An Introduction to High-Performance Computing (HPC)

By:
Tim Stitt Ph.D.

An Introduction to High-Performance Computing (HPC)

By:
Tim Stitt Ph.D.

Online:
< http://cnx.org/content/col11091/1.7/ >

C O N N E X I O N S

Rice University, Houston, Texas

Collection structure revised: October 5, 2009
PDF generated: February 6, 2011
For copyright and attribution information for the modules contained in this collection, see p. 36.

Table of Contents

Chapter 1

Introduction to HPC (slideshow)[1]

1.1 Introduction

The following presentation introduces the fundamentals of High-Performance Computing (HPC). To complement the presentation, a series of practical exercises are provided in supplementary modules, which can be used to reinforce the material introduced in the slides.

NOTE: The practical exercises are designed primarily to be completed on the Cray XT supercomputer at the **Swiss National Supercomputer Centre** (CSCS), Switzerland.

NOTE FOR NON-CSCS USERS: If you want to complete the practicals on your own system, you can directly download the source code templates for the exercises here[2].

1.2 HPC Slide-Show (Quicktime)

This media object is a video file. Please view or download it at
<HPC_Slideshow.mov>

TIP: A PDF version of the slide-show can be downloaded to your desktop here[3] [40 MB].

IMPORTANT: If you are completing the practical exercises on the CSCS Cray XT systems, please review the material in the module "**Editing, Compiling and Submitting Jobs on Ela**", before commencing.

[1] This content is available online at <http://cnx.org/content/m31999/1.7/>.
[2] See the file at <http://cnx.org/content/m31999/latest/Practicals.tgz>
[3] See the file at <http://cnx.org/content/m31999/latest/HPC_Intro.pdf>

Chapter 2

Editing, Compiling and Submitting Jobs on Ela[1]

2.1 Introduction

For this tutorial you will be using the **Ela** front-end system for editing, compiling and running your codes. Your executables will be submitted on **Gele** which is the test system for CSCS's flagship machine **Monte Rosa**.

Please **complete** this section before attempting the remaining exercises. If you require any assistance, please do not hesitate to contact the available support staff.

2.2 Logging into Ela

From your local system, you can log into **Ela** (using your supplied **username** and **password**) with the following commands:

Logging on to the Front-End System

```
ssh -X username@ela.cscs.ch
```

When you log into the front-end node **Ela**, you need to set up your **programming environment** as follows:

Setting Your Programming Environment

```
module load gele
module load PrgEnv-cray
module load pbs
```

> MODULES TIP: Loading **modules** allows the user to flexibly customize their programming environment and system software within their session. In the commands above, we have selected the mandatory system software for the Gele system and the Cray Programming Environment. For managing the submission and scheduling of jobs on the compute nodes of Gele, we have selected the PBS batch queuing system (more about this later).

[1] This content is available online at <http://cnx.org/content/m31991/1.8/>.

MODULE INFORMATION:

To view the available modules on the system use the command:

module avail

To view the currently loaded modules, use the command:

module list

2.3 Editing Files on Ela

On Ela you can use either the **emacs** or **vi** editors to modify source code or job submission scripts. To invoke the editors, use the following commands:

Invoking Emacs (with GUI)

```
emacs foo.f90
```

Invoking Emacs (without GUI)

```
emacs -nw foo.f90
```

Invoking Vi

```
vi foo.f90
```

2.4 Compiling Source Codes on Ela

To compile source codes on Ela, please use one of the available **compiler wrappers**.

The Fortran Compiler Wrapper

```
ftn -o foo foo.f90
```

The C and C++ Compiler Wrappers

```
cc -o foo foo.c
CC -o foo foo.cpp
```

COMPILERS TIP: For more information on using the compiler wrappers and adding compiler flags for **debugging** and **optimization**, please consult the respective man pages on Ela e.g.

man ftn
man cc
man CC

2.5 Submitting Jobs to Gele

Codes are executed by submitting a **job script** to a **batch submission queue** which in turn schedules the jobs to run on the compute nodes of Gele. To submit a job to Gele, please use the following submission script and submissions commands.

Sample Batch Submission Script

```
#! /bin/bash
#PBS -l mppwidth=8
#PBS -l walltime=00:10:00
#PBS -V

set -ex

cd $PBS_O_WORKDIR

aprun -n 8 ./foo.exe

exit
```

In this script we request 8 parallel tasks (**mppwidth**=8) for a maximum duration of 10 minutes (**walltime**=00:10:00). We ensure that our environment settings are exported to the compute nodes for execution (**-V**) and that our execution command is invoked within the same directory as that which the job script was submitted in.

> BATCH SCRIPT TIP: The $PBS_O_WORKDIR environment variable always contains the path to the submission directory for the batch script.

To invoke an instance of the executable on the compute nodes, you are required to use the **aprun** command. You can run multiple instances of the same executable across multiple cores by including the **-n** parameter. In the sample batch script above, we have requested that our **foo** executable be duplicated on all 8 cores requested within the batch script resources.

> IMPORTANT: Please ensure that your **aprun** command uses all the resources requested by the **mppwidth** parameter, otherwise valuable resources will be wasted.

You can modify most of the above settings if required. Please refer to the man pages (**man qsub** and **man aprun**) for further information.

To submit your job script to the batch queuing system use the following command:

Submit Job Script to Queue

```
qsub script
```

> NOTE: In this example the job script is named **script**

When you submit your script successfully, the batch queuing system will issue you with a unique **jobid** for your submitted job. You can view the queuing and running status of your job by using the following commands:

Viewing Jobs by Job ID

```
qstat jobid
```

Viewing Multiple Jobs by Username

```
qstat -u username
```

After termination of your job, you will receive stdout (standard output) and stderr (standard error) logs to your submission directory. These logs are suffixed with the id of your submitted job. Please review these logs carefully to determine whether you had successful or unsuccessful termination of your executable.

2.6 Practical Exercises

You are now in a position to attempt the "hands-on" practicals that complement the presentations. A number of practical exercises are included in this tutorial covering:

1. Compilation and Submission of a Simple Code
2. Experimentation with Compiler Optimization Flags and Timing Routines
3. Basic MPI
4. Basic OpenMP

2.6.1 Obtaining the Exercise Sources

Before attempting the practicals you should copy across the exercise source templates to your home directory. To do this complete the following instructions:

Step 1. cp ~tstitt/LinkSceem/Practicals.tgz ./
Step 2. tar -zxvf Practicals.tgz

TIP: Please feel free to attempt the practicals in any order depending on your preferences and level of experience.

NOTE: If you need any assistance with the exercises, please don't hesitate to contact the available support staff who will be glad to help with your problem.

Chapter 3

Practical 1 - Simple Compilation and Submission[1]

3.1 Introduction

In this practical you will be required to compile and submit a simple code for execution on **Gele**. If you require any assistance, please do not hesitate to contact the available support staff.

3.1.1 Objectives

The objectives of this practical are to gain experience in:

i. compiling a code within the Cray Programming Environment
ii. submitting a job to the Gele batch queuing system
iii. interrogating the execution log files

3.2 Compiling a Simple Code

Change to Exercise Directory

```
cd ../Practicals/Practical_1
```

TIP: Each practical will contain source code templates in either **Fortran90** or **C** flavours. Please feel free to choose your language of preference for completing the exercises.

3.2.1 Fortran Example

For this example, use the template code **simple.f90** provided in **../Practicals/Practical_1/Fortran90**
Briefly view the Fortran90 source code in your editor e.g.

Using Emacs

```
emacs simple.f90
```

[1] This content is available online at <http://cnx.org/content/m31992/1.8/>.

or

Using Vi

```
vi simple.f90
```

SOURCE CODE INFO: You will notice that this simple Fortran90 code sums two numbers.

Exercise 3.1 *(Solution on p. 9.)*
Compile the **simple.f90** code using the Cray Fortran compiler

3.2.2 C Example

For this example, use the template code **simple.c** provided in **../Practicals/Practical_1/C**
Briefly view the C source code in your editor e.g.

Using Emacs

```
emacs simple.c
```

or

Using Vi

```
vi simple.c
```

SOURCE CODE INFO: You will notice that the simple C code adds two numbers.

Exercise 3.2 *(Solution on p. 9.)*
Compile the **simple.c** code using the Cray C compiler

3.3 Submitting a Job

Exercise 3.3: Submit Batch Job *(Solution on p. 9.)*
Now submit the **simple** executable to the batch queue system for scheduling and execution.

3.4 Examining Job Logs

Exercise 3.4: Examining the Job Logs *(Solution on p. 9.)*
Once your job has completed execution on the compute nodes, use your editor to view the contents of the job logs.

Exercise 3.5: Modifying the Source
Modify the **simple** source code by changing the values of the two numbers to be summed. Recompile and submit your new job. Verify that the output is correct and no errors appear in the job log.

Solutions to Exercises in Chapter 3

Solution to Exercise 3.1 (p. 8)
ftn -o simple simple.f90
Solution to Exercise 3.2 (p. 8)
cc -o simple simple.c
Solution to Exercise 3.3 (p. 8)
Using your editor, create a job script (called **submit.script** for instance) as follows:

```
    #! /bin/bash
#PBS -l mppwidth=1
#PBS -l walltime=00:01:00
#PBS -V
#PBS -q linksceem

set -ex

cd $PBS_O_WORKDIR

aprun -n 1 ./simple

exit
```

and submit the script with:

```
    qsub submit.script
```

Solution to Exercise 3.4 (p. 8)
For a job with id #12345:

```
emacs submit.job.e12345
emacs submit.job.o12345
```

If all is well, you should have no errors in the error log (submit.job.e12345) and a print statement detailing the correct addition of the two numbers in the standard output log (submit.job.o12345).

Chapter 4

Practical 2 - Compiler Optimizations and Timing Routines[1]

4.1 Introduction

In this practical you will experiment with the optimization flags on the Cray compiler, to observe their effect on the runtime performance of a simple scientific kernel. Furthermore, you will be given the opportunity to perform some "hand-tuning" on the source code. You will also be introduced to methods for timing the runtime performance of your complete source code, or individual segments of it. If you require any assistance, please do not hesitate to contact the available support staff.

4.1.1 Objectives

The objectives of this practical are to gain experience in:

i. applying compiler optimization flags and observing their effect
ii. applying "hand-tuned" optimizations and observing their effect
iii. timing the runtime performance of complete codes and individual instruction blocks

4.2 Timing

Calculating the time your code requires to execute is beneficial for comparing runtime performance between various code modifications and/or the application of **compiler optimization** flags.

4.2.1 Timing Complete Program Execution

The elapsed **real time** (wallclock) of an executing program can be obtained at the command line using the **time** utility.

Example 4.1: Invoking The time Utility

```
> time app
real 0m4.314s
user 0m3.950s
sys 0m0.020s
```

The **time** utility returns 3 timing statistics:

[1] This content is available online at <http://cnx.org/content/m32159/1.4/>.

real	the elapsed real time between invocation and termination
user	the amount of CPU time used by the user's program
sys	the amount of CPU time used by the system in support of the user's program

Table 4.1: Statistics Returned By The 'time' Utility

NOTE: Typically the **real** time and **user+sys** time are the same. In some circumstances they may be unequal due to the effect of other running user programs and/or excessive disk usage.

Frequently it is useful to time specific regions of your code. This may be because you want to identify particular performance **hotspots** in your code, or you wish to time a specific **computational kernel**. Both C and Fortran90 provide routines for recording the execution time of code blocks within your source.

4.2.2 Timing Code Regions in Fortran90

Fortran Language Timers
The Fortran90 language provides two portable timing routines; **system_clock()** and **cpu_time()**.

Example 4.2: system_clock()
The **system_clock()** routine returns the number of seconds from 00:00 Coordinated Universal Time (CUT) on 1 JAN 1970. To get the elapsed time, you must call **system_clock()** twice, and subtract the starting time value from the ending time value.

IMPORTANT: To convert from the tick-based measurement to seconds, you need to divide by the clock rate used by the timer.

```
    integer :: t1, t2, rate

call system_clock(count=t1, count_rate=rate)

! ...SOME HEAVY COMPUTATION...

call system_clock(count=t2)

print *, "The elapsed time for the work is ",real(t2-t1)/real(rate)
```

Example 4.3: cpu_time()
The **cpu_time()** routine returns the processor time taken by the process from the start of the program. The time measured only accounts for the amount of time that the program is actually running, and not the time that a program is suspended or waiting.

```
    real :: t1, t2

call cpu_time(t1)

! ...SOME HEAVY COMPUTATION...

call cpu_time(2)

print *, "The elapsed time for the work is ",(t2-t1)
```

MPI Timing
To obtain the wallclock time for an individual MPI process, you can use the **mpi_wtime()** routine. This routine returns a double precision number of seconds, representing elapsed wall-clock time since an event in the past.

Example 4.4: mpi_wtime()

```
    DOUBLE PRECISION :: start, end

start = MPI_Wtime()

! ...SOME HEAVY COMPUTATION...

end   = MPI_Wtime()

print *,'That took ',(end-start),' seconds'
```

OpenMP Tip: In OpenMP codes you can can time individual threads with **omp_get_wtime()**.

4.2.3 Timing Code Regions in C

C Language Timers
The C language provides the portable timing routine **clock()**.

Example 4.5: clock()
Like the Fortran90 **system_clock()** routine, the C **clock()** routine is tick-based and returns the number of clock ticks elapsed since the program was launched.

IMPORTANT: To convert from the tick-based measurement to seconds, you need to divide the elapsed ticks by the macro constant expression **CLOCKS_PER_SEC**.

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

int main(void)
{
  clock_t t1,t2;
  double elapsed;

  t1=clock();

  // SOME HEAVY COMPUTATION

  t2=clock();
  elapsed=t2-t1;

  printf("The elapsed time for the work is %f",elapsed/CLOCKS_PER_SEC);
  return 0;
}
```

MPI Timing
Like Fortran90 codes, you can obtain the wallclock time for an individual MPI process, using the **MPI_Wtime()** routine. This routine returns a double precision number of seconds, representing elapsed wall-clock time since an event in the past.

Example 4.6: mpi_wtime()

```
    double t1, t2;

t1 = MPI_Wtime();

// SOME HEAVY CALCULATIONS

t2 = MPI_Wtime();

printf("MPI_Wtime measured an elapsed time of: %1.2f\n", t2-t1);
fflush(stdout);
```

OpenMP Tip: Also like Fortran90, C-based OpenMP codes can be timed with **omp_get_wtime()**.

4.3 The "Naïve" Matrix Multiplication Algorithm

Matrix multiplication is a basic building block in many scientific computations; and since it is an $O(n^3)$ algorithm, these codes often spend a lot of their time in matrix multiplication.

The most naïve code to perform matrix multiplication is short, sweet, simple and very very slow. The **naïve** matrix multiply algorithm is highlighted in Figure 1.

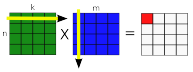

Figure 4.1: The "naïve" Matrix-Multiplication Algorithm

For each corresponding row and column, a **dot product** is formed as shown in Figure 2.

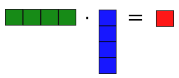

Figure 4.2: Matrix-Multiplication is composed of repeated dot-product operations

The **naïve** matrix-multiplication algorithm can be implemented as follows:

```
    for i = 1 to n

    for j = 1 to m

        for k = 1 to m

            C(i,j) = C(i,j) + A(i,k) * B(k,j)

        end for

    end for

end for
```

Naïve Matrix-Multiplication Implementation

In the following exercises you will use the naïve matrix-multiplication implementation to experiment with various compiler optimization options, as well as "hand-coded" tuning, to deliver the best performance on this simple scientific kernel.

4.4 Compiler Optimization Flags

Fortran90 Template
For this practical, use the template code **matmul.f90** provided in **../Practicals/Practical_2/Fortran90**
C Template
For this practical, use the template code **matmul.c** provided in **../Practicals/Practical_2/C**

Exercise 4.1: Compiler Flag Optimizations
Read the section on compiler optimization flags in the Cray compiler manpages i.e.
Fortran Compiler Manpages

```
man crayftn (line 678)
```

or
C Compiler Manpages

```
man craycc (line 509)
```

Listing Optimizations: If you want to know what compiler optimization options are applied at levels **-O0**, **-O1**, **-O2** and **-O3** then compile your code with the additional option **-eo** e.g. **ftn -O2 -eo -o foo foo.f90**

Exercise 4.2: Applying Optimization Flags *(Solution on p. 18.)*
Compile and execute a separate copy of the **naïve** matrix-multiplication implementation for each compiler optimization flag; -O0, -O1, -O2 and -O3. Record your observed timings in a table like the one shown in Table 2.

TIMING TIP: Use the **time** utility in your batch script to request the elapsed time for each calculation. The timings reported by the **time** utility will be displayed in the standard error logfile e.g. jobOutput.e12345

SUBMISSION TIP: Request a maximum of 10 minutes for your batch jobs

Optimization Flag	Wallclock Time (sec)
-O0	
-O1	
-O2	
-O3	

Table 4.2: Timings Table for Matrix-Multiply Kernel

Exercise 4.3: Timing The Matrix-Multiplication Kernel ***(Solutions on p. 18.)***

By using the **time** utility to record the timing statistics for the entire code, we are including the overhead time it takes to populate the matrices **A** and **B** with initial values. For large matrices, this overhead time could be quite significant and hence skew the recorded time for the matrix-multiply kernel calculation.

To ensure we are only recording the time for the matrix-multiplication kernel, we should wrap the matrix-multiply code block with source-level timing routines.

Using the language-level timing routines discussed earlier, record the time taken for the matrix-multiply kernel **only**. How do these times compare to the overall execution time?

TESTING TIP: Only record results for -O0 and -O2 compiler optimization flags

4.5 "Hand-Tuned" Optimizations

Sometimes it is possible to generate further performance by manually applying optimizations to your source code instructions. In the following exercises you will gain some experience in hand-coding simple optimizations into the naïve matrix-multiply implementation.

4.5.1 Fortran90 Programmers Only

The element order in which 2D arrays are traversed can have a significant performance impact between Fortran and C languages. In C, 2D arrays are stored in memory using **row-major** order. In Fortran, arrays are stored in memory using **column-major** order.

Exercise 4.4: Loop Re-ordering ***(Solutions on p. 18.)***

The naïve matrix-multiply Fortran90 implementation suffers in performance because its inner-most loops traverse array rows and not columns (this prevents the **cache** from being used efficiently).

Try to improve the performance of the Fortran90 implementation by maximizing column traversals. What performance gains do you achieve for **-O0** and **-O2** compiler flags? What order of indices I, J and K gives the best performance?

TIP: Modern compilers are very good at detecting sub-optimal array traversals and will try to reorder the loops automatically to maximize performance.

Exercise 4.5: Loop Unrolling (Advanced) *(Solutions on p. 19.)*

Manually unrolling loops can sometimes lead to performance gains by reducing the number of loop tests and code branching, at the expense of a larger code size. If the unrolled instructions are independent of each other, then they can also be executed in parallel.

TIP: Review loop unrolling by consulting the course slides here (Chapter 1).

Try to achieve performance improvement on the **original** naïve matrix-multiplication implementation by applying the loop unrolling technique. Compare your unrolled version against the results obtained with the **-O0** and **-O2** compiler flags.

What performance improvement do you get when you unroll **8** times?

Solutions to Exercises in Chapter 4

Follow-Up to Exercise 4.2 (p. 15)
Are the results exactly the same for each flag?
Fortran Solution A to Exercise 4.3 (p. 16)

```
    real :: t1,t2

...MATRIX INITIALIZATION...

call cpu_time(t1)

! Perform the matrix-multiplication

do I=1,N
   do J=1,N
      do K=1,N
         C(I,J)=C(I,J)+A(I,K)*B(K,J)
      end do
   end do
end do

call cpu_time(t2)

print *,"The time (in seconds) for the matrix-multiply kernel is ",t2-t1
```

C Solution B to Exercise 4.3 (p. 16)

```
    clock_t t1,t2;
double elapsed;

... MATRIX INITIALIZATION ...

t1=clock();

// Perform Matrix-Multiply Kernel

for( i = 0; i < n; i++ )
  for( j = 0; j < n; j++ )
    for( k = 0; k < n; k++ )
       c[i][j] = c[i][j] + a[i][k] * b[k][j];

t2=clock();
elapsed=t2-t1;

printf("The time (in seconds) for the matrix-multiply kernel is %f\n",elapsed/CLOCKS_PER_SEC);
```

Hint A to Exercise 4.4 (p. 16)
Try to nest deeper the loop over **I**.
Solution B to Exercise 4.4 (p. 16)

```
    ! Perform the matrix-multiplication

do K=1,N
   do J=1,N
      do I=1,N
         C(I,J)=C(I,J)+A(I,K)*B(K,J)
      end do
   end do
end do
```

Hint A to Exercise 4.5 (p. 17)

Step 1. Unroll the outer loop **I** 4 times
Step 2. Initialize 4 accumulating variables at the start of inner loop **J** e.g. C0, C1, C2 and C3
Step 3. Within the inner-most loop **K** do the following:

i. Create a temporary variable equal to **B(K,J)**
ii. Replace the matrix-multiply statement with 4 separate accumulators

Step 4. After the inner-most loop is completed, update **C** with the accumulated totals.

Fortran90 Solution B to Exercise 4.5 (p. 17)

```
    ! Perform the matrix-multiplication

do I=1,N,4
   do J=1,N
      C0=0.D0
      C1=0.D0
      C2=0.D0
      C3=0.D0
      do K=1,N
         TEMP=B(K,J)
         C0=C0+A(I,K)*TEMP
         C1=C1+A(I+1,K)*TEMP
         C2=C2+A(I+2,K)*TEMP
         C3=C3+A(I+3,K)*TEMP
      end do
      C(I,J)=C(I,J)+C0
      C(I+1,J)=C(I+1,J)+C1
      C(I+2,J)=C(I+2,J)+C2
      C(I+3,J)=C(I+3,J)+C3
   end do
end do
```

C Solution C to Exercise 4.5 (p. 17)

```
    // Perform Matrix-Multiply Kernel

for( i = 0; i < n; i=i+4 )
  {
    for( j = 0; j < n; j++ )
      {
```

```
c0=0;
c1=0;
c2=0;
c3=0;

for( k = 0; k < n; k++ )
  {

    temp=b[k][j];
            c0=c0+a[i][k]*temp;
    c1=c1+a[i+1][k]*temp;
    c2=c2+a[i+2][k]*temp;
    c3=c3+a[i+3][k]*temp;
  }

 c[i][j]=c[i][j]+c0;
 c[i+1][j]=c[i+1][j]+c1;
 c[i+2][j]=c[i+2][j]+c2;
 c[i+3][j]=c[i+3][j]+c3;

      }
  }
```

Chapter 5

Practical 3 - Basic MPI[1]

5.1 Introduction

In this practical you will develop and execute some simple codes, which incorporate the most fundamental MPI routines. Exercises include the querying of MPI ranks and size for a trivial MPI code, to the calculation of **speedup** for computing **pi** in parallel.

If you require any assistance, please do not hesitate to contact the available support staff.

> DISCLAIMER: This course (and accompanying practical exercises) are not designed to teach you MPI programming but are provided to give you a feel for the MPI programming paradigm. A detailed introduction to MPI will be given in further courses that are under development.

5.1.1 Objectives

The objectives of this practical are to gain experience in:

i. executing a MPI code with different process counts
ii. using basic MPI routines to query the **size** and **rank** of a MPI program
iii. calculating **speedup** for a simple MPI program

5.2 MPI: A Crash Course

MPI generally follows a **Single Program Multiple Data** (SPMD) programming model, whereby a single executable (called a **process**) is duplicated and executed on multiple processors. Each process (within the **MPI Communication World**) is uniquely identified by its **rank** and individual processes can communicate between each other using MPI **message-passing** routines.

> NOTE: MPI can also accommodate a Multiple Programming Multiple Data (MPMD) programming paradigm (where each process is a different executable) but this approach is less used (and supported) compared to the SPMD approach.

Figure 1 shows a typical MPI communication world (initially referred to as **MPI_COMM_WORLD**) with 4 processes (labeled **P**) each identified with a unique rank in the range 0-3. Each process executes the

[1] This content is available online at <http://cnx.org/content/m32191/1.5/>.

same task **T**. Processes communicate among other processes in the communication world by **sending** and **receiving** messages.

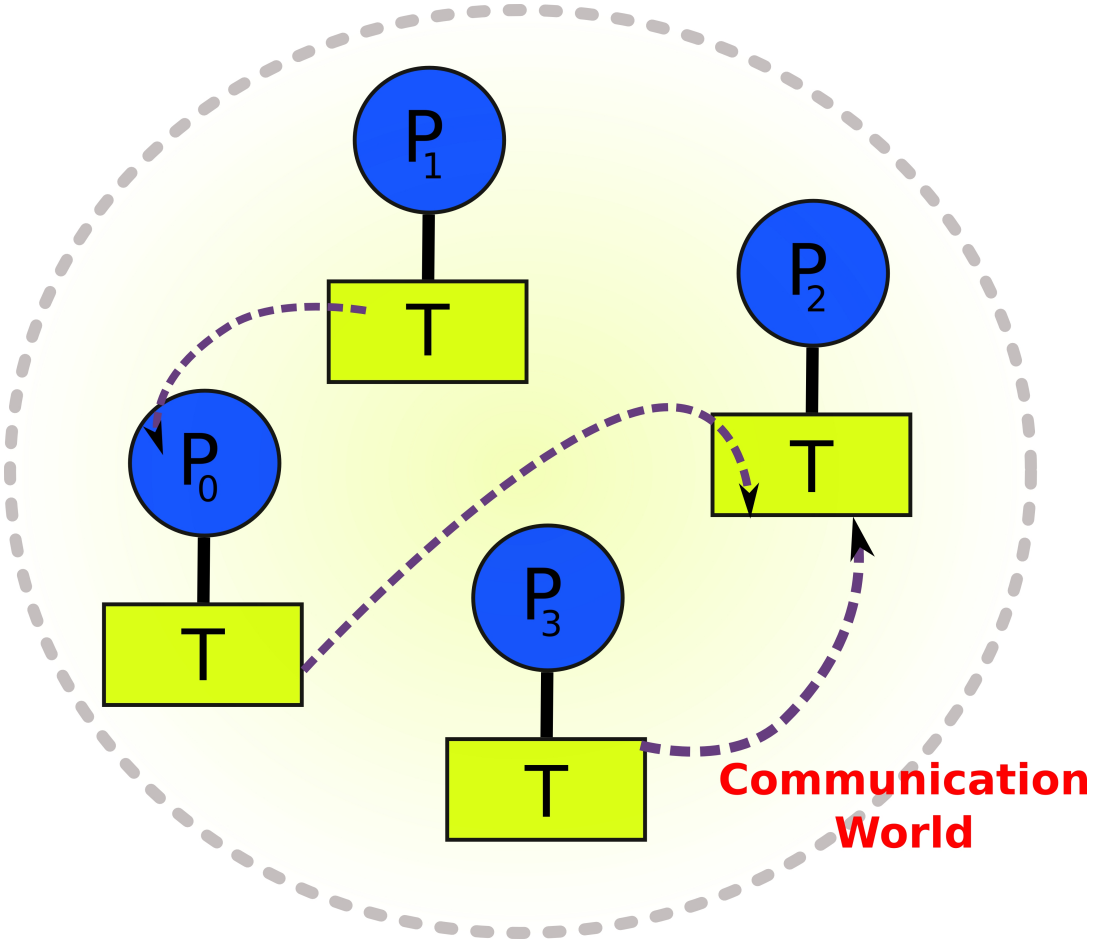

Figure 5.1: The Standard MPI Communication World

MPI INFO: Even though each process contains an instance of the same executable, different instruction streams can be executed by embedding rank dependent code into the executable e.g.

```
    if (myrank .eq. 0) then
  call do_this_work_on_process_0()
else if (myrank .eq. 1) then
  call do_this_work_on_process_1()
else
  call do_this_work_on_remaining_processes()
end if
```

The most fundamental MPI operations can be classified into 3 groups:

1. MPI Initialization and Finalization

   Initialization: MPI_Init()
   Finalization: MPI_Finalize()

2. Querying Process Rank and Communication World Size

   Query Process Rank: MPI_Comm_rank()
   Query World Size: MPI_Comm_size()

3. Sending and Receiving of Messages

   Send Message: MPI_Send()
   Receive Message: MPI_Receive()

NOTE: For more detailed information on MPI, download and review the latest MPI standard specification[2].

5.3 Basic MPI: "Hello World"

Exercise 5.1: Executing A Simple MPI Code
Develop, compile and execute a code (which uses the MPI framework) to display a "Hello World" message on multiple processes. Test your MPI code works correctly for 1, 2 4 and 8 cores.

CODE TIP: If you are not brave enough (yet) to write the code, you can use the MPI template provided in the **../Practicals/Practical_3/Exercise_1** folder.

BATCH SCRIPT TIP: Make sure you modify your submission script to request multiple processes and cores in your job e.g. to test your code for 8 processes modify your script to contain the following:

```
    ...
#PBS -l mppwidth=8        # Request 8 cores
...
...
aprun -n 8 ./hello_world  # Request 8 processes
```

Exercise 5.2: Querying MPI Rank and Size ***(Solutions on p. 25.)***
Modify your solution in Exercise 1 to print the "Hello World" message, along with the rank of the current process and the total number of processes in the MPI communication world.

5.4 Measuring Speedup

In parallel computing, **speedup** refers to how much a parallel algorithm is faster than a corresponding sequential algorithm.

Speedup is defined as:

$S_p = \frac{T_1}{T_p}$
where

- **p** is the number of processers

[2] See the file at <http://cnx.org/content/m32191/latest/mp21_spec.pdf>

- T_1 is the time of the sequential algorithm
- T_p is the time of the parallel algorithm with **p** processors

Exercise 5.3: Calculating PI
The value of **pi** can be computed approximately using an integration technique (see the course slides[3] for more information). Using the serial and parallel code implementations found in ../Practicals/Practical_3/Exercise_2, calculate the speedup obtained for 1, 2, 4 and 8 processors.

TIP: Briefly review the serial and parallel code implementations so you are familiar with the computational method and its MPI parallelisaton.

Record your results in a table similar to Table 1.

Processors	Time Taken(sec)	Speedup
1		
2		
4		
8		

Table 5.1: Speedup Results for PI Calculation

[3]See the file at <http://cnx.org/content/m32191/latest/HPC_Intro.pdf>

Solutions to Exercises in Chapter 5

Hint A to Exercise 5.2 (p. 23)
Use the **MPI_COMM_RANK()** and **MPI_COMM_SIZE()** routines to find the rank of the current process and the size of the communication world, respectively.
Fortran Solution B to Exercise 5.2 (p. 23)

```
...
! Initialize MPI
call MPI_INIT(error)

call MPI_COMM_RANK(MPI_COMM_WORLD,myrank,error)
call MPI_COMM_SIZE(MPI_COMM_WORLD,np,error)
print *, 'Hello World! I am process', myrank, ' of ', np

! Shut down MPI
call MPI_FINALIZE(error)
...
```

C Solution C to Exercise 5.2 (p. 23)

```
    ...
MPI_Init(&argc,&argv);

MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

printf("Hello World! I am process %d of %d\n", myrank, np);

MPI_Finalize();
...
```

Chapter 6

Practical 4 - Basic OpenMP[1]

6.1 Introduction

In this practical you will develop and execute some simple codes, which incorporate the most fundamental OpenMP directives. Exercises include the querying of OpenMP threads for a trivial OpenMP code, to the calculation of **speedup** for computing **pi** in parallel.

If you require any assistance, please do not hesitate to contact the available support staff.

> DISCLAIMER: This course (and accompanying practical exercises) are not designed to teach you OpenMP programming but are provided to give you a feel for the OpenMP programming paradigm. A detailed introduction to OpenMP will be given in further courses that are under development.

6.1.1 Objectives

The objectives of this practical are to gain experience in:

i. executing an OpenMP code with different thread counts
ii. using basic OpenMP directives to query the **threads** of an OpenMP program
iii. calculating **speedup** for a simple OpenMP program

6.2 OpenMP: A Crash Course

OpenMP is a portable **thread-based** programming model, whereby a single thread of execution (referred to as the **initial thread**) can spawn (or fork) additional threads to perform work in parallel. Ideally each thread of execution is mapped onto an individual processing core for maximum performance. Each thread (within a **thread team**) is uniquely identified by its **thread ID** and individual threads can communicate with each other through **shared memory** resources.

> OPENMP NOTE: OpenMP is not a programming language but rather is a specification that enables shared-memory parallelism, in base languages such as Fortran and C, through the following components:
>
> 1. Compiler Directives

[1] This content is available online at <http://cnx.org/content/m32284/1.2/>.

2.Runtime Library Routines
3.Environment Variables

Figure 1 shows a typical OpenMP thread-based execution whereby the initial thread creates a **parallel region** to perform some work in parallel. Within the parallel region a team of **n+1** threads are forked (spawned) and assigned individual tasks to complete. After the final task is complete, the threads are synchronised (joined) and serial execution continues with the initial thread.

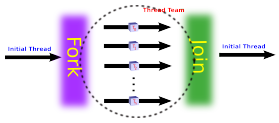

Figure 6.1: The OpenMP Programming Model

The most fundamental OpenMP operations can be classified into three (3) groups:

1. Invoking a Parallel Region **Fortran Example**

```
!$omp parallel private(var1,var2,...),shared(var1,var2,...)
... code block ...
!$omp end parallel
```

C Example

```
#pragma omp parallel private(var1,var2,...),shared(var1,var2,...)
{
... code block ...
}
```

PRIVATE AND SHARED VARIABLES: Variables that are in scope at the invocation of a parallel region must be declared either **private** or **shared** within the ensuing parallel region. If a variable is designated as **private**, then all threads within the parallel region will create an individual instance of the variable, and can modify it without any conflict from other threads. A variable that is designated as **shared** remains shared among all threads, and needs to be accessed carefully to avoid race conditions.

2. Querying Thread ID and Team Size (within the current parallel region)**Fortran Examples**

```
integer :: id, threads

id=OMP_GET_THREAD_NUM() ! returns ID of this thread
threads=OMP_GET_NUM_THREADS() ! returns the number of threads in parallel region
```

C Examples

```
int id, threads;

id=OMP_GET_THREAD_NUM(); // returns ID of this thread
threads=OMP_GET_NUM_THREADS(); // returns the number of threads in parallel region
```

3. Loop-level Parallelism (splitting loop iterations over threads)**Fortran Example**

```
!$omp parallel do private(var1,...),shared(var1,...)
... do loop ...
!$omp end parallel do
```

C Example

```
#pragma omp parallel for private(var1,...),shared(var1,...)
... for loop ...
```

NOTE: For more detailed information on the OpenMP standard, download and review the latest OpenMP specification[2].

6.3 Basic OpenMP: "Hello World"

Exercise 6.1: Executing A Simple OpenMP Code
Develop, compile and execute a code (which uses the OpenMP framework) to display a "Hello World" message on multiple threads. Test your OpenMP code works correctly for 1 and 2 threads.

CODE TIP: If you are not brave enough (yet) to write the code, you can use the OpenMP template provided in the **../Practicals/Practical_4/Exercise_1** folder.

BATCH SCRIPT TIP: Make sure you modify your submission script to request multiple threads in your job e.g. to test your code for 2 threads modify your script to contain the following:

```
    ...
#PBS -l mppwidth=1        # Request 1 process
#PBS -l mppnppn=1         # Request 1 process per node
#PBS -l mppdepth=2        # Request 2 threads per process
...
...
export OMP_NUM_THREADS=2
aprun -n 1 -N 1 -d 2 ./hello_world  # Execute code with 1 process and 2 threads
```

OPENMP TIP: For each execution, you need to set the **OMP_NUM_THREADS** environment variable to the default number of threads for each parallel region in your code.

APRUN TIP: To notify aprun that you require multiple threads per process, you need to set the **-d** option e.g. to request 4 threads per process you should use

```
aprun -n 1 -N 1 -d 4 ./foo
```

For more information see **man aprun**

Exercise 6.2: Querying Thread ID and Team Size ***(Solutions on p. 31.)***
Modify your solution in Exercise 1 to print the "Hello World" message, along with the ID of the current thread and the total number of threads in the current thread team.

[2]See the file at <http://cnx.org/content/m32284/latest/openmp3_spec.pdf>

6.4 Measuring Speedup

In parallel computing, **speedup** refers to how much a parallel algorithm is faster than a corresponding sequential algorithm.

Speedup is defined as:

$S_p = \frac{T_1}{T_p}$
where

- **p** is the number of processers
- T_1 is the time of the sequential algorithm
- T_p is the time of the parallel algorithm with **p** processors

Exercise 6.3: Calculating PI
The value of **pi** can be computed approximately using an integration technique (see the course slides[3] for more information). Using the serial and OpenMP code implementations found in ../Practicals/Practical_4/Exercise_2, calculate the speedup obtained for 1 and 2 threads.

TIP: Briefly review the serial and parallel code implementations so you are familiar with the computational method and its OpenMP parallelisaton.

Record your results in a table similar to Table 1.

Threads	Time Taken(sec)	Speedup
1		
2		

Table 6.1: Speedup Results for PI Calculation

[3]See the file at <http://cnx.org/content/m32284/latest/HPC_Intro.pdf>

Solutions to Exercises in Chapter 6

Hint A to Exercise 6.2 (p. 29)
Use the **OMP_GET_THREAD_NUM()** and **OMP_GET_NUM_THREADS()** runtime routines to find the ID of the current thread and the size of the thread team, respectively.
Fortran Solution B to Exercise 6.2 (p. 29)

```
    ...
integer :: id, threads

!$OMP PARALLEL private(id,threads)

id=OMP_GET_THREAD_NUM()
threads=OMP_GET_NUM_THREADS()

print *,"Hello World from thread ",id," of ",threads

!$OMP END PARALLEL
...
```

C Solution C to Exercise 6.2 (p. 29)

```
    ...
int id,threads;

#pragma omp parallel private(id,threads)
{
  id=omp_get_thread_num();
  threads=omp_get_num_threads();

  printf("Hello World from thread %d of %d\n",id,threads);
}
...
```

Glossary

B Batch Queue

A job queue maintained by the scheduler software for determining the order in which jobs are executed based on user priority, estimated execution time and available resources.

C Cache

Area of high-speed memory that contains recently referenced memory addresses.

Column-Major Ordering

Array elements are stored in memory as contiguous columns in the matrix. For best performance (and optimal cache use) elements should be traversed in column order.

This is an unsupported media type. To view, please see http://cnx.org/content/m32159/latest/

Figure 4.3: Column-Major Ordering

Compiler Optimizations

Transformations to the source code which are applied by the compiler to improve the runtime performance of the executing code e.g. loop unrolling, instruction reordering, in-lining etc.

Compiler Wrapper

A compiler wrapper is a script that invokes the low-level compiler command with the required link libraries for a specific computing system and message-passing framework. Typically standard scientific and vendor-specific libraries are also included in the wrapper script.

D Debugging

The art of identifying and removing compile-time and runtime errors from a code.

Dot Product

Also known as the scalar product, it is an operation which takes two vectors over the real numbers R and returns a real-valued scalar quantity. It is the standard inner product of the orthonormal Euclidean space.

E Ela

Ela is a 16 core front-end node which is used to edit and compile source code and submit jobs for execution on the compute nodes of Monte Rosa.

Emacs

Emacs is a feature-rich editor that has been in development since the mid-70s.

Please take a look at the emacs tutorial[4] for more information on using the editor.

G Gele

Gele is a 76 node dual-core system. It is the test system for the CSCS flagship machine Monte Rosa.

H Hotspot

A block of source code instructions that account for a significant amount of the CPU execution time.

J Job Script

A script which describes the command line execution of an executable along with the core, memory and time resources required by the calculation.

K Kernel

[4] http://www.gnu.org/software/emacs/tour/

A block of source code instructions that represent a particular algorithm or calculation.

M

Message Passing

A communication protocol whereby processes communicate by sending and receiving data through messages.

Module

A script the configures the search paths and link commands for a given software package or library.

Monte Rosa

Monte Rosa is a Cray XT5 supercomputer with a peak performance of 140 TeraFlop/s. Its current specification is:

- 3688 AMD quad core Opteron @ 2.4 GHz
- 29.5 TB DDR2 RAM
- 290 TB Disk
- 9.6 GB/s interconnect bandwidth

This is an unsupported media type. To view, please see http://cnx.org/content/m31991/latest/

Figure 2.1: Monte Rosa

MPI Communication World

A collection of MPI processes which can communicate through passing messages. The default communication world for a MPI process is **MPI_COMM_WORLD**

MPI Rank

Each process in a MPI communication world of **P** processes is assigned an unique id in the range 0..(P-1). This id is called the **rank** of the process.

mppwidth

The total number of cores requested by the user for the completion of the job.

O

omp_get_wtime()

```
use omp_lib

DOUBLE PRECISION START, END

START = omp_get_wtime()

! ...HEAVY COMPUTATION...

END = omp_get_wtime()

PRINT *, 'That took ', &
   (END - START), ' seconds.'
```

Optimization

The art of improving the memory performance and runtime performance of an executing code.

P

Parallel Region

Within the OpenMP specification, a **parallel region** encapsulates a block of instructions which will be executed by a team of threads.

Programming Environment

The Programming Environment is the collection of compiler and pre-built libraries that are specific to an individual compiler suite.

R

Real Time

The elapsed time between the invocation of a program and its termination.

Row-Major Ordering

Array elements are stored in memory as contiguous rows in the matrix. For best performance (and optimal cache use) elements should be traversed in row order.

This is an unsupported media type. To view, please see http://cnx.org/content/m32159/latest/

Figure 4.4: Row-Major Ordering

S

Speedup

In parallel computing, **speedup** refers to how much a parallel algorithm is faster than a corresponding sequential algorithm.

This is an unsupported media type. To view, please see http://cnx.org/content/m32191/latest/

Figure 5.2: Speedup Diagram

SPMD

Single Program Multiple Data: multiple autonomous processors simultaneously execute the same program at independent points.

Further information: http://en.wikipedia.org/wiki/SPMD[5]

Swiss National Supercomputing Centre (CSCS)

Founded in 1991, CSCS, the Swiss National Supercomputing Centre, develops and promotes technical and scientific services for the Swiss research community in the fields of high-performance computing.

Located at Manno near Lugano, in the southern, Italian-speaking part of Switzerland, CSCS is an autonomous unit of the Swiss Federal Institute of Technology in Zurich (ETH Zurich[6]).

Further details can be found at www.cscs.ch[7]

T

Thread ID

Each thread within a team of **T** threads is uniquely identified by an ID in the range **0..(T-1)**

Thread Team

A group of threads (including the initial thread that spawned the team) which can work in parallel on some task.

Thread-Based

A programming model whereby work is carried out in parallel using concurrent threads of execution.

V

Vi

Vi is a family of screen-oriented text editors which has been developed since 1976.

Please take a look at the Vi tutorial[8] for more information on using the editor.

W

walltime

The total real time for completing a task. In a batch script, it refers to the maximum wall-clock time required by the job.

[5] http://en.wikipedia.org/wiki/SPMD
[6] http://www.ethz.ch/index_EN
[7] http://www.cscs.ch
[8] http://www.unix-manuals.com/tutorials/vi/vi-in-10-1.html

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). Keywords do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

Attributions

Collection: *An Introduction to High-Performance Computing (HPC)*
Edited by: Tim Stitt Ph.D.
URL: http://cnx.org/content/col11091/1.7/
License: http://creativecommons.org/licenses/by/3.0/

Module: "Introduction to HPC (slideshow)"
By: Tim Stitt Ph.D.
URL: http://cnx.org/content/m31999/1.7/
Page: 1
Copyright: Tim Stitt Ph.D.
License: http://creativecommons.org/licenses/by/3.0/

Module: "Editing, Compiling and Submitting Jobs on Ela"
By: Tim Stitt Ph.D.
URL: http://cnx.org/content/m31991/1.8/
Pages: 3-6
Copyright: Tim Stitt Ph.D.
License: http://creativecommons.org/licenses/by/3.0/

Module: "Practical 1 - Simple Compilation and Submission"
By: Tim Stitt Ph.D.
URL: http://cnx.org/content/m31992/1.8/
Pages: 7-9
Copyright: Tim Stitt Ph.D.
License: http://creativecommons.org/licenses/by/3.0/

Module: "Practical 2 - Compiler Optimizations and Timing Routines"
By: Tim Stitt Ph.D.
URL: http://cnx.org/content/m32159/1.4/
Pages: 11-20
Copyright: Tim Stitt Ph.D.
License: http://creativecommons.org/licenses/by/3.0/

Module: "Practical 3 - Basic MPI"
By: Tim Stitt Ph.D.
URL: http://cnx.org/content/m32191/1.5/
Pages: 21-25
Copyright: Tim Stitt Ph.D.
License: http://creativecommons.org/licenses/by/3.0/

Module: "Practical 4 - Basic OpenMP"
By: Tim Stitt Ph.D.
URL: http://cnx.org/content/m32284/1.2/
Pages: 27-31
Copyright: Tim Stitt Ph.D.
License: http://creativecommons.org/licenses/by/3.0/
Based on: Practical 3 - Basic MPI
By: Tim Stitt Ph.D.
URL: http://cnx.org/content/m32191/1.4/

An Introduction to High-Performance Computing (HPC)
This course introduces students to the fundamentals of High-Performance Computing (HPC) through a slide-show and a series of practical exercises. The course was developed by the Swiss National Supercomputing Centre (CSCS), Switzerland as part of the LinkSCEEM Conference, 6th-8th October, Cyprus, 2009.

About Connexions
Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.