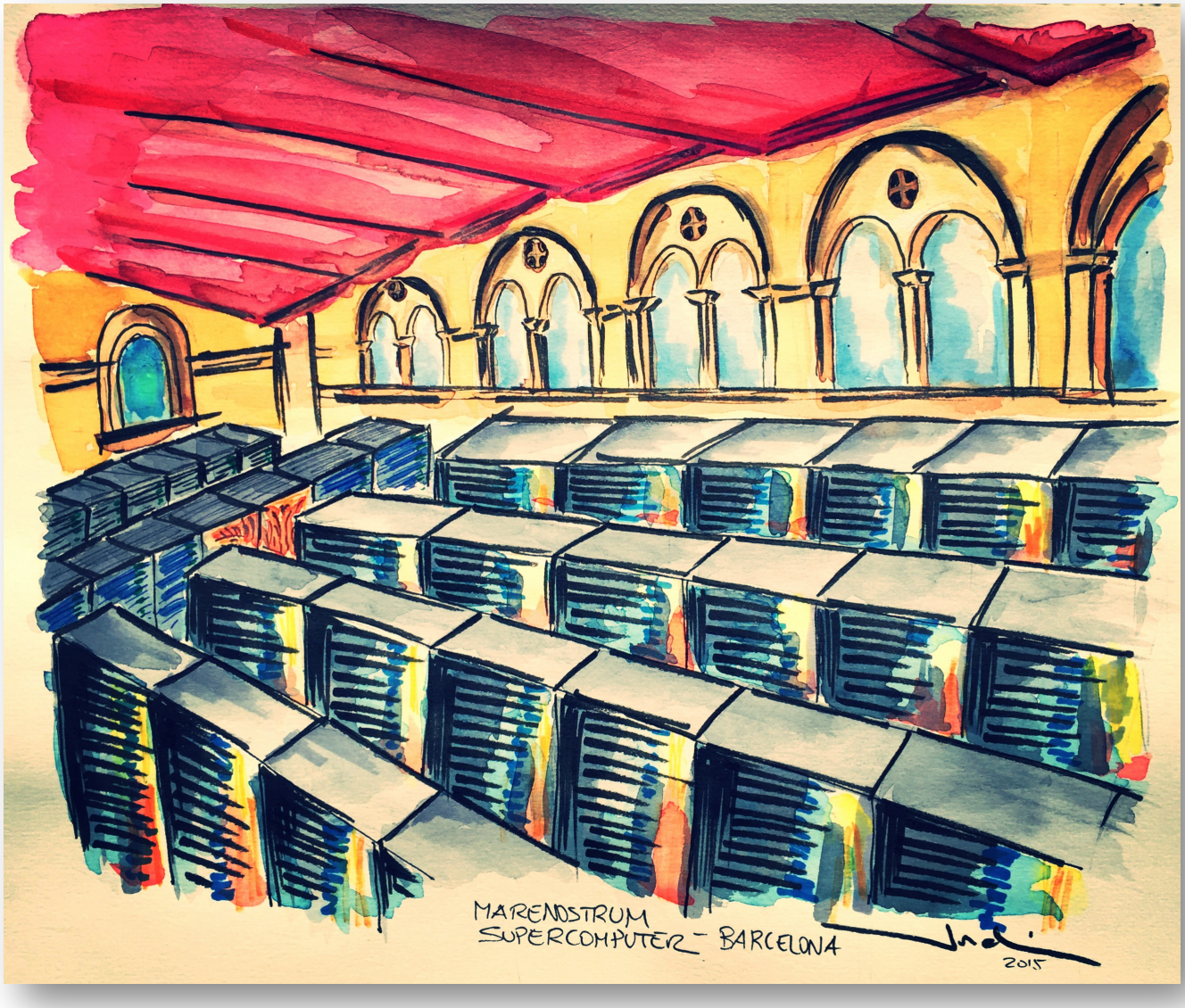
Hands-on 2:
Getting Started with Distributed Memory Parallelism & Batch Processing Support



SUPERCOMPUTERS ARCHITECTURE

MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

Barcelona, Fall 2015

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Hands-on 2

Getting Started with Distributed Memory Parallelism & Batch Processing Support

SUPERCOMPUTERS ARCHITECTURE

Master in innovation and research in informatics

(Specialization High Performance Computing)

UPC Barcelona Tech & Barcelona Supercomputing Center

Version 2.1 - 30 September 2015

Professor: Jordi Torres jordi.torres@bsc.es www.JordiTorres.eu

Table of Contents

1 Hands-on description

This Hands-on will get you started with MPI in a very straightforward and easy way. First we will present the MPI compilers and later the LSF options required to appropriately request resources from computing nodes according to the needs of your job. In this Lab session we will use the Marenostrum III supercomputer from Barcelona Supercomputing Center – Centro Nacional de Supercomputation as a platform.

2 Background required to follow these hands-on

These hands-on are designed to accompany the student in his introduction to the real world of supercomputing. Required previous knowledge could be summed up in parallel programming with OpenMP and MPI (to be introduced in the theoretical part of the course). All other skills will be presented in a self-contained hands-on.

However these hands-on are also designed for a student that can progress on their own in this learning. Prior knowledge required can be obtained from SA-MIRI theory classes or through books as *An Introduction to Parallel Programming by Peter S. Pacheco*, among others.

3 Distributed Memory Parallelism

As we show in theory class, the world of parallel multiple instruction, multiple data, or MIMD, computers is divided into distributed-memory and shared-memory systems.

From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core. In this hands-on we are going to start practice at how to program distributed-memory systems using message-passing.

3.1 MPI?

In message-passing programs two processes can communicate by calling functions: one process calls a send function and the other calls a receive function. The implementation of message-passing that we will use in this hands-on is called MPI , which is an abbreviation of Message-Passing Interface .

MPI is not a new programming language. It defines a library of functions that can be called from C, C++ , and Fortran programs. In this hands-on we will learn about some of MPI's different send and receive functions. We will leave the communication functions that can involve more than two processes (collective communications) for the next hands-on.

3.2 Getting Started

To compile MPI programs it is recommended to use the following handy wrappers: mpicc, mpicxx for C and C++ source code. You could choose the Parallel environment first: module load openmpi /module load impi /module load poe. By default the loaded module is openmpi.
These wrappers will include all the necessary libraries to build MPI applications without having to specify all the details by hand.

```
% mpicc a.c -o a.exe
% mpicxx a.C -o a.exe
```

Exercise 1: Create a "Hello World" program in MPI and Compile it.

Note: Because all lab groups will launch jobs at the same time in the same login server we could have performance problems. We suggest using a command like the following one in order to open an interactive session:

```
% bsub -Is -n 4 -q interactive -W 00:15 /bin/bash
```

This command opens an interactive terminal for 15 minutes with 4 processors available.

You can use a as skeleton the following code to create your program `mpi-hello-world.c`:

```
#include "mpi.h"
#include <stdio.h>
int main (int argc, char **argv)
{

MPI_Init (&argc, &argv);
MPI_Comm_rank (                                    );
MPI_Comm_size (                                    ); /* How many are we? */
printf ( "I am %d of %d\n", rank, size );
MPI_Finalize();
return 0;
}
```

Now you are ready to compile the program:

```
% mpicc -o hello-world mpi-hello-world.c
```

For this test purpose only you can run the command (nteractive)

```
%  mpirun -np 4 ./hello-world >hello.out
```

Exercise 2: Run the “Hello World” program. How many lines contain the file hello.out? Why?

Now you area ready to submit mpi jobs at Marenostrum supercomputer.

Warning: If you are connecting using ssh from a MAC environment remember to disable the option “Set locale environment variables on startup” from menu Terminal->Preferences->Settings

4 LSF

4.1 Overview

LSF is the utility used at MareNostrum III for batch processing support, so all jobs must be run through it.

The Platform LSF ("LSF", short for load sharing facility) software is leading enterprise-class software that distributes work across existing heterogeneous IT resources creating a shared, scalable, and fault-tolerant infrastructure, delivering faster, more reliable workload performance while reducing cost. LSF balances load and allocates resources, while providing access to those resources.

LSF provides a resource management framework that takes your job requirements, finds the best resources to run the job, and monitors its progress. Jobs always run according to host load and site policies. Detailed information can be found at http://www.bsc.es/support/LSF/9.1.2

4.2 Basic Commands

These are the basic commands to submit, control and check your jobs:

```
bsub < job_script
```

submits a “job script” passed through standard input (STDIN) to the queue system. Job directives explains the available options to write a jobscript

```
bjobs [-w][-X][-l job_id]
```

shows all the submitted jobs.

```
bkill <job_id>
```

remove the job from the queue system, cancelling the execution of the processes, if they were still running.

Detailed information can be found at http://www.bsc.es/support/LSF/9.1.2

4.3 BSC Commands

BSC Commands are a suite of commands useful to the users of Barcelona Supercomputing Center's (BSC) HPC facilities. The commands are developed and maintained by the Support Team at BSC. Two useful command for our labs are:

```
bsc_jobs
```

This command shows all the pending or running jobs from your group.

```
bsc_jobcheck
```

This command simulates the launch of a jobscript to the queue system and reports the amount of resources the jobscript requested. This is especially useful when tweaking jobscripts for efficiency and debugging the resources requested by a certain job.

```
bsc_queues
```

This command shows all the queues available in your group.

Detailed information can be found with *man bsc.*

5 Job directives

A job must contain a series of directives to inform the batch system about the characteristics of the job. We recommend that you read the bsub command's manual from any of MareNostrum's terminals:

```
% man bsub
```

5.1 Most common directives

Here we provide a short summary of most common directives:

```
#BSUB -J job_name
```

Specify the name (description) of the job.

```
#BSUB -q debug
```

Specify the queue for the job to be submitted. The *debug* queue is only intended for small tests, so there is a limit of 1 job per user, using up to 64 cpus (4 nodes), and one hour of wall clock limit. The queue might be reassigned by LSF internal policy, as with the *sequential* queue.

```
#BSUB -W HH:MM
```

Specify how much time the job will be allowed to run. This is a mandatory field. NOTE: take into account that you can not specify the amount of seconds in LSF. You must set it to a value greater than the real execution time for your application and smaller than the time limits granted to the user. Notice that your job will be killed after the elapsed period.

```
#BSUB -cwd pathname
```

The working directory of your job (i.e. where the job will run). If not specified, it is the current working directory at the time the job was submitted.

```
#BSUB -e/-eo file
```

The name of the file to collect the stderr output of the job. You can use %J for job_id. -e option will APPEND the file, -eo will REPLACE the file.

```
#BSUB -o/-oo file
```

The name of the file to collect the standard output (stdout) of the job. -o option will APPEND the file, -oo will REPLACE the file.

```
#BSUB -n number
```

The number of processes to start.

```
#BSUB -x
```

Use the nodes exclusively. This is the default behaviour except for sequential executions.

```
#BSUB -R "span[ptile=number]"
```

The number of processes assigned to a node. Note that full nodes will be allocated except for sequential executions.

Examples:

```
# if you want to use 4 processes per node and 4 threads:
#BSUB -R "span[ptile=4]"
```

```
# if your program has high memory
# consumption you can reduce the number
# of processes per node
#BSUB -R "span[ptile=14]"
```

5.2 SA-MIRI students

For students enrolled at SA-MIRI course they will have a reservation of 10 nodes during lab sessions with the name `training`. In order to use this reservation you can use

```
% bsub -q training <job.lsf
```

or use the parameter in the LSF file

```
#BSUB -q training
```

If you need to execute jobs outside of class time, according your user other queues are available as . `debug`. Execute the following command to know the name of available queues:

```
% bsc_queues
```

Note: you can use the queues training or debug.

5.3 Basic Exercises

Exercise 3: submit the previous mpi *hello world* program. Create the LSF file according the following suggestions:

- The name of the job is hello_world
- Start 16 processes
- Max time to be executed 15 min.
- The name to collect the standard output is “output_%J.out”
- The name to collect the standard error is “output_%J.err”
- Assign 16 processes to a node
- Use the node exclusively
-

Exercise 4: Do exactly the same as previous exercise, but now with 4 processes per node. Compare the two executions (exercise 3 vs 4).

6 MPI point-to-point communication directives

6.1 Basic functions

• MPI can be very simple (or not!). In this hands.-on we are covering the six functions that enable you to write many programs: initiate and terminate a computation, identify processes, and send and receive messages:

- MPI_INIT : Initiate an MPI computation.
- MPI_FINALIZE : Terminate a computation.
- MPI_COMM_SIZE : Determine number of processes.
- MPI_COMM_RANK : Determine my process identifier.
- MPI_SEND : Send a message.
- MPI_RECV : Receive a message.

6.2 Basic example: trapezoidal rule

Remember that in the theory lab we presented a step by step MPI program to implement a parallel version of the trapezoidal rule.

Exercise 5.a: Write a mpi_trap.c program based on the code given in the theory class that estimates the integral from a to b of f(x) in n intervals. Compile and run it.

Note 1: f(x) is all hardwired, e.g.

```
double f(double x) {
   double return_val;

   return_val = x*x;
   return return_val;
}
```

(you can change it for a more interesting function)

Note 2: The endpoints of the interval and the number of trapezoids are hardwired.

Note 3: you can use “mpirun –np" to run it.

Exercise 5.b (OPTIONAL): The same mpi_trap.c program that accept the endpoints of the interval of integration and the number of trapezoids as an input.

Exercise 5.c: Create a lsf jobscript and submit it with 8 processors. Include the lsf jobscript in the answer.

7 Taking Timing

You may have been wondering how to measure time. There are a lot of different approaches, and with parallel programs the details may depend on the API.

7.1 Marenostrum III approach

We can use this approach at MN III that may make things a little easier. See the following example.

```
/* timesample.c */
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#define SIZE 1000
typedef double matrix[SIZE][SIZE];
matrix m1;
struct timeval start_time, end_time;

static void foo (void) {
  int i,j;
  for (i = 0;i < SIZE; ++i)
    for (j = 0; j < SIZE; ++j)
      m1[j][i] = 1.0;
}
main () {
  int i,j;

gettimeofday(&start_time, NULL);

    for (i = 0; i < 10; ++i) foo();

   gettimeofday(&end_time, NULL);
   print_times();

}

print_times()
{
   int total_usecs;
   float total_time, total_flops;
   total_usecs = (end_time.tv_sec-start_time.tv_sec) * 1000000 +
        (end_time.tv_usec-start_time.tv_usec);
   printf(" %.2f mSec \n", ((float) total_usecs) / 1000.0);
   total_time = ((float) total_usecs) / 1000000.0;
}
```

7.2 Sequential version

Exercise 6: Take the time for executing the sequential version of exercise 10 of previous Hands-on 1 for different N values (matrix multiplication). Compare the results. In the answer also include the LSF file used.

1024	2048	4096	8192	16384	32.768

7.3 Parallel version

Exercise 7: Answer exercises 3 and 4 again taking the time.

Exercise 8: Take the time for executing the program version of exercise 5 for different input values and with different number of processors (describe the results).

8 Lab Report

You have one week to deliver a report with the answers to the exercises.

Acknowledgement: Part of this hands-on is based on "Marenostrum III User's Guide". Special thanks to David Vicente and Miguel Bernabeu from BSC Operations department for his invaluable help preparing this hands-on.