Hands-on 5:

Performance Evaluation of MPI programs

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 5

Performance evaluation of MPI programs

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

In this Hands-On we will center our attention on the matrix-vector multiplication case study in order to go deeper into the performance analysis of parallel programs in our supercomputer Marenostrum. We will use results from the previous hands-on. However in order to be a self-contained hands-on we will review and add all the required previous results and knowledge in the annex section.

2 Case study

For the most part we write parallel programs because we expect that they'll be faster than a serial program that solves the same problem. Then we will compare a serial vs a parallel program.

2.1 Matrix-vector multiplication

Let's take a look at the performance of the matrix-vector multiplication program. Get the code that you created in previous hands-ons:

- Sequential matrix-vector multiplication program (exercise/hands-on 1).
- Parallel matrix-vector multiplication program (exercise 10/hands-on 3).

We will use these two codes as a test in this hands-on section. For simplicity we will consider that the matrix are square (m=n).

Also, in order to have a "more" parallel code to help the learning process, we suggest using the version of matrix-vector multiplication introduced in the previous hands-on: **x1000** the multiplication part. The resulting code that we will use as a case study will be something like:

```
for (noreal_i = 0; noreal_i < 1000; noreal_i++ ){
  for (local_i = 0; local_i < local_m; local_i++) {
     local_y[local_i] = 0.0;
     for (j = 0; j < n; j++)
        local_y[local_i] += local_A[local_i*n+j]*x[j];
  }
}
```

Versions of both codes, serial and sequential, are available in the annex of this hands-on.

2.2 Building the testbed

Exercise 1:

- Create the Matrix-Vector Multiplication SEQuential program "mvmseq.c" program.

- Compile it

```
icc -o mvmseq mvmseq.c
```

Exercise 2:

- Create the Matrix-Vector Multiplication MPI program "mvmmpi.c" program.

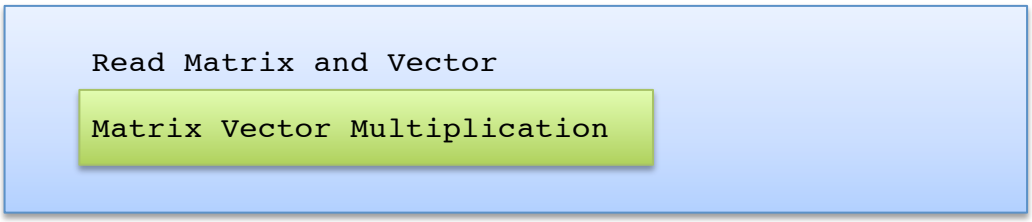
- Compile it

```
mpicc -o mvmmpi mvmmpi.c
```

2.3 Parts of the code

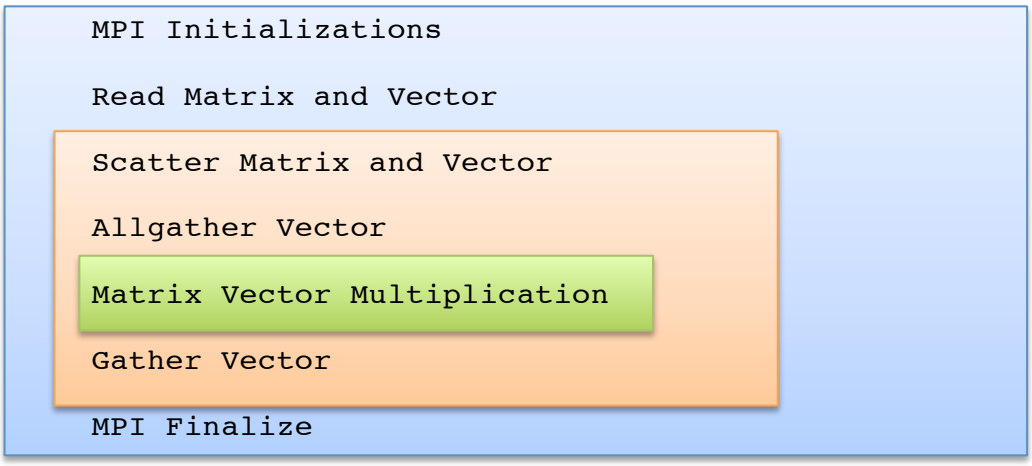
As we already discussed in the previous hands-on, we can distinguish different parts in the codes.

Sequential version:



We suggest timing only the matrix vector multiplication (green box) in order to compare with the parallel version. It doesn't not makes sense to compute the matrix population routine.

Parallel version:



We suggest to timing two parts of them in order to compare with the sequential version: the computation required for the matrix vector multiplication (green box) and the communication (orange box).

2.4 JSF files

We already learned how to build and submit a JSF file. In order to facilitate the way you can obtain the results required to compare, we provide you with the following JSF files for the timing process.

2.4.1 job.seq.lsf

```
#!/bin/bash

#BSUB -J "SA-MIRI"
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -W 00:15
#BSUB -x
#BSUB -R"span[ptile=16]"

SIZE=1024
echo $SIZE >>seq.time
mvmseq $SIZE >>seq.time
SIZE=2048
echo $SIZE >>seq.time
mvmseq $SIZE >>seq.time
SIZE=4096
echo $SIZE >>seq.time
mvmseq $SIZE >>seq.time
SIZE=8192
echo $SIZE >>seq.time
mvmseq $SIZE >>seq.time
SIZE=16384
echo $SIZE >>seq.time
mvmseq $SIZE >>seq.time
SIZE=32768
echo $SIZE >>seq.time
mvmseq $SIZE >>seq.time
```

```
bsub < job.seq.lsf
```

In the file seq.time you will find the timing for each size.

2.4.2 job.mpi.lsf

In order to submit the set of tests required in the next section we suggest using variables to parameterize the JSF files (e.g. LSB_MAX_NUM_PROCESSORS contain the number of processors. It will be useful to use this variable in order to generate different names to the output files).

```
#!/bin/bash

#BSUB -J "SA-MIRI"
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -W 00:55
#BSUB -x
#BSUB -R"span[ptile=16]"

SIZE=1024
echo $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
mpirun mvmmpi $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
SIZE=2048
echo $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
mpirun mvmmpi $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
SIZE=4096
echo $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
mpirun mvmmpi $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
SIZE=8192
echo $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
mpirun mvmmpi $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
SIZE=16384
echo $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
mpirun mvmmpi $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
SIZE=32768
echo $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
mpirun mvmmpi $SIZE >>$LSB_MAX_NUM_PROCESSORS.proc.mpi.time
```

Then you can use this command line for executing the same job for 2,4,8, 16, 32 and 64 number of processors:

```
for i in 2 4 8 16 32 64; do bsub -q debug -n $i <job.lsf;done
```

> Note: In order to specify the number of processors, “bsub –n” is equivalent to “#BSUB –n”. If you define the number of processors in one of these ways, the “mpirun” command does not require the “-np” parameter.

You will have 6 files, one file for each number of processors, which contain the timing for all the sizes with this number of processors. This will help you to populate the tables in the next sections.

3 Taking Timing

3.1 Sequential version

Exercise 3: Taking the time of sequential version.

Timing only the computation part, excluding the **time required to populate the matrix and vector**. Give your global view of the results

Table 5.1: Sequential version:

	1024	2048	4096	8192	16384	32.768
$T_{sequential}$						

3.2 Parallel Computation

Exercise 4: Timing the matrix-vector multiplication program for different number of processes and for different sizes.

Timing only the computation part, excluding the communication part (e.g. the execution time of process 0). Give your global view of the results.

We suggest the following experiments (keep ptile=16):

Table 5.2: Parallel Computing time

		N					
		1024	2048	4096	8192	16384	32.768
$T_{parallel}$	2						
	4						
	8						
	16						
	32						
	64						

Warning: Check in the <jobid>.out file that appears in all the executions the phrase “Successfully completed.”. Otherwise check also <jobid>.err.

3.3 Timing the communication

However in order to know if we are taking advantage of parallelization process we need to compare the sequential time versus the parallel time including the time required for communication.

Exercise 5: Timing the matrix-vector multiplication program for different number of processes and for different order of Matrix **including the required communication time of the collective communication between processes (do not include the time required to populate the matrix and vector).**

Suggestion: Create a new version of the program mvmmpi.com.c and move the timing calls. Compile with `mpicc -o mvmmpi mvmmpi.com.c` and use the same JSF file.

Table 5.3: Communication + computation time

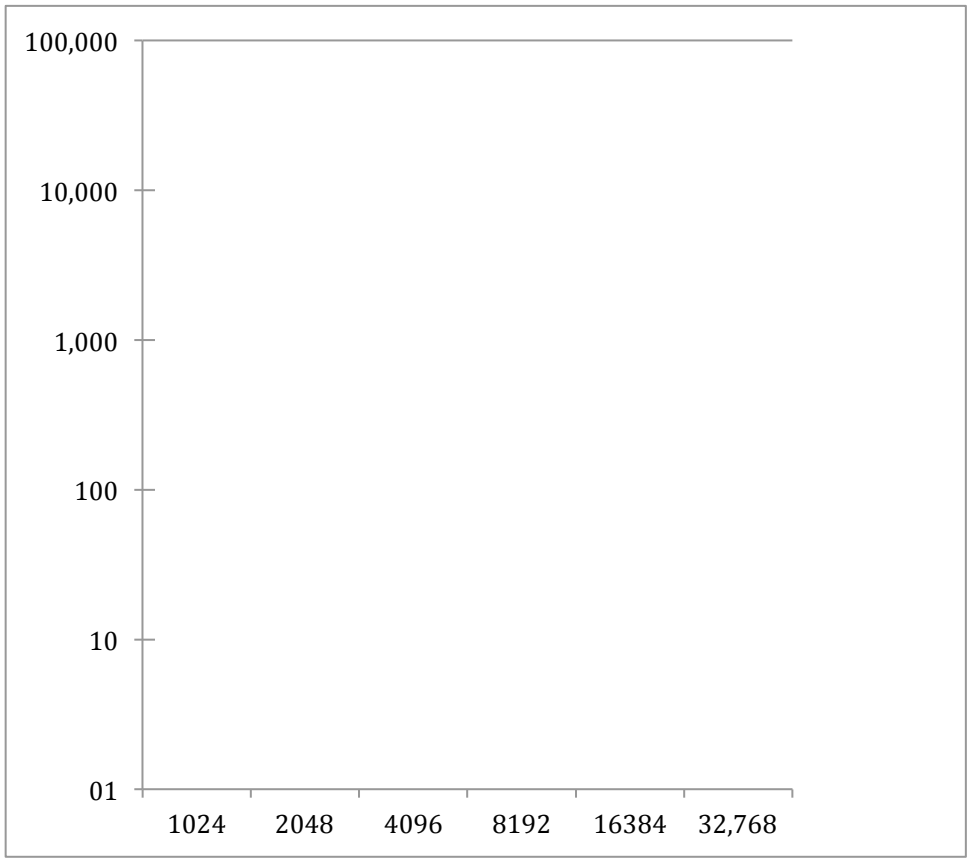
	N					
	1024	2048	4096	8192	16384	32.768
$T_{parallel}$ 2						
with com 4						
8						
16						
32						
64						

Warning: The results are approximate because the time mesured can vary from one execution to another. If you have time you could repeat the experiments and obtain an average value (out of scope of this hands-on the design correct experiments).

4 Data Analysis

4.1 Analyzing the computation time

Exercise 6: Plot the results of table 5.2 comparing the size problem with the execution time. Use a logarithmic scale for the time axis.



Observe that if we fix the number of processors and increase N , the order of the matrix, the run-times increase. And if we fix the order of the matrix and increase the number of processors the run-time decreases.

In this graph there is something strange, What? Why is this? Can you make a guess?

Exercise 7: Why when we use 16 processors the execution time is similar to 8 processors? The first hypothesis could be a problem of memory (you can notice that it occurs from a given matrix size). In order to see the memory usage we can generate a log file for example for a N=8192. With a grep (e.g. cat profile.log | grep "Mem:") we can determine the memory usage. What do you think?

Remember that when we talked about Von Neumann Model Improvements in theory class, the bandwidth between processors and memory can be a bottleneck for some type of applications. Let's try check it

Exercise 8: Execute again the problem with different number of threads per node (e.g. 4,8,16). Remember that you can use the job directive

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

that indicates the number of processes assigned to a node.

Table 5.4:

	ptile=4	ptile=8	ptile=16
	8192	8192	8192
16			

What is your conclusion?

4.2 Analyzing the communication cost

Exercise 9: Plot the results of table 5.3 comparing the size problem with the execution time. Use a logarithmic scale for the time axis. Why are the results very similar to the previous one plot?

Exercise 10: Create a new version of the program mvmmpi.com.c (compting the communication time) but reduce the parallel factor x10 (multiply x100) (`mvmmpi100.com.c`). Compile with `mpicc -o mvmmpi mvmmpi100.com.c` and use the same previous JSF file. Timing the matrix-vector multiplication program `mvmmpi100.com.c` to populate the following matrix.

Table 5.5:

		N					
		1024	2048	4096	8192	16384	32.768
$T_{parallel}$	2						
	4						
	8						
	16						
	32						
	64						

Exercise 11: plot the results and compare with the previous one. What happened with small values of N? Why?

What is your conclusion?

4.3 Program Speedup

Remember that 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:

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

The ideal value for S(n,p) is p. Then our parallel program with p processors is running p times faster than the serial program. In practice this speedup, sometimes called linear speedup, is rarely achieved.

Exercise 12: Plot the graph of the Speedup vs the number of processing elements for matrix multiplication. Also print the linear speedup. Discuss your results with your partner and your teacher during the lab. Include in your report the explanation of its behavior.

What is your conclusion?

5 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.

6 Annex

6.1 Matrix-Vector Multiplication Sequential program

```
// mvmseq.c
#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, jj;

   for (jj = 0; jj < 1000; jj++) {

     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;
}
```

6.2 Matrix-Vector Multiplication MPI program

```
//mvmmpi.c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

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

struct timeval start_time, end_time; //taking time

int main(int argc, char *argv[]) {
   double* local_A;
   double* local_x;
   double* local_y;
   int m, local_m, n, local_n;
   int my_rank, comm_sz;
   MPI_Comm comm;
   double* vec = NULL;
   double* v_vec = NULL;
   double* A = NULL;
   int i, j;
   double* x;
   int local_i,  noreal_i;

   MPI_Init(NULL, NULL);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &comm_sz);
   MPI_Comm_rank(comm, &my_rank);

   if (my_rank == 0) {
      if (argc<2) {
         printf("the number of parameters is not correct\n");
         exit(-1);
      }
      n= atoi(argv[1]);
      m= n; // for simplicity nxn matrix
   }

   MPI_Bcast(&m, 1, MPI_INT, 0, comm);
   MPI_Bcast(&n, 1, MPI_INT, 0, comm);
   local_m = m/comm_sz;
   local_n = n/comm_sz;

   // Allocate storage for local parts of A, x, and y
   local_A = malloc(local_m*n*sizeof(double));
   local_x = malloc(local_n*sizeof(double));
   local_y = malloc(local_m*sizeof(double));

   //Read_matrix_vector
   if (my_rank == 0) {
      A = malloc(m*n*sizeof(double));
      v_vec = malloc(n*sizeof(double));
      for (i = 0; i < m; i++)
         for (j = 0; j < n; j++)
              A[i*n+j]=rand();
      for (i = 0; i < n; i++)
           v_vec[i]=rand();
     }
```

```
   //taking time
//   if (my_rank == 0) { gettimeofday(&start_time, NULL); }

   //Scatter
   MPI_Scatter(A, local_m*n, MPI_DOUBLE, local_A, local_m*n, MPI_DOUBLE,
0, comm);
   MPI_Scatter(v_vec, local_n, MPI_DOUBLE, local_x, local_n, MPI_DOUBLE,
0, comm);

   x = malloc(n*sizeof(double));

   MPI_Allgather(local_x, local_n, MPI_DOUBLE, x, local_n, MPI_DOUBLE,
comm);

  // Mat_vect_mult

if (my_rank == 0) { gettimeofday(&start_time, NULL); }

   for (noreal_i = 0; noreal_i < 100; noreal_i++ ){  // x100 iterations
     for (local_i = 0; local_i < local_m; local_i++) {
        local_y[local_i] = 0.0;
        for (j = 0; j < n; j++)
           local_y[local_i] += local_A[local_i*n+j]*x[j];
     }
   }

   if (my_rank == 0) { gettimeofday(&end_time, NULL); print_times(); }
//taking time
   if (my_rank == 0) { vec = malloc(n*sizeof(double)); }

   //Vector Gather
   MPI_Gather(local_y, local_m, MPI_DOUBLE, vec, local_m, MPI_DOUBLE, 0,
comm);

//   if (my_rank == 0) { gettimeofday(&end_time, NULL); print_times(); }
//taking time

   if (my_rank == 0) {
       free(vec);
      free(A);
      free(v_vec);
      }
   free(x);
   free(local_A);
   free(local_x);
   free(local_y);
   MPI_Finalize();
   return 0;
}  /* main */

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;
}
```