Fabien Labarbe-Desvignes
nov 21 2019

Assignment 4, heat simulation

Summary

Overview of the project

The goal of this project's to create three programs that have for main goal to parallelize a heat simulation problem. The programs are written in C++ (14), they use pthread, openmp and mpi 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 heat simulation and Jacobi iteration ?

Temperature differences between surfaces and the ambient temperature cause hot air particles to move and therefore transfer **heat**. Using a **heat simulator**, a designer can identify the **heat** transfer amount which is usually calculated as surface temperature minus ambient temperature.

In numerical linear algebra, the Jacobi method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations. Each diagonal element is solved for, and an approximate value is plugged in. The process is then iterated until it converges. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization. The method is named after Carl Gustav Jacob Jacobi.

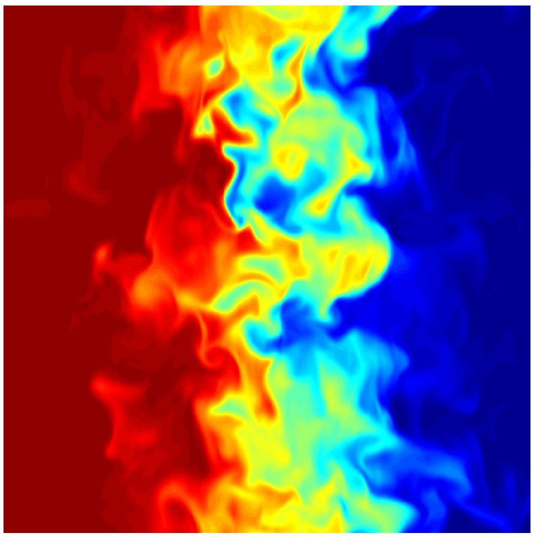


Figure 1 - Heat simulation of fluids

Build & usage

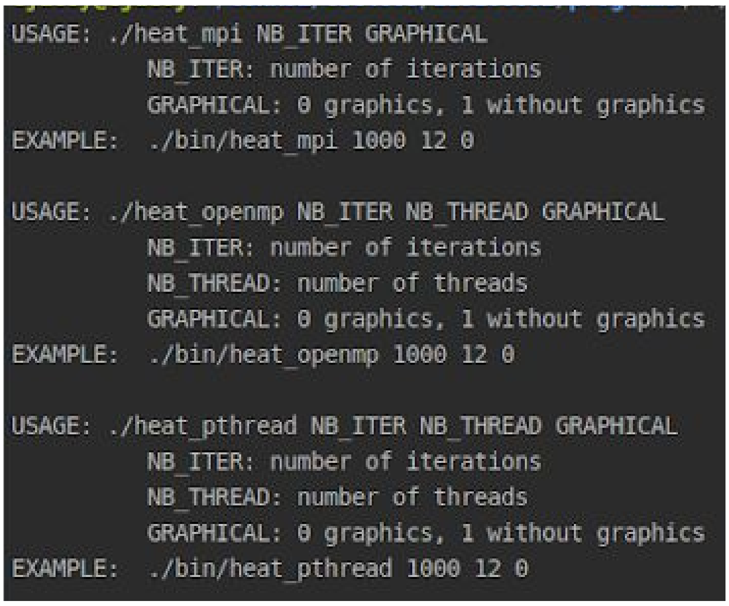
Requirements:
cmake (for compilation), pthread, mpi

How to build the pthread / mpi version?
1: go to 119100002/pthread etc...
> 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.

Usages



```
USAGE: ./heat_mpi NB_ITER GRAPHICAL
          NB_ITER: number of iterations
          GRAPHICAL: 0 graphics, 1 without graphics
EXAMPLE:  ./bin/heat_mpi 1000 12 0

USAGE: ./heat_openmp NB_ITER NB_THREAD GRAPHICAL
          NB_ITER: number of iterations
          NB_THREAD: number of threads
          GRAPHICAL: 0 graphics, 1 without graphics
EXAMPLE:  ./bin/heat_openmp 1000 12 0

USAGE: ./heat_pthread NB_ITER NB_THREAD GRAPHICAL
          NB_ITER: number of iterations
          NB_THREAD: number of threads
          GRAPHICAL: 0 graphics, 1 without graphics
EXAMPLE:  ./bin/heat_pthread 1000 12 0
```

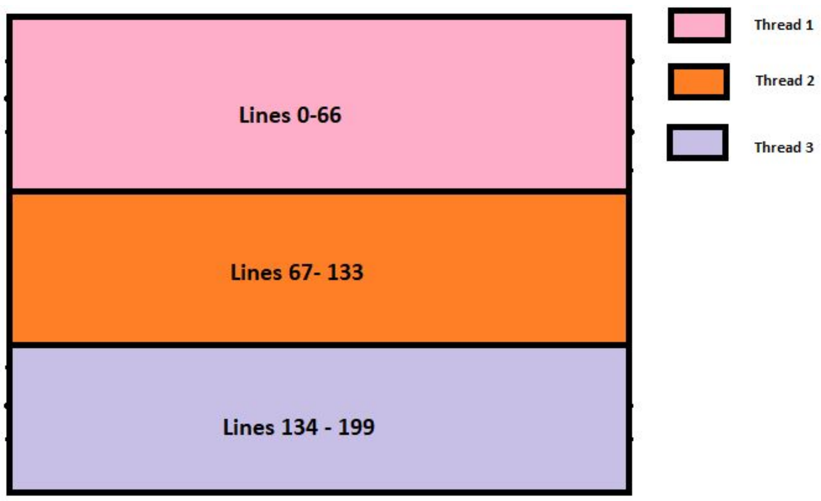
note: videos can be found in the data folder.

Partitioning

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

Zone partitioning

I implemented this partitioning because it's simple to do but still relevant in our case since all the lines got the same computation complexity.
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.

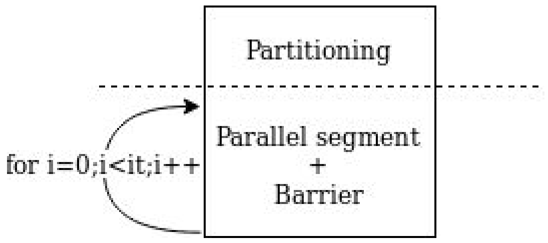
Phase 1:
Fill vectors with positions depending on the partitioning selected.
Those vectors define the line that each thread/process will compute as well as its starting position and the offset it'll do at each computation.

My algorithm can be divided in 2 main phases for the pthread and openMP program:

1. Partitioning
2. Processing + display

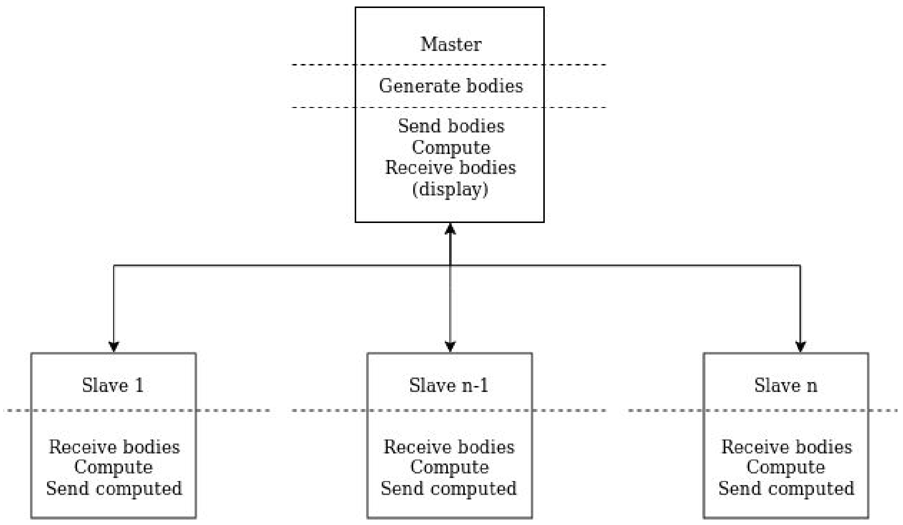
Phase 1:
Generate bodies with random values and add them to a vector in order to handle them more easily and separate the n vectors between the different processes/threads.

Phase 2 (pthread / openmp):
Each thread perform the calculation of its assigned bodies and then wait at a barrier so that the computation ain't rigged by faster threads. The main thread (id 0) display the bodies.



Phase 2 (mpi):

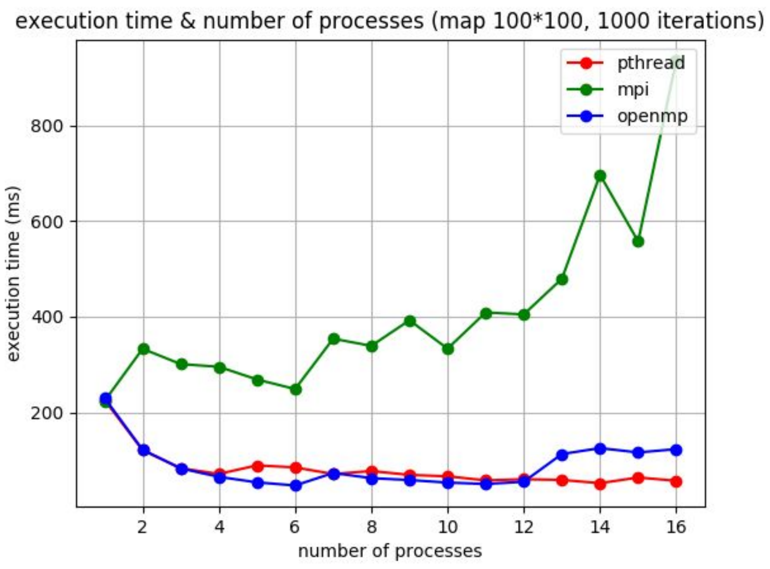
- master send bodies (with broadcast for perf) / slaves receive bodies
- master / slave compute new values for their assigned bodies (in the mpi + openmp version the computation is parallelized with n threads)
- mater receive the computed bodies / slaves send only the bodies they computed (for performance gain)


Master
Generate bodies
Send bodies
Compute
Receive bodies
(display)
Slave 1
Receive bodies
Compute
Send computed
Slave n-1
Receive bodies
Compute
Send computed
Slave n
Receive bodies
Compute
Send computed

Obtained results

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

100*100 map:



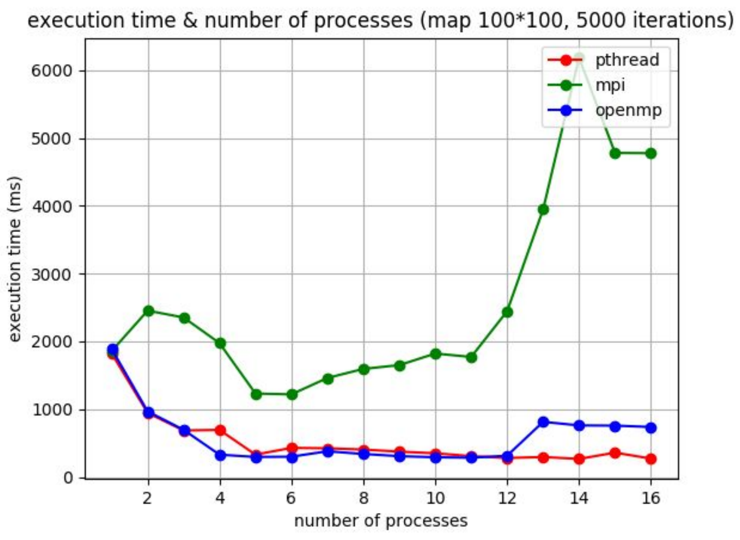
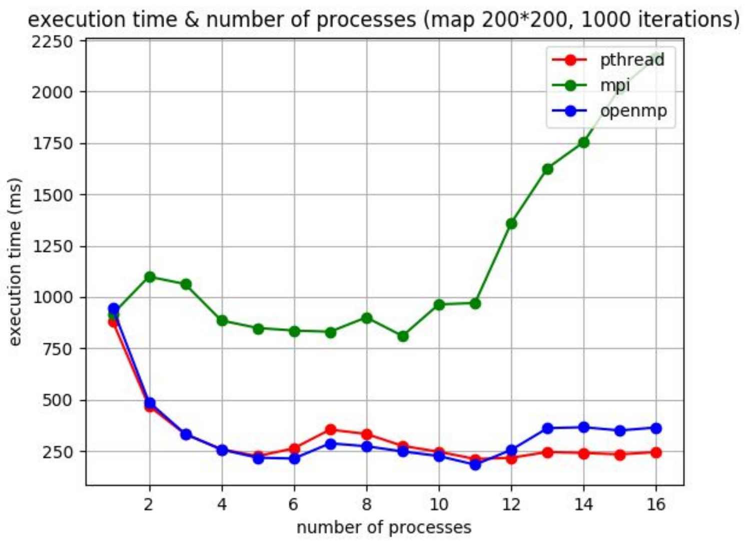


*Figure 2 - 100 * 100 map, 1000 and 5000 iterations*

It's good to note that I didn't uploaded any benchmark below 100*100 because
my benchmarks were too fast so the results weren't relevant.
On this benchmark we can see that both OpenMP and PTHREAD are always faster than the sequential version.
Pthread is always slightly faster than OpenMP above 12 cores because since 12 is the total number of cores on the test machine it got hardware limited. Since OpenMP use processes the load balancing is less well distributed than pthread that divide the utilization of its processes evenly across the core. MPI is almost always slower than the sequential version as it needs to communicate a lot with the main program. The more communication the more the process takes time to execute.

We can note that the programs scale well depending on the number of iterations.

200*200 map:



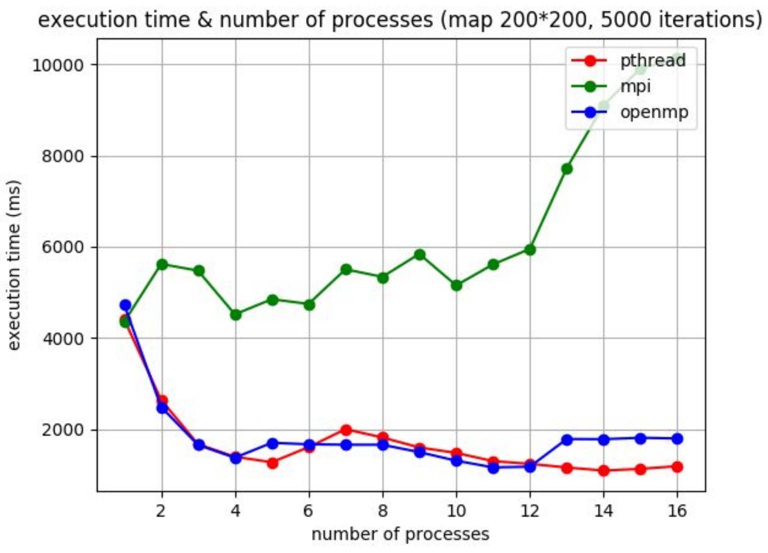


*Figure 2 - 200 * 200 map, 1000 and 5000 iterations*

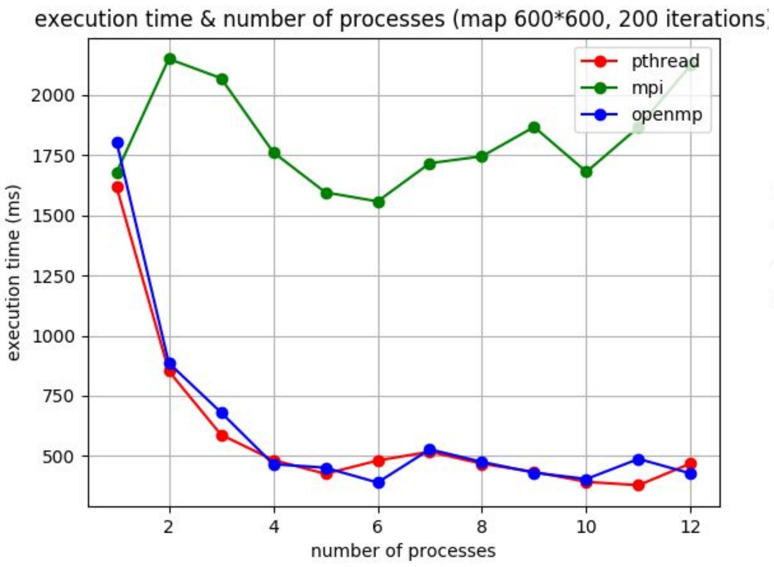
Here again, we can see that the program is scalable, the runtime and the number of iterations are proportional. When under 12 cores (my physical limit) the different parallelization methods are taking almost the exact same amount of time which's what I expected to happen.

Since pthread and openmp are implemented the same way and the results are still pretty much the same except for the load balancing as explained earlier.

Why are MPI versions Slower after 12 cores ?

There are a lots communications between each steps with mpi because on each iteration we need to send the lines that are around our actual thread zone, it makes this parallel version a lot worse than the other ones.

600*600 map:



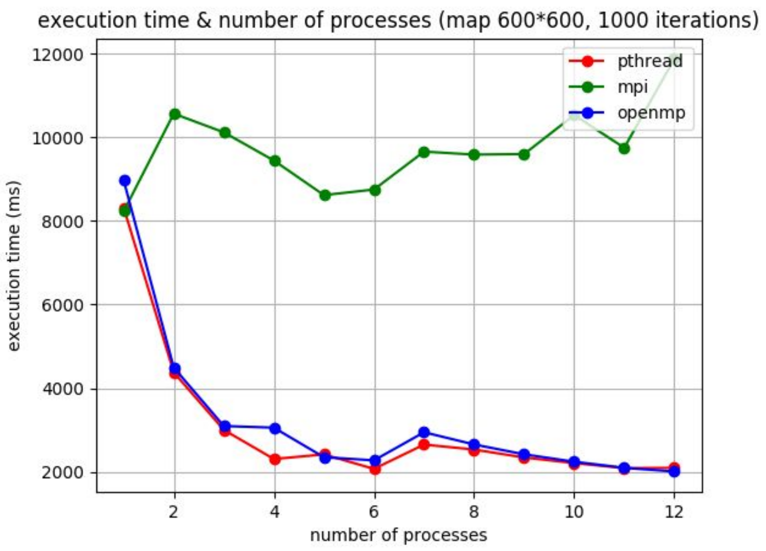
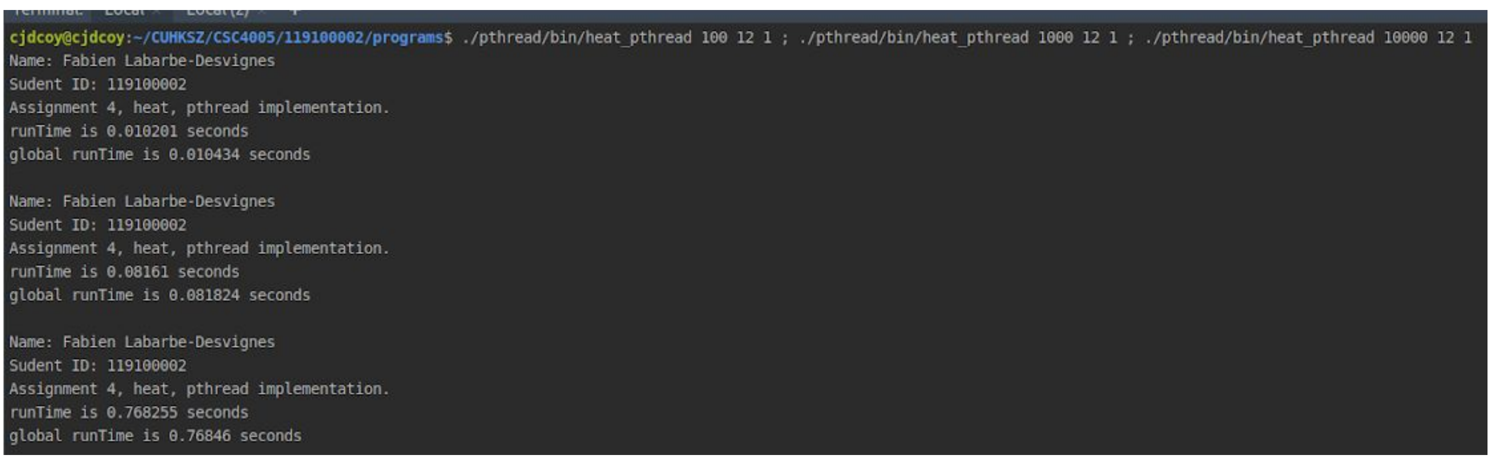


*Figure 3 - 600 * 600 map, 200 and 1000 iterations*

Here again we can see that both the mpi and openmp implementation got a behavior that's still the same. The scaling in really good but mpi's still not worth using, the communications are way to frequent and represent too many different slowdown areas.

jacobi heat computation runtime vs global runtime



```
cjdcoy@cjdcoy:~/CUHKSZ/CSC4005/119100002/programs$ ./pthread/bin/heat_pthread 100 12 1 ; ./pthread/bin/heat_pthread 1000 12 1 ; ./pthread/bin/heat_pthread 10000 12 1
Name: Fabien Labarbe-Desvignes
Sudent ID: 119100002
Assignment 4, heat, pthread implementation.
runTime is 0.010201 seconds
global runTime is 0.010434 seconds

Name: Fabien Labarbe-Desvignes
Sudent ID: 119100002
Assignment 4, heat, pthread implementation.
runTime is 0.08161 seconds
global runTime is 0.081824 seconds

Name: Fabien Labarbe-Desvignes
Sudent ID: 119100002
Assignment 4, heat, pthread implementation.
runTime is 0.768255 seconds
global runTime is 0.76846 seconds
```

Figure 7 - 100 to 10 000 iterations, heat + 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 0,0002 seconds (200 us).

Performance analysis

First, if we focus on the heat computation there's no communication between processes with pthread and openmp the pixel's information gathering take such a small time that it's negligible. Due to that, the heat computation set is highly scalable which means that the time to compute the heat 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.

$$Speedup\ =\ 1/((1-0.9998)+0.98/12)$$
$$Speedup\ = 12,215$$

Using Gustafson Law

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

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

$$Speedup\ =\ 12+(1-12)(0.0002)$$
$$Speedup\ =\ 11,9978$$

The results from both laws are definitely good and that it is worth to parallelize the heat computation with jacobi iteration. 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.

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.