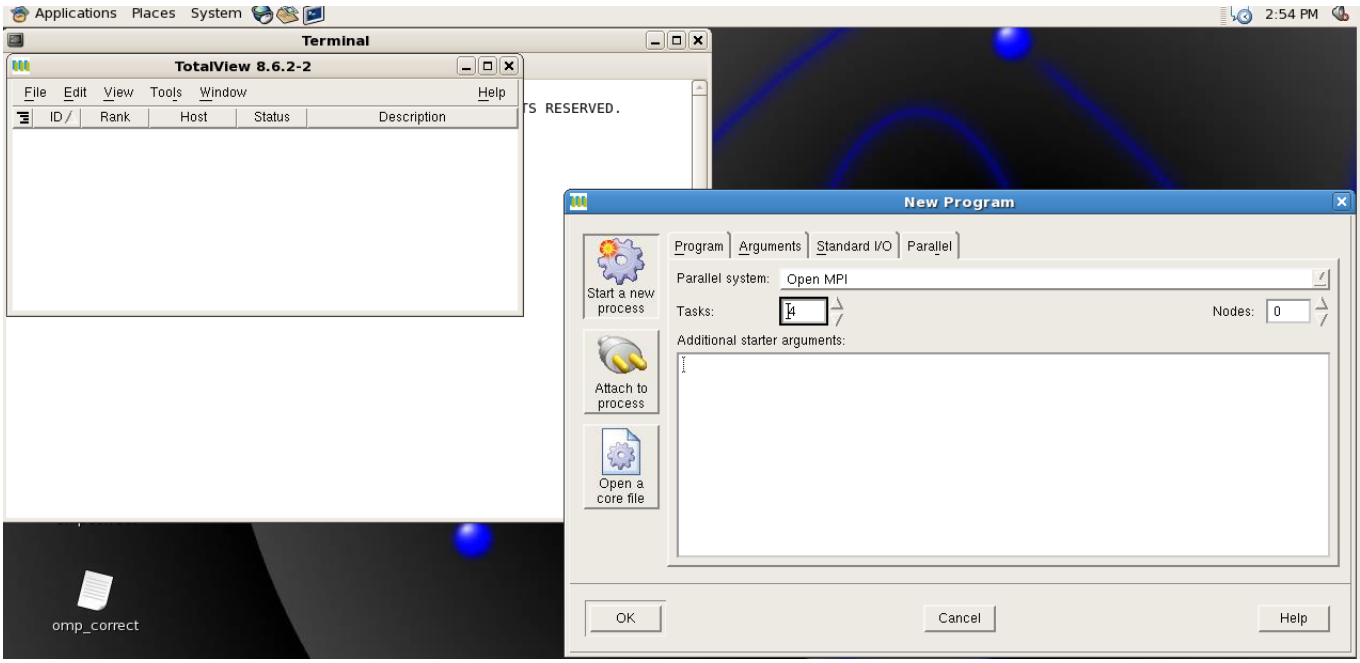
Now you should be able to see the graphical interface of TotalView. Here are some pictures to help you interact with it. Next you have to select the executable file corresponding to your program, which has been previously compiled using the "-g" option, from the appropriate folder placed on the cluster(the complete path to your home folder).



Next you need to select the parallel environment you want to use.



The next step is to set the number of tasks you want to run:

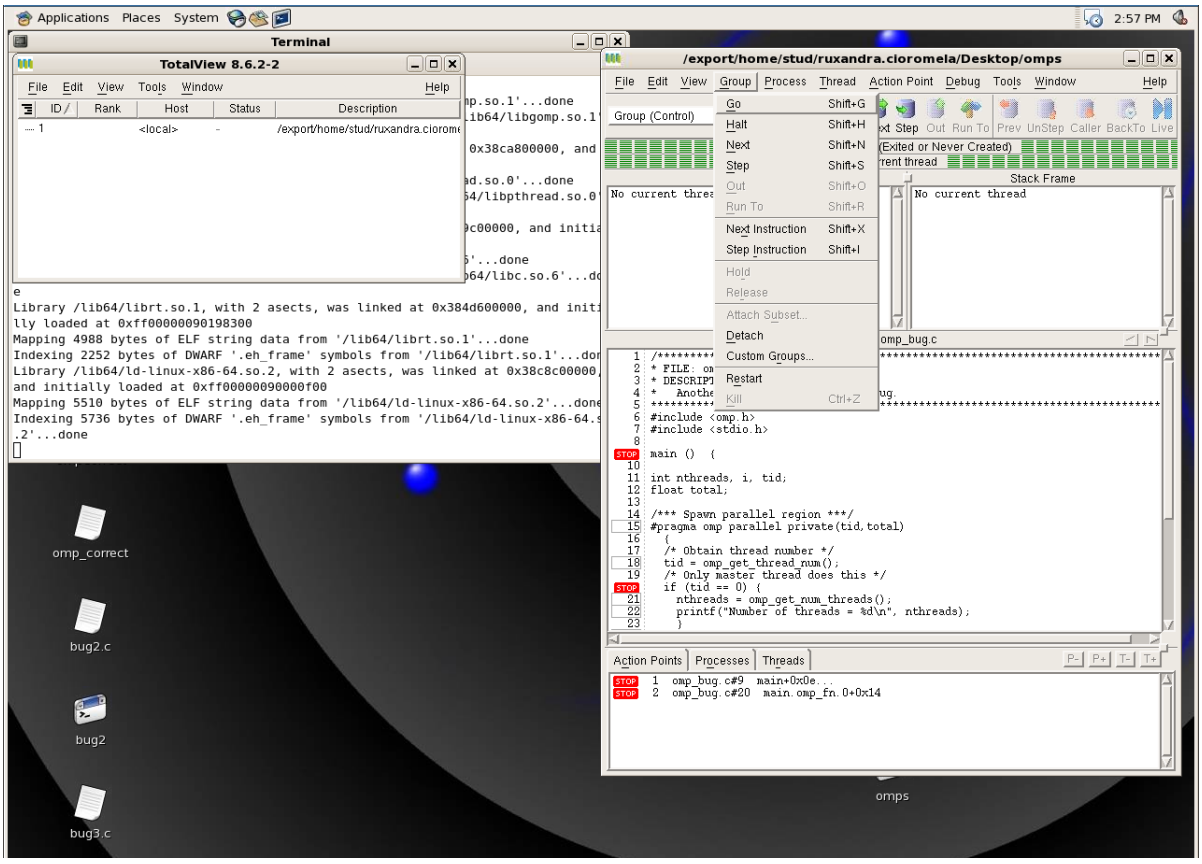


Here is an example of how to debug your source. Try to use the facilities and options offered by TotalView, combined with the examples shown in the tutorials below.

Some helpful links, to help you with Totalview:

Total View Tutorial

Total View Excercise



TotalView Command Line Interpreter

The TotalView Command Line Interpreter (CLI) provides a command line debugger interface. It can be launched either stand-alone or via the TotalView GUI debugger.

The CLI consists of two primary components:
o The CLI commands
o A Tcl interpreter

Because the CLI includes a Tcl interpreter, CLI commands can be integrated into user-written Tcl programs/scripts for "automated" debugging. Of course, putting the CLI to real use in this manner will require some expertise in Tcl.

Most often, the TotalView GUI is the method of choice for debugging. However, the CLI may be the method of choice in those circumstances where using the GUI is impractical:
o When a program takes several days to execute.
o When the program must be run under a batch scheduling system or network conditions that inhibit GUI interaction.
o When network traffic between the executing program and the person debugging is not permitted or limits the use of the GUI.

See the TotalView documentation located at Total View Official Site for details:
o TotalView User Guide - relevant chapters
o TotalView Reference Guide - complete coverage of all CLI commands, variables and usage.

Starting an Interactive CLI Debug Session:
Method 1: From within the TotalView GUI:

1. Use either path:

```
Process Window  >  Tools Menu  > Command Line
Root Window  >  Tools Menu  > Command Line
```

2. A TotalView CLI xterm window (below) will then open for you to enter CLI commands.
3. Load/Start your executable or attach to a running process
4. Issue CLI commands

Method 2: From a shell prompt window:

1. Invoke the totalviewcli command (provided that it is in your path).
2. Load/Start your executable or attach to a running process
3. Issue CLI commands

CLI Commands:

As of TotalView version 8, there are approximately 75 CLI commands. These are covered completely in the TotalView Reference Guide.

Some representative CLI commands are shown below.

Environment Commands	
alias	creates or views user-defined commands
capture	allows commands that print information to send their output to a string variable
dgroups	manipulates and manages groups
dset	changes or views values of CLI state variables
dunset	restores default settings of CLI state variables
help	displays help information
stty	sets terminal properties
unalias	removes a previously defined command
dworker	adds or removes a thread from a workers group
CLI	initialization and termination
dattach	attaches to one or more processes currently executing in the normal run-time environment
ddetach	detaches from processes
dkill	kills existing user process, leaving debugging information in place
dload	loads debugging information about the target program and prepares it for execution
dreload	reloads the current executable
drerun	restarts a process
drun	starts or restarts the execution of users processes under control of the CLI
dstatus	shows current status of processes and threads
quit	exits from the CLI, ending the debugging session
Program Information	
dassign	changes the value of a scalar variable
dlist	browses source code relative to a particular file, procedure or line
dmstat	displays memory use information
dprint	evaluates an expression or program variable and displays the resulting value
dptsets	shows status of processes and threads
dwhat	determines what a name refers to
dwhere	prints information about the target thread's stack
Execution Control	
dcont	continues execution of processes and waits for them
dfocus	changes the set of process, threads, or groups upon which a CLI command acts
dgo	resumes execution of processes (without blocking)
dhalt	suspends execution of processes
dhold	holds threads or processes
dnext	executes statements, moving into subfunctions if required
dnexti	executes machine instructions, stepping over subfunctions
dout	runs out from the current subroutine
dstep	executes statements, moving into subfunctions if required
dstepi	executes machine instructions, moving into subfunctions if required
dunhold	releases a held process or thread
duntil	runs the process until a target place is reached
dwait	blocks command input until processes stop

Action Points	
dactions	views information on action point definitions and their current status
dbarrier	defines a process or thread barrier breakpoint
dbreak	defines a breakpoint
ddelete	deletes an action point
ddisable	temporarily disables an action point
denable	reenables an action point that has been disabled
dwatch	defines a watchpoint
Miscellaneous	
dcache	clears the remote library cache
ddown	moves down the call stack
dflush	unwinds stack from suspended computations
dlappend	appends list elements to a TotalView variable
dup	moves up the call stack

7 Parallelization

Parallelization for computers with shared memory (SM) means the automatic distribution of loop iterations over several processors(autoparallelization), the explicit distribution of work over the processors by compiler directives (OpenMP) or function calls to threading libraries, or a combination of those.

Parallelization for computers with distributed memory (DM) is done via the explicit dis- tribution of work and data over the processors and their coordination with the exchange of messages (Message Passing with MPI).

MPI programs run on shared memory computers as well, whereas OpenMP programs usu- ally do not run on computers with distributed memory.

There are solutions that try to achieve the programming ease of shared memory paralleliza- tion on distributed memory clusters. For example Intels Cluster OpenMP oers a relatively easy way to get OpenMP programs running on a cluster.

For large applications the hybrid parallelization approach, a combination of coarse-grained parallelism with MPI and underlying ne-grained parallelism with OpenMP, might be attrac- tive, in order to use as many processors eciently as possible.

Please note, that large computing jobs should not be started interactively, and that when submitting use of batch jobs (see chapter 4.6 on page 25), the GridEngine batch system determines the distribution of the MPI tasks on the machines to a large extent.

7.1 Shared Memory Programming

For shared memory programming, OpenMP is the de facto standard (http://www.openmp.org). The OpenMP API is dened for Fortran, C and C++ and consists of compiler directives (resp. pragmas), runtime routines and environment variables.

In the parallel regions of a program several threads are started. They execute the contained program segment redundantly, until they hit a worksharing construct . Within this construct, the contained work (usually do- or for-loops) is distributed among the threads. Under normal conditions all threads have access to all data (shared data). But pay attention: if data, accessed by several threads, is modied, then the access to this data must be protected with critical regions or OpenMP locks .

Also private data areas can be used, where the individual threads hold their local data. Such private data (in OpenMP terminology) is only visible to the thread owning it. Other threads will not be able to read or write private data.

Note: In many cases, the stack area for the slave threads must be increased by changing a compiler specic environment variable (e.g. Sun Studio: stacksize, Intel:kmp_stacksize), and the stack area for the master thread must be increased with the command ulimit -s xxx (zsh shell, specication in KB) or limit s xxx (C-shell, in KB).

Hint: In a loop, which is to be parallelized, the results must not depend on the order of the loop iterations! Try to run the loop backwards in serial mode. The results should be the same. This is a necessary, but not a sucient condition!

The number of the threads has to be specied by the environment variable omp_num_threads.

Note: The OpenMP standard does not specify the value for omp_num_threads in case it is not set explicitly. If omp_num_threads is not set, then Sun OpenMP for example starts only 1 thread, as opposed to the Intel compiler which starts as many threads as there are processors available.

On a loaded system fewer threads may be employed than specied by this environment variable, because the dynamic mode may be used by default. Use the environment variable omp_dynamic to change this behavior. If you want to use nested OpenMP, the environment variable omp_nested=true has to be set.

The OpenMP compiler options have been sumarized in the following. These compiler flags will be set in the environment variables FLAGS_AUTOPAR and FLAGS_OPENMP (as explained in section 6.1.1)

Compiler	**flags_openmp**	**flags_autopar**
Sun	-xopenmp	-xautopar -xreduction
Intel	-openmp	-parallel
GNU	-fopenmp (4.2 and above)	n.a. (planned for 4.3)
PGI	-mp	-Mconcur -Minline

An example program using OpenMP is /export/home/stud/alascateu/PRIMER/PROFILE/openmp_only.c

7.1.1 Automatic Shared Memory Parallelization of Loops

The Sun Fortran, C and C++-compilers are able to parallelize loops automatically. Success or failure depends on the compilers ability to prove it is safe to parallelize a (nested) loop. This is often application area specic (e.g. nite dierences versus nite elements), language (pointers and function calls may make the analysis dicult) and coding style dependent. The appropriate option is -xautopar which includes -depend -xO3. Although the -xparallel option is also available, we do not recommend to use this. The option combines auto- matic and explicit parallelization, but assumes the older Sun parallel programming model is used instead of OpenMP. In case one would like to combine automatic parallelization and OpenMP, we strongly suggest to use the -xautopar -xopenmp combination. With the option -xreduction, automatic parallelization of reductions is also permitted, e.g. accumulations, dot products etc., whereby the modication of the sequence of the arithmetic operation can cause dierent rounding error accumulations.

Compiling with the option -xloopinfo supplies information about the parallelization. The compiler messages are shown on the screen.

If the number of loop iterations is unknown during compile time, then code is produced, which decides at run-time whether a parallel execution of the loop is more ecient or not (alternate coding).

Also with automatic parallelization, the number of the used threads can be specied by the environment variable omp_num_threads.

7.1.2 GNU Compilers

With version 4.2 the GNU compiler collection supports OpenMP with the option **-fopenmp**. It supports nesting using the standard OpenMP environment variables. In addition to these variables you can set the default thread stack size in kilobytes with the variable **GOMP_STACKSIZE**.

CPU binding can be done with the **GOMP_CPU_AFFINITY** environment variable. The variable should contain a space or comma-separated list of CPU's. This list may contain different kind of entries: either single CPU numbers in any order, a range of CPU's (M-N) or a range with some stride (M-N:S). CPU numbers are zero based. For example, **GOMP_CPU_AFFINITY="0 3 1-2 4-15:2"** will bind the initial thread to CPU 0, the second to CPU 3, the third to CPU 1, the fourth to CPU 2, the fifth to CPU 4, the sixth through tneth to CPU's 6, 8, 10, 12 and 14 respectively

and then start assigning back from the beggining of the list. **CGOMP_CPU_AFFINITY=0** binds all threads to CPU 0.

Automatic Shared Memory Parallelization of Loops Since version 4.3 the GNU compilers are able to parallelize loops automatically with the option **-ftree-parallelize-loops=[threads]**. However the number of threads to use have to be specified at compile time and thus are fix at runtime.

7.1.3 Intel Compilers

By adding the option -openmp the OpenMP directives are interpreted by the Intel compilers. Nested OpenMP is supported too.

The slave threads stack size can be increased with the environment variable kmp_stacksize=megabytes M.

Attention: By default the number of threads is set to the number of processors. It is not recommended to set this variable to larger values than the number of processors available on the current machine. By default, the environment variables omp_dynamic and omp_nested are set to false.

Intel compilers provide an easy way for processor binding. Just set the environment variable kmp_affinity to compact or scatter, e.g.

```
$ export KMP_AFFINITY=scatter
```

Compact binds the threads as near as possible, e.g. two threads on dierent cores of one processor chip. Scatter binds the threads as far away as possible, e.g. two threads, each on one core on dierent processor sockets.

Automatic Shared Memory Parallelization of Loops The Intel Fortran, C and C++ compilers are able to parallelize certain loops automatically. This feature can be turned on with the option **-parallel**. The number of the used threads is specied by the environment variable OMP_NUM_THREADS.

Note: using the option -O2 enables automatic inlining which may help the automatic parallelization, if functions are called within a loop.

7.1.4 PGI Compilers

By adding the option -mp the OpenMP directives, according to the OpenMP version 1 specications, are interpreted by the PGI compilers.

Explicit parallelization can be combined with the automatic parallelization of the compiler. Loops within parallel OpenMP regions are no longer subject to automatic parallelization. Nested parallelization is not supported. The slave threads stack size can be increased with the environment variable mpstkz=megabytes M

By default omp_num_threads is set to 1. It is not recommended to set this variable to a larger value than the number of processors available on the current machine. The environment variables omp_dynamic and omp_nested have no eect!

The PGI compiler oers some support for NUMA architectures like the V40z Opteron systems with the option -mp=numa. Using NUMA can improve performance of some parallel applications by reducing memory latency. Linking -mp=numa also allows you to use the environment variables

mp_bind, mp_blist and mp_spin. When mp_bind is set to yes , parallel processes or threads are bound to a physical processor. This ensures that the operating system will not move your process to a dierent CPU while it is running. Using mp_blist, you can specify exactly which processors to attach your process to. For example, if you have a quad socket dual core system (8 CPUs), you can set the blist so that the processes are interleaved across the 4 sockets (MP_BLIST=2,4,6,0,1,3,5,7) or bound to a particular (MP_BLIST=6,7).

Threads at a barrier in a parallel region check a semaphore to determine if they can proceed. If the semaphore is not free after a certain number of tries, the thread gives up the processor for a while before checking again. The mp_spin variable denes the number of times a thread checks a semaphore before idling. Setting mp_spin to -1 tells the thread never to idle. This can improve performance but can waste CPU cycles that could be used by a dierent process if the thread spends a signicant amount of time in a barrier.

Automatic Shared Memory Parallelization of Loops The PGI Fortran, C and C++ compilers are able to parallelize certain loops automatically. This feature can be turned on with the option -Mconcur. The number of the used threads is also specied by the environment variable OMP_NUM_THREADS.

Note: Using the option -Minline the compiler tries to inline functions. So even loops with function calls may be parallelized.

7.2 Message Passing with MPI

MPI (Message-Passing Interface) is the de-facto standard for parallelization on distributed memory parallel systems. Multiple processes explicitly exchange data and coordinate their work flow. MPI specifies the interface but not the implementaion. Therefore, there are plenty of implementations for PC's as well as for supercomputers. There are freely available implementations and commercial ones, which are particularly tuned for the target platform. MPI has a huge number of calls, although it is possible to write meaningful MPI applications just employing some 10 of these calls.

An example program using MPI is /export/home/stud/alascateu/PRIMER/PROFILE/mpi.c.

7.2.1 OpenMPI

The Open MPI Project (www.openmpi.org) is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available. Open MPI offers advantages for system and software vendors, application developers and computer science researchers.

The compiler drivers are **mpicc** for C, **mpif77** and **mpif90** for FORTRAN, **mpicxx** and **mpiCC** for C++.

mpirun is used to start a MPI program. Refere to the manual page for a detailed description of mpirun (`$ man mpirun`).

We have several Open MPI implementations. To use the one suitable for your programs, you must load the appropriate module (remember to also load the corresponding compiler module). For example, if you want to use the **PGI** implementation you should type the following:

```
$ module list
```

```
Currently Loaded Modulefiles:
  1) batch-system/sge-6.2u3            4) switcher/1.0.13
  2) compilers/sunstudio12.1           5) oscar-modules/1.0.5
  3) mpi/openmpi-1.3.2_sunstudio12.1
$ module avail

[...]
-------------------- /opt/modules/modulefiles --------------------
apps/hrm                          debuggers/totalview-8.6.2-2
apps/matlab                       grid/gLite-UI-3.1.31-Prod
apps/uso09                        java/jdk1.6.0_13-32bit
batch-system/sge-6.2u3            java/jdk1.6.0_13-64bit
cell/cell-sdk-3.1                 mpi/openmpi-1.3.2_gcc-4.1.2
compilers/gcc-4.1.2               mpi/openmpi-1.3.2_gcc-4.4.0
compilers/gcc-4.4.0               mpi/openmpi-1.3.2_intel-11.0_081
compilers/intel-11.0_081          mpi/openmpi-1.3.2_pgi-7.0.7
compilers/pgi-7.0.7               mpi/openmpi-1.3.2_sunstudio12.1
compilers/sunstudio12.1           oscar-modules/1.0.5(default)
```

Load the PGI implementation of MPI

```
$ module switch mpi/openmpi-1.3.2_pgi-7.0.7
```

Load the PGI compiler

```
$ module switch compilers/pgi-7.0.7
```

Now if you type **mpicc** you'll see that the wrapper calls **pgcc**.

7.2.2 Intel MPI Implementation

Intel-MPI is a commercial implementation based on mpich2 which is a public domain implementation of the MPI 2 standard provided by the Mathematics and Computer Science Division of the Argonne National Laboratory.

The compiler drivers **mpifc**, **mpiifort**, **mpiicc**, **mpiicpc**, **mpicc** and **mpicxx** and the instruction for starting an MPI application mpiexec will be included in the search path.

There are two dierent versions of compiler driver: **mpiifort**, **mpiicc** and **mpiicpc** are the compiler driver for Intel Compiler. **mpifc**, **mpicc** and **mpicxx** are the compiler driver for GCC (GNU Compiler Collection).

To use the Intel implementation you must load the apporpriate modules just like in the PGI exemple in the OpenMPI section.

Examples:

```
$ mpiifort -c ... *.f90
$ mpiicc -o a.out *.o
$ mpirun -np 4 a.out:
$ ifort -I$MPI_INCLUDE -c prog.f90
$ mpirun -np 4 a.out
```

7.3 Hybrid Parallelization

The combination of MPI and OpenMP and/or autoparallelization is called hybrid parallelization. Each MPI process may be multi-threaded. In order to use hybrid parallelization the MPI library has to support it. There are 4 stages of possible support:

1. single - multi-threading is not supported.
2. funneled - only the main thread, which initializes MPI, is allowed to make MPI calls.
3. serialized - only one thread may call the MPI library at a time.
4. multiple - multiple threads may call MPI, without restrictions.

You can use the MPI_Init_thread function to query multi-threading support of the MPI implementation.

A quick example of a hybrid program is /export/home/stud/alascateu/PRIMER/PROFILE/hybrid.c. It is a standard Laplace equation program, with MPI support, in witch a simple OpenMP matrix multiply program was inserted. Thus, every process distributed over the cluster will spawn multiple threads that will multiply some random matrices. The matrix dimensions where augmented so the program would run sufficient time to collect experiment data with the Sun Analyzer presented in the **Performance / Runtime Analysis Tools** section.

Tu run the program (C environment in the example) compile it as a MPI program but with OpenMP support:

```
$ mpicc -fopenmp hybrid.c -o hybrid
```

Run with (due to the Laplace layout you need 4 processors):

```
$ mpirun -np 4 hybrid
```

7.3.1 Hybrid Parallelization with Intel-MPI

Unfortunately Intel-MPI is not thread safe by default. Calls to the MPI library should not be made inside of parallel regions if the library is not linked to the program. To provide full MPI support inside parallel regions the program must be linked with the option -mt_mpi. Note: If you specify either the -openmp or the -parallel option of the Intel C Compiler, the thread safe version of the library is used. Note: If you specify one of the following options for the Intel Fortran Compiler, the thread safe version of the library is used:

1. -openmp
2. -parallel
3. -threads
4. -reentrancy
5. -reentrancy threaded

8 Performance / Runtime Analysis Tools

This chapter describes tools that are available to help you assess the performance of your code, identify potential performance problems, and locate the part of the code where most of the execution time is spent. It also covers the installation and run of an Intel MPI benchmark.

8.1 Sun Sampling Collector and Performance Analyzer

The Sun Sampling Collector and the Performance Analyzer are a pair of tools that you can use to collect and analyze performance data for your serial or parallel application. The Collector gathers performance data by sampling at regular time intervals and by tracing function calls. The performance information is gathered in so called experiment files, which can then be displayed with the analyzer GUI or the er_print line command after the program has nished. Since the collector is part of the Sun compiler suite the studio compiler module has to be loaded. However programs to be analyzed do not have to be compiled with the Sun compiler, the GNU or Intel compiler for example work as well.

8.1.1 Collecting experiment data

The first step in profiling with the Sun Analyzer is to obtain experiement data. For this you must compile your code with the **-g** option. After that you can either run **collect** like this

```
$ collect a.out
```

or use the *GUI*.

To use the GUI to collect experiment data, start the analyzer (**X11 forwarding must be enabled** - `$ analyzer`), go to *Collect Experiment* under the *File* menu and select the **Target**, **Working Directory** and add **Arguments** if you need to. Click on **Preview Command** to view the command for collecting experiment data only. Now you can submit the command to a queue.

Some example of scripts used to submit the command (the path to **collect** might be different):

```
$ cat script.sh
#!/bin/bash

qsub -q [queue] -pe [pe] [np] -cwd -b y  \
  "/opt/sun/sunstudio12.1/prod/bin/collect -p high -M CT8.1 -S on -A on -L none \
                   mpirun -np 4 -- /path/to/file/test"

$ cat script_OMP_ONLY.sh
#!/bin/bash

qsub -q [queue]  [pe] [np] -v OMP_NUM_THREADS=8 -cwd -b y  \
  "/opt/sun/sunstudio12.1/prod/bin/collect  -L none \
                -p high -S on -A on /path/to/file/test"

$ cat scriptOMP.sh
```

```
#!/bin/bash

qsub -q [queue] [pe] [np] -v OMP_NUM_THREADS=8 -cwd -b y  \
  "/opt/sun/sunstudio12.1/prod/bin/collect -p high -M CT8.1 -S on -A on -L none \
                   mpirun -np 4 -- /path/to/file/test"
```

The first one uses MPI tracing for testing MPI programs, the second one is intended for Openmp programs (that's why it sets the OMP_NUM_THREADS variable) and the last one is for hybird programs (they use both MPI and Openmp).

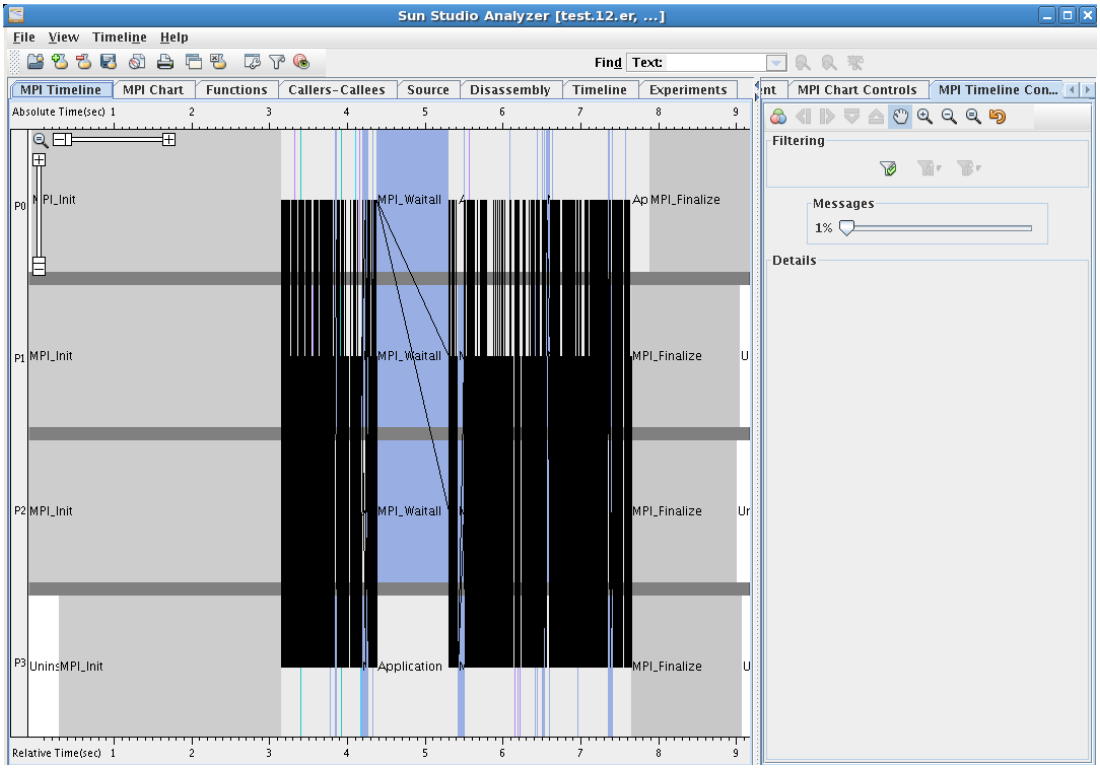
Some of the parameters used are explained in the following. You can find more information in the manual (`$ man collect`).

- -M CT8.1; Specify collection of an MPI experiment. CT8.1 is the MPI version installed.
- -L size; Limit the amount of profiling and tracing data recorded to size megabytes. None means no limit.
- -S interval; Collect periodic samples at the interval specified (in seconds). **on** defaults to 1 second.
- -A option; Control whether or not load objects used by the target process should be archived or copied into the recorded experiment. **on** archive load objects into the experiment.
- -p option; Collect clock-based profiling data. **high** turn on clock-based profiling with the default profiling interval of approximately 1 millisecond.

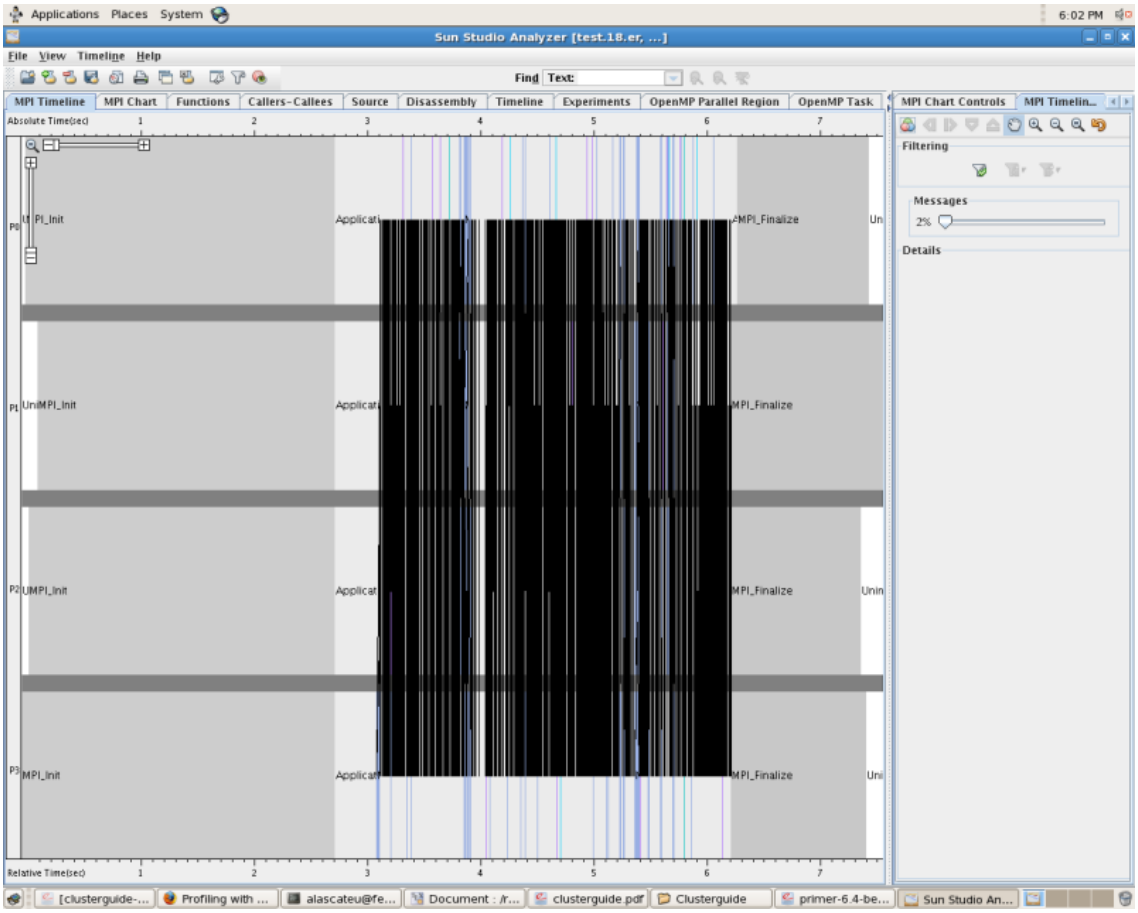
8.1.2 Viewing the experiment results

To view the results, open the analyzer and go to *File - Open Experiment* and select the experiment you want to view. A very good tutorial for analyzing the data can be found here. *Performance Analyzer MPI Tutorial* is a good place to start.

The following screenshots where taken from the analysis of the programs presented in the **Parallelization section** under **Hybrid Parallelization**.



MPI only version



Hybrid (MPI + Openmp) version

8.2 Intel MPI benchmark

The Intel MPI benchmark - **IMB** is a tool for evaluating the performance of a MPI installation. The idea of IMB is to provide a concise set of elementary MPI benchmark kernels. With one executable, all of the supported benchmarks, or a subset specified by the command line, can be run. The rules, such as time measurement (including a repetitive call of the kernels for better clock synchronization), message lengths, selection of communicators to run a particular benchmark (inside the group of all started processes) are program parameters.

8.2.1 Installing and running IMB

The first step is to get the package from here. Unpack the archive. Make sure you have the Intel compiler module loaded and a working OpenMPI installation. Go to the */imb/src* directory. There are three benchmarks available: IMB-MPI1, IMB-IO, IMB-EXT. You can build them separately with:

```
$ make <benchmark name>
```

or all at once with:

```
$ make all
```

Now you can run any of the three benchmarks using:

```
$ mpirun -np <nr_of_procs> IMB-xxx
```

NOTE: there are useful documents in the /imb/doc directory detailing the benchmarks.

8.2.2 Submitting a benchmark to a queue

You can also submit a benchmark to run on a queue. The following two scripts are examples:

```
$ cat submit.sh
#!/bin/bash
qsub -q [queue] -pe [pe] [np] -cwd -b n [script_with_the_run_cmd]

$ cat script.sh
#!/bin/bash
mpirun -np [np] IMB-xxx
```

To submit you just have to run:

```
$ ./submit.sh
```

After running the IMB-MPI1 benchmark on a queue with 24 processes the following result was obtained (only parts are shown):

```
#---------------------------------------------------
#    Intel (R) MPI Benchmark Suite V3.2, MPI-1 part
#---------------------------------------------------
# Date                  : Thu Jul 23 16:37:23 2009
# Machine               : x86_64
# System                : Linux
# Release               : 2.6.18-128.1.1.el5
```

```
# Version               : #1 SMP Tue Feb 10 11:36:29 EST 2009
# MPI Version           : 2.1
# MPI Thread Environment: MPI_THREAD_SINGLE

# Calling sequence was:

# IMB-MPI1

# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
#

# List of Benchmarks to run:

# PingPong
# PingPing
# Sendrecv
# Exchange
# Allreduce
# Reduce
# Reduce_scatter
# Allgather
# Allgatherv
# Gather
# Gatherv
# Scatter
# Scatterv
# Alltoall
# Alltoallv
# Bcast
# Barrier

[...]

#---------------------------------------------------------------------------
# Benchmarking Exchange
# #processes = 2
# ( 22 additional processes waiting in MPI_Barrier)
#---------------------------------------------------------------------------
      #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]   Mbytes/sec
      524288           80      1599.30      1599.31      1599.31      1250.54
```

```
     1048576           40      3743.45      3743.48      3743.46      1068.53
     2097152           20      7290.26      7290.30      7290.28      1097.35
     4194304           10     15406.39     15406.70     15406.55      1038.51

[...]

#----------------------------------------------------------------------------
# Benchmarking Exchange
# #processes = 24
#----------------------------------------------------------------------------
      #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]   Mbytes/sec
           0         1000        75.89        76.31        76.07         0.00
           1         1000        67.73        68.26        68.00         0.06
           2         1000        68.47        69.29        68.90         0.11
           4         1000        69.23        69.88        69.57         0.22
           8         1000        68.20        68.91        68.55         0.44
      262144          160     19272.77     20713.69     20165.05        48.28
      524288           80     63144.46     65858.79     63997.79        30.37
     1048576           40     83868.32     89965.37     87337.56        44.46
     2097152           20     91448.50    106147.55     99928.08        75.37
     4194304           10    121632.81    192385.91    161055.82        83.17

[...]

#---------------------------------------------------------------
# Benchmarking Alltoallv
# #processes = 8
# ( 16 additional processes waiting in MPI_Barrier)
#---------------------------------------------------------------
      #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
           0         1000         0.10         0.10         0.10
           1         1000        18.49        18.50        18.49
           2         1000        18.50        18.52        18.51
           4         1000        18.47        18.48        18.47
           8         1000        18.40        18.40        18.40
          16         1000        18.42        18.43        18.43
          32         1000        18.89        18.90        18.89
          68         1000       601.29       601.36       601.33
       65536          640      1284.44      1284.71      1284.57
      131072          320      3936.76      3937.16      3937.01
      262144          160     10745.08     10746.09     10745.83
      524288           80     22101.26     22103.33     22102.58
     1048576           40     44044.33     44056.68     44052.76
     2097152           20     88028.00     88041.70     88037.15
     4194304           10    175437.78    175766.59    175671.63
```

```
[...]

#---------------------------------------------------------------
# Benchmarking Alltoallv
# #processes = 24
#---------------------------------------------------------------
      #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
           0         1000         0.18         0.22         0.18
           1         1000       891.94       892.74       892.58
           2         1000       891.63       892.46       892.28
           4         1000       879.25       880.09       879.94
           8         1000       898.30       899.29       899.05
      262144           15    923459.34    950393.47    938204.26
      524288           10   1176375.79   1248858.31   1207359.81
     1048576            6   1787152.85   1906829.99   1858522.38
     2097152            4   3093715.25   3312132.72   3203840.16
     4194304            2   5398282.53   5869063.97   5702468.73
```

As you can see, if you specify 24 processes then the benchmark will also run the 2,4,8,16 tests. You can fix the minimum number of processes to use with:

```
$ mpirun [...] <benchmark> -npmin <minimum_number_of_procs>
```

NOTE: Other useful control commands can be found in the Users Guide (/doc directory) under section 5.

9 Application Software and Program Libraries

9.1 Automatically Tuned Linear Algebra Software (ATLAS)

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software, LAPACK for example.

The ATLAS (Automatically Tuned Linear Algebra Software) project is an ongoing research effort that provides C and Fortran77 interfaces to a portably efficient BLAS implementation, as well as a few routines from /htmladdnormallinkLAPACKhttp://math-atlas.sourceforge.net/

9.1.1 Using ATLAS

To initialize the environment use:

- module load blas/atlas-9.11_sunstudio12.1 (compiled with gcc)
- module load blas/atlas-9.11_sunstudio12.1 (compiled with sun)

To use level1-3 functions available in atlas see the prototypes functions in cblas.h and use them in your examples. For compiling you should specifiy the necessary libraries files.

Example:

- gcc -lcblas -latlas example.c
- cc -lcblas -latlas example.c

It is recommended the version compiled with gcc.It is almost never a good idea to change the C compiler used to compile ATLAS's generated double precision (real and complex) and C compiler used to compile ATLAS's generated single precision (real and complex), and it is only very rarely a good idea to change the C compiler used to compile all other double precision routines and C compiler used to compile all other single precision routines . For ATLAS 3.8.0, all architectural defaults are set using gcc 4.2 only (the one exception is MIPS/IRIX, where SGI's compiler is used). In most cases, switching these compilers will get you worse performance and accuracy, even when you are absolutely sure it is a better compiler and flag combination!

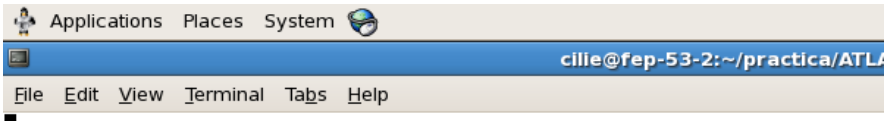
9.1.2 Performance

```
./xsl3blastst


---------------------------------- GEMM -----------------------------------
TST# A B    M    N    K ALPHA  LDA  LDB  BETA  LDC  TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ===== ==== ==== ===== ==== ===== ===== ==== =====
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.01 250.0 1.00 -----
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.00   0.0 0.00 PASS
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.02 800.0 1.00 -----
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.01 1333.2 1.67 PASS
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.07 749.9 1.00 -----
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.00   0.0 0.00 PASS
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.16 800.0 1.00 -----
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.02 6399.7 8.00 PASS
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.33 762.1 1.00 -----
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.03 8928.3 11.71 PASS
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.65 666.6 1.00 -----
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.06 7199.5 10.80 PASS
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  1.03 664.7 1.00 -----
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  0.08 9025.7 13.58 PASS
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  1.72 596.7 1.00 -----
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  0.14 7110.7 11.92 PASS
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  2.18 667.5 1.00 -----
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  0.19 7754.8 11.62 PASS
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  2.90 688.7 1.00 -----
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  0.26 7691.8 11.17 PASS

10 tests run, 10 passed

~
~
```




```
./xdl3blastst


---------------------------------- GEMM -----------------------------------
TST# A B    M    N    K ALPHA  LDA  LDB  BETA  LDC  TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ===== ==== ==== ===== ==== ===== ===== ==== =====
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.01 166.7 1.00 -----
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.00   0.0 0.00 PASS
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.04 400.0 1.00 -----
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.01 2000.0 5.00 PASS
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.09 586.9 1.00 -----
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.01 6750.0 11.50 PASS
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.18 711.1 1.00 -----
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.03 4571.1 6.43 PASS
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.39 637.7 1.00 -----
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.07 3676.3 5.76 PASS
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.68 635.3 1.00 -----
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.10 4319.7 6.80 PASS
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  1.14 601.7 1.00 -----
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  0.22 3118.0 5.18 PASS
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  2.06 497.1 1.00 -----
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  0.29 3555.3 7.15 PASS
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  2.26 646.2 1.00 -----
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  0.32 4613.6 7.14 PASS
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  2.97 672.9 1.00 -----
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  0.47 4273.2 6.35 PASS

10 tests run, 10 passed
```

```
./xsl3blastst


---------------------------------- GEMM -----------------------------------
TST# A B    M    N    K ALPHA  LDA  LDB  BETA  LDC  TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ===== ==== ==== ===== ==== ===== ===== ==== =====
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.01 250.0 1.00 -----
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.00   0.0 0.00 PASS
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.02 800.0 1.00 -----
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.01 1333.2 1.67 PASS
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.07 749.9 1.00 -----
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.00   0.0 0.00 PASS
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.16 800.0 1.00 -----
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.02 6399.7 8.00 PASS
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.33 762.1 1.00 -----
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.03 8928.3 11.71 PASS
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.65 666.6 1.00 -----
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.06 7199.5 10.80 PASS
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  1.03 664.7 1.00 -----
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  0.08 9025.7 13.58 PASS
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  1.72 596.7 1.00 -----
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  0.14 7110.7 11.92 PASS
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  2.18 667.5 1.00 -----
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  0.19 7754.8 11.62 PASS
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  2.90 688.7 1.00 -----
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  0.26 7691.8 11.17 PASS

10 tests run, 10 passed

~
~
```

```
./xzl3blastst


---------------------------------- GEMM -----------------------------------
TST# A B    M    N    K     ALPHA  LDA  LDB      BETA  LDC TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ==== ==== ==== ==== ==== ==== ==== ===== ==== =====
   0 N N  100  100  100  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 666.6 1.00 -----
   0 N N  100  100  100  1.0  0.0 1000 1000  1.0  0.0 1000  0.0   0.0 0.00 PASS
   1 N N  200  200  200  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 640.0 1.00 -----
   1 N N  200  200  200  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 3199.8 5.00 PASS
   2 N N  300  300  300  1.0  0.0 1000 1000  1.0  0.0 1000  0.2 1124.9 1.00 -----
   2 N N  300  300  300  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 2454.4 2.18 PASS
   3 N N  400  400  400  1.0  0.0 1000 1000  1.0  0.0 1000  0.5 1066.6 1.00 -----
   3 N N  400  400  400  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 4266.4 4.00 PASS
   4 N N  500  500  500  1.0  0.0 1000 1000  1.0  0.0 1000  1.1 909.0 1.00 -----
   4 N N  500  500  500  1.0  0.0 1000 1000  1.0  0.0 1000  0.2 4166.4 4.58 PASS
   5 N N  600  600  600  1.0  0.0 1000 1000  1.0  0.0 1000  1.7 1002.3 1.00 -----
   5 N N  600  600  600  1.0  0.0 1000 1000  1.0  0.0 1000  0.4 3999.8 3.99 PASS
   6 N N  700  700  700  1.0  0.0 1000 1000  1.0  0.0 1000  2.4 1135.7 1.00 -----
   6 N N  700  700  700  1.0  0.0 1000 1000  1.0  0.0 1000  0.7 3897.5 3.43 PASS
   7 N N  800  800  800  1.0  0.0 1000 1000  1.0  0.0 1000  4.2 978.9 1.00 -----
   7 N N  800  800  800  1.0  0.0 1000 1000  1.0  0.0 1000  1.0 4031.2 4.12 PASS
   8 N N  900  900  900  1.0  0.0 1000 1000  1.0  0.0 1000  5.7 1016.7 1.00 -----
   8 N N  900  900  900  1.0  0.0 1000 1000  1.0  0.0 1000  1.3 4365.0 4.29 PASS
   9 N N 1000 1000 1000  1.0  0.0 1000 1000  1.0  0.0 1000  6.7 1186.2 1.00 -----
   9 N N 1000 1000 1000  1.0  0.0 1000 1000  1.0  0.0 1000  1.9 4319.4 3.64 PASS

10 tests run, 10 passed
```

9.2 MKL - Intel Math Kernel Library

Intel Math Kernel Library (Intel MKL) is a library of highly optimized, extensively threaded math routines for science, engineering, and financial applications that require maximum performance. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers, Fast Fourier Transforms, Vector Math, and more. Offering performance optimizations for current and next-generation Intel processors, it includes improved integration with Microsoft Visual Studio, Eclipse, and XCode. Intel MKL allows for full integration of the Intel Compatibility OpenMP run-time library for greater Windows/Linux cross-platform compatibility.

9.2.1 Using MKL

To initialize the environment use:

- module load blas/mkl-10.2

To compile with gcc an example that uses mkl functions you should specify necessary libraries. Example:

- gcc -lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm example.c

9.2.2 Performance

```
./xcl3blastst


---------------------------------- GEMM -------------------------------------
TST# A B    M    N    K     ALPHA  LDA  LDB      BETA  LDC TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ==== ==== ==== ==== ==== ==== ==== ===== ==== =====
   0 N N  100  100  100  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 195.2 1.00 -----
   0 N N  100  100  100  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 190.5 0.98 PASS
   1 N N  200  200  200  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 64000.0 1.00 ----
   1 N N  200  200  200  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 1016.0 0.02 PASS
   2 N N  300  300  300  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 9001.1 1.00 -----
   2 N N  300  300  300  1.0  0.0 1000 1000  1.0  0.0 1000  0.2 1285.9 0.14 PASS
   3 N N  400  400  400  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 9482.9 1.00 -----
   3 N N  400  400  400  1.0  0.0 1000 1000  1.0  0.0 1000  0.5 1123.0 0.12 PASS
   4 N N  500  500  500  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 11766.5 1.00 ----
   4 N N  500  500  500  1.0  0.0 1000 1000  1.0  0.0 1000  0.8 1297.2 0.11 PASS
   5 N N  600  600  600  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 12001.8 1.00 ----
   5 N N  600  600  600  1.0  0.0 1000 1000  1.0  0.0 1000  1.3 1367.3 0.11 PASS
   6 N N  700  700  700  1.0  0.0 1000 1000  1.0  0.0 1000  0.2 12647.1 1.00 ----
   6 N N  700  700  700  1.0  0.0 1000 1000  1.0  0.0 1000  1.5 1886.2 0.15 PASS
   7 N N  800  800  800  1.0  0.0 1000 1000  1.0  0.0 1000  0.3 13300.7 1.00 ----
   7 N N  800  800  800  1.0  0.0 1000 1000  1.0  0.0 1000  1.3 3089.5 0.23 PASS
   8 N N  900  900  900  1.0  0.0 1000 1000  1.0  0.0 1000  0.4 13347.6 1.00 ----
   8 N N  900  900  900  1.0  0.0 1000 1000  1.0  0.0 1000  1.4 4064.7 0.30 PASS
   9 N N 1000 1000 1000  1.0  0.0 1000 1000  1.0  0.0 1000  0.6 13159.9 1.00 ----
   9 N N 1000 1000 1000  1.0  0.0 1000 1000  1.0  0.0 1000  1.7 4625.0 0.35 PASS

10 tests run, 10 passed

~
~
```

```
./xsl3blastst


---------------------------------- GEMM -----------------------------------
TST# A B    M    N    K ALPHA  LDA  LDB  BETA  LDC  TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ===== ==== ==== ===== ==== ===== ===== ==== =====
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.06  33.9 1.00 -----
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.01 153.9 4.54 PASS
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.00   0.0 1.00 -----
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.02 1000.2 0.00 PASS
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.01 9001.5 1.00 -----
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.05 1149.1 0.13 PASS
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.01 16002.0 1.00 -----
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.11 1153.3 0.07 PASS
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.03 8065.6 1.00 -----
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.21 1207.9 0.15 PASS
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.03 13502.1 1.00 -----
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.33 1317.3 0.10 PASS
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  0.06 11829.4 1.00 -----
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  0.52 1309.4 0.11 PASS
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  0.08 12801.9 1.00 -----
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  0.81 1269.1 0.10 PASS
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  0.11 12791.4 1.00 -----
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  1.06 1380.9 0.11 PASS
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  0.14 13891.0 1.00 -----
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  0.69 2915.9 0.21 PASS

10 tests run, 10 passed

~


./xdl3blastst


---------------------------------- GEMM -----------------------------------
TST# A B    M    N    K ALPHA  LDA  LDB  BETA  LDC  TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ===== ==== ==== ===== ==== ===== ===== ==== =====
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.04  44.5 1.00 -----
   0 N N  100  100  100   1.0 1000 1000   1.0 1000  0.03  76.9 1.73 PASS
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.00   0.0 1.00 -----
   1 N N  200  200  200   1.0 1000 1000   1.0 1000  0.03 500.1 0.00 PASS
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.02 3375.6 1.00 -----
   2 N N  300  300  300   1.0 1000 1000   1.0 1000  0.08 675.1 0.20 PASS
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.02 5334.0 1.00 -----
   3 N N  400  400  400   1.0 1000 1000   1.0 1000  0.19 670.3 0.13 PASS
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.05 4902.7 1.00 -----
   4 N N  500  500  500   1.0 1000 1000   1.0 1000  0.19 1330.0 0.27 PASS
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.07 6085.4 1.00 -----
   5 N N  600  600  600   1.0 1000 1000   1.0 1000  0.59 731.1 0.12 PASS
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  0.11 6181.1 1.00 -----
   6 N N  700  700  700   1.0 1000 1000   1.0 1000  0.95 723.7 0.12 PASS
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  0.17 6132.7 1.00 -----
   7 N N  800  800  800   1.0 1000 1000   1.0 1000  1.04 980.1 0.16 PASS
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  0.22 6509.9 1.00 -----
   8 N N  900  900  900   1.0 1000 1000   1.0 1000  0.41 3591.7 0.55 PASS
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  0.30 6558.4 1.00 -----
   9 N N 1000 1000 1000   1.0 1000 1000   1.0 1000  0.54 3697.4 0.56 PASS

10 tests run, 10 passed
```

```
./xzl3blastst


---------------------------------- GEMM -----------------------------------------
TST# A B    M    N    K     ALPHA  LDA  LDB      BETA  LDC TIME MFLOP SpUp  TEST
==== = = ==== ==== ==== ==== ==== ==== ==== ==== ==== ==== ===== ==== =====
   0 N N  100  100  100  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 258.1 1.00 -----
   0 N N  100  100  100  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 186.1 0.72 PASS
   1 N N  200  200  200  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 7112.7 1.00 -----
   1 N N  200  200  200  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 581.9 0.08 PASS
   2 N N  300  300  300  1.0  0.0 1000 1000  1.0  0.0 1000  0.0 5400.8 1.00 -----
   2 N N  300  300  300  1.0  0.0 1000 1000  1.0  0.0 1000  0.3 660.7 0.12 PASS
   3 N N  400  400  400  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 6096.2 1.00 -----
   3 N N  400  400  400  1.0  0.0 1000 1000  1.0  0.0 1000  0.1 3737.8 0.61 PASS
   4 N N  500  500  500  1.0  0.0 1000 1000  1.0  0.0 1000  0.2 6580.0 1.00 -----
   4 N N  500  500  500  1.0  0.0 1000 1000  1.0  0.0 1000  0.4 2793.7 0.42 PASS
   5 N N  600  600  600  1.0  0.0 1000 1000  1.0  0.0 1000  0.3 6546.4 1.00 -----
   5 N N  600  600  600  1.0  0.0 1000 1000  1.0  0.0 1000  1.1 1597.3 0.24 PASS
   6 N N  700  700  700  1.0  0.0 1000 1000  1.0  0.0 1000  0.4 6534.3 1.00 -----
   6 N N  700  700  700  1.0  0.0 1000 1000  1.0  0.0 1000  0.7 3838.3 0.59 PASS
   7 N N  800  800  800  1.0  0.0 1000 1000  1.0  0.0 1000  0.6 7112.2 1.00 -----
   7 N N  800  800  800  1.0  0.0 1000 1000  1.0  0.0 1000  1.5 2643.0 0.37 PASS
   8 N N  900  900  900  1.0  0.0 1000 1000  1.0  0.0 1000  0.8 7113.3 1.00 -----
   8 N N  900  900  900  1.0  0.0 1000 1000  1.0  0.0 1000  1.7 3405.1 0.48 PASS
   9 N N 1000 1000 1000  1.0  0.0 1000 1000  1.0  0.0 1000  1.1 7080.7 1.00 -----
   9 N N 1000 1000 1000  1.0  0.0 1000 1000  1.0  0.0 1000  2.2 3610.7 0.51 PASS

10 tests run, 10 passed

~
```

9.3 ATLAS vs MKL - level 1,2,3 functions

The BLAS functions were tested for all 3 levels and the results are shown only for level 3. To summarize the performance tests, it would be that ATLAS loses for Level 1 BLAS, tends to be beat MKL for Level 2 BLAS, and varies between quite a bit slower and quite a bit faster than MKL for Level 3 BLAS, depending on problem size and data type.

ATLAS's present Level 1 gets its optimization mainly from the compiler. This gives MKL two huge advantages: MKL can use the SSE prefetch instructions to speed up pretty much all Level 1 ops. The second advantage is in how ABS() is done. ABS() *should* be a 1-cycle operation, since you can just mask off the sign bit. However, you cannot standardly do bit operation on floats in ANSI C, so ATLAS has to use an if-type construct instead. This spells absolute doom for the performance of NRM2, ASUM and AMAX.

For the Level 2 and 3, ATLAS has it's usual advantage of leveraging basic kernels to the maximum. This means that all Level 3 ops follow the performance of GEMM, and Level 2 ops follow GER or GEMV. MKL has the usual disadvantage of optimizing all these routines seperately, leading to widely varying performance.

References

[1] The RWTH HPC Primer, http://www.rz.rwth-aachen.de/go/id/pil/lang/en

[2] Wikipedia page on SPARC processors, http://www.rz.rwth-aachen.de/go/id/pil/lang/en

[3] Sunsolve page on the Sun Enterprise 220R, http://sunsolve.sun.com/handbook_pub/validateUser.do?target=Systems/E220R/E220R

[4] Sunsolve page on the Sun Enterprise 420R, http://sunsolve.sun.com/handbook_pub/svalidateUser.do?target=Systems/E420R/E420R