Hands-on 4:
Getting Started with Performance

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 4

Getting Started with Performance

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

In this Hands-On we will present basic procedures in order to improve the performance of your program. We will discuss how compilers, modules and implementations can affect the performance of a program. We will also review some of the linux performance monitoring tools such as top, *vmstat* or *perf* and how we can use it in a batch mode. Finally we will use the Amdahl's Law.

2 Cache and programs

Remember that the working of the CPU cache is controlled by the system hardware (programmers don't determine which data is in the cache). However, knowing the principle of spatial and temporal locality allows us to have some indirect control over caching. E.g. C stores two-dimensional arrays in "row-major" order. Look at the following example:

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

#define SIZE 8000
typedef double matrix[SIZE][SIZE];
matrix m1, m2;
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] = 8.0/m2[j][i];
}

main () {
  int i,j;

for(i=0;i<SIZE;i++) {
     for(j=0;j<SIZE;j++) {
         m2[i][j]=2.5;
    }
    }
   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;
}
```

Exercise 1:

a) Compile and execute the test.c program. Use the flag “-O0”.

b) Create a new program test.ji.c based in .test.c changing “ m1[j][i] = 8.0/m2[j][i];” by “ m1[i][j] = 8.0/m2[i][j];

c) Justify why the execution time of the two loops are so different.

3 Compilers

Why are the flags in a compiler important?

3.1 icc

Exercise 2: Execute the previous example test.c program using different flags of Intel compiler

```
icc -O0 test.c -o test
echo "icc -O0"
./test
icc -O2 test.c -o test
echo "icc –O1"
./test
icc -O3 test.c -o test
echo "icc -O3"
./test
```

Exercise 3: using the man icc explain why the execution time is different between them. Tip: search what is “dead code elimination (DCE)”.

3.2 gcc

Try now with gcc.

Exercise 4: Do the same with gcc compiler. What is your opinion?

```
gcc -O0 test.c -o test
        echo "gcc -O0"
        ./test
        gcc -O2 test.c -o test
        echo "gcc -O2"
        ./test
        gcc -O3 test.c -o test
        echo "gcc -O3"
        ./test
```

4 Modules environment

4.3 Overview

The Environment Modules package (http://modules.sourceforge.net/) provides a dynamic modifica- tion of a user's environment via modulefiles. Each modulefile contains the information needed to configure the shell for an application or a compilation. Modules can be loaded and unloaded dynam- ically, in a clean fashion. All popular shells are supported, including bash, ksh, zsh, sh, csh, tcsh, as well as some scripting languages such as perl.

Installed software packages are divided into five categories:

- Environment: modulefiles dedicated to prepare the environment, for example, get all necessary variables to use openmpi to compile or run programs
- Tools: useful tools which can be used at any time (php, perl, . . .)
- Applications: High Performance Computers programs (GROMACS, . .)
- Libraries: Those are typically loaded at a compilation time, they load into the environment the correct compiler and linker flags (FFTW, LAPACK, . . .)
- Compilers: Compiler suites available for the system (intel, gcc, . . .)

Modules can be invoked in two ways: by name alone or by name and version. Invoking them by name implies loading the default module version. This is usually the most recent version that has been tested to be stable (recommended) or the only version available.

```
% module load intel
```

Invoking a module by version loads the version specified by the application.

```
% module load intel/13.0.1
```

4.4 Modules commands

The most important commands for modules are the following:

- *module list* shows all the loaded modules
- *module avail* shows all the modules the user is able to load
- *module purge* removes all the loaded modules
- *module load <modulename>* loads the necessary environment variables for the selected module- file (PATH, MANPATH, LD_LIBRARY_PATH. . .)
- *module unload <modulename>* removes all environment changes made by module load command
- *module switch <oldmodule> <newmodule>* unloads the first module (oldmodule) and loads the second module (newmodule)

You can run “module help” any time to check the command’s usage and options or check the module(1) manpage for further information.

4.5 MPI implementation

How to compile MPI programs at Marenostrum; We have different options:

LANGUAGE	Commad
C	mpicc
C++	mpiCC/mpicxx/mpic++
Fortran 77	mpif77
Fortran 90	mpif90

Note: we will use C language throughout all the hands-on.

All the backend compilers are managed by environment variables (modules intel and gnu). Remember to use the same module to compile and execute.

Several MPI implementations are available:

- *Intel MPI*
- *OpenMPI*
- *MVAPICH2*
- *IBM Parallel Environment (PE)*

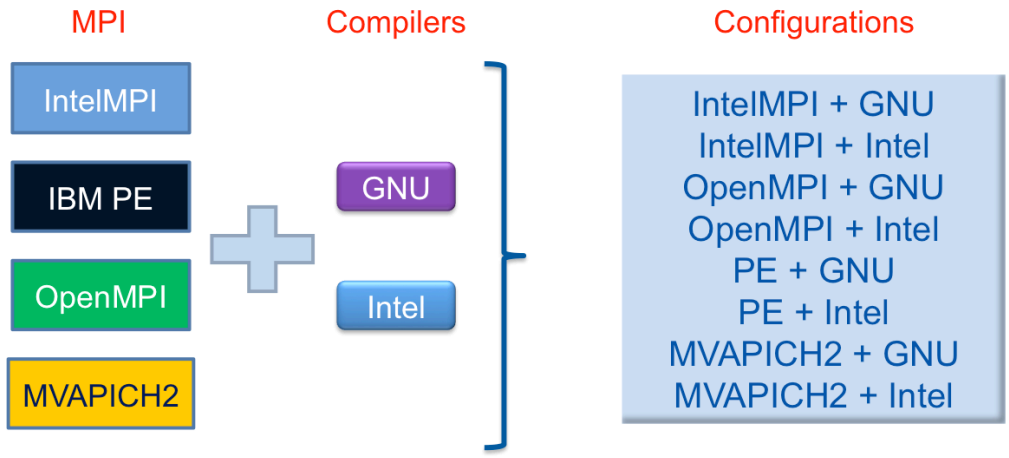
OpenMPI 1.5.4 is the default loaded module. You can change our environment easily using 'module' commands. As we mentioned in a previous section, for example, to change the OpenMPI by IntelMPI module:

```
$ module unload openmpi
remove openmpi/1.5.4 (PATH, MANPATH, LD_LIBRARY_PATH)
$ module load impi
load impi/4.1.1.036 (PATH, MANPATH, LD_LIBRARY_PATH)
```

Recall that you can check the loaded module by

```
$ module list
Currently Loaded Modulefiles:
  1) intel/13.0.1     2) transfer/1.0     3) impi/4.1.1.036
$ mpicc -show
icc -ldl -I/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/include -
L/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/lib -Xlinker --enable-new-
dtags -Xlinker -rpath -Xlinker
/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/lib -Xlinker -rpath -Xlinker
/opt/intel/mpi-rt/4.1 -lmpi -lmpigf -lmpigi -lrt -lpthread
```

You can not have 2 MPI implementations loaded at the same time (module command will not allow you). We have different possible configurations at Marenostrum:



Note: There is no “best” MPI implementation in MareNostrum, it depends on your program.

Exercise 5: Check the current module. Run the same program (e.g. your matrix multiplication program from exercise 10 in hands-on 3 for N=16384 and 64 processors) in two different environments: OpenMPI and IntelMPI. Compare the execution time obtained in boths cases and give your opinion. Include in your answer the data obtained and the JSF files used.

5 Command tools to monitor Linux Systems

As you know Marenostrum is a Linux environment. There are different command line tools which are very useful to monitor and debug Linux System Performance. Here we will review some of them just to give you the sense that we are in a Linux environment but we could also use them in a bach mode.

In the “Taking Timing” section in Hands-on 2 we discovered how much time it takes to run a job. With the linux performance monitoring commands we can find out if a user’s job will take a longer time than necessary to execute. In hands-on 6 and hands-on 7 we will explore more powerful performance tools developed at BSC.

However the question that appears is … How can we execute some of these commands when our job is running in a bach mode? We don’t have access to the execution context from our terminal. Lets book at some examples.

As a test code you could use any of the previous codes you created.

5.1 top

Linux Top command is a performance monitoring program which is used frequently to monitor Linux performance and it is available under many Linux/Unix like operating systems. The top command is used to display all the running and active real-time processes in an ordered list and updates it regularly. It displays CPU usage, Memory usage, Swap Memory, Cache Size, Buffer Size, Process PID, User, Commands and much more. It also shows high memory and cpu utilization of running processes.

This exercise shows you how to execute a top command in your batch job script to monitor the master node of your job.

Exercise 6:

a) Create the bash script file *top.sh* with the following code:.

```
#!/bin/bash

for (( c=1; c<=5; c++ )) ; do

top -n 1 -b -u $USER
sleep 2

done
```

The idea is to launch in background the *top.sh* command and then execute the standard mpirun command with any of the MPI jobs previously executed during this training.

b) Create a JSF or take one used in a previous exercise and add the following line before executing the mpirun:

```
./top.sh &
```

c) Submit the JSF again Check the .out file. What is its content? Please, explain.

5.2 vmstat

Another example is Linux Vmstat command. VmStat command used to display statistics of virtual memory, kernel threads, disks, system processes, I/O blocks, interruptions, CPU activity and much more. By default vmstat command is not available under Linux systems. You need to install a package called sysstat that includes a vmstat program. The common usage of command format is

```
%  vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0 4464728 24484412      0 2865196    0    1     0     3    1    0  3  1 96  0  0
```

It helps to find memory usage, disk usage, etc

Exercise 7 (optional):

a) Create the bash script file vmstat.sh with the following code:.

```
#!/bin/bash

for (( c=1; c<=5; c++ )) ; do

vmstat 3 3

sleep 2
done
```

Help about vmstat: http://linux.about.com/library/cmd/blcmdl8_vmstat.htm

b) Create a LSF job using the vmstat.sh bash script. The idea is to launch in background the vmstat.sh command and then execute the standard mpirun command with any of the MPI jobs previously executed during this training.

c) Check the meaning of all the fields shown by the vmstat command.

5.3 perf

PERF command is useful to monitor the performance of a serial or parallel job. The output of perf is a list of different system performance metrics. This exercise will show you how to use the PERF command in a LSF job script.

Exercise 8 (OPTIONAL):

a) Try the following command.

```
perf stat ls
```

Check and explain the output generated by the PERF performance tool , Help about PERF command can be found at the following page :

https://perf.wiki.kernel.org/index.php/Main_Page

b) Create the bash script file perf.sh with the following code:.

```
#!/bin/bash
if [ -z "$PMI_RANK" ] ; then

  perf stat $@ 2> $1.out.perf

else
  perf stat $@ 2> $1.$PMI_RANK.perf

fi
```

c) Create a LSF job using the previous perf.sh bash script. The idea is to launch it in background and then execute the standard mpirun command with any of the MPI jobs previously executed during this training.

5.4 Log file

In order to have a more clear log file separated from standard output file and your timing file, we suggest to following the template below in order to build your LSF file.

```
#!/bin/bash
#!/bin/bash

#BSUB -J "SA-MIRI-log.file.generator"
#BSUB -n 16
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -W 00:15

LOG=profile.log

cont=0
# Launch vmstat for infinity into logfile
( while [ $cont -eq 0 ]; do top -n 1 -b -u $USER -H >> $LOG; vmstat >>
$LOG ; do
ne ) &

sleep 1
echo "#####################################" >> $LOG
echo "#####     Starting execution     ####" >> $LOG
echo "#####################################" >> $LOG

mpirun {your program + parameters} >timing.out

echo "#####################################" >> $LOG
echo "####     Stopping execution     #####" >> $LOG
echo "#####################################" >> $LOG
sleep 1
```

In this bash script template "{your program + parameters}" indicate your program (and parameters if necessary).

Exercise 9: Launch some of the previous codes used in this course (e.g. your matrix multiplication program from exercise 10 in hands-on 3 for N=32768 and 16 processors). Inspect the *log* file generated. Determine and describe the evolution of memory and CPU use. Discuss your conclusions with the teacher at lab session.

Exercise 10 (OPTIONAL): Include in your log file information extracted from vmstat and perf. Include in the answer your new LSF file.

6 Visiting the Amdahl's Law

Remember that in a previous hands-on we argued 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.

6.1 Overview

Amdahl's law can be used to find the maximum expected improvement to an overall system when only part of the system is improved (to predict the theoretical maximum speedup using multiple processors).

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. However the speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.

Remember the simple example of a program that needs 20 hours using a single processor core, and a particular portion of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (95%) of execution time can be parallelized. In conclusion, regardless of how many processors are devoted to a parallelized execution of this program, assuming millions, the minimum execution time cannot be less than that critical one hour. Hence the speedup is limited to at most 20×.

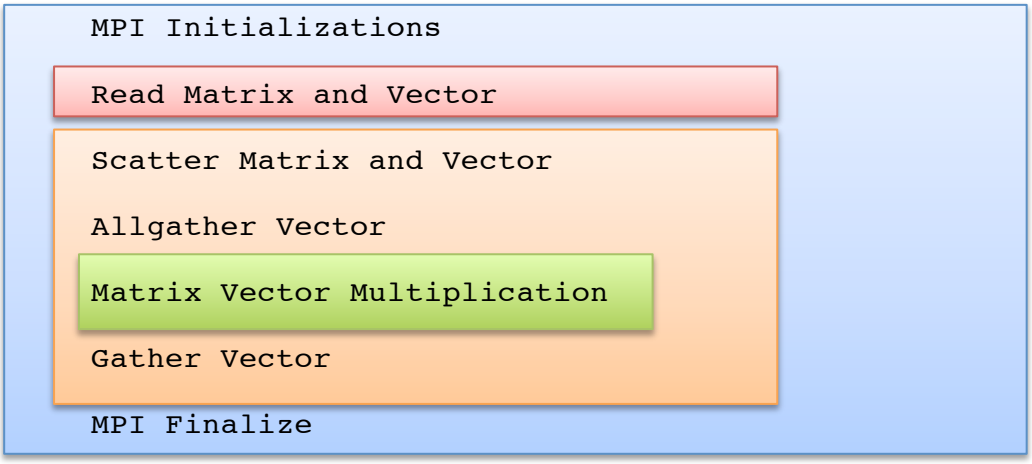
6.2 Case study: Matrix-vector multiplication

Let's take a look at the performance of the matrix-vector multiplication program. Get the code that you created in previous hands-ons.. From now on, for simplicity we will consider that the matrix are square (m=n).

Versions of this code are available as an annex in this hands-on (you can do a copy&paste).

6.3 Timing the parallel code

If we have a look at this code we can distinguish different parts:



The only parallel part that could be parallelized is the matrix vector multiplication code (green box). Note that communication (orange box), despite being parallel, is actually an overload.

Exercise 11: Now timing again the parallel MPI matrix-vector multiplication from exercise 10 in Hands-on 3 for 2, 4, 8, 16, 32 and 64. Divide the previous time obtained for each process in two: the serial part executed by process 0 and the parallel part. Compute the percentage.

	2	4	8	16	32	64
serial						
parallel						
% serial						
% parallel						

6.4 Speedup vs number of processors

Exercise 12: Plot the results of the previous exercise in the form speedup vs number of processors. What are your conclusions?

Now, in order to have a "more" parallel code we suggest to **x100** the matrix-vector multiplication part. The resulting code that we will use as a case study will be something like:

```
for (ii = 0; ii < 100; ii++ ){
  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];
  }
}
```

Exercise 13: Compute again the following table with the new code and plot the results. Compare the results with the previous ones. What are your conclusions?

	2	4	8	16	32	64
serial						
parallel						
% serial						
% parallel						

Exercise 14 (OPTIONAL): Do the same as previous exercice with a "more more" parallel code with "x1000" in the matrix-vector multiplication. Plot and compare the results with the previous ones.

7 Report Lab

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

Now, we are ready for the next hands-on. We will center our attention in the previous matrix-vector multiplication case discussed in order to go deeper into the performance analysis of parallel programs.

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.

8 Annex: Matrix-Vector Multiplication MPI program

8.1 Assumptions in this code

We are distributing A by rows so that the computation of y[i] will have all of the needed elements of A , so we should distribute y by blocks. Now the computation of y[i] also involves all the components of x, so we could minimize the amount of communication by simply assigning all of x to each process.

However, I assumed that in general applications that uses matrix-vector multiplication will execute the multiplication many times and the result vector y from one multiplication will be the input vector x for the next iteration. In practice, then, I assumed that the distribution for x is the same as the distribution for y.

8.2 Code

```
//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;
}
```