HEAT DIFFUSION PROJECT

XIE JAROD

Introduction

To begin with, I ran the sequential code to understand the key points of its operation. I then tried a naive approach to parallelization, but it didn't work. I then discussed with the project manager the possibility of projecting the heat sink in one dimension.

This approach proved more promising, and I was able to develop a functional and realistic solution. However, it was not optimized enough to run the "CHALLENGE" mode in realistic time. The following results therefore come from this one-dimensional parallelization, which is described in the "HeatSink1d.c" file.

I also tried a two-dimensional projection of the heat sink. This approach was more complex than the one-dimensional projection, and required the implementation of more advanced parallelization techniques and algorithms. However, it proved unsuccessful due to too many complications.

In conclusion, optimizing the heatsink.c program is a challenging task that demands a deep understanding of the sequential code, parallelization techniques, and hardware performance. The one-dimensional projection approach holds promise, but further refinements are necessary to achieve the performance required for the "CHALLENGE" mode.

Comparison of sequential code with parallelized code in "Fast" mode

In this section, we'll look at the difference in terms of speed, based on the total time it takes the function to finish. To do that, I will perform all of the computation on the same host, here “grisou” .I'll calculate 10 times for each function and take the average for each function (Moyenne_Temps.py jobs).

(Note: for the parallelized version, I'll fix the number of nodes used, which will be 10).

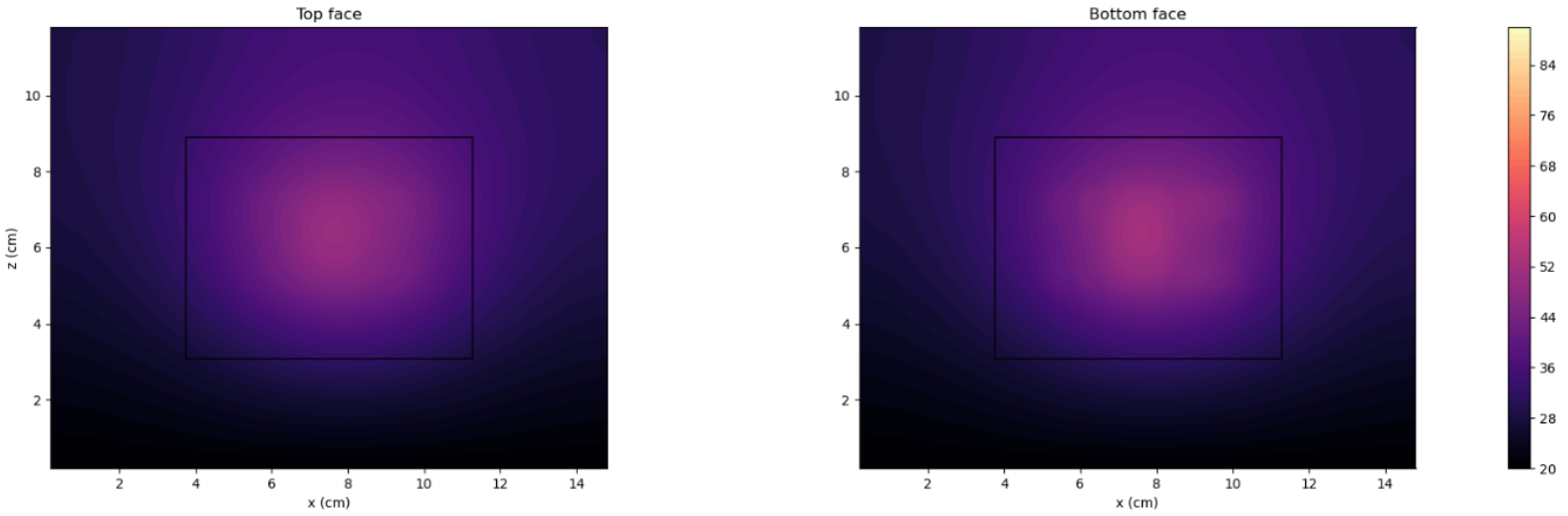
We'll start with the sequential function and obtain the following results :

```
jxie@grisou-13:~/PROJET_PPAR$ python3 Moyenne_Temps.py TimeSequentialFast.txt
[9.023717, 9.047999, 9.006738, 9.083381, 9.061069, 8.998534, 9.206068, 9.027547, 9.037107, 9.085326]
The average time of the function 10 iterations is 9.0577486 sec
```

We do the same thing on the parallelized code with 10 cores allocated and we get :

```
jxie@grisou-13:~/PROJET_PPAR$ python3 Moyenne_Temps.py Fast1d.txt
[0.938009, 0.923712, 0.912278, 0.976512, 0.928248, 0.928988, 0.928328, 0.932475, 1.113079, 1.041375]
The average time of the function 10 iterations is 0.9623004 sec
```

A speed ratio on these instances shows that the parallelized function is approximately 9.4 times faster than the sequential one.These are satisfactory results for our parallelized code, which was supposed to be faster. Here is a visual rendering of the result with the “rendu_pictures_steady.py” file.



Quick comparison of sequential code with parallelized code in other modes

Since we already know that it's faster in "Fast" mode, we can assume it will always be faster with parallelized code since the other mode should take more time to compute.

Let's try with the "Medium" and "Normal" mode for both of the code with the same parameters as before.

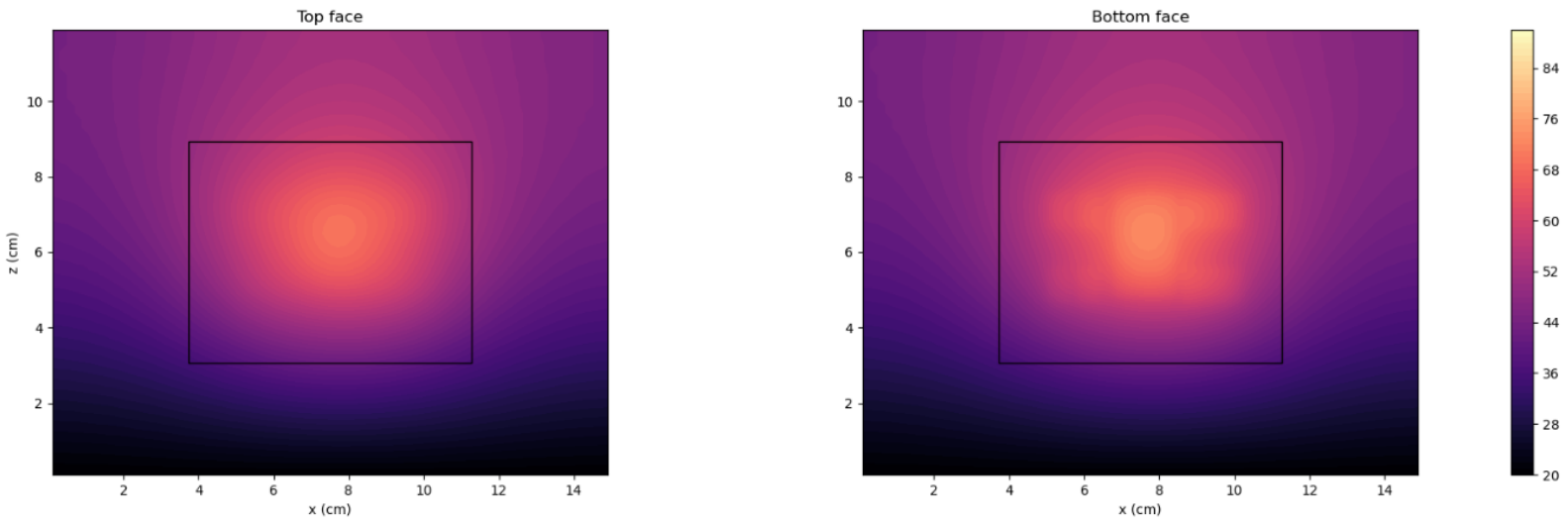
First with "Medium" mode we get those results.

```
jxie@grisou-25:~/PROJET_PPAR$ python3 Moyenne_Temps.py TimeSequentialMedium.txt
[148.946731, 148.726987, 148.561012, 149.356324, 149.643185, 149.654135, 148.918841, 148.686913, 149.001004, 148.920807]
The average time of the function 10 iterations is 149.041594 sec
```

```
jxie@grisou-35:~/PROJET_PPAR$ python3 Moyenne_Temps.py Medium1d.txt
[15.367318, 14.087185, 19.002861, 17.731884, 14.214772, 15.752941, 14.997791, 17.813324, 14.811616, 17.068371]
The average time of the function 10 iterations is 16.084806 sec
```

As expected, the time for sequential computation is greater than for parallelized computation, with a time ratio of 9.3. Our results are therefore consistent with previous ones and satisfactory.

Here is a visual rendering of the result.



Now let's try the "Normal" mode. For the sequential computation we won't run it ten times since it will take too much time, but we will still do it for the parallelized computation

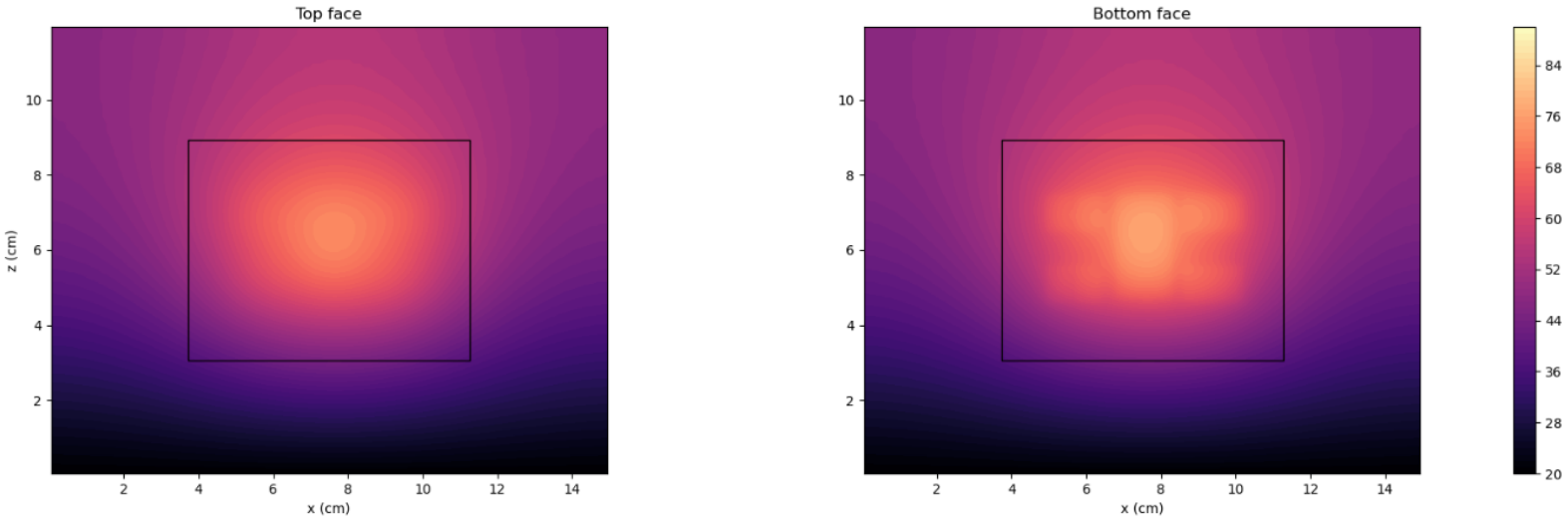
```
jxie@grisou-25:~/PROJET_PPAR$ cat TimeSequentialNormal.txt
911.935371
```

```
jxie@fnancy:~/PROJET_PPAR$ python3 Moyenne_Temps.py Normal1d.txt
[129.367227, 122.973843, 119.628681, 119.604363, 120.432176, 121.6018, 119.510763, 119.08639, 119.944405, 119.872532, 120.750237]
The average time of the function 11 iterations is 121.161129 sec
```

As usual we got what we were expecting for the difference in speed. The speed ratio between sequential and parallelized computation is 7.5. (Reminder : we used 10 nodes to do the parallelized computation).

So for every mode except "CHALLENGE" that I did not try, the parallelized version is always faster than his sequential version.

Here is a visual rendering of the result.



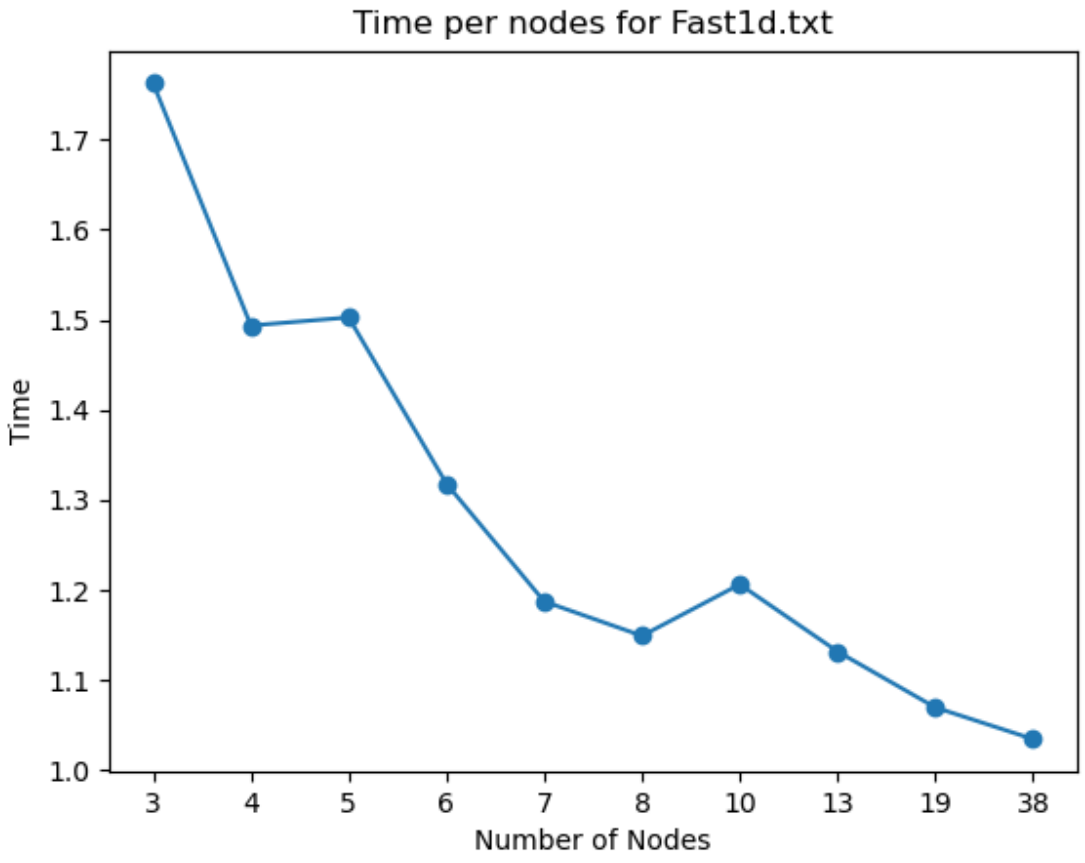
Comparison on how much the number of nodes affect the time of execution spent on parallelized code

In this section, we'll look at the impact of the number of nodes used to execute parallelized code. So we will just run the different modes while we increase the number of nodes little by little.

At first I ran the “Fast” mode with the number of nodes increasing by 1 from 3 to 100 with the command “for i in {3..100} ; do mpiexec -hostfile $OAR_NODE_FILE -n $i HeatSink1d > test1d.txt;done” .

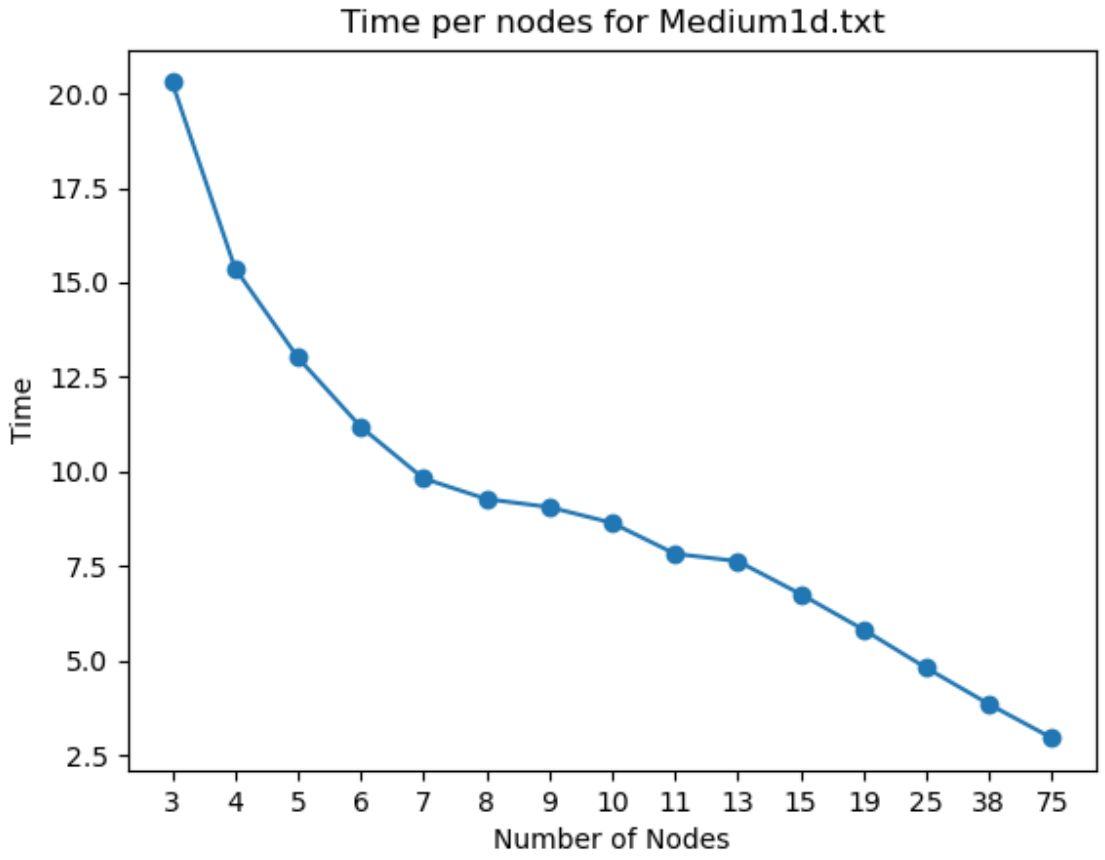
Since I remove all the nodes that aren't useful for computation, I end up with a graph like the following, calculated by taking the average time for each number of nodes. They are all from the “gros” cluster from Nancy.

We obtain the graph using the curve.py file, passing as parameters the text file we're using to analyze the data (here Fast1d.txt). The following command was use “python3 courbe.py Fast1d.txt”



As expected, the curve is decreasing, so the time decreases in relation to the number of cores used, and the result is convincing compared with expectations.

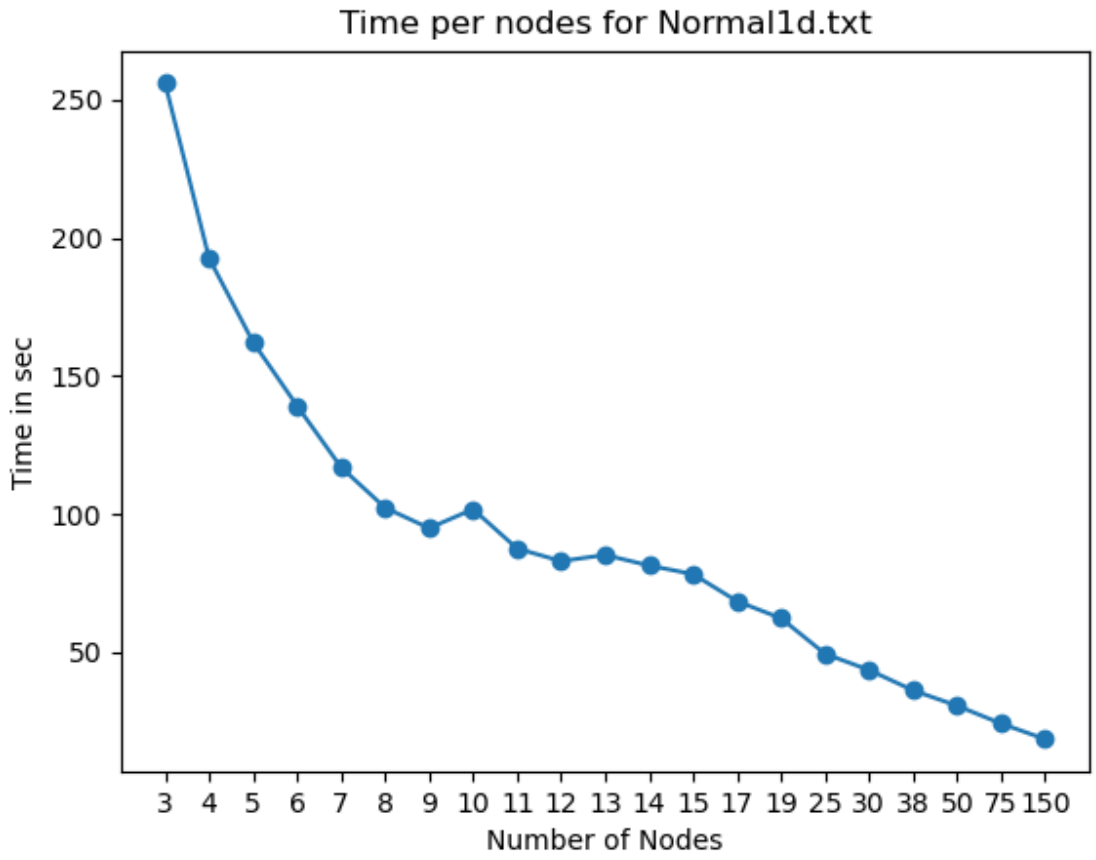
So now I will compute for the “MEDIUM” mode with the following command “for i in {3..100} ; do mpiexec -hostfile $OAR_NODE_FILE -n $i HeatSink1d > test1d.txt ;done” . Don’t forget to change the value in HeattSink1.c of FAST to MEDIUM and compile it again with make.



After computation, we have again obtained a decreasing curve which follows the expected result of our experiments. We can now conjecture that we'll get a similar result for the next test with "NORMAL".

So now I will compute for the “MEDIUM” mode with the following command “for i in {3..20} ; do mpiexec -hostfile $OAR_NODE_FILE -n $i HeatSink1d > test1d.txt ;done” and the command “for i in {21..150..5} ; do mpiexec -hostfile $OAR_NODE_FILE -n $i HeatSink1d > test1d.txt ;done ``.I did it in two steps, since the number of nodes used between 3 and 20 varies more, and between 21 and 150 it varies less, so I don't have to make redundant calculations.

Don’t forget to change the value in HeatSink1.c of MEDIUM to NORMAL and compile it again with make.



We finally obtain a very nice decreasing curve which decreases quite rapidly with a difference of 250 seconds with 3 cores and 18 seconds only with 150, which makes a ratio of about 14 times faster but at the cost of 50 times more cores used. So our conjecture is proven true by our numerous tests.

Conclusion

To conclude, optimizing the heatsink.c program has been a rewarding journey that has underscored the transformative power of parallel programming. The one-dimensional projection approach successfully accelerated the program's execution, achieving a significant speedup compared to the sequential code. This achievement highlights the potential of parallel programming to tackle computationally demanding tasks with enhanced efficiency.

As we move forward, exploring alternative parallelization techniques, such as block parallelization or domain parallelization, could further amplify the performance gains. Additionally, implementing communication reduction techniques between threads holds promise in optimizing data exchange and minimizing overhead.

By pursuing these avenues, we can propel the heatsink.c program to new heights of performance. Moreover, the program's ability to scale seamlessly with increasing nodes opens up exciting possibilities for harnessing the power of distributed computing to tackle even more challenging computational problems.

As we continue to refine and optimize the heatsink.c program, we can unlock its full potential for real-world applications.