Iowa State University - COM S 424
High perfomance computing project report

Yann DROY

December 8, 2017

Abstract

This document presents the high performance computing project we had to do for the COM_S 424 class at Iowa State University. My program uses parallelism to find the limits of tectonic plates on the surface of a geoid model. We will first see how the computing is done then we will see the two parallelization strategies that can be used to have a better output time. Finally, after showing some results, we will study the performances achieved with different parameters and with two different machines.

Contents

1 Introduction

1.1 Chosen topic

The chosen subject for this project is a geology-related subject. It deals with the computing of tectonic plates limits on the surface of a model, which can obviously be the Earth or any random geoid model created by a simulator. The geoid model is described in a file which contains a list of points located on its surface. Because of the tectonic activity of models, the points at the surface are moving according a given direction, which is the direction of the plate itself they are located on. Thus each point is defined by 4 values:

1. latitude
2. longitude
3. velocity towards the North
4. velocity towards the East

Latitude and longitude give us the point location, and the vector V defined by $V = [v_{north}; v_{east}]$ tells us how the point is moving on the geoid surface.
With just these informations we are able to compute the plate limits of each model, supposing that we have enough points to be precise. It is important to have close to each other points, so we must have a lot of points to be accurate.

1.2 Detailed computing

1.2.1 Prerequisites

Since each point has 2 coordinates: latitude and longitude, instead of dealing with a roughly spheric model (we assume that all the points are at the same elevation, sea level), we can see the model as a matrix M with n rows and m columns where n is the number of different latitudes (horizontal lines from North pole to South pole, so from 90° to -90°, we call them *parallels*) and m the number of different longitudes (vertical lines from longitude -180° to longitude 180°, we call them *meridians*).
It is hence crucial that there is the same number of points on each parallel, otherwise the model can not be reduced as a rectangular matrix.

1.2.2 Algorithm

Once the data file containing the model is loaded, the algorithm to compute the plates limits is quite simple. Basically what it does is comparing all the close points by pair. The program loops through the matrix and for each point of the matrix, it will compare it with the point next to it in the East direction, and the next point in the South direction. For each pair, it computes the velocity difference and if this difference is over a given cutoff, it means the points are probably not on the same plate. There are 3 different cases: either the points are on the same plate; or if they are not, there are two cases whether the point are getting closer to each other or if they are splitting apart. For each of these cases, We set an appropriate value to the cell of the first point of the pair, which will be considered the limit (arbitrarely, because we can not set the limit

between the two points). So if $M[i][j]$ and $M[i][j+1]$ are not on the same plate, $M[i][j]$ will be set to be the limit.
At the beginning, the matrix is initialized with all the points on the same plate. Here is the pseudo-code of the algorithm:

```
// for all the points
for i from 0 to n:
  for j from 0 to m:
    // East velocity difference first
    diff = (V_east of M[i][j]) - (V_east of M[i][(j+1)%j])

    // cutoff testing
    if diff > 0 and diff > cutoff:
      M[i][j] = CONVERGENT
    else if diff < 0 and |diff| > cutoff:
      M[i][j] = DIVERGENT

    // then South velocity difference if not already a limit
    if(M[i][j] == DEFAULT){
      // preventing out of bonds
      if(j + 1 < m){
        diff = (V_north of M[i][j]) - (V_north of M[i+1][j])

        if diff > 0 and diff > cutoff:
          M[i][j] = DIVERGENT
        else if diff < 0 and |diff| > cutoff:
          M[i][j] = CONVERGENT
      }
    }
```

Where `DEFAULT`, `CONVERGENT` and `DIVERGENT` are constant values in the program.

1.2.3 Getting the best result

Because the cutoff is a parameter selected by a human being, depending on the model, the cutoff to have the most representative result is not always the same For example, if all the points are barely moving, they could all be on the same plate if the cutoff is high. However, if the points are moving very fast, the whole geoid could be covered with plate limits with a too low cutoff. Since we, humans, can not know in advance what would be the appropiate cutoff for a given model, the program will compute the limits for different cutoffs: from a low bound to a higher bound with a given number of computation. These 3 parameters are however entered by the user or are set to default ones that I chose in the program.
The algorithm shown above is included in a `for` loop that manages the different cutoffs.

1.3 Result visualization

For each plate limits computation (*ie* for each cutoff), the program produces an output of the model in a file, with the points positions and their "value".

This file is readable by GMT (Generic Mapping Tool) which can process the file to create an image of the computation in which you can see the model and the limits. This output is way better and understandable that the one in the console. You can then compare the obtained results with the the actual model. For each example file (except for Earth), an image is furnished to do the comparison.

2 Parallelization strategy

Even if the code of our program to get the plates limits result is pretty short, there are ways to parallelize it because it uses a relatively "big" amout of data and processes it with loops.
The solution used to parallelized the code is **OpenMP**. OpenMP lets us use preprocessor directives to parallelize our code.
There are two main ways to parallelize the program. We will see each one in detail here.

2.1 Cutoff parallelization

The first parallelization strategy is to parallelize each whole plates limits computation according to the different cutoffs. Each created thread will be assigned a cutoff to work with and will run the velocities differences function on the whole matrix, going through all the points. With this strategy, at each time, there are n threads processing the entire matrix. The figure below summarizes the program behavior with cutoff parallelization. This is an example for 4 threads and 4 cutoffs. As you can see, thread 0 (leftmost) deals with cutoff 1, thread 1 is dealing with cutoff 2, thread 2 is dealing with cutoff 3 and thread 3 is dealing with cutoff 4. As we can also see, the threads are "long", because they process all the matrix.

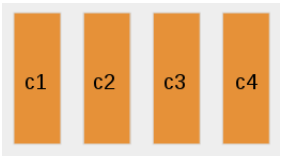

Figure 1: Example of cutoff parallelization for 4 threads and 4 cutoffs

During this paper, I will sometimes refer to cutoff parallelization with a "-c" notation.

2.2 Matrix parallelization

The second parallelization strategy is to let the main thread handle itself the cutoff loop and to parallelize the matrix process for each cutoff. For each new cutoff, when it comes to process the matrix, n threads are created and each one will go through a part of the matrix. With this strategy, at each time, there are n threads processing a small part of the matrix, but these threads will be created again for each cutoff. The figure below this time summarizes the program behavior for matrix parallelization. Again, this is an example for

4 threads and 4 cutoffs. As you can see on this one, the cutoffs are treated one after the other, and the matrix is split between the threads. So for cutoff 1, all the threads compute the output on one quarter of the matrix, same for cutoff 2, 3 and 4. As we can also see, the threads are shorter, because they indeed just use a quarter of the matrix.

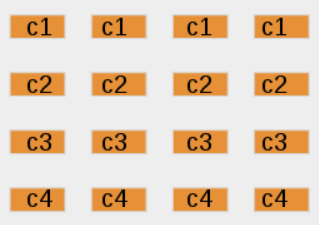

Figure 2: Example of cutoff parallelization for 4 threads and 4 cutoffs

During this paper, I will sometimes refer to cutoff parallelization with a "-m" notation.

2.3 Parallelization parameters

The goal is to determine what is the best strategy to use for running the program. For both strategies, since we are dealing with `for` loops, we can play with the scheduling of threads and the assigned chunk sizes and observe the different results we obtain for the different program parameters. We can also see the difference between the program running without any parallelization (*ie* just 1 thread) and the program running with multiple threads. During the performance analysis section, we will see how we can impact the program throughput using OpenMP environment variables or the parallelization parameters, and hence we will answer that question.

3 Results

In this section I will show you the output of my program to actually see what it does. No matter if we use the cutoff or matrix parallelization, the obtained result is the same, because we are using the same algorithm. Below is a couple examples of our Earth, with 3 different cutoffs.

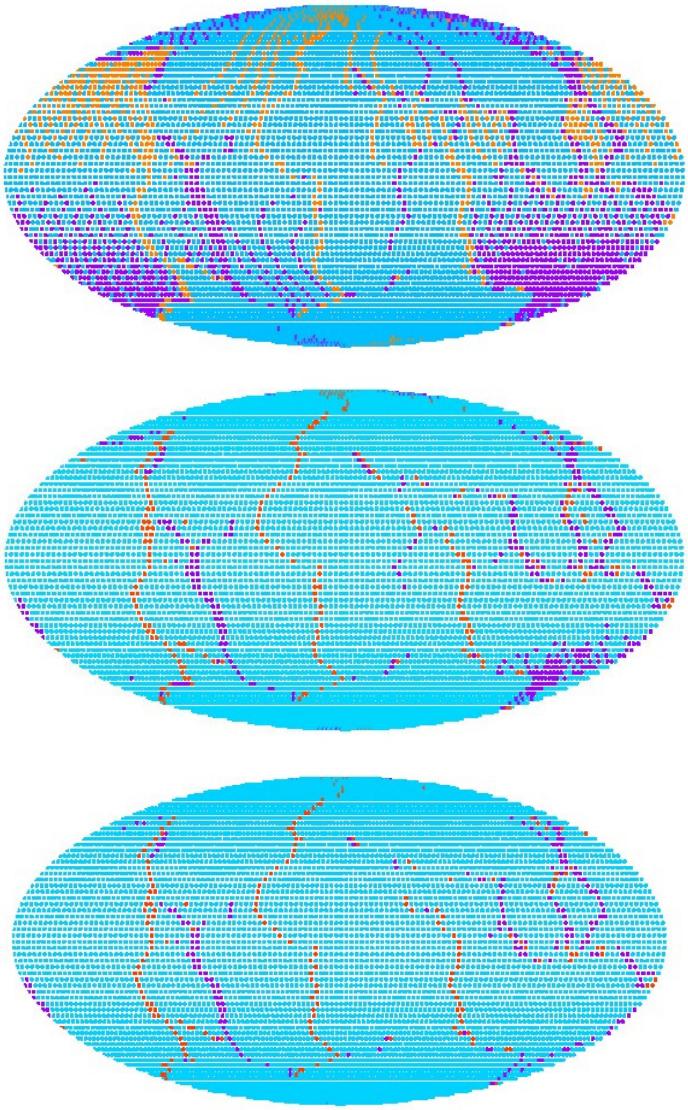

Figure 3: Earth visualization output with cutoff=5, 10 and 21.0

We can see that the first one seems to low, because we have plenty of plates limits that sould obviously not be here, the second one seems good but we still have a lot of subduction points near Australia, the third one though gives a really decent result. Below are two other examples, run on software created geoid models. Pictures with green background are the expected output.

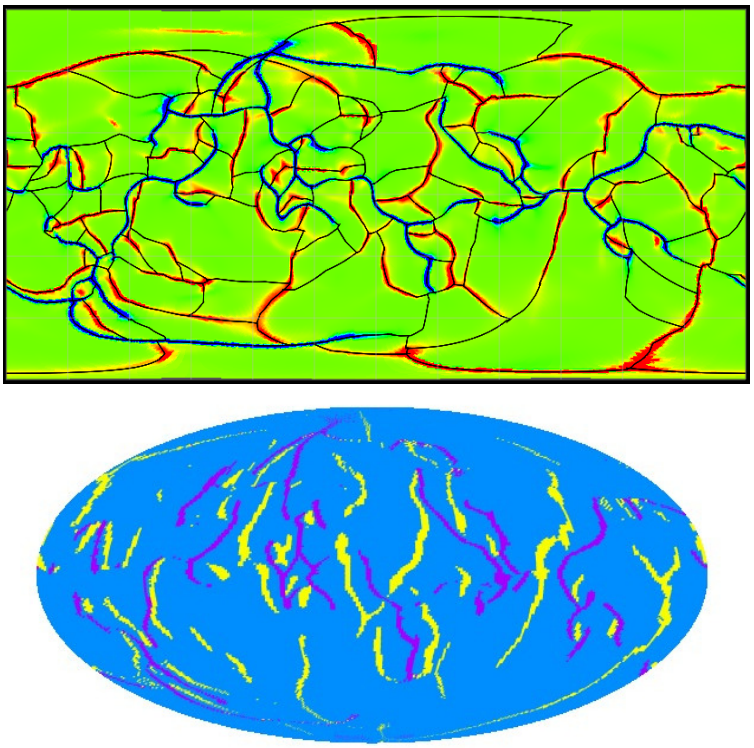

Figure 4: Model 1 visualization expected and obtained output

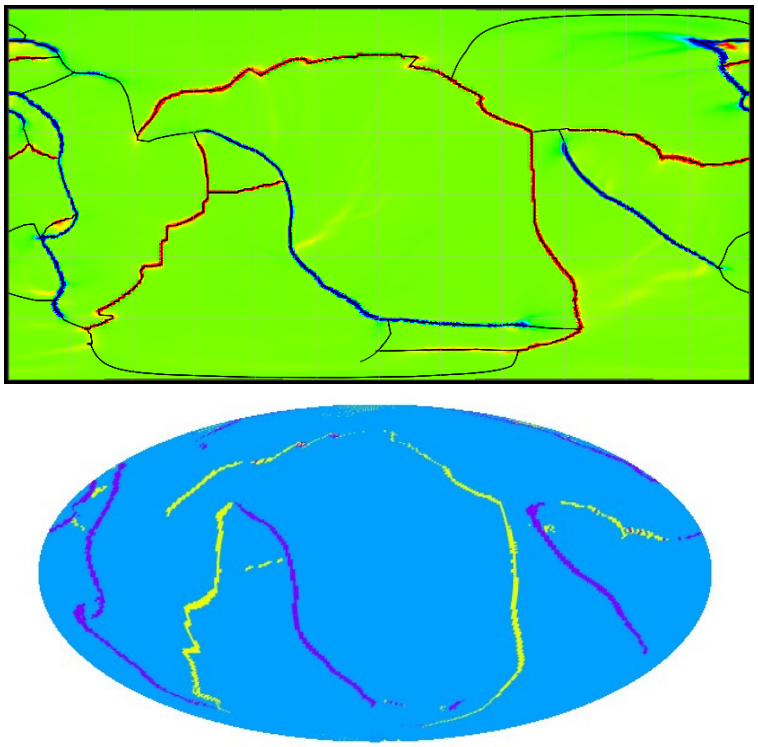

Figure 5: Model 2 visualization expected and obtained output

As you can see, the results are pretty accurate.

4 Performance analysis

4.1 Introduction

So we already have two different output times ans performances to analyze: cutoff and matrix parallelization. But I did not only run the program on a single machine: I used two different ones. One of them is my personal machine, the other one is a high performance computing compute node that we can use thanks to HPC class. This allowed me to make like a cross-analysis of the performances: -c on my machine, -c on HPC node, -m on my machine and -m on HPC node. In this section I will be putting a lot of screenshots to prove my results, and I would like to apologize for the language set to French even if the content is not quite hard to understand.

4.2 CPU architectures

The two machines obviously don't have the same architecture, it would have been useless if they did. Here are two screenshots with a short description of the two architectures uses and what are the differencies. I removed the non relevant lines.

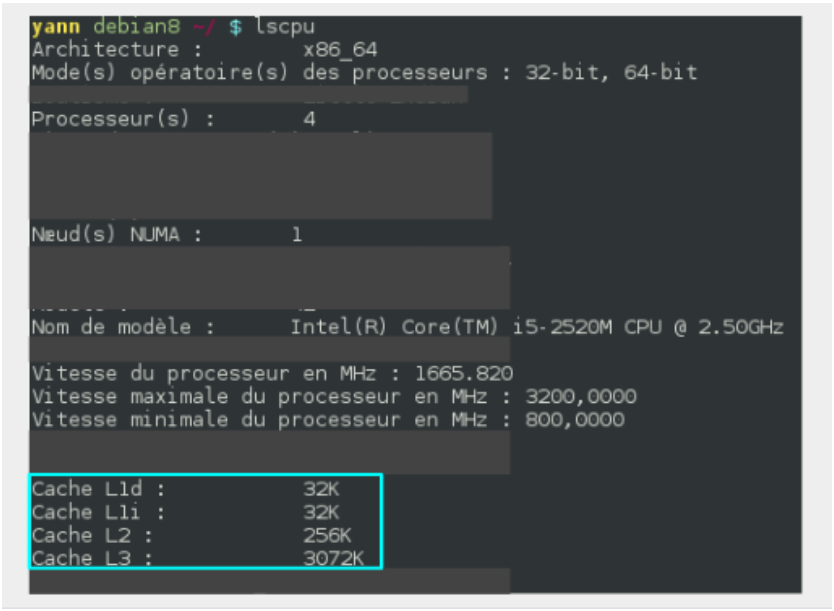

Figure 6: My machine CPU architecture

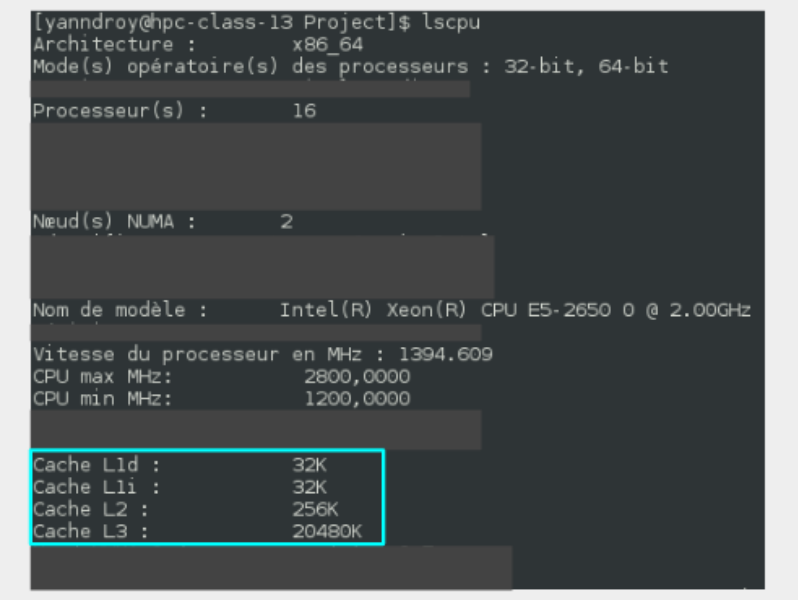

Figure 7: HPC node CPU architecture

As we can see, they both support 64-bit architecture and are embedded with Intel processors. My has an i5 and HPC node has a Xeon. My processor seem to have a better frequency but the difference is not very big, however my machine has only 4 cores and the HPC node has 16, which is 4 times more. I have 1 NUMA node and HPC has 2. But the really big difference between these two machines is concerning their caches. Even though they both have the same L1

cache (instruction and data) and the same L2 cache, HPC node machine has an L3 cache of 20 MB, which is more than 6 times the size of my L3 cache which is only 3 MB. This will for sure have an impact on cache misses while the program is running.

4.3 Cache sizes impact on performances

For this comparison, I ran the program 4 times:

1. -c on my machine
2. -c on HPC node
3. -m on my machine
4. -m on HPC node

The table below summarizes the output that I have got from `valgrind` with the `cachegrind` tool that analyzes caches misses. The output only gave me informations about L1 (instructions and data) and L3.

	My machine	HPC node
-c	0.00001% L1 instruction misses 0.14% L1 data misses 41.3% L3 cache misses	0.00001% L1 instruction misses 0.11% L1 data misses 0.71% L3 cache misses
-m	0.00001% L1 instruction misses 0.18% L1 data misses 95.2% L3 cache misses	0.00001% L1 instruction misses 0.13% L1 data misses 0.53% L3 cache misses

Figure 8: Cache misses summary table

As we can see in the table, the L3 cache really makes the difference. Although there is a first huge difference between -c and -m on my machine, with -m more than doubling L3 cache misses raising it to almost 100% (on HPC node it is quite the same), the difference between L3 cache misses on my machine and the HPC node are insane: for -c we have nearly 58 times more cache misses on my machine, and for -m it is nearly 180 times more. And I recall that L3 on HPC node is "only" 6 times bigger.

4.4 CPU affinity and binding

4.4.1 Affinity

CPU affinity spedifies on which CPU a thread has to run. With CPU affinity, we can control on how many CPUs are running the program and how many threads on each CPU are running simultaneously. CPU affinicty is an OpenMP environment variable, so it is a string that represents the affinity. For example, "0" means that all the threads will be run on CPU 0, and if we have 4 threads, "0 1 0 1" means that threads 0 and 2 will be run on CPU 0 and threads 1 and

3 on CPU 1. Below is a screenshot that shows that affinity affects the output time. This was tested on the HPC compute node. I tried with -c and -m and with 3 different CPU affinities with 8 threads.

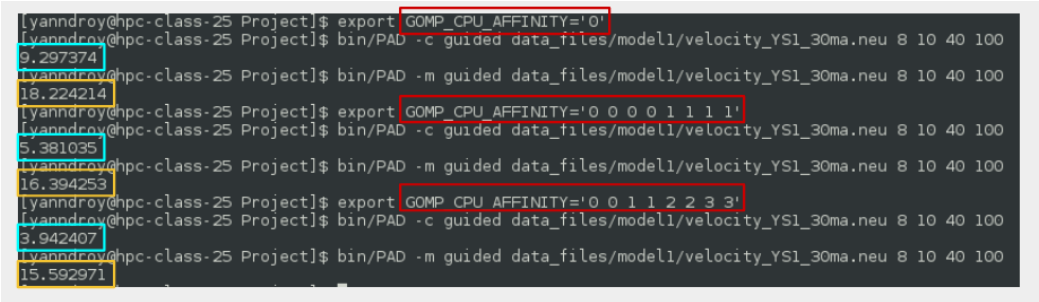

Figure 9: Some output times with different CPU affinities

We can already see that it looks like -c is way more effective than -m. But anyways we can see that the more we use cores, the better is the output time. Below is a table with average output times over 10 runs, with 16 threads and 4 different CPU affinities, from all the threads running on one CPU, to all threads on a different CPU. The program was run with the same parameters: -c, static scheduling, 100 cutoffs; on the HPC compute node.

CPU Affinity	Average output time
All threads on CPU 0	9.32s
8 threads on CPU 0 and 8 on CPU 1	5.22s
4 threads for each CPU from 0 to 3	3.64s
2 threads for each CPU from 0 to 7	3.48s
Each thread run on a different CPU	3.11s

Figure 10: Average output times for 16 threads and different CPU affinities

So these tests were run with 8 and 16 threads, which is still less or equal to the number of cores. What happens if we specify more threads than cores, also with different CPU affinities ? I tried that, this time on my machine, with 4 threads and 3 different affinities (all threads on one CPU, 2 thrads on 2 CPUs and on thread on each) and then with 8 threads. Again, -c, static scheduling and 100 cutoffs. 3 computations to have an average.

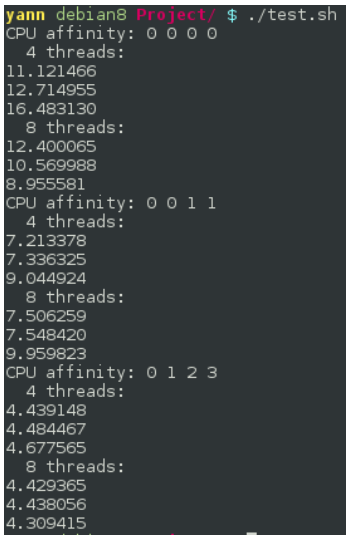

Figure 11: CPU affinities with 4 and 8 threads on my machine

As we can see, affinities affect the output time and we have better results when more CPUs are used, but we can also see that we obtain the same results for 4 and 8 threads for the same affinities.

4.4.2 Binding

CPU binding is also an environment variable, which this time can take only two different values: true or false. CPU binding specifies if threads can move between cores during process scheduling. If set to true, they can, otherwise they can not. To run the binding test I had first to unset the CPU affinity environment variable of course. Below is a screenshot of the obtained results for both machines. I ran the program with both -c and -m, with both variables set to true and false, and with respectively 4, 8, 16, 32 and 64 threads, asking 100 computations with static scheduling. There are 5 runs per test to have an avergae idea.

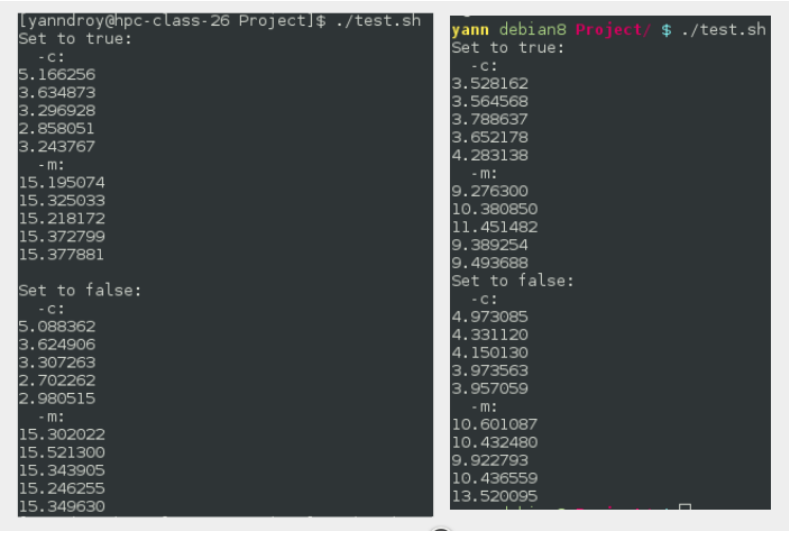

Figure 12: CPU binding results on both machines (left HPC node, right my machine)

This screenshot is really interesting because we have really different results on each machine. On the HPC node, -m is of course longer, but the binding setting does not change anything. However, on my machine, when threads are bound the computing is a bit faster for both -c or -m parallelizations.

4.5 Cutoff vs. Matrix parallelization

You may have already seen that -c seems better that -m, but I want to go more in deep in analyzing the results we have, depending on the number of threads and. I performed these tests on the HPC node because we can use more threads (because it has 16 cores) and hence have better results. I will also change the loop scheduling and chunk sizes to define what is really the faster way to run the program to get the desired output.

4.5.1 Performance

I used the `perf` command to get perfomance informations. Below is a screenshot with the performance output, on average on 10 runs, with the same parameters (guided schedule, 10 threads, 50 cutoffs).

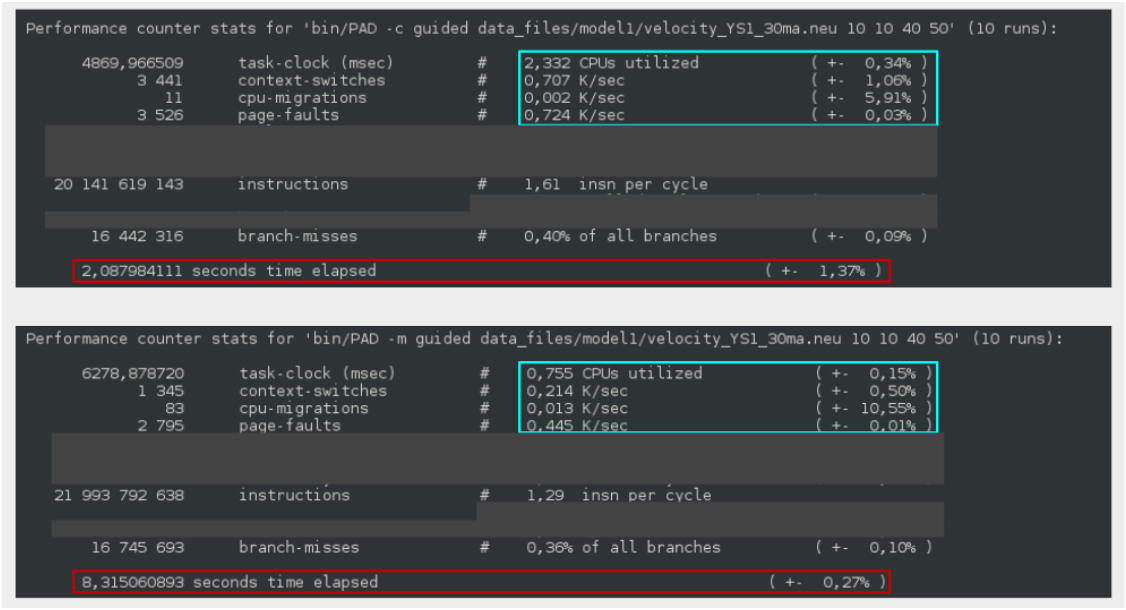

Figure 13: Performance comparison for -c and -m on HPC node

As we can see, -m (on bottom) is 4 times slower than -c (on top), this gives us time information. In the blue rectangle we can find 4 intersting informations. The most relevant one is basically CPU usage, and we can see that -c uses more CPU. It can explains why it is faster. We have more context switches for -c but less page faults.

4.5.2 Loop scheduling

I created a benchmark script that runs the program 100 times with the same parameters, increasing the number of threads from 1 to 100. That, for the 3 different scheduling types: static, dynamic, guided; and for both -c and -m. This gives me 6 different output time files and below are the curves obtained from those files using Gnuplot.

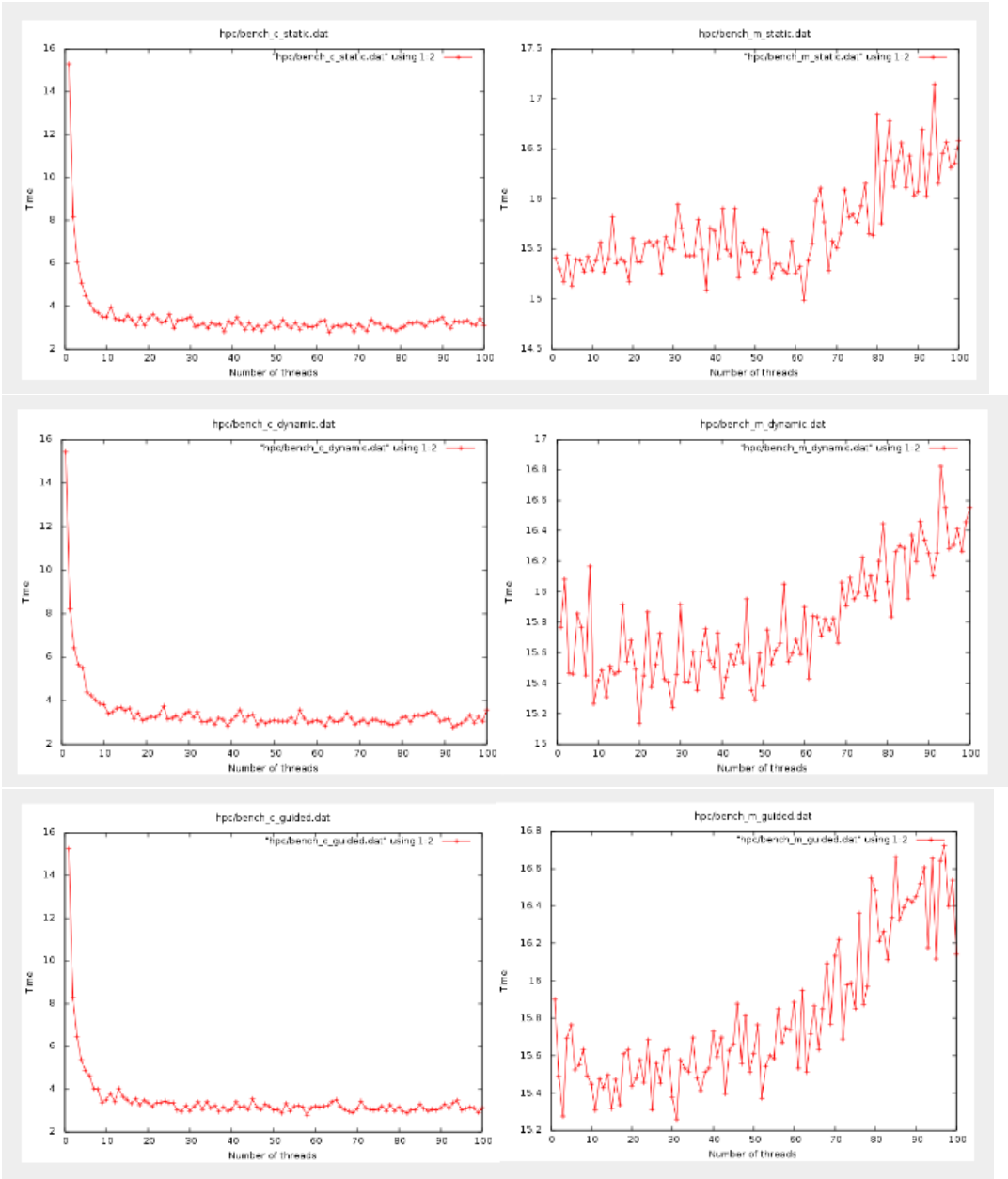

Figure 14: Loop scheduling differences. Top to bottom: static, dynamic, guided; left -c, right -m

We can straight away see that -c is way better overall. For the three schedulings we have pretty much the same curve, which decreases very fast. In the end, we achieve an output time of nearly 3.5 seconds. Guided scheduling seems to be a bit better though. On the other hand -m is really messy. The times are

not stable at all when the number of thread increases but the most surprising thing is that the program goes slower and slower, from around 15 seconds for one thread (same as -c, which is normal) to almost 17 seconds. We can also note that the time difference is not the same for both parallelization methods, we have more than 10 seconds for -c but less than 2 for -m.

4.5.3 Chunk sizes

I created another benchmark script, this time it runs the program 100 times with 100 different chunk sizes (1 to 100) with the given scheduling. The program parameters are: 16 threads and 200 cutoffs. The schedule used for this test is guided, since it is the best one. That was run on the HPC compute node. Below are the curves for both -c and -m.

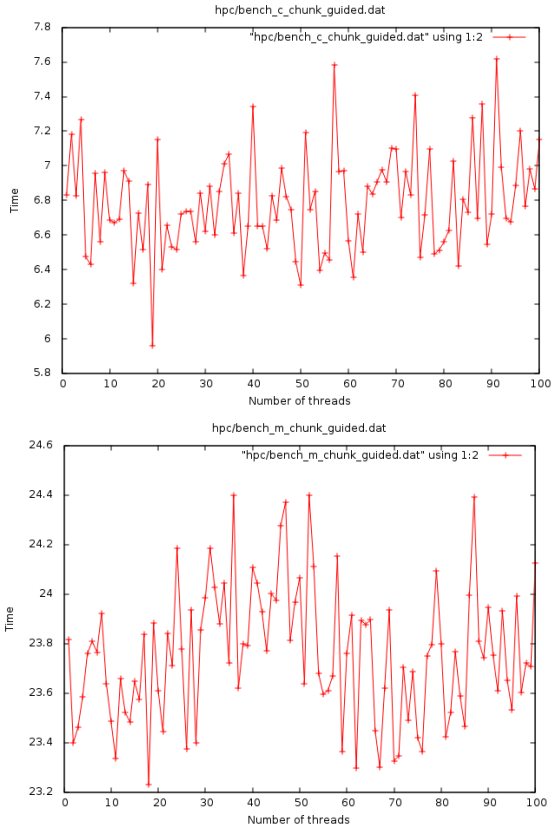

Figure 15: Chunk sizes differences for guided scheduling. Top is -c and bottom is -m

There is not a lot of things to say about chunk sizes, as we can see the chunk have no real effect on the computation time, for any of the schedulings. The output time is nearly 6 or 7 seconds for -c, which is what we could expect since computations for 100 cutoffs were taking around 3 or 4 seconds; and for -m we are around 24 seconds which is also logic comparing the approximately 15 we

had with 100 cutoffs.

4.5.4 Comparison with my machine

Even though I ran more tests for the chunk sizes and schedulings on the HPC compute node than on my machine, here is a quick comparison of output times for -c, guided scheduling, 100 computations and a number of threads going from 1 to 100.

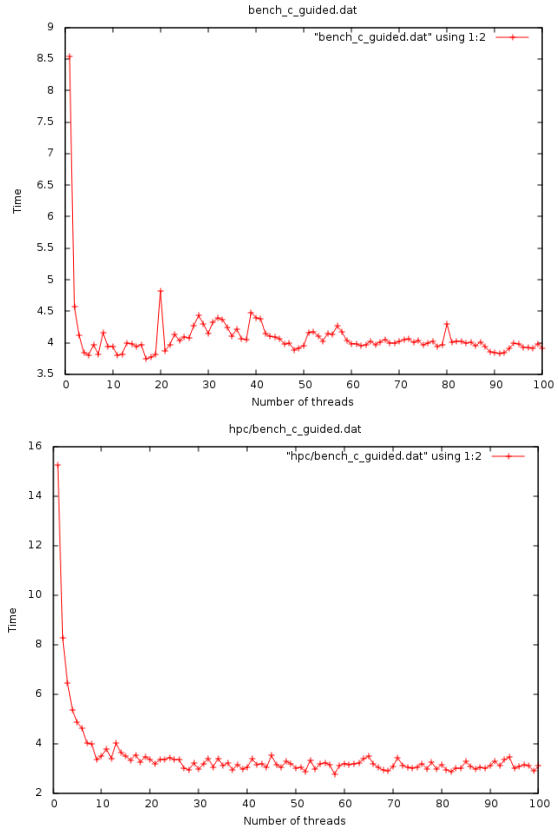

Figure 16: -c guided scheduling difference between my machine and HPC node

What we can say about these two curves is that, first, the output time is a bit better on the HPC compute node when we increase the number of threads: indeed, we are near 3.5 seconds for the HPC node and more near 4 seconds for my machine. Second interesting thing is that the computations on the HPC compute node are way more stable. This is what we could expect since the compute node has more cores. A really strange things though is that from 1 to 10 threads, my machine is faster. And for just 1 thread, a lot faster: 8.5 seconds for my machine and nearly 16 for the compute node. That might be because of the processor frequency, since just one thread is running; same when we have less than 10 threads.

5 Conclusion

We can make two conclusions for this paper:

1. To compute plates limits on a geoid model using my program, the best parameters/environment to setup is processors binding, spreading out the threads between the processors with the CPU affinity and of course using cutoff parallelization; even though we can say that the program is quite "long" since for only "100" cutoffs we have results in 4 seconds. Imagine you have even bigger files, need more cutoffs or just need to run the program 100 times (which is quite what I did), this could take a long time.

2. 0verall the HPC compute node is better than my machine to compute the program. It has more cores so can handle more threads before having quite constant results. It has better performances thanks to his L3 cache size which leads to a lot less instruction and data misses. Knowing that, I ran more tests on the HPC compute node.

6 References

Augury team: I have worked 3 years ago in a French geology laboratory as a computer science intern. I was helping the Augury Team (Augury project, a rewarded European project) developing some Python scripts on massive amount of data for different purposes: sorting, retrieving interesting values, searching plates poles of rotation... For the HPC project, I contacted professor Nicolas Coltice, leader of Augury, to give me some ideas, because I knwo that HPC is used a lot in science. I remember they have a huge cluster. He then gave me this idea of plates limits along with the exmaple files I am using.
Augury website: http://geologie.ens-lyon.fr/augur/

GMT: I used GMT to write the visualization script. I was helped by some sample scripts on their website and by command manuals.
GMT website: http://gmt.soest.hawaii.edu/

OpenMP: I used OpenMP for program parallelization. I used what we learned in class and also the website for documentation.
OpenMP website: http://www.openmp.org/specifications/