The RWTH HPC-Cluster User's Guide

Short Version 6.5

Release: February 2009

Build: February 3, 2009

Dieter an Mey, Samuel Sarholz, Christian Terboven

Center for Computing and Communication, RWTH Aachen University
(Rechen- und Kommunikationszentrum der RWTH Aachen)

{anmey|sarholz|terboven}@rz.rwth-aachen.de

Ruud van der Pas, Eugene Loh

Sun Microsystems

{ruud.vanderpas|eugene.loh}@sun.com

What's New

These topics were added or changed significantly [1] compared to the prior release of this primer.

- We bought a new cluster from Sun and thus updated the description of our machines, see 2.1 on page 8
- We added more options to our mpiexec wrapper for interactive use, which offloads MPI processes from the frontends to dedicated machines. For more information see chapter 6.2 on page 18.

[1] see the changebar on the border of the page

Table of Contents

1 Introduction **6**
- 1.1 The HPC-Cluster . . . 6
- 1.2 Software Overview . . . 7
- 1.3 Examples . . . 7
- 1.4 Further Information . . . 8

2 Hardware **8**
- 2.1 Configuration . . . 8

3 Operating Systems **8**
- 3.1 Solaris . . . 8
- 3.2 Linux . . . 9
- 3.3 Windows . . . 9
- 3.4 Addressing Modes . . . 9

4 The RWTH Environment **9**
- 4.1 Login to Unix . . . 9
  - 4.1.1 Commandline Login . . . 9
  - 4.1.2 Graphical Login . . . 10
- 4.2 Login to Windows . . . 10
  - 4.2.1 Remote Desktop Connection . . . 10
  - 4.2.2 rdesktop, Linux Client . . . 11
- 4.3 The RWTH User File Management . . . 11
  - 4.3.1 Transferring Files to the Cluster . . . 11
- 4.4 Defaults of the RWTH User Environment (US-Sol + x86-Sol + Lin) . . . 12
  - 4.4.1 Z Shell Configuration Files . . . 12
- 4.5 The Module Package (US-Sol + x86-Sol + Lin) . . . 13
- 4.6 The RWTH Batch Job Administration . . . 14

5 Programming / Serial Tuning **14**
- 5.1 Introduction . . . 14
- 5.2 General Compiling and Linker hints (US-Sol + x86-Sol + Lin) . . . 15
- 5.3 Tuning Hints . . . 16
- 5.4 Hardware Performance Counters (All) . . . 17

6 Parallelization **17**
- 6.1 Shared Memory Programming . . . 17
- 6.2 Message Passing with MPI . . . 18
- 6.3 Hybrid Parallelization . . . 19

7 Debugging **19**
- 7.1 Static Program Analysis . . . 20
- 7.2 Dynamic Program Analysis . . . 20
- 7.3 Debuggers . . . 20
- 7.4 Runtime Analysis of OpenMP Programs . . . 21

8 Performance / Runtime Analysis Tools **21**

9 Application Software and Program Libraries **21**
- 9.1 Application Software . . . 21

A Beginners Introduction to the Unix HPC-Cluster **21**
A.1 Login 22
A.1.1 Notes to the Unix HPC-Cluster 22
A.2 The Example Collection 22
A.3 Compilation, Modules and Testing 23
A.4 Computation in batch mode 24

Keyword Index **28**

1 Introduction

The Center for Computing and Communication of the RWTH Aachen University has been operating a Unix cluster since march 1994.

The cluster is operated to serve the computational needs of researchers from the RWTH Aachen University and other universities in North-Rhine-Westphalia. Thus every employee from one of the universities may use the cluster for research purposes. Furthermore students of the RWTH Aachen University may get an account in order to get acquainted with parallel computers and learn how to program for them[2].

This primer serves as a practical introduction to the HPC-Cluster. It describes the hardware architecture, selected aspects of the operating environments, the programming environment and provides references for further information. It gives you a quick start in using the HPC-Cluster at the RWTH Aachen University and other systems hosted for institutes, which are integrated into the cluster.

1.1 The HPC-Cluster

The layout of the cluster is heterogeneous and contains a variety of hardware platforms and operating systems.
The goal is to keep the special features while offering a homogeneous environment as far as possible. The cluster keeps changing, since parts of it get replaced by newer faster and probably different machines. Therefore this document is updated regularly to keep up with the changes.

The HPC-Cluster mainly consists of *Xeon and Opteron-based*[3] *4 to 24 -way SMP nodes* and a few *Ultra*SPARC*-based, 64-way, SMP nodes*.[4] Whereas all UltraSPARC-based nodes are running Solaris, the Opteron- and Xeon-based nodes are either running Linux, Solaris or Windows.
Thus, there are four different platforms on the cluster available:

1. SPARC-Solaris (US-Sol)
2. x64-Solaris (x64-Sol)
3. Linux (x64-Lin)
4. Windows (x64-Win)

Accordingly we offer different frontends, where you can login for interactive access to the cluster:

Frontend name	**short alias**	**Proc - OS**
cluster.rz.RWTH-Aachen.DE		US-Sol
cluster-solaris-sparc.rz.RWTH-Aachen.DE	cl-sol-s	US-Sol
cluster-solaris-opteron.rz.RWTH-Aachen.DE	cl-sol-o	Opt-Sol
cluster-linux.rz.RWTH-Aachen.DE		Linux
cluster-linux-opteron.rz.RWTH-Aachen.DE	cl-lin-o	Opt-Lin
cluster-linux-xeon.rz.RWTH-Aachen.DE	cl-lin-x	Xeon-Lin
cluster-win.rz.RWTH-Aachen.DE		Windows
cluster-win2003.rz.RWTH-Aachen.DE		Windows 2003

All other machines are running in non-interactive mode and can be used by means of batch jobs (see chapter 4.6 on page 14).

[2] see appendix A on page 21 for a quick introduction the Linux cluster
[3] since both Opteron and Xeon processors belong to the same processor architecture, they will be called x64
[4] a complete overview is given in table 1 on page 25

1.2 Software Overview

A variety of different development tools as well as other ISV[5] software is available. However this primer focuses on describing software development tools. An overview is given in table 1 full primer on page 8.

1.3 Examples

We offer a collection of example programs and scripts, which should demonstrate the different aspects explained in the user's guide and are thus often referenced.

The *example scripts* demonstrate the use of many tools and commands. Command lines, for which an example script is available, have the following notation in this document:

```
$ $PSRC/pex/100|| echo ‘‘Hello World’’
```

You can either run the script **$PSRC/pex/100** to execute the example. The script includes all necessary initializations. Or you can do the initialization yourself and then run the command after the "pipes", in this case **echo "Hello World"**. However most of the scripts are offered for Unix only.

The *example programs* demonstrating e.g. the use of parallelization paradigms like OpenMP or MPI are available on a shared cluster filesystem where the environment variable **$PSRC** points to the base directory. On our Windows system the examples are located on the drive P:.

The code of the examples is usually available in the programming languages C++, C and FORTRAN (F). The directory name contains the programming language, the parallelization paradim and the name of the code, e.g. the directory **C++-omp-pi** contains the Pi example written in C++ and parallelized with OpenMP. Available paradims are:

- **ser** : serial version, no parallelization. See chapter 5 on page 14
- **aut** : automatic parallelization done by the compiler for shared memory systems. See chapter 6.1 on page 17
- **omp** : shared memory parallelization with OpenMP directives. See chapter 6.1 on page 17
- **mpi** : parallelization using the message passing interface (MPI). See chapter 6.2 on page 18.
- **hyb** : hybrid parallelization, mixing MPI and OpenMP. See chapter 6.3 on page 19

The example directories contain Makefiles for Unix and Visual Studio project files for Windows. Furthermore there are some more specific examples in project subdirectories like *vihps*.

You have to copy the examples to a writeable directory before using them. On Unix you can copy an example to your home directory by changing into the example directory with e.g.

```
$ cd $PSRC/F-omp-pi
```

and running

```
$ gmake cp
```

After copying a new shell is started and instructions how to build the example are given.

```
$ gmake
```

will invoke the compiler to build the example program and then run it.

Additionally we offer a detailed beginners introduction for the Unix cluster as an appendix, see A on page 21. It contains a step by step description for building a first program and should be a good starting point in helping to understand many explanations of this document. It may also be interesting for advanced Unix users who are new to our HPC-Cluster to get a quick start.

[5] Independent Software Vendor

1.4 Further Information

Please check our web pages for more up-to-date information and the latest version of this document:
http://www.rz.rwth-aachen.de/hpc/primer/

News, like new software or maintenance announcements about the HPC-Cluster, are provided through the **rzcluster** mailing list.
Interested users are also invited to join the **rzhpc** discussion list at **http://mailman.rwth-aachen.de/mailman/listinfo/[rzhpc|rzcluster]**

The histories of these mailing lists are accessible at
http://mailman.rwth-aachen.de/pipermail/[rzhpc|rzcluster]

Please feel free to send feedback, questions or problem reports to **hpc@rz.rwth-aachen.de**.

Have fun using the RWTH HPC-Cluster!

2 Hardware

This chapter describes the different machine architectures available at the RWTH Aachen HPC-Cluster.

2.1 Configuration

Table 1 on page 25 lists all the nodes of the HPC-Cluster. The node names depend on the operating system running.

The following list includes only machines, which are dedicated to general usage. In the course of the proceeding implementation of our integrative hosting concept[6] there is a number of hosted machines, which occasionally might be used for batch production jobs. Their machine names will not show up in this list.

The Center for Computing and Communication part of the HPC-Cluster has a peak performance of 10.6 TFlops, other hosted systems[7] , that of over 40 TFlops.

3 Operating Systems

We are running several different operatings systems on the RWTH Aachen HPC-Cluster machines. Solaris is the only choice for the SPARC systems, whereas we run three different operating systems, Solaris, Linux and Windows, on the x86 machines to accommodate user needs. The differences between these operating systems are explained in this chapter.

3.1 Solaris

Solaris is a Unix Operating System. The UltraSPARC based systems are running Solaris 10, as well as some Opteron based machines.

The command
```
$ uname -r
```

[6] Furher Information can be found at http://www.rz.rwth-aachen.de/go/id/pgo/

[7] The Center for Computing and Communication offers institutes of the RWTH Aachen University to integrate their computers into the HPC-Cluster, where they will maintained as part of the cluster. The computers will be installed in the computer room at the center where cooling and power is provided. Some institutes choose to share compute resources with others, thus being able to use more machines when the demand is high and giving unused compute cycles to others when they do not use their machines.

will displays the corresponding SunOS release level.

3.2 Linux

Linux is a UNIX-like operating system. We are running the 64-bit version of CentOS Linux, with support for 32-bit binaries, on our systems. CentOS is a clone of RedHat Enterprise Linux.

The CentOS release is displayed by the command:

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

The Linux kernel version can be printed with the command

```
$ uname -r
```

3.3 Windows

The nodes of the Windows part of the HPC-Clusterruns Windows Server 2008. All interactive services are disabled on the compute nodes in order to not interfer with compute jobs.

3.4 Addressing Modes

All operating systems on our machines (Solaris, Linux and Windows) support 64-bit addressing. Programs can be compiled and linked either in 32-bit mode or in 64-bit mode. This affects memory addressing, the usage of 32- or 64-bit pointers, but has no influence on the capacity or precision of floating point numbers (4- or 8-byte real numbers). Programs requiring more than 4 GB of memory have to use 64-bit addressing mode. You have to specify the addressing mode at compile and link time. The default mode is 32-bit on Solaris and Windows and 64-bit on Linux. On the Opteron processor, the 64-bit mode is usually faster, because it offers more registers.

Note: *long int* data and pointers in C/C++ programs are stored with 8 bytes when using 64-bit addressing mode, thus being able to hold larger numbers. This example will print out two times "4" in 32-bit mode and two times "8" in 64-bit mode.

Listing 1: $CC $FLAGS_ARCH32 $PSRC/pis/addressingModes.c; a.out

```c
#include <stdio.h>
int main (int argc, char **argv)
{
   int* p;
   long int li;
   printf ("%d %d\n", sizeof(p), sizeof(li));
   return 0;
}
```

4 The RWTH Environment

4.1 Login to Unix

4.1.1 Commandline Login

The secure shell **ssh** is used to log into the Unix-like systems. Usually ssh is installed by default on Linux and Unix systems. Therefore you can log into the cluster *from a local Unix machine* using the command

```
$ ssh cluster.rz.rwth-aachen.de -l username
```

Depending on the platform you want to log into, use one of the following hosts:

Frontend name	short alias	Proc - OS
cluster.rz.RWTH-Aachen.DE		
cluster-solaris-sparc.rz.RWTH-Aachen.DE	cl-sol-s	US-Sol
cluster-solaris-opteron.rz.RWTH-Aachen.DE	cl-sol-o	Opt-Sol
cluster-linux.rz.RWTH-Aachen.DE		Linux
cluster-linux-opteron.rz.RWTH-Aachen.DE	cl-lin-o	Opt-Lin
cluster-linux-xeon.rz.RWTH-Aachen.DE	cl-lin-x	Xeon-Lin

Depending on your local configuration it may be necessary to use the **-Y**[8] flag to enable the forwarding of graphical programs.

To log into the cluster *from a Windows machine*, you need to have an ssh client installed. Such a software is provided e.g. by the cygwin (http://www.cygwin.com) environment, which is free to use. Other software is available under different licenses, for example putty (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html) or SSH Client for Windows (ftp://ftp.cert.dfn.de/pub/tools/net/ssh).

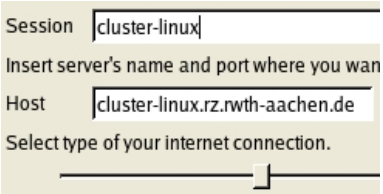
The SSH Client for Windows provides a graphical file manager for copying files to and from the cluster as well, another tool providing such a functionality is WinSCP (http://winscp.net/eng/docs/start).

4.1.2 Graphical Login

We are running an NX server on our Linux frontend machines. NX allows you to run remote X11 sessions even across low-bandwidth network connections, as well as reconnecting to running sessions.

You can download the NX client from http://www.nomachine.com/download.php

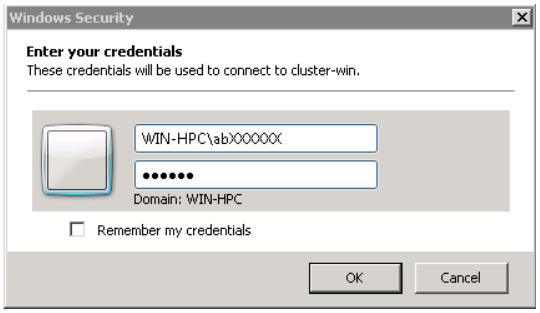
The first time NX is started the *NX Connection Wizard* will help setting up the connection. To get started all you need is to enter the session information. By default you will get KDE desktop from which you can start other programs.



4.2 Login to Windows

4.2.1 Remote Desktop Connection

To log into a Windows system from a *Windows environment* the program *Remote Desktop Connection* is used. You will find it under *"Start"* → *"Programs"* → *"Accessories"* → *"Remote Desktop Connection"*. („Start" → „Programme" → „Zubeh"or" →„Remotedesktopverbindung") After starting the program, you need to enter the client name



(e.g. **cluster-win.rz.rwth-aachen.de**) into the appearing window and then enter your username and password.

Note: Make sure to connect to the *WIN-HPC* domain, as local login will not work.

[8] older versions of ssh have to use the **-X** option

You can export local drives or printers to the remote desktop session. Choose **Options** and then the **Local Resources** tab, on the login window of the remote desktop connection, and check the local devices you want to export.

If the *Remote Desktop Connection* program is not installed, it can be downloaded from the Microsoft homepage. *Note*: Administrator privileges are required for installation.

4.2.2 rdesktop, Linux Client

To log into a Windows system *from a Linux client*, the **rdesktop** program is used. By calling
`$ rdesktop cluster-win.rz.rwth-aachen.de`[9]

you will get a graphical login screen. Frequently used rdesktop options are listed in table 2 on page 25. If called without parameter **rdesktop** will give information about further options. The following line gives an example of rdesktop usage:

```
$ rdesktop -a 24 -g 90% -r sound:local -r disk:tmp=/tmp -k de
cluster-win.rz.rwth-aachen.de
```

4.3 The RWTH User File Management

Every user owns directories on shared file systems. These are **$HOME**=/**home**/***username*** or **H:** for long-term user files and a fast **$WORK**=/**work**/***username*** or **W:** for large, medium-term work files. The work file system is indented for intermediate compute results. Additionally we offer an X: drive, which is only accessible on Windows.

The $HOME data will be backed-up in regular intervals. Unused data in the work file system is deleted automatically after 4 weeks, thus permanent data has to be stored somewhere else, for example the home directory.
Note: There is no backup of the work file system! Do not store important, non reproducable or non recomputable data, like source code or input data, here.

Every user has limited space (quota) on these file systems. Use the **quota** command, to see how much of your space is already occupied and how much is still left. With ZFS every user has its own filesystem of a given size. There is no file count limit and no soft limit anymore. Furthermore we offer snapshots of the home directories so that you can access older versions of accidentally erased or modified files, without restoring them from the backup. If deleted files still exist in a snapshot, they occupy space which is counted to the quota. This space will be freed at a later point in time when the snapshot is deleted automatically. The snapshots are located in the **~/.zfs/snapshot** directory.

Due to technical limitations the quota in the home directory is rather small. If you need more space, please let us know (mailto:unix@rz.rwth-aachen.de).

Furthermore a directory **$TMP** is available for storing session-dependent temporary data. Use **%TMP%** on the Windows commandline. It will be automatically created before and deleted after the terminal session or a batch job.

You can use the archive service, to store large long-term data, e.g. simulation result files, for future use. A description how to use the archive service can be found at http://www.rz.rwth-aachen.de/li/k/qgy/lang/en

4.3.1 Transferring Files to the Cluster

If you login to the *Windows* cluster you can export your local drives or directories and access the files as usual. See chapter 4.2.1 on page 10 or 4.2.2 on page 11 for more details. Furthermore you can use the hot keys *ctrl+c* and *ctrl+v* to copy files to and from the remote host.

To transfer files to the *Unix* cluster the secure copy command **scp**, on Unix, or the *Secure File Transfer Client*, on Windows, can be used. Usually the latter is located in *"Start"* →

[9] some versions of rdesktop need the **-4** option to work with our windows frontend.

"Programs" → *"SSH Secure Shell"* → *"Secure File Transfer Client"*, if installed. To connect to a system use the menu *"File"* → *"Quick Connect"*. Enter the hostname and username and select *Connect*. You will get a split window, the left half represents the local computer and the right half the remote system. Files can be exchanged by drag-and-drop.

4.4 Defaults of the RWTH User Environment (US-Sol + x86-Sol + Lin)

The default login shell is the **zsh** shell. Its prompt is symbolized by the dollar sign.

With the special "." dot command a shell script is executed as part of the current process. Thus changes made to the variables from within this script affect only the current shell, which is the main purpose of initialization scripts.

```
$ .  $PSRC/pex/440
```

Environment variables are set with

```
$ export VARIABLE=value
```

This corresponds to the C shell command

```
% setenv VARIABLE value
```

If you prefer to use a different shell, keep in mind to run initialization scripts before you change to your preferred shell or inside of it, otherwise they will run after the shell exits.

```
$ .  init_script
$ exec tcsh
```

The C shell prompt is indicated with a "percentage" symbol.

4.4.1 Z Shell Configuration Files

This section describes how to configure the zsh to your needs.

The user configuration files for the zsh are ~/**.zshenv** and ~/**.zshrc**, which are sourced (in this order) during login. The file ~/**.zshenv** is sourced on *every* execution of a zsh. If you want to initialize something e.g. in scripts which use the zsh to execute, put it in ~/.zshenv, but be aware that this file is sourced during login, too.
Note: Never put a command which calls a zsh in the ~/.zshenv, as this will cause an endless recursion and you will not be able to login anymore. Do not write to standard output in ~/.zshenv or you will run into problems using **scp**.

In login mode the file ~/**.zshrc** is sourced also, therefore ~/.zshrc is suited for interactive zsh configuration like setting aliases or setting the look of the prompt. If you want more information, like the actual path in your prompt, export a format string in the environment variable **PS1**. Example:

```
$ export PS1='%n@%m:%~$'
```

This will look like *user@cluster:~/directory$*

Since there are different platforms and not all software is available everywhere, it might be useful to differ to which platform you are logging-in to and e.g. load appropriate modules in your zsh configuration files. This example prints the current Unix platform you are using.

Listing 2: $PSRC/pis/diffplatform.sh

```
if [ $R_OSTYPE = LINUX ]; then
    if [ $R_ADDRW = 64 ]; then
        echo 'Linux 64'
    fi
fi
if [ $R_OSTYPE = SUNOS ]; then
    if [ $R_HWTYPE = sunsparc ]; then
        echo 'Solaris Sparc'
    fi
    if [ $R_HWTYPE = i386 ]; then
        echo 'Solaris Opteron'
    fi
fi
```

You can find an example **.zshrc** in $PSRC/psr/zshrc.

For further information see: http://www.rz.rwth-aachen.de/go/id/owu/lang/en

4.5 The Module Package (US-Sol + x86-Sol + Lin)

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 environment variables such as $PATH, to choose for example a specific compiler version or use software packages. The need to load modules will be denoted in the according software sections in this document.
The advantage of the modules system is that environment changes can easily be undone by unloading a module. Furthermore dependencies and conflicts between software packages can be easily controlled. Color coded warning and error messages will be printed if conflicts are detected.

The **module** command is available for the ksh, zsh and tcsh shells. csh users should switch to the tcsh since it is backward compatible.
Note: bash users have to add the line **. /usr/local_host/etc/bashrc** into the **~/.bashrc**, in order to make the module function available.

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 options.

```
$ module avail
```

prints a list of available initialization scripts. This list depends on the platform you are logged in to. The modules are sorted in categories, e.g. CHEMISTRY and DEVELOPER. The output may look like the following example, but will usually be much longer.

```
-------- /usr/local_rwth/modules/modulefiles/linux/linux64/CHEMISTRY ------
gaussian/03(default)                gaussian/98

------- /usr/local_rwth/modules/modulefiles/linux/linux64/DEVELOPER -------
intel-32/10.0                        intel-mpi-64/3.0
intel-32/10.1(default)               intel-mpi-64/3.1(default)
intel-64/10.0                        intel-64/10.1(default)
```

An available module can be loaded with

```
$ module load file
```

This will set all necessary environment variables for the use of the respective software.

You can for example either enter the full name like **intel-32/9.0** or just **intel-32**, in which case the default **intel-32/9.1** will be loaded. The module file naming schema is **{*name*}[optional -*32* or -*64* depending on the addressing mode]/[*version*]**.

A module which has been loaded before but is lo longer needed can be removed by
`$ module unload file`

If you want to use another version of a software or e.g. another compiler, we *strongly recommend* to switch between modules.
`$ module switch oldfile newfile`

This will unload all modules from bottom up to the *oldfile*, unload the *oldfile*, load the *newfile* and then reload all previously unloaded 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]`

You can add a directory with your own module files with
`$ module use path`

By default, for all available software categories modules are loaded. As long as such a module is loaded the available software in this category is shown by the **module avail** command. If you never use software from e.g. the CHEMISTRY category you may want to unload this module with **module unload CHEMISTRY**. This will shorten the lengthy and unhandy list of available modules.
On systems running *Solaris*, the modules for the Sun Studio compilers and the HPC Cluster Tools with Sun's MPI are also loaded by default. On *Linux* systems the GNU Compiler Collection (gcc) comming with the distribution is loaded by default, as well as the Intel compiler and MPI modules.
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!

4.6 The RWTH Batch Job Administration

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 running longer than 15 minutes or using a lot of resources should use the batchsystem in order to reduce load on the frontend machines. Please refere to 4.6.1 full primer on page 24 and 4.6.2 full primer on page 27 for more information.

5 Programming / Serial Tuning

5.1 Introduction

The basic tool in programming is the compiler, which translates the program source to executable machinecode. However, not every compiler is available on different operating systems.

On the **Solaris** Systems the Sun Studio 12 development tools are now in production mode. They include the Sun Fortran, C and C++ compilers. Also available is the GNU compiler collection.

On the **Linux** operating system the freely available GNU compilers are the somewhat "natural" choice. Code generated by the gcc C/C++ compiler performs acceptably on the Opteron-based machines. The 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.

The Intel compiler family can also be used on the Opteron machines. Although the Intel compilers in general generate a very efficient code, it can be expected that AMD's processors are not the main focus of the Intel compiler team.
From version 12 on the Sun Compilers are available on Linux, too.

The Intel compiler offers interesting features and tools for OpenMP programmers. See chapter 6.1.3 full primer on page 44 and 7.4.2 full primer on page 57. The Sun Compiler offers comparable tools (see chapter 7.4.1 full primer on page 57). It also may, depending on the code, offer better performance.

A word of 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.

On **Windows**, the Microsoft Visual Studio environment is installed supporting the Microsoft Visual C++ compiler as well as Intel Fortran95 and C++ compilers.

5.2 General Compiling and Linker hints (US-Sol + x86-Sol + Lin)

To access non-default compilers you have to load the appropriate module file.[10] You can then 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 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. The offered variables are listed below. Values for different compilers are listed in tables 3 on page 25 and 4 on page 25.

- FC, CC, CXX – a variable containing the appropriate 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

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.
Example:

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

[10] see chapter 4.5 on page 13.

The makefiles of the example programs also use these variables. See chapter 1.3 on page 7 for further advice on using theses examples.

In general we recommend to use the same flags 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 from the library yyy, the library yyy has to be right of xxx in the command line, e.g **ld ... -lxxx -lyyy**.

The rightmost compiler option, in the command line, takes precedence over the ones on the left, e.g. **cc ... -O3 -O2**. In this example the optimization flag O3 is overwritten by O2. Special care has to be taken if macros like **-fast** are used, because they may overwrite other options unintentionally. Therefore it is advisable to put macro options at the beginning of the command line.

The search path for header files is extended with the **-I*directory*** option and the library search path with the **-L*directory*** option.

The environment variable LD_LIBRARY_PATH specifies 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.

Consider static linking of libraries. This will generate a larger executable, which is however a lot more portable. Especially on Linux it may be a good idea, since every distribution has slightly different library versions, which may not be compatible with each other.

5.3 Tuning Hints

You have developed a program and want to make it run faster. Here are some general program tuning hints:

- 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.

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

- Write efficient code, which can be optimized by the compiler. We offer videos of several talks that are a good introduction into this topic, see http://www1.rz.rwth-aachen.de/computing/events/material.php. Some more detailed tuning tips for the Sun compiler, which also hold true for other compilers, can be found in chapter 5.5.2 full primer on page 34.

- Access memory continuously in order to reduce cache and TLB misses. This especially effects multidimensional arrays and structures. A tool like Acumem (see chapter 8.4 full primer on page 62 may help to identify problems easily. Furthermore we offer videos from tuning workshops on our website.

- Use optimized libraries, e.g. the Sun Performance Library or the ACML library (see chapter 9 on page 21).

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

- Consider parallelization to reduce the runtime of your program.

5.4 Hardware Performance Counters (All)

Hardware Performance Counters are used to measure how certain parts, like floating point units or caches, of a CPU or memory system are used. They are very helpfull in finding performance bottlenecks in programs.
The Opteron and Xeon processor core offers 4 programmable 48-bit performance counters.

6 Parallelization

Parallelization for computers with shared memory (SM) means the automatic distribution of loop iterations over several processors (automatic parallelization), 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 distribution 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 usually do not run on computers with distributed memory. [11] As a consequence, MPI programs can use all available processors of the HPC-Cluster, whereas OpenMP programs can use up to 16 processors of a x86 node or up to 144 processors of a Sun Fire E25K node.

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

Please note, that large computing jobs should not be started interactively. Please use the batch system (see chapter 4.6 on page 14), which determines the distribution of the MPI tasks to the machines to a large extent.

We offer examples using the different parallelization paradigms. See chapter 1.3 on page 7 for information how to use them.

6.1 Shared Memory Programming

For shared memory programming, OpenMP is the de facto standard (http://www.openmp.org, http://www.compunity.org). The OpenMP API is defined 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 modified, 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 worker threads must be increased by changing a compiler specific 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, specification 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 sufficient condition!

[11] We are testing Cluster OpenMP at the moment, see http://www.rz.rwth-aachen.de/go/id/qgt/lang/en. If you have a project suitable for it and you want to give it a try please contact us.

The number of the threads has to be specified 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 specified 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.

In table 4 on page 25 we summarized the OpenMP compiler options. These compiler flags will be set in the environment variables FLAGS_AUTOPAR and FLAGS_OPENMP as explained in section 5.2 on page 15.

6.2 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 implementation. Therefore, there are plenty of implementations for PCs 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.

Like the compiler environment flags set by the compiler modules, we offer MPI environment variables in order to make it easier to write platform independent makefiles. However these variables are only available on our Unix systems. Since the compiler wrappers and the MPI libraries relate to a specific compiler, a compiler module has to be loaded before the MPI module. Some MPI libraries do not offer a C++ or a FORTRAN90 interface for all compilers, e.g. the Intel MPI does not offer such interfaces for the Sun compiler. If this is the case there will be an info printed while loading the MPI module.

- MPIEXEC – the MPI command used to start MPI applications, e.g. mprun or mpiexec.
- MPIFC, MPICC, MPICXX – compiler driver which automatically set the include path and links the MPI library automatically.
- FLAGS_MPI_BATCH – options necessary for executing in batch mode

This example shows how to use the variables.

```
$ $PSRC/pex/620|| $MPICXX -I$PSRC/cpmp $PSRC/cpmp/mpi_pi.cpp -o a.out
$ $PSRC/pex/620|| $MPIEXEC -np 2 a.out
```

On Linux we offer dedicated machines for interactive MPI tests. These machines will be used automatically by our mpiexec and mpirun wrapper. The goal is to avoid overloading the frontend machines with MPI tests and to enable larger MPI test with more processes.

The wrapper works transparently so you can start your MPI programs with the usuall MPI options. I order to make sure that MPI programs do not hinder each other the wrapper will check the load on the available machines and choose the least loaded ones. The chosen machines will get one processes per available processor. However this default setting may not work for a jobs, if for a jobs need more memory per processes than there is available per core, the jobs has to be spread to more machines. Therefore we added the **-m** <***processes per node***> option, which determines how many processes should be started per node. You can get a list of the mpiexec wrapper options with

```
$ mpiexec --help
```

which will print the list of option as shown in table 5 on page 26.

Passing environment variables from the master, where the MPI program is started, to the other hosts, is handled differently by the MPI implementations.

We recommend that if your program depends on environment variables, you let the master MPI process read them and broadcast the value to all other MPI processes.

6.3 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.

7 Debugging

If your program is causing problems, it might be a good opportunity to lean back and think for a while.

- What were the last changes that you made? (A source code revision system e.g. SVN, CVS or RCS might help.)
- Reduce the optimization level of your compilation.
- Look for compiler messages and warnings. Use tools for a static program analysis (see chapter 7.1 on page 20).
- Chose a smaller data set. Try to build a specific test case for your problem.
- Try a dynamic analysis with appropriate compiler options (see chapter 7.2 on page 20).
- Reduce the number of CPUs in a parallel program, try a serial program run if possible.
- Use a debugger like TotalView (see chapter 7.3.1 full primer on page 54). Use the smallest case which shows the error.
- In case of an OpenMP program, use a threadchecking tool like the Sun Thread Analyzer (see chapter 7.4.1 full primer on page 57) or the Intel ThreadChecker (see chapter 7.4.2 full primer on page 57).
- If it is an OpenMP program, try to compile without optimization, e.g. with **-g -O0 -xopenmp=noopt** for the Sun compilers

7.1 Static Program Analysis

First, an exact static analysis of the program is recommended for error detection. Watch for compiler warnings and turn on a high verbosity level while compiling. Furthermore the tools listed in table 6 can be used for static analysis.

Sometimes, program errors occur only with high compiler optimization. This can be a compiler error or a program error. If the program runs correctly without compiler optimizations, the module causing the trouble can be found by systematic bisectioning. With this technique, one compiles half of the application with the *right* options and the other half with the *wrong* options. If the program then fails, one will know which part causes the problem. Likewise, if the program runs fine now, repeat the process for the part of the program causing the failure.

7.2 Dynamic Program Analysis

Many compilers offer options to perform runtime checks of the generated program. This could be e.g. array bound checks or checks for not initialized variables. Study the compiler documentation and search for compiler options which enable additional runtime checks.

The **Sun Fortran** compiler for example does array bound checking with the option **-C** and global program analysis with the option **-Xlist**. Compiling with **-xcheck=init_local** initializes local variables to a value that is likely to cause an arithmetic exception if it is used before it is assigned by the program. Memory allocated by the ALLOCATE statement will also be initialized in this manner. SAVE variables, module variables, and variables in COMMON blocks are not initialized. Floating point errors like division by zero, overflows and underflows are reported with the option **-ftrap=%all**.

The Sun compilers also offer an option **-xcheck=stkovf** to detect stack overflows at runtime. In case of a stack overflow a core file will be written, which then can be analyzed by a debugger. The stack trace will contain a function name indicating the problem.

The GNU compiler offers the option -fmudflap to trace memory accesses during runtime. If an illegal access is detected, the program will halt.

To *detect common errors with dynamic memory allocation*, one can use the library **Electric Fence**. It helps to detect two common programming bugs: software that overruns the boundaries of a *malloc()* memory allocation, and software that touches a memory allocation that has been released by *free()*. If an error is detected, the program stops with a segmentation fault, then the error can easily be found with a debugger. To use the library, link with **-lefence**. For more information see the manual page **man libefence**.

Memory leaks can be detected using TotalView (see chapter table.1 full primer on page 76) or with the sampling collector (**collect -H**, see chapter 8.1 full primer on page 59).

If a program with optimization delivers other results than without, floating point optimization may be responsible. There is a possibility to test this by optimizing the program carefully. The Sun compilers for example turn floating point optimization off with the options **-fsimple=0 -xnolibmopt**. Thus, the sequence of the floating point operations is not changed by the optimization, perhaps increasing the runtime. Also, you have to consider that on the x86 platform floating point calculation do not necessarily conform to IEEE standard by default, so rounding effects may differ between platforms.

7.3 Debuggers

A Debugger is a tool to control and look into a running program. It allows a programmer to follow the program execution step by step and see e.g. values of variables. It is a powerful tool for finding problems and errors in a program.

For debugging, the program must be translated with the option **-g** and optimization should be turned off to facilitate the debugging process. If compiled with optimization, some variables may not be visible while debugging and the mapping between the source code and the

executable program may not be accurate.

A core dump can be analyzed with a debugger, if the program was translated with **-g**. Do not forget to increase the core file size limit of your shell, if you want to analyze the core that your program may have left behind.

```
$ ulimit -c unlimited
```

But please do not forget to purge core files afterwards!

In general we recommend to use a full-screen debugger like TotalView or SunStudio to

- start your application and step through it
- analyze a core dump of a prior program run
- attach to a running program

In some cases, e.g. in batch scripts or when debugging over a slow connection, it might be preferable to use line mode debugger like **dbx** or **gdb**.

7.4 Runtime Analysis of OpenMP Programs

If an OpenMP program runs fine on one thread, but not on many, there is probably a data sharing conflict or data race condition. This is the case if e.g. a variable which should be private is shared or a shared variable is not protected by a lock.

The presented tools will detect data race conditions during runtime and point out the portions of code which are not thread safe.

Recommendation: Never put an OpenMP code into production, before having used a thread checking tool.

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. See the full version of this document to view this section.

9 Application Software and Program Libraries

9.1 Application Software

You can find the list of available application software and program libraries form several ISVs at http://www.rz.rwth-aachen.de/go/id/ond/lang/en/

As for the compiler and MPI suites we offer environment variables for the mathematical libraries to make usage and switching easier. These are **FLAGS_MATH_INCLUDE** for the include options and **FLAGS_MATH_LINKER** for linking the libraries. If loading more than one mathematical module the last loaded will overwrite this variables. However each module sets extra variables which will not be overwritten.

A Beginners Introduction to the Unix HPC-Cluster

This chapter aims to give new users a short tutorial on how to use the Unix HPC-Cluster. It will be explained how to set up the environment correctly in order to build a simple example program. Hopefully this can easily be adapted to your own code. In order to get more information on the steps performed you need to read the referenced chapters.

The first step you need to do is login[12] to the HPC-Cluster.

[12] If you do not have a HPC-Clusteraccount you can create one in TIM

A.1 Login

You have to use secure shell(ssh) to login. Therefore it might be necessary to install an ssh client on your local machine. If you are running Windows please see chapter 4.1 on page 9 on where to get such a ssh client. Depending on which client you choose there are different ways to enter the necessary information. The hostname you need to connect to is **cluster-linux.rz.rwth-aachen.de** and your user name is usually your TIM id.

On Unix systems like Linux ssh is usually installed or at least included in the distribution. Therefore you can open a console and enter the command

```
$ ssh -X <username>@cluster-linux.rz.rwth-aachen.de
```

Now you are prompted to enter your password. After entering your password you should be logged in to the HPC-Cluster and see a shell prompt like this:

```
LINUX:linuxoc68:/home/av006ss[5]$
```

The first word is the operating system, in this case `LINUX`, separated with a colon from the machine name `linuxoc68`. Note that the name is different from cluster-linux, which is just an alias to one of the HPC-Cluster machines. After the next colon the current directory is prompted, in this case `/home/av006ss`. This is your home directory, for more information on available directories please see chapter 4.3 on page 11. Please note that your user name, contained in the path, is of course different from av006ss. The number in the brackets counts the entered commands. The prompt end with the $ character. If you want to change your prompt please see chapter 4.4 on page 12.

A.1.1 Notes to the Unix HPC-Cluster

You are now logged-in to a Linux frontend machine. As we offer other Unix systems running Solaris you can use the appropriate other frontend machines to compile programs or prepare jobs. If you are are familiar to Linux and its GNU commands, note, that these commands may differ on Solaris. Some GNU commands are installed and have a "g" prepended, e.g. gmake, gtar or g_cp. The cluster consists of machines that are interactively accessible and machines that are only accessible by batch jobs, see chapter 4.6 on page 14. The interactive machines are not meant for time consuming jobs. Please keep in mind that there are other users on the system which are affected if the system gets overloaded.

A.2 The Example Collection

As a first step, we want to compile an example program from our Example Collection (chapter 1.3 on page 7) The Example Collection is located at /rwthfs/rz/SW/HPC/examples. You can access this directory with the help of the environment variable $PSRC, which stores the path.

To list the content of the examples directory, use the command ls with the environment variable as argument:

```
$ ls $PSRC
```

The examples differ in the parallelization paradigm and their programming language. See chapter 1.3 on page 7 or the README file for more information:

```
$ cat $PSRC/README.txt | more
```

The examples have to be copied to your home directory (~), as the global directory is readonly. We choose the jacobi examples written in C++ and parallelized with OpenMP and copy it.

```
$ mkdir ~/C++-omp-jacobi/; cp $PSRC/C++-omp-jacobi/* ~/C++-omp-jacobi/
```

Now change to the directory with

```
$ cd ~/C++-omp-jacobi
```

A.3 Compilation, Modules and Testing

Before we start compiling, we have to ensure the environment is properly set up. Because there are different and even contradicting needs, regarding software, we offer the modules system to easily adapt the environment. See chapter 4.5 on page 13 for more detailed information.

On login some modules are loaded by default. To get the list of these modules use the

```
$ module list
```

command. Now a list with the loaded modules is displayed, that probably ends with the lines
8) LIBRARIES
9) gcc/system-default
The modules written in capital are categories to make more modules for e.g. chemistry software available. Unloading such a module will shorten the long list of available modules, which can be printed with

```
$ module avail
```

gcc/system-default is the module setting up the Gnu compiler collection which is installed on the system. As we want to compile an OpenMP [13] program we need to switch to an OpenMP capable compiler, e.g. the gcc version 4.2. You can get a list of all available gcc versions by adding the name to the module avail command

```
$ module avail gcc
```

In order to switch between the two gcc versions we use the command

```
$ module switch gcc gcc/4.2
```

The version of the first gcc module in the command line doesn't need to be further specified. If there's no gcc in the loaded module list, e.g. because you are logged into a Solaris cluster machine, go ahead with

```
$ module load gcc/4.2
```
[14]

The compiler environment is now set up and you can use it. In order to make the use of different compilers easier and to be able to compile with the same command, several environment variables are set. You can look up the list of variables in chapter 5.2 on page 15.

As there is often more than one step needed to build a program, the **make** program offers a nice way do define theses steps in a *Makefile*. We offer such makefiles for the examples, which use the environment variables. Therefore when starting

```
$ gmake
```

the example will be build and executed according to the specified rules. Have a look at the Makefile if you are interested on more details.

As the Makefile already does everything but to explain the steps, we will go through and compile again while explaining the steps. We start with compiling the source files, in this case *main.cpp* and *jacobi.cpp*, with the C++ compiler:

```
$ $CXX $FLAGS_DEBUG $FLAGS_FAST $FLAGS_OPENMP -DREAD_INPUT -c jacobi.cpp
main.cpp
```
[15]

This command invokes the C++ compiler saved in the environment variable $CXX, in this case g++ as we are using the GNU compiler collection. The compiler reads both source files and it writes two objects files, which contain machine code. The variables $FLAGS_DEBUG $FLAGS_FAST $FLAGS_OPENMP contain compiler flags to write debugging information into the objects, to optimize the code for high performance and to enable OpenMP parallelization. The *-D* option specifies C preprocessor directives to allow conditional compilation of parts of the source code. The command line above is equivalent to writing just the content of the

[13] see chapter 6.1 on page 17

[14] we use the gcc for this example here because it is a widely used compiler and default on the most Linux systems. However there are compilers which offer better performance. Therefore we recommend to use another compiler, e.g. the Intel or Sun compiler, for production on our systems.

[15] If you are not on one of our cluster systems the values are probably not set and you cannot use these variables. However as every compiler has its own compiler flags, it makes live a lot easier on our systems, because one need not to remember or look up all the flags.

variables:

```
$ g++ -g -O3 -ffast-math -mtune=opteron -fopenmp -DREAD_INPUT -c jacobi.cpp
main.cpp
```

You can print the values of variables with the *echo* command, which should print the line above

```
$ echo $CXX $FLAGS_DEBUG $FLAGS_FAST $FLAGS_OPENMP -DREAD_INPUT -c jacobi.cpp
main.cpp
```

After compiling the object files, we need to link them to an executable. You can either use the linker *ln* directly, but it is recommended to let the compiler invoke the linker and add appropriate options for e.g. linking the OpenMP library automatically. Therefore you should use the same compiler options, you have used for compiling, for linking. Otherwise the compiler may not generate all needed linker options. To link the objects to the program *jacobi.exe* we use

```
$ $CXX $FLAGS_DEBUG $FLAGS_FAST $FLAGS_OPENMP jacobi.o main.o -o jacobi.exe
```

Now after building the executable, we can run it. The example program is an iterative solve algorithm with built-in measurement of time and megaflops per second. Because the *jacobi.exe* program needs input we supply an input file and start

```
$ ./jacobi.exe < input
```

After a few seconds you will get the output, including the runtime and megaflop rate, which depend on the load of the machine.

As we build a parallel OpenMP program it depends on the compiler with how many threads the program is executed if the environment variable $OMP_NUM_THREADS is not explicitly set. In the case of the GNU compiler the default is to use as many threads as processors are available.

We can set the variable and run again with one thread

```
$ export OMP_NUM_THREADS=1; ./jacobi.exe < input
```

Now the execution should take more time and the megaflop rate should be about 500 MFlops.

A.4 Computation in batch mode

After compiling the example and making sure it runs fine, we want to compute. However the interactive nodes are not suited for larger computations. Therefore we want so submit our example to the batch queue. For detailed information see chapter 4.6 on page 14. It will be executed when a compute node is available.

We specify the command to run with *echo* and pipe the string to *qsub*, which submits the batchjob. qsub needs several options in order to specify the needed resources, e.g. the number of CPUs and amount of memory to reserve, the operating system and the runtime.

```
$ echo "module switch gcc gcc4.2; jacobi.exe < input" | qsub -o
‘pwd‘/output.txt -j y-l h_rt=00:01:00 -l h_vmem=200M -l num_proc=2 -l
ostype=linux -m beas -wd ‘pwd‘ -M <your_email_address>
```
[16]

You will get an email when the job is finished, if you enter your email address instead of *<your_email_address>*. The output of the job will be written to *output.txt* in the current directory.

[16] This is not the recommended way to submit jobs, however we do not need a job script here. You can find several example scripts in chapter 4.6 on page 14. The used options are explained there as well.

Model	Processor type	Sockets/Cores /Threads	Memory Flops/node	Hostname
Sun T5120 20 nodes	UltraSparc T2 "Niagara2", 1.4 GHz	1 / 8 / 64	32 GByte 22.4 GFlops	suntc01-20
Sun Fire X4450 10 nodes	Intel Xeon 7460 2.66 GHz	4 / 24 / -	128 - 256 GByte 255.4 GFlops	linuxdc01..10
Fujitsu-Siemens RX600S4/X 2 nodes	Intel Xeon E7350 2.93 GHz	4 / 16 / -	64 GByte 187.5 GFlops	linuxhtc01..02
Fujitsu-Siemens RX200S4/X 60 nodes	Intel Xeon E5450 3.0 GHz	2 / 8 / -	16 - 32 GByte 96 GFlops	linuxhtc03..62 winhtc03..62
Sun Fire V40z 64 nodes	AMD Opteron 848 2,2 GHz	4 / 4 / -	8 GByte 17.6 Gflops	sunoc00..63 linuxoc00..63 winoc00..63
Sun Fire V40z 4 nodes	AMD Opteron 875 2,2 GHz (dualcore)	4 / 8 / -	16 GByte 35.2 GFlops	sunoc64..67 linuxoc64..67
Sun Fire X4600 2 nodes	AMD Opteron 885 2,6 GHz (dualcore)	8 / 16 / -	32 GByte 83.2 GFLops	sunoc69 linuxoc68

Table 1: Node Overview

Parameter	Description
-u *user*	login as *user*.
-d *domain*	use Windows-domain *domain* for authentication
-g *WxH* **-g** *P%*	desktop geometry *W* (width) x *H* (height), in pixel. use P% of you current screen resulution
-a *depth*	colordepth, *depth*=[8 | 16 | 24], 24 recommended, 8 default.
-f	full-screen mode.
-r *device*	enable specified device or directory redirection.
-k *value*	keyboard layout, e.g. ”de“ or ”us“.

Table 2: rdesktop options overview

Compiler	FLAGS_DEBUG	FLAGS_FAST	._FAST_NO_FPOPT	FLAGS_ARCH32|64
Sun	-g | -g0	-fast	-fast -fsimple=0	-xarch=*architecture* -m32 | -m64 (V>11)
Intel	-g	-O3 -xW -fp-model fast=2	-O3 -xW -fp-model precise	module load intel-32|64
GNU	-g	-O3 -ffast-math	-O3	-m32 | -m64

Table 3: Compiler options overview

Compiler	FLAGS_OPENMP	FLAGS_AUTOPAR
Sun	-xopenmp	-xautopar -xreduction
Intel	-openmp	-parallel
GNU	-fopenmp (4.2 and above)	-ftree-parallize-loops=<*threads*> (4.3 and above)

Table 4: OpenMP compiler options overview

--show	prints out which machines are used
-m <nr>	starts <nr> processes on every host
--cluster	prints out all available clusters
--cluster=<*clname*>	uses only cluster <*clname*>
--help | -h	prints this help and the help information of normal mpiexec
--onehost	starts all processes on one host
--listonly	just writes the machinefile, without starting the program
MPIHOSTLIST	specify which file contains the list of hosts to use if not specified, default list is taken
MPIMACHINELIST	if --listonly is used, this variable specifies the name of the created hostfile, default is host.list

Table 5: mpiexec wrapper options

cc -v | **cc -pedantic**	stricter semantic checks of C programs by the compiler (Sun | GNU compiler)
lint	more accurate syntax check of C programs
ftnchek	syntax check of Fortran77 programs (with some Fortran90 features) (US-Sol + Lin)

Table 6: Static program analysis tools

Index

bash, 13
batchsystem, 14

CC, 15
csh, 12, 13
CXX, 15

diskspace, 11

example, 7
export, 12

FC, 15
flags, 15
    arch32, 15
    arch64, 15
    autopar, 15, 18
    debug, 15
    fast, 15
    fast_no_fpopt, 15
    mpi_batch, 18
    openmp, 15, 18

hardware
    overview, 8
home, 11

interactive, 6

ksh, 12

library
    efence, 20
Linux, 9, 14
login, 6, 9

module, 13
MPICC, 18
MPICXX, 18
mpiexec, 18
MPIFC, 18

OMP_NUM_THREADS, 18
Opteron, 8

quota, 11

rdesktop, 11

Solaris, 8, 14
Sparc, 8
ssh, 9

tcsh, 13
tmp, 11

ulimit, 21
uname, 8, 9

work, 11

zsh, 12
zshenv, 12
zshrc, 12

Notes