CSC4005 - Distributed and parallel computing

Fabien Labarbe-Desvignes

Parallel Mandelbrot set

Summary

Overview of the project

The goal of this project's to create two programs that have for main goal to parallelize the mandlebrot set. Both programs are written in C++ (14), the firist one uses pthread parallelization and the second one use MPI (Message Passing Interface). Finally, in order to visualize the computations I used the graphic library Xlib.

What's a Mandelbrot set ?

The Mandelbrot set is generated by *iteration*, which means to repeat a process over and over again. In mathematics this process is most often the application of a mathematical function. For the Mandelbrot set, the functions involved are some of the simplest: they all are what is called *quadratic polynomials* and have the form $f(x) = x^2 + c$, where c is a constant number. As we go along, we will specify exactly what value c takes.

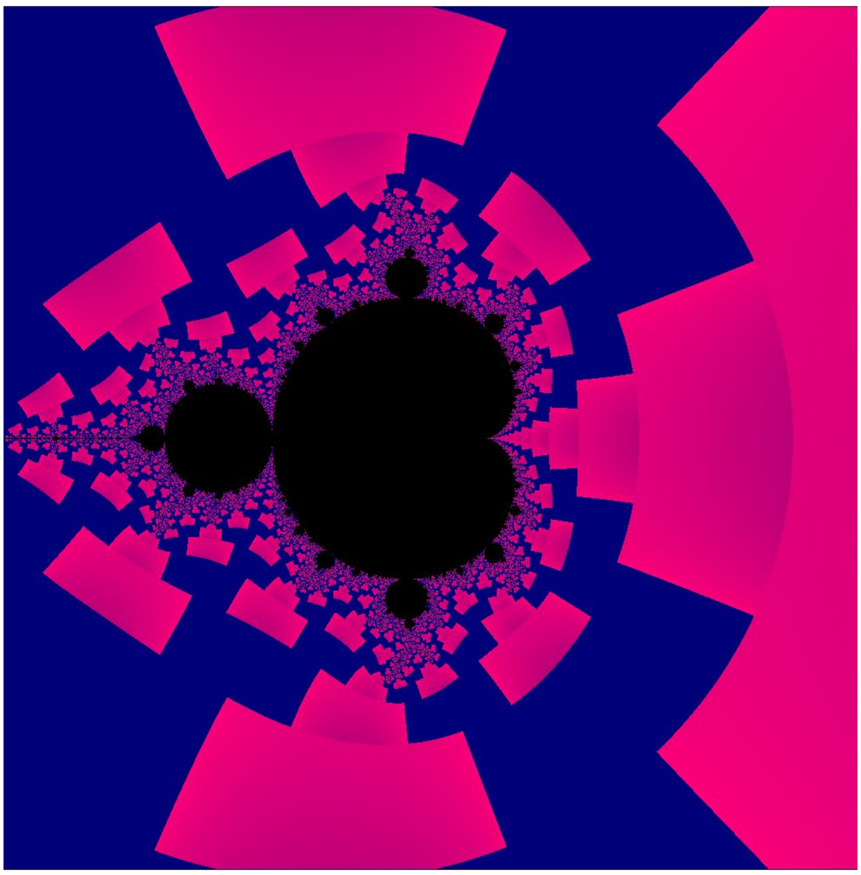


Figure 1 - My Mandelbrot graphical output

Build & usage

Requirements:

cmake (for compilation), pthread, mpi

How to build the pthread / mpi version?

1: go to 119100002/pthread or 119100002/mpi

> cd 119100002/pthread

2: create a build folder, go in it,compile the cmake, make and come back

> mkdir build && cd build && cmake .. && make && cd ..

note: the command line to build is the same for both versions.

How to run the program manually (pthread)

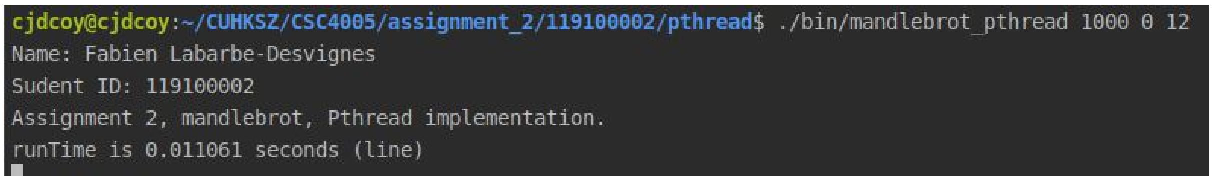
./mandlebrot iteraition NB_ITER MODE [NB_THREAD]

NB_ITER: number of iterations for the mandelbrot

MODE: -1: all modes, 0: line, 1: row, 2: zone partitionning.

[NB_THREAD]: number of threads that will be used for pthread version

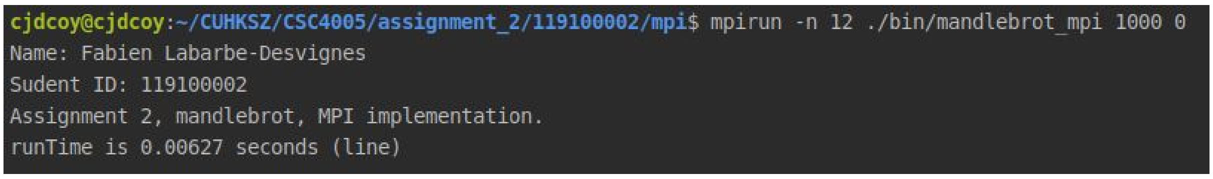
Examples with pthread **(iterations 1000, mode 0, processes 12)**



```
cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_2/119100002/pthread$ ./bin/mandlebrot_pthread 1000 0 12
Name: Fabien Labarbe-Desvignes
Sudent ID: 119100002
Assignment 2, mandlebrot, Pthread implementation.
runTime is 0.011061 seconds (line)
```

./bin/mandlebrot_pthread 1000 0 12

Example with mpi **(iterations 1000, mode 0, processes 12)**



```
cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_2/119100002/mpi$ mpirun -n 12 ./bin/mandlebrot_mpi 1000 0
Name: Fabien Labarbe-Desvignes
Sudent ID: 119100002
Assignment 2, mandlebrot, MPI implementation.
runTime is 0.00627 seconds (line)
```

mpirun -n 12 ./bin/mandlebrot_mpi 1000 0

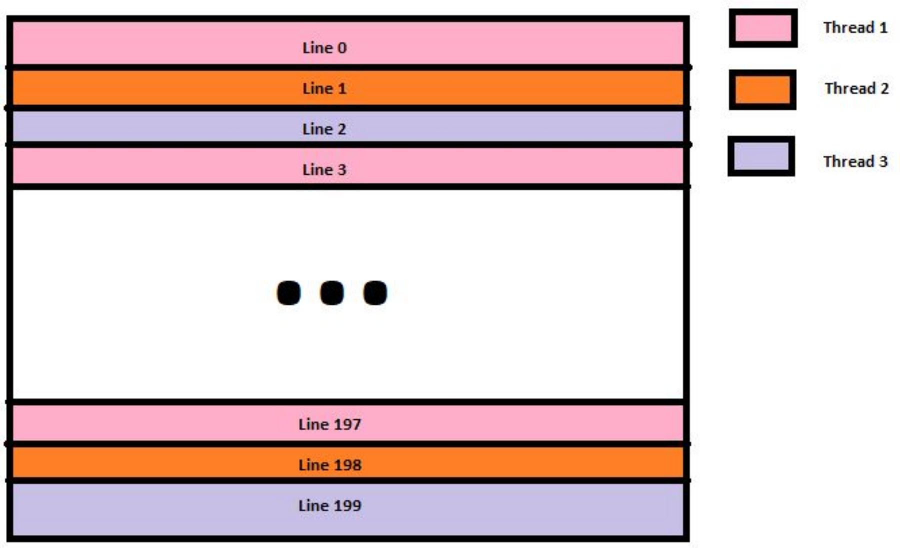
Partitioning

Since I didn't find any material giving name to the kinds of partitioning I used I named them myself and I will explain you in what they consist.

Line partitioning:

In this kind of partition you assign one line to each process/thread, one after the other and you repeat this process until all of the image lines have been assigned.
Example:

3 threads, 200 lines

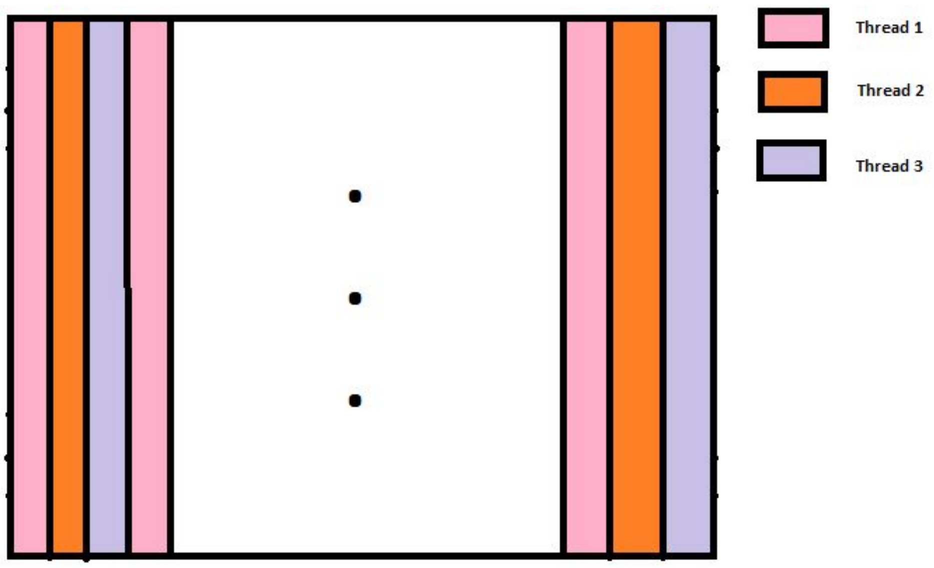


first thread compute lines: 0, 3, 6, 9, 12 ... 191, 194, 197
second thread compute lines: 1, 4, 7, 10, 13 ... 192, 195, 198
and so on for each thread.

Pixel (column) partitioning

Here it's basically the opposite, instead of assigning one line to each process you assign one column to each process one after the other and repeat the process until all columns have been assigned.

Example:
3 threads, there 200 columns

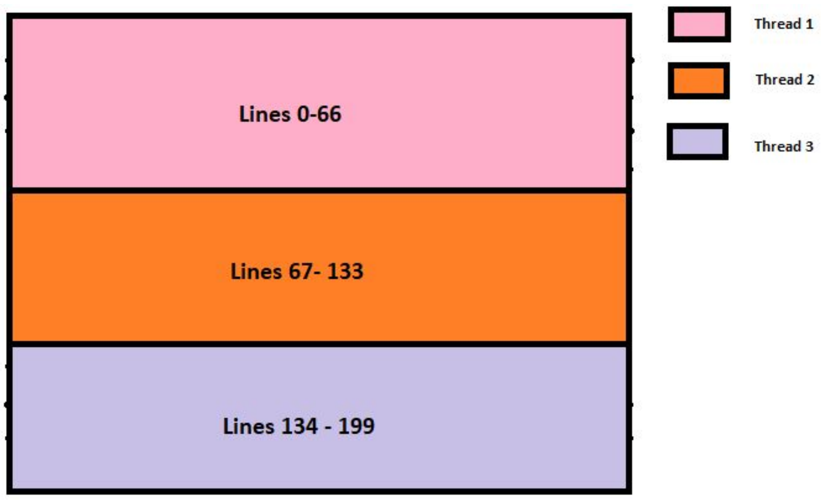


first thread compute column: 0, 3, 6, 9, 12 ... 191, 194, 197
second thread compute column: 1, 4, 7, 10, 13 ... 192, 195, 198
and so on for each thread.

Note: I called it pixel partitioning first because you assign pixels one a line to the thread but after a while I realized that column partitioning is a better name…

Zone partitioning

I implemented this partitioning because I know that it’s one of the worst to use for the Mandelbrot, it allows me to do good comparaisons and justify my choices.
For this partitioning you take the number of lines, divide it by the number of process and assign each offset to a thread.



Example: 3 threads, 200 lines
First thread computes lines 0 to 66
Second thread computes lines 67 to 133 and so on.

Algorithm design (pthread & mpi)

My algorithm can be divided in 3 main phases:

1. Partitioning
2. Processing
3. Display

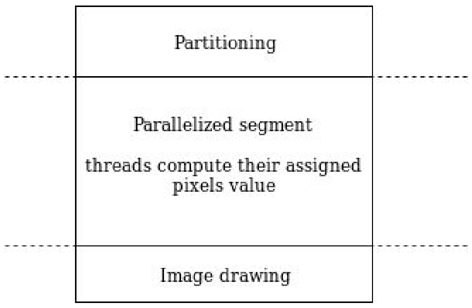
Only the second phase differs in each programs.

Phase 1:
Fill vectors with positions depending on the partitioning selected.
Those vectors define the line that each thread/process will compute as well as its starting position and the offset it'll do at each computation.

Phase 2 (pthread):
Each thread perform its calculations independently and once all the iterations of a pixel are done it adds the pixel's color to the pixel's map until all pixels are computed. After computation each thread exit except for the 'main' program.

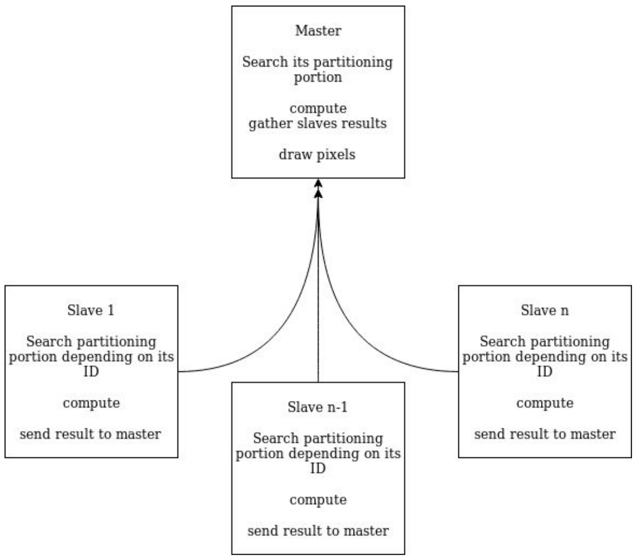
Phase 2 (mpi):
Each processes perform its calculations independently and store the pixel values and position in a vector.
Once all the calculations are done, the slaves send back their vectors to the master which will add each pixel value to the pixel map.

Phase 3:
I put a sleep of 0.5 seconds after the computations (before calling XPutPixels) otherwise there's a risk that the pixels won't be displayed if the computation is really fast (under 0.005 seconds for my computer).



< Pthread design

MPI design >


Master
Search its partitioning portion
compute
gather slaves results
draw pixels
Slave 1
Search partitioning portion depending on its ID
compute
send result to master
Slave n-1
Search partitioning portion depending on its ID
compute
send result to master
Slave n
Search partitioning portion depending on its ID
compute
send result to master

Obtained results

Note**:** The following computation have been performed on a i7-9750H CPU (2.60GHz ×12) with an image of size 200*200. The display's not timed in those benchmarks

100 iterations:

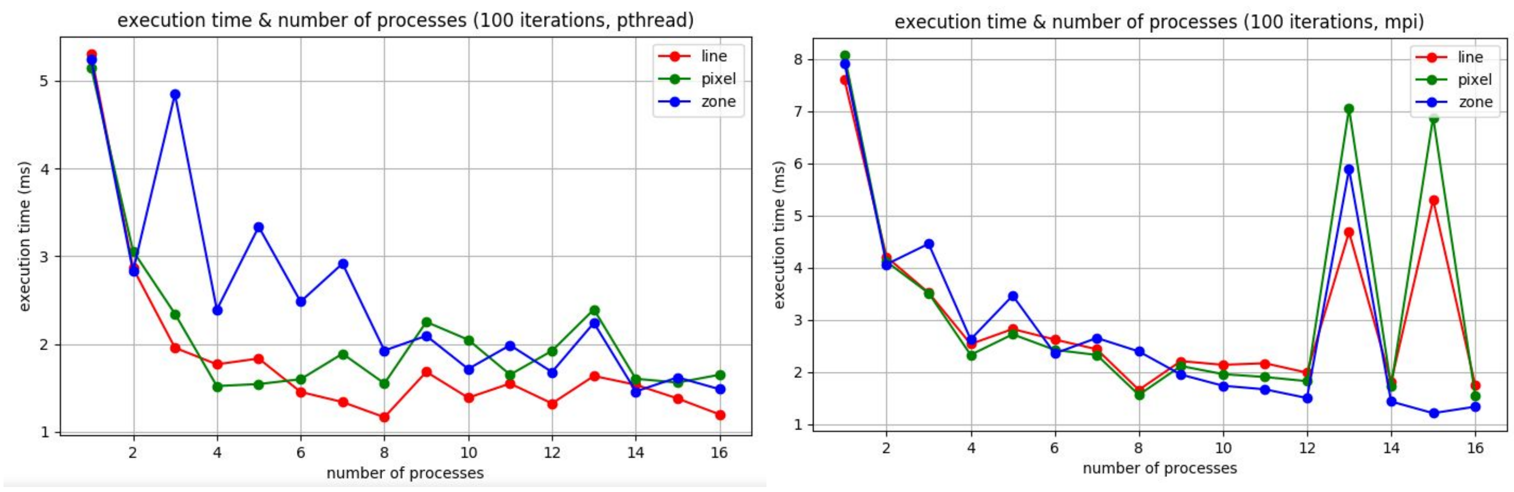


Figure 2 - pthread vs mpi ,100 iterations (line, pixel & zone partitioning)

It's good to note that I didn't uploaded any benchmark below 100 iterations because my benchmarks with 10 iterations were sometimes taking below ONE millisecond (around 800 microseconds) which I don't think is relevant for us.

On this benchmark we can see that both MPI and PTHREAD are faster than the sequential version of the Mandelbrot and this with EVERY partitioning methods. Pthread is always slightly faster than MPI because it there's no communication between threads whereas the MPI slaves have to send back their vectors to the master. If we assume that this is the only difference then we can assume that this last communication takes around 500us to 1500us which is negligible on slightly larger scales

1000 iterations:

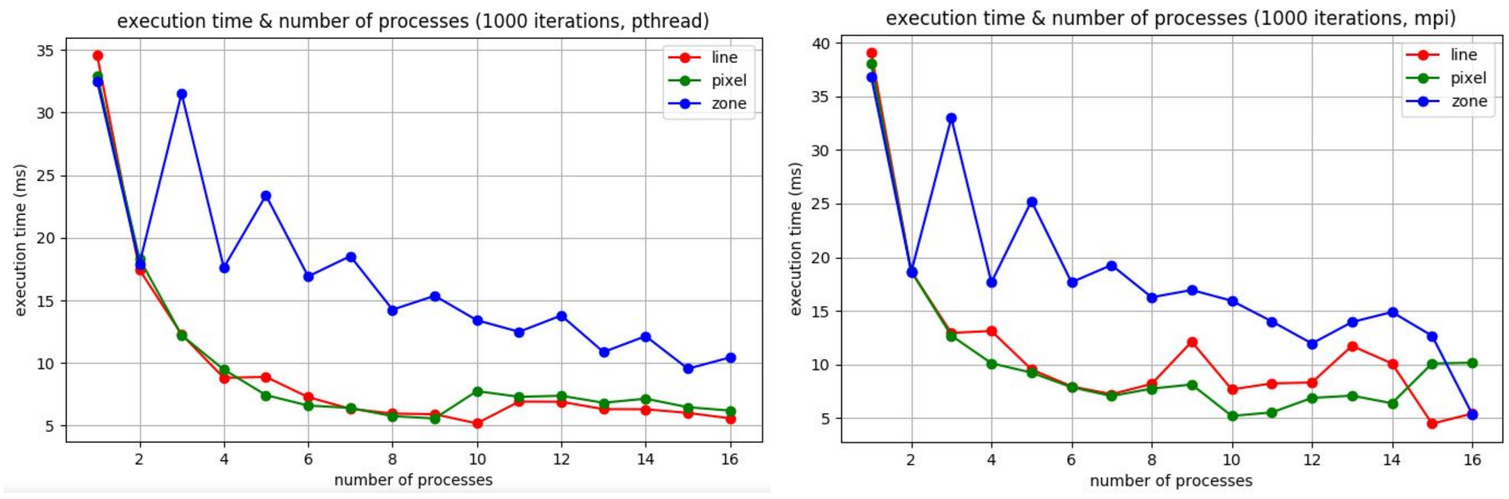
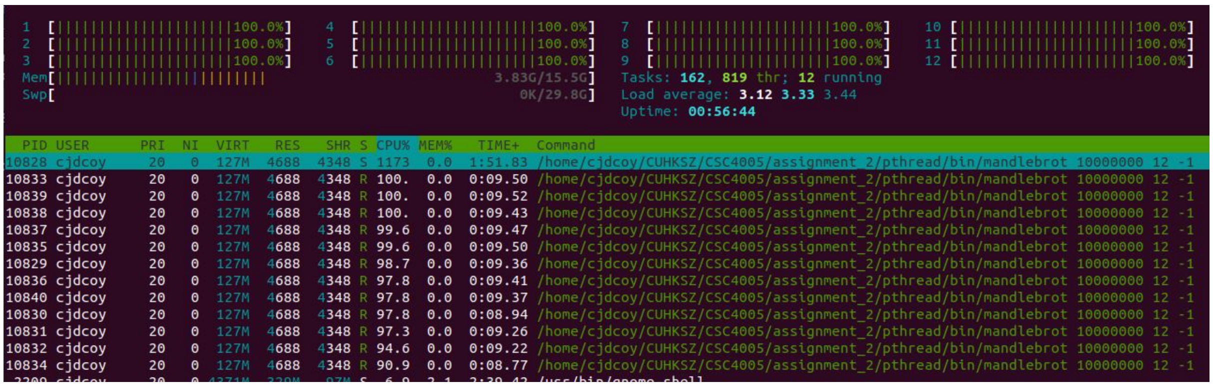


Figure 3 - pthread vs mpi ,1000 iterations (line, pixel & zone partitioning)

Even though the execution time is really fast with 1000 iterations (40 ms) we can already see some recurrent pattern appear. We can see that the zone partitioning is far from being as competitive as the other partitioning methods.

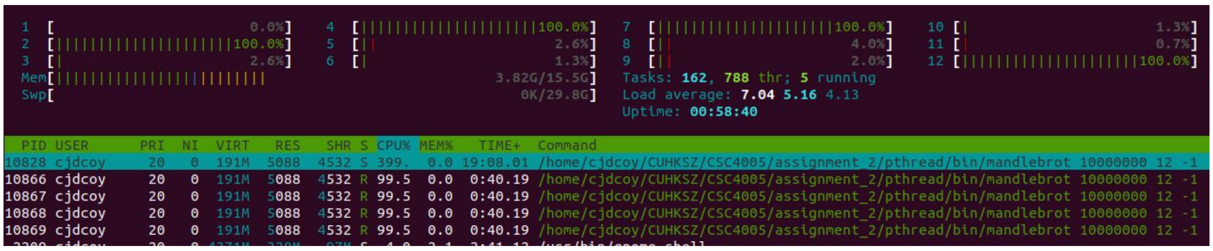
Concretely, what’s happening, why’s the zone partitioning so slow ?



```
  1  [||||||||||||||||||||100.0%]    4  [||||||||||||||||||||100.0%]    7  [||||||||||||||||||||100.0%]    10 [||||||||||||||||||||100.0%]
  2  [||||||||||||||||||||100.0%]    5  [||||||||||||||||||||100.0%]    8  [||||||||||||||||||||100.0%]    11 [||||||||||||||||||||100.0%]
  3  [||||||||||||||||||||100.0%]    6  [||||||||||||||||||||100.0%]    9  [||||||||||||||||||||100.0%]    12 [||||||||||||||||||||100.0%]
  Mem[||||||||||||||||||||||||                   3.83G/15.5G]   Tasks: 162, 819 thr; 12 running
  Swp[                                              0K/29.8G]   Load average: 3.12 3.33 3.44
                                                                Uptime: 00:56:44

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
10828 cjdcoy     20   0  127M  4688  4348 S 1173  0.0  1:51.83 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10833 cjdcoy     20   0  127M  4688  4348 R 100.  0.0  0:09.50 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10839 cjdcoy     20   0  127M  4688  4348 R 100.  0.0  0:09.52 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10838 cjdcoy     20   0  127M  4688  4348 R 100.  0.0  0:09.43 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10837 cjdcoy     20   0  127M  4688  4348 R 99.6  0.0  0:09.47 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10835 cjdcoy     20   0  127M  4688  4348 R 99.6  0.0  0:09.50 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10829 cjdcoy     20   0  127M  4688  4348 R 98.7  0.0  0:09.36 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10836 cjdcoy     20   0  127M  4688  4348 R 97.8  0.0  0:09.41 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10840 cjdcoy     20   0  127M  4688  4348 R 97.8  0.0  0:09.37 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10830 cjdcoy     20   0  127M  4688  4348 R 97.8  0.0  0:08.94 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10831 cjdcoy     20   0  127M  4688  4348 R 97.3  0.0  0:09.26 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10832 cjdcoy     20   0  127M  4688  4348 R 94.6  0.0  0:09.22 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10834 cjdcoy     20   0  127M  4688  4348 R 90.9  0.0  0:08.77 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
```

Figure 4 - CPUs usage on mandlebrot set using pthread, 12 threads and line partitioning (1 000 000 iterations)



```
  1  [                       0.0%]    4  [||||||||||||||||||||100.0%]    7  [||||||||||||||||||||100.0%]    10 [|                      1.3%]
  2  [||||||||||||||||||||100.0%]    5  [||                     2.6%]    8  [||                     4.0%]    11 [|                      0.7%]
  3  [|                      2.6%]    6  [|                      1.3%]    9  [||                     2.0%]    12 [||||||||||||||||||||100.0%]
  Mem[||||||||||||||||||||||||                   3.82G/15.5G]   Tasks: 162, 788 thr; 5 running
  Swp[                                              0K/29.8G]   Load average: 7.04 5.16 4.13
                                                                Uptime: 00:58:40

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
10828 cjdcoy     20   0  191M  5088  4532 S 399.  0.0 19:08.01 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10866 cjdcoy     20   0  191M  5088  4532 R 99.5  0.0  0:40.19 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10867 cjdcoy     20   0  191M  5088  4532 R 99.5  0.0  0:40.19 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10868 cjdcoy     20   0  191M  5088  4532 R 99.5  0.0  0:40.19 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
10869 cjdcoy     20   0  191M  5088  4532 R 99.5  0.0  0:40.19 /home/cjdcoy/CUHKSZ/CSC4005/assignment_2/pthread/bin/mandlebrot 10000000 12 -1
```

Figure 5 - CPUs usage on Mandlebrot set using pthread, 12 threads and zone partitioning (1 000 000 iterations)

As you can see with the line and pixel partitioning the workload is well assigned: during 99% of the computing time all the threads/processes are working. Whereas, with the zone partitioning the workload’s badly assigned. On the figure 5 we can see that 8 threads finished their work while 4 others are still computing the mandlebrot. The spikes in the graphic correspond to the thread/process that’s computing the middle pixels (which are the longest to compute since they contain the most iterations.

100 000 & 1 000 000 iterations

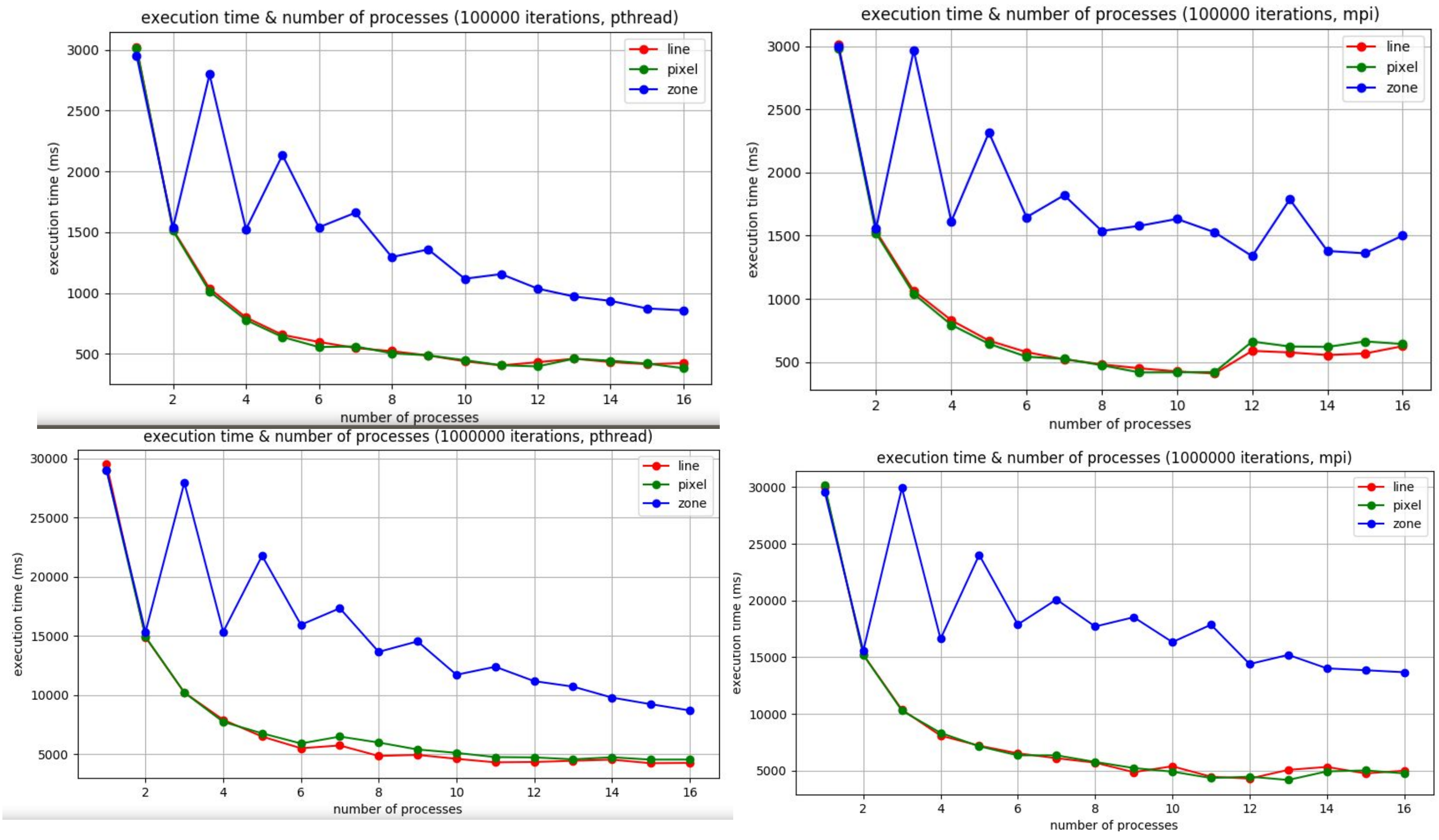
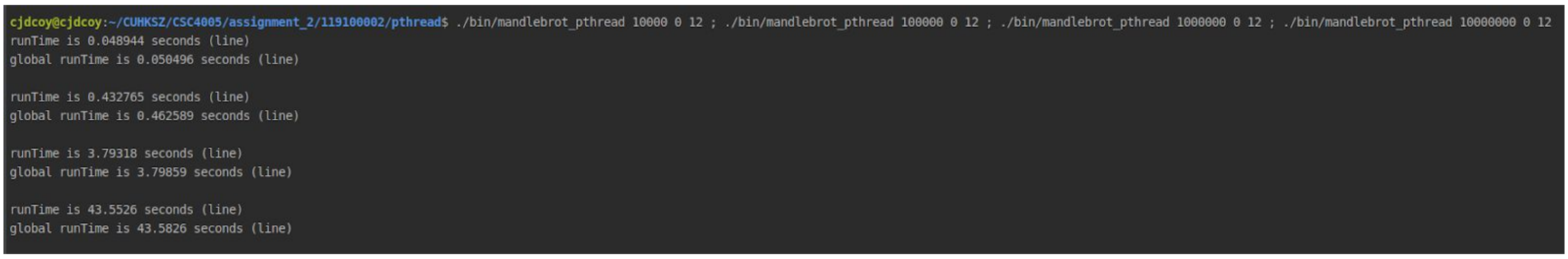


Figure 6 - pthread vs mpi ,many iterations (line, pixel & zone partitioning)

On those last figues we can see that the Mandelbrot's scaling almost perfectly: the computing time grows as the number of iterations grows (3000ms of 100 000 iterations, 30 000ms for 1 000 000 iterations with a single core/process). The computing time's pretty much the same of both MPI and PTHREAD version which's due to the threads/processes working individually.

Mandelbrot runtime vs global runtime



```
cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/assignment_2/119100002/pthread$ ./bin/mandlebrot_pthread 10000 0 12 ; ./bin/mandlebrot_pthread 100000 0 12 ; ./bin/mandlebrot_pthread 1000000 0 12 ; ./bin/mandlebrot_pthread 10000000 0 12
runTime is 0.048944 seconds (line)
global runTime is 0.050496 seconds (line)

runTime is 0.432765 seconds (line)
global runTime is 0.462589 seconds (line)

runTime is 3.79318 seconds (line)
global runTime is 3.79859 seconds (line)

runTime is 43.5526 seconds (line)
global runTime is 43.5826 seconds (line)
```

Figure 7 - 10 000 to 10 000 000 iterations, mandelbrot + overall runtime

Beside the mandelbrot that's fully parallelizable there are few parts that we cannot parallelize sur as verifying the arguments, compute partitioning and drawing the image. From the image above we can see that those non-parallelizable features take a constant time O(1) which's around 0,02 seconds (20 ms).

Performance analysis

First, if we focus on the Mandelbrot set computation there's no communication between processes with pthread and the pixel's information gathering take such a small time in the MPI version that it's negligible. Due to that, the mandelbrot set is highly scalable which means that the time to compute the Mandelbrot will grow linearly depending on the number of iterations to perform.

Using Amdahl's Law

we know that $Speedup = \frac{1}{(1-p)+p/N}$

We also know that the non scalable part is constant O(1) and takes around 20ms so let's assume that the program is 99% parallelizable.
Maximum Speedup for the Pthread version on 12 CPUs:

$Speedup\ pthread\ =\ 1/((1-0.99)+0.99/12)$

$Speedup\ pthread\ = 10,81$

For MPI we know that there's also the gathering part that cannot be parallelized and we saw in the first part that it takes around 5ms so let's say that 98% of the program is parallelizable with MPI.

$Speedup\ MPI\ =\ 1/((1-0.98)+0.98/12)$

$Speedup\ MPI\ = 9.84$

Using Gustafson Law

we know that $S = N + (1-N)s$

let's take the same parallelization values for both pthread and MPI.

$Speedup\ pthread\ =\ 12+(1-12)(0.01)$

$Speedup\ pthread\ =\ 11.89$

$Speedup\ MPI\ =\ 12+(1-12)(0.02)$

$Speedup\ MPI\ =\ 11.78$

The results from both laws are definitely good and that it is worth to parallelize the Mandlebrot set. Keep in mind that both laws only gives a very rough estimate. Indeed, there are other key components such as the partitioning that play a big role into a program's parallelization. Just by comparing the different partitioning methods I implemented we ca see that the loss can be huge (27500ms for zone partitioning vs 10 000ms for line and pixel partitioning in figure 6, 3 threads/processes).

Conclusion

During this project the Mandelbrot parallelization was successfully implemented using MPI and Pthread. The performance analysis shows that both version avec decent performance upgrade on the sequential and that they also have a pretty similar runtime which is expected since the program's 99% parallelized.

During the last project I used pthread as a bonus but I didn't have the time to experiment much with it. Even though we didn't use mutex/semaphores/atomic variable in this project I know that it definitely improve my capacity to use parallelization libraries. Moreover I think that the performance analysis will help me in the future to choose the right algorithm/way to iterate, depending on its complexity.