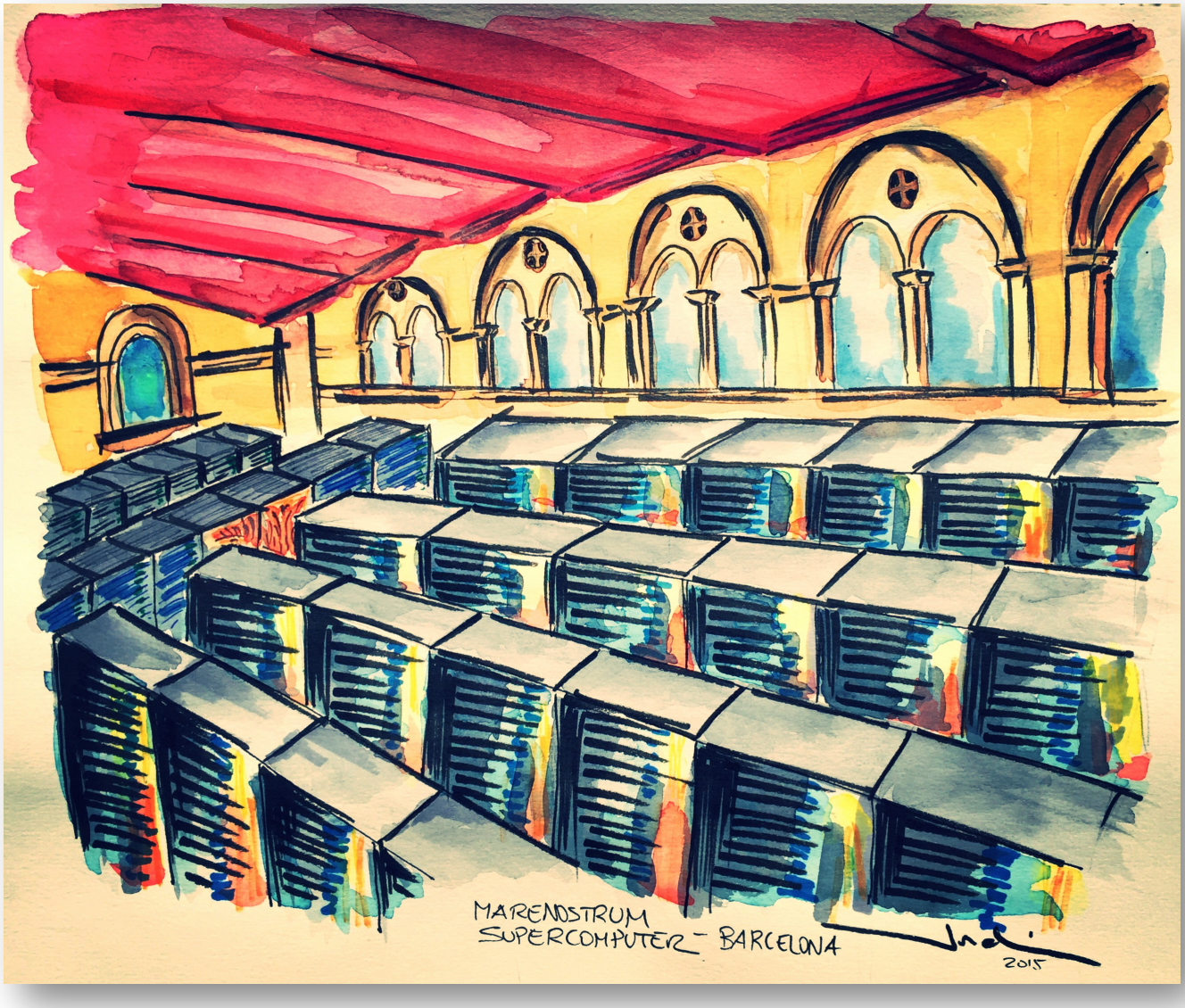
Hands-on 3:

Getting Started with OpenMP Parallelism & Advanced MPI Parallelism



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 3

Getting Started with OpenMP Parallelism & Advanced MPI Parallelism

SUPERCOMPUTERS ARCHITECTURE

Master in innovation and research in informatics

(Specialization High Performance Computing)

UPC Barcelona Tech & Barcelona Supercomputing Center

Version 2.0 - 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 OpenMP in a very straightforward and easy way. First we will present the OpenMP compilers and the LSF options required to appropriately request resources from computing nodes according to the needs of your job. In the second part of the hands-on we will explore some Advanced MPI features.

2 OpenMP basics

2.1 Why OpenMP?

OpenMP is the most used programming model for shared memory in High Performance Computing. OpenMP is an API for shared-memory parallel programming. The “MP” in OpenMP stands for “multiprocessing,” a term that in general is synonymous with shared-memory parallel computing. OpenMP is designed for systems in which each process/thread has access to all available memory.

As we discussed in the theory class there are mainly two standard APIs for shared memory programming: Pthreads are lower level and provide us with the power to program thread behavior. This power, however, comes with some associated cost in terms of programmability. OpenMP, on the other hand, force the compiler and run-time system to determine some of the details of thread behavior, so it can be simpler to code some parallel behaviors using OpenMP. However, in this case, some low-level thread interactions can be more difficult to program.

OpenMP was developed by a group of programmers and computer scientists who believed that writing large-scale high-performance programs using APIs such as Pthreads was too difficult, and they defined the OpenMP specification so that shared-memory programs could be developed at a higher level.

In fact, OpenMP was explicitly designed to allow programmers to incrementally parallelize existing serial programs; this is virtually impossible with MPI and fairly difficult with Pthreads.

BSC has more than 15 years’ experience in proposing and implementing parallel programming models. Its researchers have been involved in OpenMP since the beginning, participating in the definition of the tasking model. With the inclusion of task dependences, BSC has recently become an auxiliary member of the OpenMP consortium.

2.2 Compilers

OpenMP directives are fully supported by the Intel C and C++ compilers. To use it, the flag -openmp must be added to the compile line.

```
% icc -openmp -o exename filename.c
% icpc -openmp -o exename filename.C
```

It allows also mix MPI + OPENMP code using -openmp with the mpi wrappers (out of scope of the lab).

With gcc you can use the flag:

```
% gcc -fopenmp -o exename filename.c
```

Exercise 1: Compile and run the following sequential "Hello World" program. What is the output?

```
#include <stdio.h>
int main()
{
int ID = 0;
printf("hello(%d)" , ID);
printf("world(%d) \n", ID);
}
```

2.3 Getting started

OpenMP provides what's known as a "directives-based" shared-memory API. In C and C++ , this means that there are special preprocessor instructions known as pragmas.

Exercise 2: Parallelize (compile and run) the "hello world" sequential code adding the most basic parallel directive. How many threads are created? Why?

Note: remember to include the prototypes and types from the file <omp.h>

Exercise 3: Write a multithreaded version of the same program where each thread prints its thread_num as a ID. What happened? Why?

3 Programming with OpenMP

Remember from the theory class the trapezoidal rule example for estimating the area under a curve, we will use this example in order to practice the OpenMP basics.

3.1 Basic OpenMP program

Exercise 4

Create a OpenMP program to estimate definite integral (or area under curve) using trapezoidal rule. The number of theads is a parameter. The input is a, b, n (estimate of integral from a to b of f(x) using n trapezoids). You can assume that the function f(x) is hardwired (*).

Use a critical directive for global sum (each thread explicitly computes the integral over its assigned subinterval). You can assume that n is evenly divisible by the number of threads.

You can use (copy & paste) the sequential code in (*) as a template.

(*) Some help:

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

double f(double x);    /* Function we're integrating */
void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);

   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}
```

```
double f(double x) {
   double return_val;
return_val = x*x;
   return return_val;
}
```

```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;


   *global_result_p += my_result;
}
```

3.2 OpenMP batch execution

Exercise 5: Create and submit a LSF jobscript with the execution of OpenMP version with 16 processors. Add the LSF jobscript in the answer.

4 Comparing programming models

In the previous hands-on we presented how to time executions at Marenostrum. Now we will start to use it to compare the trapezoidal rule problem programmed in different programming models.

4.1 OpenMP parallelism

Exercise 6:

Create a lsf jobscript with the execution of OpenMP version with 2,4,8 and 16 threads/processors and obtain the execution time of each of them. Compare the execution time of all the executions with some value for N, a and b (very high values). Answer: table with the execution time for each number of processors.

4.2 MPI parallelism

Exercise 7:

Create a lsf jobscript with the execution of MPI version with 2,4,8 and 16 processors and obtain the execution time of each of them. Use the same values for N, a and b. Answer: table with the execution time for each number of processors.

4.3 MPI vs OpenMP parallelism

Exercise 8:

Compare and discuss the results of the previous two exercises 6 and 7.

5 Advanced MPI parallelism

So far in the previous hands-on we have examined point-to-point communication, which is communication between two processes. Now we will start practicing collective communication. Remember that collective communication is a method of communication that involves participation of all processes in a communicator.

5.1 Programming Collective Communication

Parallel algorithms often call for coordinated communication operations involving multiple/all processes. For example, all processes may need to cooperate to transpose a distributed matrix or to sum a set of numbers distributed one per process. As we have seen these global operations can be implemented by a programmer using the send and receive functions. For convenience, and to permit optimized implementations, MPI also provides a suite of specialized collective communication that includes:

- Barrier: Synchronizes all processes.
- Broadcast: Sends data from one process to all processes.
- Gather: Gathers data from all processes to one process.
- Scatter: Scatters data from one process to all processes.
- Reduction operations: Sums, multiplies, etc., distributed data.

5.2 Advanced example: MPI Matrix-vector multiplication

Exercise 9:

Implement a parallel MPI matrix-vector multiplication using one-dimensional arrays to store the vectors (x) and the matrix (A). Vectors use block distributions and the matrix is distributed by block rows. As a input parameters of the program indicate the dimensions of the matrix (m = number of rows, n = number of columns). The output (printf) will be the product vector y = Ax. For simplicity you may assume that the number of processes should be evenly divisible by both m and n.

The program will use the MPI directives in “MPI advanced” module presented in the theory class. I suggest using the following calls:

- MPI_Bcast ( )
- MPI_Scatter ( )
- MPI_Allgather ( )
- MPI_Gather ( )

And obviously the default ones too:

- MPI_Init( )
- MPI_Comm_size( )
- MPI_Comm_rank( )
- MPI_Finalize()

5.3 Batch execution

Exercise 10: Create a LSF jobscript and submit it with 64 processors. Add the LSF jobscript in the answer. Remember that we have the reservation of SA-MIRI with 5 nodes. We can acces this with "#BSUB –U SA-MIRI".

6 Program Speedup

Remember that in the theory class we discussed that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited regardless of the number of cores available. Look what happens to our example of matrix-vector multiplication.

Exercise 11:

Recovered from your previous hands-on the time for executing the sequential version of matrix-vector multiplication (exercise 12 of Hands-on 1) for N=32.768:

sequential

Note: In the annex 8 you can find the code for one possible answer of exercise 12 of hands-on 1).

Now timing the parallel MPI matrix-vector multiplication from exercise 10 in Hands-on 3 for 2, 4, 8, 16, 32 and 64:

2	4	8	16	32	64

One widely used measure of the relation between the serial and the parallel run-times is the speedup . It's just the ratio of the serial run-time to the parallel run-time:

$$S(n,p) = T_{serial}(n)/T_{parallel}(n,p)$$

Obtain the Speedup of this application and discuss the results with your partner and the professor in the lab session. Plot the results in your report lab.

2	4	8	16	32	64

What is your conclusion?

We will come back to this example in the next hands-on.

7 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. Part of the code used for preparing this hands-on is obtained from the support material of the book An Introduction of Parallel Computer of Peter Pacheco.

8 Annex:

8.1 C two dimensional arrays

C programmers frequently use one-dimensional arrays to “simulate” two-dimensional arrays. The most common way to do this is to list the rows one after another.

```
void Mat_vect_mult(double  A[], double  x[], double  y[], int m, int n) {
   int i, j;


     for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
           y[i] += A[i*n+j]*x[j];
     }
```

8.2 Matrix-Vector Multiplication Sequential program

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

#include <sys/time.h>
#include <time.h>

void Read_matrix(char prompt[], double A[], int m, int n);
void Read_vector(char prompt [], double x[], int n);
void Mat_vect_mult(double A[], double x[], double y[], int m, int n);

struct timeval start_time, end_time;

int main(int argc, char *argv[]) {
   double* A = NULL;
   double* x = NULL;
   double* y = NULL;
   long long int size;
   int m, n;

   n= atoi(argv[1]);
   m= n;

   size=m*n*sizeof(double);
   A = malloc(size);
   x = malloc(n*sizeof(double));
   y = malloc(m*sizeof(double));

   Read_matrix("A", A, m, n);
   Read_vector("x", x, n);

   gettimeofday(&start_time, NULL);
```

```
   Mat_vect_mult(A, x, y, m, n);

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

   free(A);
   free(x);
   free(y);
   return 0;
}  /* main */

void Read_matrix(char    prompt[], double  A[], int m, int n) {
   int i, j;
for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
              A[i*n+j]=rand();
}


void Read_vector( char    prompt[], double x[], int n) {
   int i;
for (i = 0; i < n; i++)
     x[i]=rand();
}

void Mat_vect_mult(double  A[], double  x[], double  y[], int m, int n) {
   int i, j;


     for (i = 0; i < m; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
           y[i] += A[i*n+j]*x[j];
     }


}

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;
}
```