Distributed HPC programming with POP-C++

Distributed Matrix Multiplication

2016-2017

Professors: Pierre Kuonen, Peter Kropf

Assistants: Veronica Estrada Galinanes, Andrei Lapin

1. Performance analysis of a distributed matrix multiplication program

The objective of this exercise is to execute and to analyze the performances of a parallel dense matrices multiplication program written in POP-C++. This program computes the following product:

A x **B** = **R**

where **A**, **B** and **R** are **NxN** square matrices).

The program uses a « Master/Worker » approach. The master prepares the matrices, creates the workers, sends the work to do to each workers, waits for the partial result of each worker and finally reconstructs the full result e.i. the **R** matrix.

The detail of the program is as follows. The master divides the matrix **A** into severals blocks of lines and the matrix **B** into several blocks of columns. Each worker receives one block of lines of the matrix **A** and one block of columns of the matrix **B**. Using these data, each worker can compute one block of the matrix **R**. This process is illustrated in the figure 1.

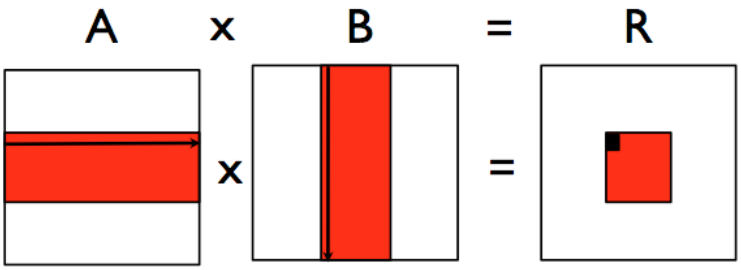

Figure 1: Distribution of the matrices

As mentioned above, each worker compute a block of the resulting matrix. To do this work each work can use one core (only one thread of execution) or several cores (several threads of execution). This depends of the choice of the user who can consider each core as a different machine (only one thread of execution) or can use all the available cores of each machine (several threads of execution).

The number of workers (**W**) is equal to the product of the number of blocs of lines of the matrix **A** and the number of blocs of columns of the matrix **B**. In order to ease the implementation of the program the following constraints must be fulfill : *the numbers of blocs the matrices **A** and **B**, must be divisors of the size **N** of the matrices.*

In the rest of this document we will use the following naming:

- **N** : The size of matrices
- **L** : The number of blocs of lines of the matrix **A** is divided
- **C** : The number of blocs of columns of the matrix **B** is divided
- **T** : The number of cores (threads) per worker

These values are independent parameters and they influence the computing time of the program. The number of workers **W** being equal to **LxC.**

Your first task is to play with these values and to record computing times. Then you need to analyse and explain the obtained results.

2. Computations to run

The tests on the cluster will be done by group of students. For each group some values for these parameters are imposed. You must use ***at least*** these values but, of course, you can play with other values as long as you respect the constraint mentioned above. The provided program record the following time values:

- The ***initialization time*** = the time used by the master to create all workers.
- The ***sending time*** = the time used by the master to send data to all workers.
- The ***computing time*** = the time spent on the Master to obtain all results from the worker and to reconstruct the result matrix **R**
- The ***worker waiting times*** = the time each worker waits since he has been created until he receives all the data he needs to do the computation.
- The ***worker computing times*** = the time used by each worker to do the computation after they have received all the data they need.

The times presented above are illustrated on the figure 2.

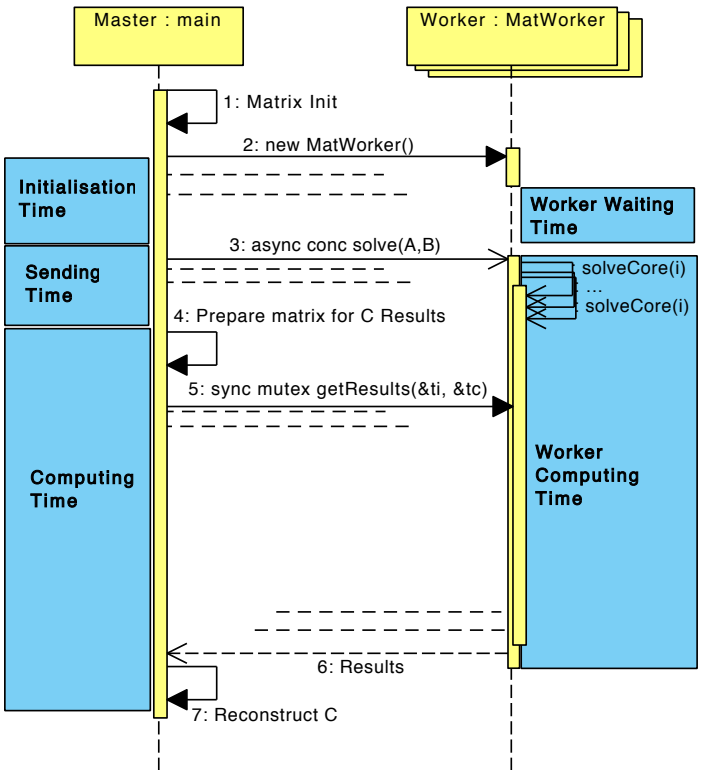

Figure 2: Sequence diagram of the execution of the matrix multiplication program and times recorded

On Figure 2 we can notice that, when the Master (`main`) asks a Worker to compute his block (call the method `solve(A,B)`), the worker launches several instances of the `solveCore` method. In the case of "one core execution" only one instance of the `solveCore` method is launched, when in the case of a "multi-core execution" several instances of the `solveCore` method is launched (one for each core).

The program uses the file `machines.ip`. This file contains the list of the available machines for the execution. The fact that you execute a multi-cores or a single core version depends on how you describe the available machines in the `machines.ip` file. See section 5 of this document for more details on this feature.

3. Work to do

Each group will run “one core executions” and “multi-cores executions”. Nevertheless before doing parallel executions, we need to have the reference sequential times which will allow us to compute the speedup. Each group will have to compute sequential times for five different sizes of matrices. Then, we will put all these sequential times together to construct a table containing all necessary sequential reference times.

3.1. Computation of sequential references times

The sequential time is the time used to do the computation using only one core. To do so, you will use the `scriptSequential.sh` script.

Make the four following plots (for all plots X-axis=size of the matrix (**`N`**))

- plot 1 : Y-axis = initialisation time
- plot 2 : Y-axis = sending time
- plot 3 : Y-axis = computing time
- plot 4 : Y-axis = total of the initialisation, sending and computation time

With each plot also give a table which contained the measured value.

Remarks: Always indicate the units of the measured values. Use linear X and Y axis.

For each plot (curves) make a fit in order to give the mathematical function which fit the best the curve.

3.2. Computation of parallel times

To do so you will use the `scriptParallel.sh` script. Each group will have to compute for five different sizes of the matrix (**`N`**), the time for five different numbers of workers (**`W`**) (depending of the group, single or multi-core)

For all values of **`W`** (X axis) assigned to your group (use linear X and Y axis)

A) Tables (using same units for all values)

Initialisation time

Sending time

Computing time

B) Two different plots (use linear X and Y axis) :

The speed up for each value of **`N`** assigned to your group.

The efficiency for each value of **`N`** assigned to your group.

For each plot comment, explain and justify the obtained results.

Hint:

- All results must be an average of several executions (min. 3) in order to decrease the influence of local conditions.
- Times measured by the program are LT (local time= wall clock time on the machine).

4. Technical information:

The sources of the programs are in the directory `/home/$USER/shared_public (replace $USER with your user account id)`. Do the following to install and to test the execution of the matrix multiplication program:

1. Go in your private directory

```
cd /home/$USER/shared_private
```

2. Copy `project_1.tgz` file in this directory :

```
cp ../shared_public/project_1.tgz .
```

3. Untar the file: `tar -zxvf project_1.tgz`
4. Go in the directory where the execution have to be done : `cd project_1/matrix/Standard`
5. Compile the program: `make all`
6. Now, you can run a test to check that everything is working: `make run`

   You should obtain something like:

```
popcrun obj.map ./mainpopc 1500 1 1

POP-C++: mainpopc has started with 1 tasks.
Initializing arrays...

Parameters are:
Matrix Size=1500, Blocs (lines=1, columns=1), Workers=1, Cores=4

POP-C++: Starting standard Matrix multiplication program...

Times (init, send and computing) = 0.0456, 0.0382, 2.62 sec

...End of matrix multiplication

A[1500x1500]:
   9.0   19.0    9.7    5.4    7.6   22.7   17.3   18.4 ..   15.6
  28.1   26.6    3.9   22.3   14.1   11.7    3.1   25.9 ..    0.6
  22.6   16.7   27.9    8.1    1.9    0.6   14.6    0.9 ..   16.0
  16.6   23.6   23.9   25.9    4.4    6.1   10.1   10.9 ..    1.6
   1.3    2.4   14.3   13.3   20.9    2.0   11.1   20.1 ..    1.6
  15.9   11.3    5.1   19.0   19.7    3.1   18.6   14.7 ..   28.1
  13.3    2.0   22.6   10.1   24.3   10.7    5.6   18.1 ..   25.9
  24.0   22.0    5.7    3.0   27.0   27.4   14.1   17.4 ..    9.4
    ...  ...  ...
  10.0   25.9   19.0   20.3    3.7   25.3   21.1   23.1 ..   10.3
-----------------------
B[1500x1500]:
   9.0   28.1   22.6   16.6    1.3   15.9   13.3   24.0 ..   10.0
  19.0   26.6   16.7   23.6    2.4   11.3    2.0   22.0 ..   25.9
   9.7    3.9   27.9   23.9   14.3    5.1   22.6    5.7 ..   19.0
   5.4   22.3    8.1   25.9   13.3   19.0   10.1    3.0 ..   20.3
   7.6   14.1    1.9    4.4   20.9   19.7   24.3   27.0 ..    3.7
  22.7   11.7    0.6    6.1    2.0    3.1   10.7   27.4 ..   25.3
  17.3    3.1   14.6   10.1   11.1   18.6    5.6   14.1 ..   21.1
  18.4   25.9    0.9   10.9   20.1   14.7   18.1   17.4 ..   23.1
    ...  ...  ...
  15.6    0.6   16.0    1.6    1.6   28.1   25.9    9.4 ..   10.3
-----------------------
C[1500x1500]=A*B:
414065.4 305604.8 294139.8 306499.2 305618.9 301913.5 317061.5 307363.8 .. 302333.0
305604.8 404674.7 294508.2 305363.4 303034.6 302890.6 310925.7 309742.1 .. 303773.0
294139.8 294508.2 384836.5 289505.3 291311.6 292743.8 302075.3 295835.6 .. 283864.3
306499.2 305363.4 289505.3 414737.8 303186.6 303692.6 315883.5 305761.7 .. 297611.5
305618.9 303034.6 291311.6 303186.6 397727.9 298465.4 308633.2 303596.6 .. 292481.9
301913.5 302890.6 292743.8 303692.6 298465.4 408607.7 307320.1 303330.6 .. 295313.3
317061.5 310925.7 302075.3 315883.5 308633.2 307320.1 417991.5 315925.7 .. 301844.1
307363.8 309742.1 295835.6 305761.7 303596.6 303330.6 315925.7 412560.3 .. 300046.7
    ...  ...  ...
302333.0 303773.0 283864.3 297611.5 292481.9 295313.3 301844.1 300046.7 .. 397597.3
-----------------------
```

5. Information on POP-C++ environment *(reminder)*

The syntax of the command to run a POP-C++ program is the following:

```
popcrun ObjectFile ./NameOfProgram Parameters
```

where:

- **`ObjectFile`** is a text file containing information on where the executable files for workers (remote objects) are located.
- **`NameOfProgram`** is the name of the file containing the executable code to launch.
- **`Parameters`** are parameters of the program.

In our case:

- the file **`ObjectFile`** is automatically generated by the line:
  `/MatWorker.obj -listlong > obj.map`
  of the makefile. Its name is `obj.map`.
- the name of the executable to launch: `/mainpopc`
- the parameter of the program are: the size **`N`** of the matrices, the number **`L`** of blocs lines of the matrix **`A`** and the number **`C`** of blocs columns of the matrix **`B`** and finally (optional) the name of the file where the results must be written.

The following command line allows you to multiply matrices of size 1000, on 4 workers, dividing the matrix **`A`** into 2 blocs and the matrix **`B`** in 2 blocs and finally to write the result in the file `results.txt`

```
popcrun obj.map ./mainpopc 1000 2 2 results.txt
```

Please note that the «`./`» before the name of the executable file is mandatory.

The program generates random **`A`** and **`B`** matrices of size **`NxN`** (1000x1000 in this case), divides the matrix **`A`** in 2 blocs of 500 lines, the matrix **`B`** in 2 blocs of 500 columns, set the numbers of workers to 2x2=4, does the computation and writes results in the `results.txt` file. Each execution writes a new line in the file `results.txt` using the following format:

```
1000    2    2    0.151508    0.052309    0.705574    4    0.121091    0.721429 .....
```

Each value is separated by a « tab » character. The meanings on these values are the following:

- first value `1000`: size of the matrices
- second value `2`: numbers of blocks of lines of matrix **`A`**
- third value `2`: number of blocks of columns of matrix **`B`**
- fourth value `0.151508`: initialisation time (sec)
- fifth value `0.052309`: sending time (sec)
- sixth value `0.705574`: computing time (sec)
- seventh value `4`: number of cores
- ……..the remaining values are the worker waiting times, the worker computing times followed by the number of cores for each workers.

The names of the available machines are provided in the `machines.xt` file. Each name of machine is followed by the number of cores this machine is supposed to have. The way the program sees the infrastructure depends on how names of the machines are written in this file. Suppose you have **`p`** machines, each machine having **`n`** cores. Thus you have two ways to see the infrastructure:

- You consider that you have **`pxn`** machines, each machine having one core and the program launches **`pxn`** workers, each worker using one core
- You consider that you have **`p`** machines, each machine having **`n`** cores and the program launches **`p`** workers, each worker using n cores.

This file `machines.txt` is read by the program and must contain the list of the names of available machines written in the format corresponding to the way you want to use the infrastructure. Thus, before

starting an execution you have to copy in the `machines.txt` file, the correct description of the available machines. The file `machines_cores.txt` contains the description for the multi-cores execution (**p** machines with **n** cores) when the file `machines_nocores.txt` contains the description for the single core execution (**pxn** machines of 1 cores). At the beginning of the script `scriptParallel.sh` you will find the line where this copy is done. You have to modify this line accordingly to what you want. Note that to exploit the fact that a machine is multi-core, the program uses the indication of the number of cores contained in the file `machines.txt` to launch several instances of the method `solveCore(…)` of the `parclass MatWorker`.

Two scripts are provided: . Edit these files to indicate your values (see comments at the beginning of the file).

To launch the scripts without being logged on the machine use the `crontab` command (see next section).

7. Information on the usage of `crontab`

To ease the automation of executions, two scripts files are provided. Edit these files to indicate your values (see comments at the beginning of the file).

To launch the scripts without being logged on the machine use the `crontab` command.

Below are few information on the usage of crontab.

- `crontab -l` : show the current programmed crontab
- `crontab -e` : edit crontab

General crontab format to use:

For the sequential execution

```
min hour day month * source <your working directory>scriptSequential.sh > <your working directory>log.log
```

For the parallel execution

```
SHELL=/bin/bash
min hour day month * source <your working directory>scriptParallel.sh > <your working directory>log.log
```

EXEMPLE:

```
SHELL=/bin/bash
19 15 21 10 * source /home/project/MATRIX/scriptParallel.sh > /home/project/MATRIX/log.log
```

The effect is: at 15h19 the 21 October, lunch the script:

`/home/project/MATRIX/scriptParallel.sh`

and write the output (console) on the file `log.log`

Note:

- You must indicate the **full path name** of the script file you want to execute.
- The line `SHELL=/bin/bash` is mandatory to be sure that the correct shell (`bash`) will be used.

For more information on the `crontab` command type: `man crontab`.

8. Handout

The following documents must be provided for each run:

1. A «pdf» version of your report (max. 12 pages!). Each student has to realise its own version of the report even if the measurements have been made by groups of several students. This report should contain all your interesting results (as required in this statement), as well as a pertinent analysis and evaluation.
   On the first page of your report, indicate after you name, the login username you have used to run your tests (see template on the next page).

2. A «zip» file containing all the result files (text format) produced by the program that you used to draw plots of your report.This «zip» file can be the same for the two students of the same group, but each student must deposit this zip file

9. Values for each group

Matrix size for sequential measurements:

Group 1	Group 2	Group 3	Group 4	Group 5
720	960	864	1260	1152
1620	1800	1980	1728	1440
2304	2592	2304	2520	2352
2880	3240	3960	4140	3456
3600	4500	5184	5040	4320

Parallel measurements:

Group 1	Group 2	Group 3	Group 4	Group 5
960	1152	720	864	1260
1440	1440	2352	1800	1620
1728	2304	2592	2520	1980
4320	3600	2880	3240	4140
5184	5040	3456	3960	4500

Workers:

Group 1	Group 2	Group 3	Group 4	Group 5
2x1=2	2x1=2	2x1=2	2x1=2	2x2=4
2x2=4	2x2=4	2x2=4	2x4=8	2x3=6
2x4=8	2x3=6	2x3=6	3x2=6	3x1=3
3x4=12	3x4=12	3x4=12	3x3=9	3x3=9
4x4=16	4x4=16	4x4=16	4x3=12	4x3=12

Distributed HPC programming with POP-C++
Distributed Matrix Multiplication

2016-2017

Professors: Pierre Kuonen, Pierre Kuonen

Assistants: Veronica Estrada Galinanes, Andrei Lapin

Student Name

login name

1. Title of section 1 of your report

Section 1 text....

2. Title of section 2 of your report

Section 2 text....