Project : TOP

Chutzpa William YEUMO BARKWENDE

May 1, 2022

1 Introduction

The goal of this project is to check our understanding of how threads and process work with libraries such as OpenMP and MPI. We also need to know how to use profiling programs to help us quickly analyze source codes effectively. For this project, I will be using my own personal laptop, which comes with its advantages of being easily available and manageable. However, since it's a laptop not made for High Performance Computing (HPC), it's lacking in several fields for this project.

In the directory where this document is, you can find a cpu.txt file that contains all the data regarding my processor thanks to **lscpu**. I use **cpupower** to set my frequency to performance, which will not allow me to be at 4.5 GHz during any of my benchmarks due to the dynamic frequency scaling installed to prevent it from overheating too much. During all my experience I will not go above 6 process since my hardware limitation would make it counterproductive to do so. Since I have 6 real cores with hyper threads which makes 12 logical cores, however they share the same cache line and will only work against each other when charging data in the cache line.

In the next part of this rapport, I will detail step by step my thought process on how and why I intend to improve this simulation.

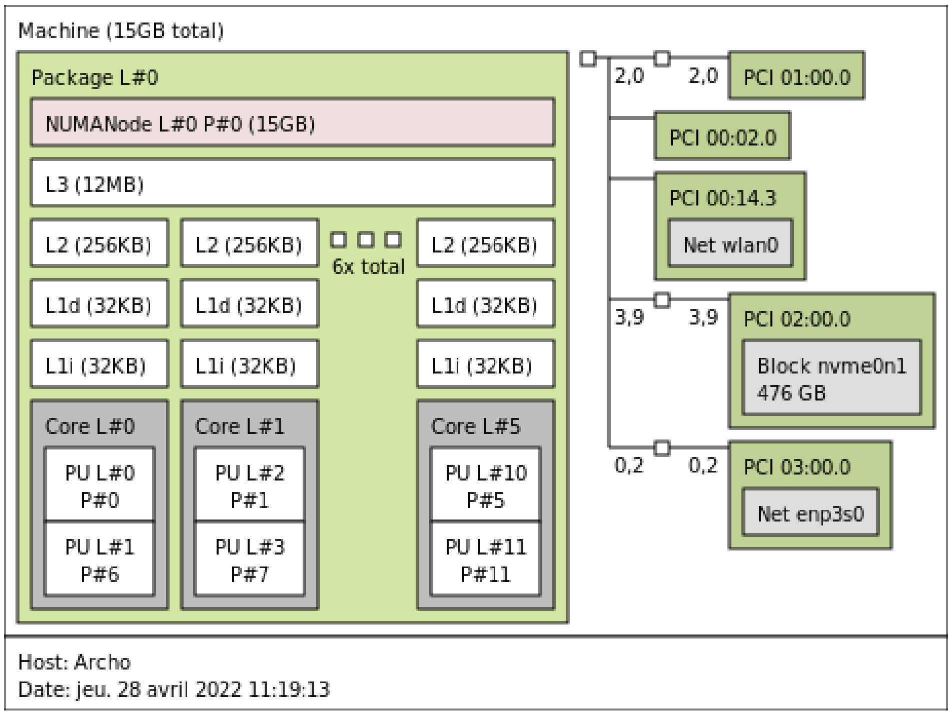


Figure 1: Graphical representation of the CPU with lstopo

2 Improving the simulation

2.1 Compiling and debugging

After reading downloading the directory and reading the instructions, I went ahead and opened the Makefile and added a -Wextra flag for more precise compiling error messages before executing a make.

This is where we encounter our first issue. In order to understand where it comes from, I first must say that I use spack which is a flexible package manager that handles dependencies for me. I used spack load GCC, which installed the 11.2.0 version of GCC and added it to my path when I compiled everything. This version of GCC will have an issue the const variables declared at the beginning of the lbm_phys.h. The compiler considers that the const variables are defined multiple times. In order to fix this issue we add the **extern** keyword at the beginning which will declare the variable as global as intended but without any memory assigned to it so it won't be defined. Next issue is that -Wall and -Wextra allowed us to find two functions that were declared and defined with unused parameters. I removed them in order to make the code a bit less cluttered remove the warnings.

After these two steps were done I managed to use make to get my executable lbm. I use **valgrind** to launch it and see how the memory management is being handled in the program. However, during the execution a segfault is signaled.

I then launch the program with **lldb** to find where the issue is. And, unsurprisingly, the program encountered an issue when we tried to assign a value to something we didn't allocate memory prior. I decide to read each file to see where memory allocation was done for the mesh. We can find it in the lbm_struct.c file. However, the reason our program crashed is simply because the allocation was commented. I removed the "//" and also the line right below that set our cells vector back to NULL.

I used **make -B** to update the executable and launch it with **valgrind**. And the program managed to launch successfully albeit it was very slow. Also in order to to make the progress line not create new lines every time, I replaced the \n with \r.

The result of the first successfully launched valgrind test can be found in the valgrind_test_first.txt file. However, this doesn't help much, we only know that we have memory leaks but not where it leaks. To help with this, we add the -ggdb3 to the compile options in the Makefile, and **make -B** again before using: valgrind –leak-check=full –show-leak-kinds=all –track-origins=yes –verbose –log-file=valgrind_test_first.txt ./lbm.
–leak-check=full allows us to see each leak in details. –show-leak-kinds=all will show all kinds of leak in the full report. –track-origins=yes will show a more complete report but will make valgrind take a longer time to execute. –verbose to tell us if our programs does something unusual. –log-file to write the report in a specific file in order to not bloat the terminal.

In the file made by valgrind we can see that every memory leak is due to MPI functions, however, I'm not using OpenMPI but MPICH 4.0.2 thanks to spack, however since valgrind is required for OpenMPI, it's possible that it's already set to use their binary. Nevertheless, as for the the structures we allocated manually they're still reachable according to valgrind. This means that they have not been freed but could have been. They are not as much of a threat because the program keeps track of those pointers and when it stops will automatically release its content.

However, after fixing, the release functions in lbm_struct.c, the only memory issues we have left come from MPI library, which, for the second report uses the spack dependency. After looking into it, this is due to the fact that I install valgrind with pacman and not spack which made it follow either one dependency randomly it seems. But it's not really an issue as MPI only causes "still reachable" memory leaks, probably due to the buffers created for the different MPI communications.

2.2 Sleep and compilation flags

To start we will try to understand why the program is so slow by looking at the trace with lldb. I launch the program in lldb and stop it with a manual SIGINT (CTRL - C), and check the backtrace. And this is how I found out that there was a sleep function somewhere in the program, and more precisely in the ghost_exchange function. However, besides useless MPI_Barriers that I will remove as well, there's only the FLUSHINOUT function. Thanks to VS code I find pretty easily where it's declared and defined. And it turned out that it was using defines in lbm_config.h to hide a sleep.

With the sleep, we can do 200 iterations in a relatively short period of time. Furthermore, when using more than one MPI process we notice a deadlock at the end. This makes it simple to debug since it means that the issue happens after or during the last iteration. To solve this issue I read the main.c file and find easily enough where the problem is. We have a close_file function where one process MPI waits for everyone else to finish before closing the file that was open. However what this does is that while it's closing the file all the other process are exiting which causes the deadlock. To fix this, I just need to add a barrier so the other process wait for the main process to finish closing the file.

But, it still takes quite a long time, so I decide to add the -O3, reasons for that below, as a compile flag to shorten it further. I then decide to launch the program with 2 process and generate a GIF to see if my simulation still matches the one from the ref_resultat_200.raw file, and use the display function, I created a shell script named check.sh to compare the results made from my simulations and the one from reference.

Now we want to get an accurate amount of time for our experiments and not just raw guesses. I will use the clock() function from the time.h library which gives us a more or less accurate of the time used by the cpu to execute the simulation. We can also use other tools like **perf** but I will use them for matters that are more interesting.

In the graph below, the time assigned to each situations is an average over 10 executions with the same configuration as the one used for reference which is 200 iterations and save frame every 1 iteration. Each of the flags tested below increase compilation time but greatly optimize codes. While O2 and O3 could be considered quite safe as they follow the standards of optimization for most programs Ofast just optimize regardless of safety protocols. Which is one of the reasons I'll be using O3 since the performance gain from Ofast isn't noticeable enough to risk increasing the instability of the code. These measures were made with 1,2 4 and 5 MPI process. 3 and 6 MPI process were not included as they produced too many errors with the checksum. 4 and 5 MPI produced the same errors across all experiences at frame 163 and 183. And above 6 would not be efficient for reasons mentioned above.

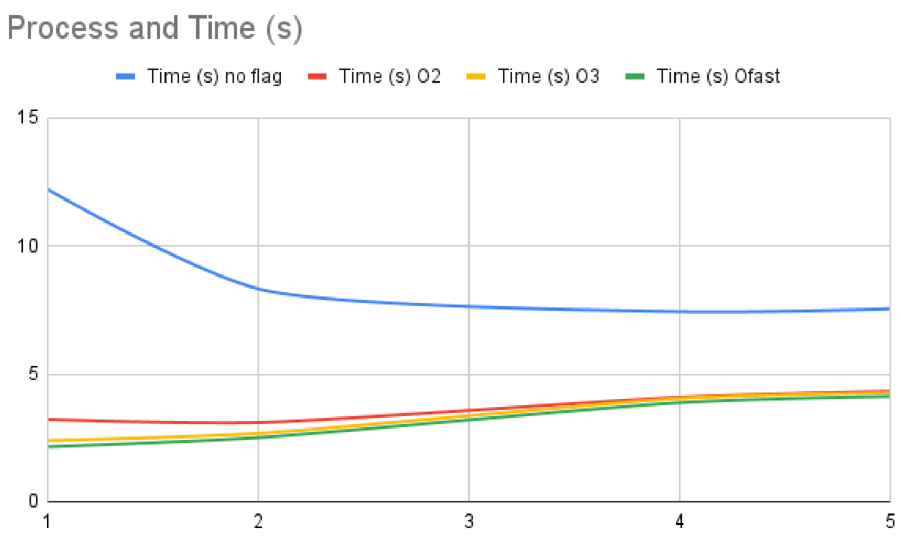


Figure 2: Graph of execution time depending on the number of process

This graph also help us understand another important point for this project, that right now it's not a strong scaling program in any case. A strong scaling program would see his execution time nearly halved if you allocate twice as many process or resources to it. The best performance increase is with experiments without the flags where from 1 to 2 process we can observe something close to a 70% decrease in time needed to execute over 200 iterations. However, it's not all true when we look at the difference between 2 and 4 MPI process, let alone for other cases with O flags.

However, this begs the question as to whether or not the program is weak scaling. To verify this, we will multiply the number of process by 2 each time and the width variable as well. For a weak scaling program, if we multiply the size of the sample size by two and the allocated resources to treat the issue by two, we should take bout the same time for the execution.

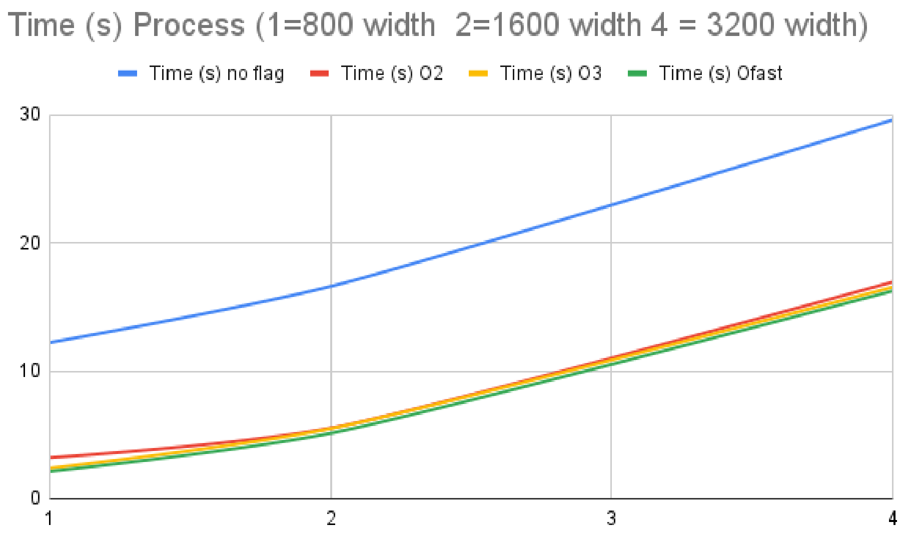


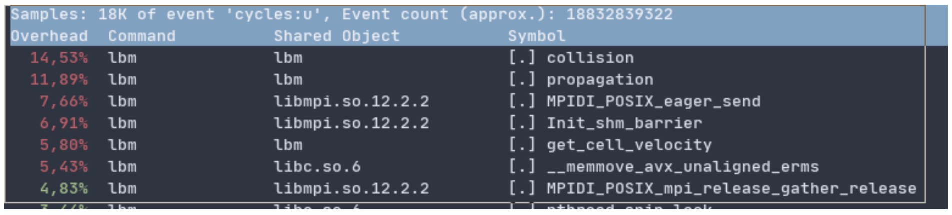
Figure 3: Graph of execution time depending on the number of process and width

As we can see in this graph, the program is neither strong scaling or weak scaling.

2.3 Perf and OpenMP

For the next part, I will use perf to know which functions I should focus on to optimize since, I realized that I didn't see any use of threads in the entire source code. And with the data recorded above and my hardware limitation, I am looking at using 2 MPI process and 3 threads so each thread can work with his own cache line and no hurdles.

In order to do this, I will use **perf record mpirun -np 2 ./lbm**, which will make perf create a data file that I can read with **perf report**. The config is set back to the values uses for reference which is 200 iterations and 800 width. And the report tells me that two functions take most of my time, which are collision and propagation from lbm_phys.c.



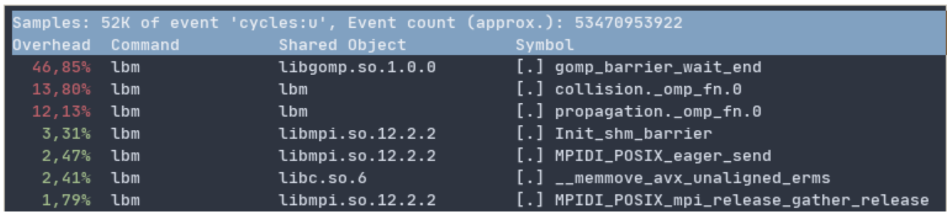
```
Samples: 18K of event 'cycles:u', Event count (approx.): 18832839322
Overhead  Command          Shared Object            Symbol
  14,53%  lbm              lbm                      [.] collision
  11,89%  lbm              lbm                      [.] propagation
   7,66%  lbm              libmpi.so.12.2.2         [.] MPIDI_POSIX_eager_send
   6,91%  lbm              libmpi.so.12.2.2         [.] Init_shm_barrier
   5,80%  lbm              lbm                      [.] get_cell_velocity
   5,43%  lbm              libc.so.6                [.] __memmove_avx_unaligned_erms
   4,83%  lbm              libmpi.so.12.2.2         [.] MPIDI_POSIX_mpi_release_gather_release
```

Figure 4: Result of the first report on the most time consuming functions

Looking at both function, I decide to use OpenMP since they have a lot of iterations over two for loops that could be done over multiple threads. First I add the -fopenmp flag in the Makefile which will enables the compiler to recognize the #pragma, then I will use them to create parallel regions and configure the first for loop in a way where they wont bother each other.

First, for the collision function I will use a dynamic schedule for the for loop since it helps balance the workload in all situations since we can and will modify the config.txt file. It is however, more costly to use dynamic scheduling since there needs to be communication between the threads but it's worth it with our problem since the fact that the settings can be changed on a whim, having a static scheduling could lead to useless waiting time.

And with my current configuration 3 threads is optimal so each can work on different core. I will also make i and j private as they were declared outside the parallel region so each thread in the same process have the same address to their memory. Same for propagation. I will also invert the for loops so we start iterating on the width before the height in order to make smaller loops for each threads and not a small one and a huge one. I also thought about using the collapse feature to "fuse" the loops and experimented with it. And here I notice something interesting, the time I get from my timer using clock() is greatly above the time it took to execute the program. And this can be explained because of of the fact that it tracks the process' cpu work time. And me allocating 3 threads per process, increase the amount of time spent on the CPU and clock() will combine all of it. For our earlier experiments, it was not an issue as the wall time and cpu time were about the same as long as we used only one of the two MPI process to track it. However with OpenMP,the wall time is significantly different, wall time being the time it took for the program to execute itself. For this reason, I'll switch from the time.h library to using the shell function **time**.



```
Samples: 52K of event 'cycles:u', Event count (approx.): 53470953922
Overhead  Command          Shared Object          Symbol
  46,85%  lbm              libgomp.so.1.0.0       [.] gomp_barrier_wait_end
  13,80%  lbm              lbm                    [.] collision._omp_fn.0
  12,13%  lbm              lbm                    [.] propagation._omp_fn.0
   3,31%  lbm              libmpi.so.12.2.2       [.] Init_shm_barrier
   2,47%  lbm              libmpi.so.12.2.2       [.] MPIDI_POSIX_eager_send
   2,41%  lbm              libc.so.6              [.] __memmove_avx_unaligned_erms
   1,79%  lbm              libmpi.so.12.2.2       [.] MPIDI_POSIX_mpi_release_gather_release
```

Figure 5: Result of the second report on the most time consuming functions

However, with OpenMP alone, there's no noticeable increase in performance, so I decide to look at the functions or operations called in both function and the other functions in the file. The function special_cells also has a double for loop iterating over every cells, which could improve with multi-threading. However, when you compile the code and test it again and making sure with check.sh that the value are correct, there's no real improvement. The time it takes to execute is about the same. And when you look at the report from perf, you see that the time you gained from multi-threading was lost because of the threads had to wait. And adding the nowait flag to the for loops doesn't help either, which means that the problem is somewhere else. Let's check, if despite the fact that we did not gain any performance increase, we managed to make the program strong and/or weak scaling. Since we are using 3 threads, we'll making tests with 1 2 an 4 MPI process, as 5 MPI process would not make any sense with my current hardware.

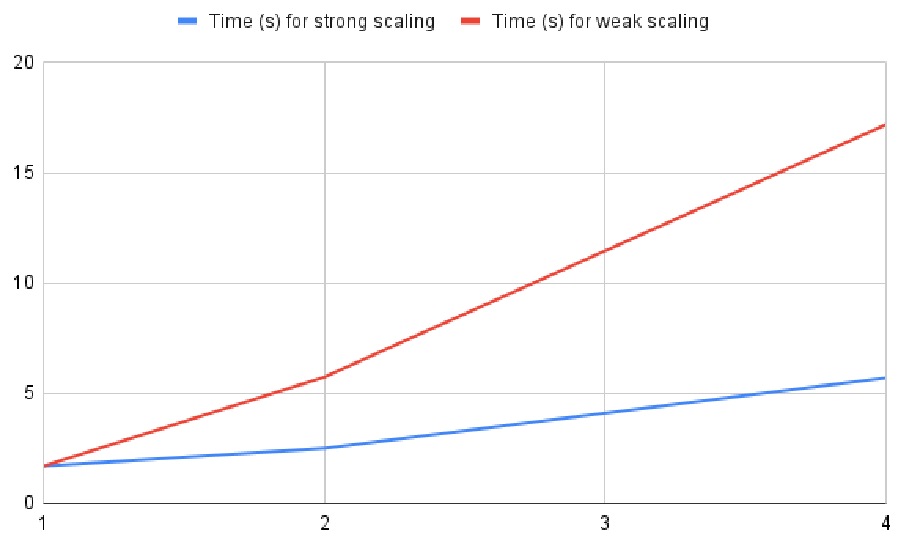


Figure 6: Scaling with OpenMP

Despite the fact that we used OpenMP to help us, the fact that the treads have so much waiting time prevents any form of scaling. Which is not so surprising because, while reading the source code we can find so many barriers and useless wait time, like the sleep, that prevents the program from executing itself smoothly. Which is why I'll focus on MPI next.

2.4 MPI

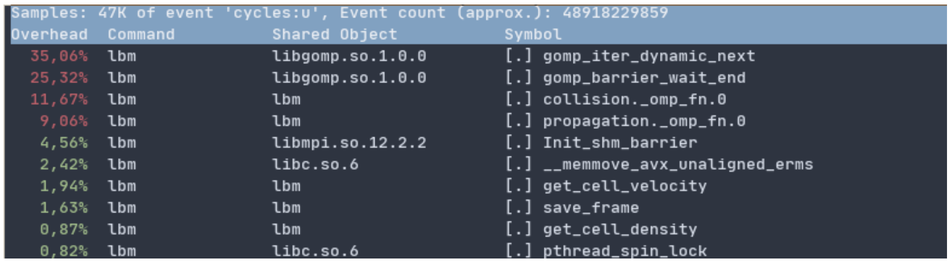
First thing first, I will remove as many MPI_Barriers as possible and recompile the code and verify if they were indeed necessary or not. The first function I want to look at is the ghost_exchange function in lbm_comm.c. The MPI communication method used is with regular Sends and Receive, there are already barriers in them and having extra MPI_Barriers called is just a waste of time. And I was proven right since after running my bash script, no errors were found with 1, and 2 process while 4 MPI process still has errors at frame 163 and 183. Next I notice that the horizontal function is called again at the end to do the same thing, which makes no sense since the data has already been sent and received, I also comment it.

Next, I focus on the exchange functions themselves. First the horizontal function uses another function declared and defined in lbm_struct.h. I will not be using Isend to improve the MPI communication here. You would use Isend to be able to make use of waiting time in between the Isend and the Recv, however in order to do this. This would make the waiting time effective and beneficial toward the program instead of being an hindrance. However, I didn't have the time to experiment with it far enough to come with something working properly. Instead, I focused on sending better messages. For the horizontal exchange, there's no need to have a for loop sending one element at a time of the array, we can just send the full array and the result will still be valid. Doing so reduces the number of MPI communication by however much the height is - 2. And although I won't test the scaling yet, it will probably make the program scale on way or another when I'm done or maybe even have both type of scaling.

Next is the vertical and diagonal functions. I'll start by saying that the diagonal function is redundant if we use the vertical one. So I'll be putting them as comment in the ghost_exchange function. Then, for the vertical exchange function, it has so many things going wrong with itself. The receiving buffer size is bigger than what is called and needed, iterating over an array to send each element one by one with regular send that stop the process until the other process receives the data is not good either. To fix this, I created a small array in which I collect the data and send in one go to the other process.

This makes that in total, only 4 MPI communications are needed every iterations and not the previous 7000+. And not only that, in the third valgrind report, we can see that compared to the second one, the number of reachable memory leak has drastically decreased because we removed most MPI communications. However, I need to mention that the second report was launched with 1 MPI process while the launched with 2. The errors are due to this detail.

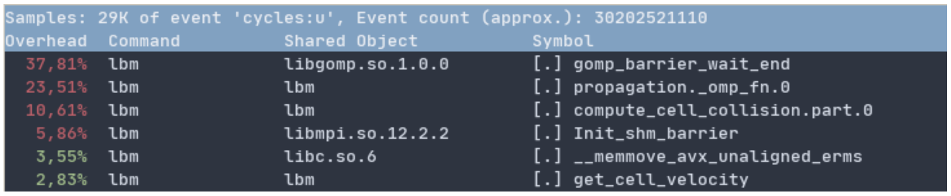
Although my OS does the work for me, in order to be sure that each threads work on different cores, you would need to add –map-by core whenever you use mpirun. In the following graphs I'll be looking at the scaling of the program again. But first, let's take a look at the time spent on the execution over 200 iterations while taking into account that compared to the 6s it took before with 2 process and 3 threads each, this execution took around 2.2s.



```
Samples: 47K of event 'cycles:u', Event count (approx.): 48918229859
Overhead  Command          Shared Object           Symbol
  35,06%  lbm              libgomp.so.1.0.0        [.] gomp_iter_dynamic_next
  25,32%  lbm              libgomp.so.1.0.0        [.] gomp_barrier_wait_end
  11,67%  lbm              lbm                     [.] collision._omp_fn.0
   9,06%  lbm              lbm                     [.] propagation._omp_fn.0
   4,56%  lbm              libmpi.so.12.2.2        [.] Init_shm_barrier
   2,42%  lbm              libc.so.6               [.] __memmove_avx_unaligned_erms
   1,94%  lbm              lbm                     [.] get_cell_velocity
   1,63%  lbm              lbm                     [.] save_frame
   0,87%  lbm              lbm                     [.] get_cell_density
   0,82%  lbm              libc.so.6               [.] pthread_spin_lock
```

Figure 7: Report with Dynamic OpenMP and 4 MPI communications

I said earlier that the time lost with dynamic scheduling would be worth it however, this report tells me it took a lot of the execution time. Let's see what the report says with static scheduling.



```
Samples: 29K of event 'cycles:u', Event count (approx.): 30202521110
Overhead  Command          Shared Object           Symbol
  37,81%  lbm              libgomp.so.1.0.0        [.] gomp_barrier_wait_end
  23,51%  lbm              lbm                     [.] propagation._omp_fn.0
  10,61%  lbm              lbm                     [.] compute_cell_collision.part.0
   5,86%  lbm              libmpi.so.12.2.2        [.] Init_shm_barrier
   3,55%  lbm              libc.so.6               [.] __memmove_avx_unaligned_erms
   2,83%  lbm              lbm                     [.] get_cell_velocity
```

Figure 8: Report with Static OpenMP and 4 MPI communications

Some part of the program do take a larger portion of the execution time but the execution went from 2.2s to 1.6s. Which means that I severely underestimated the time lost when OpenMP has to balance each threads. However, it's quite interesting to see how these parameters influence the scaling of the program.

2.5 Scaling

When I first started working on this project, I showed how this program was neither weak or strong scaling. Let's start with the static scheduling tests. During those tests I will be using 3 threads and will be doing the measures on 1, 2 and 4 MPI process. The time used will be an average of multiple executions. Also as a reminder, for the weak scaling time, I multiply the number of process by 2 = I multiply the width by 2 as well.

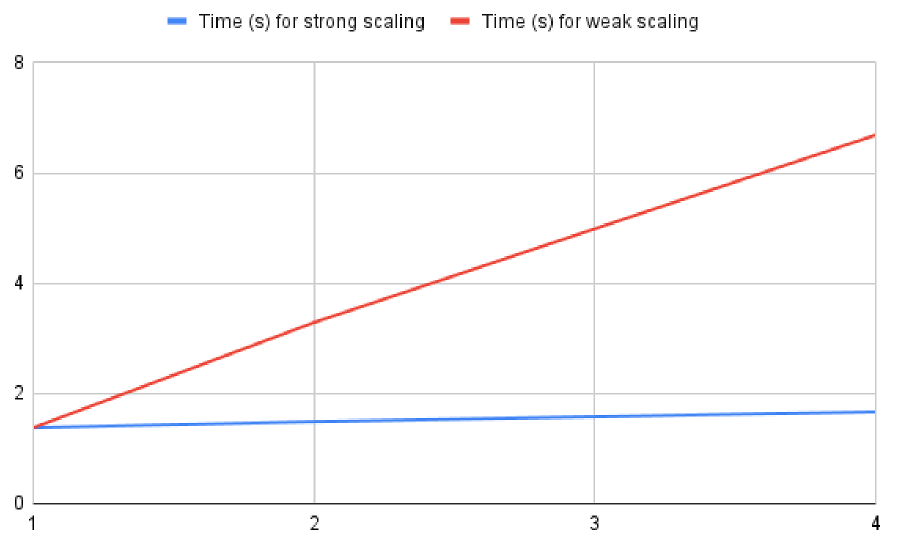


Figure 9: Graph with Static

While the static parameter allows for a faster execution the program is neither weak or strong scaling. Now let's check for dynamic.

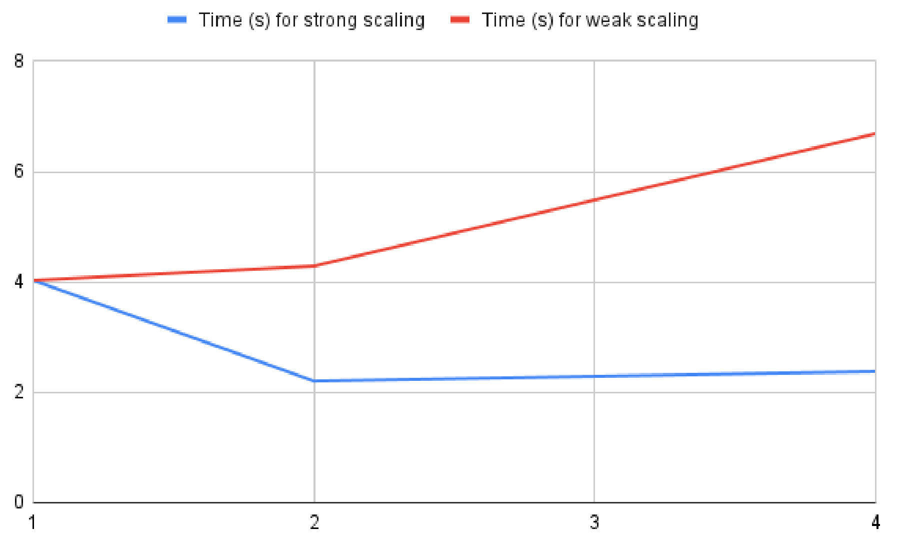


Figure 10: Graph with Dynamic

Here, we can see that the program is both strong and weak scaling. The reason why it doesn't work with 4 MPI process is because my hardware is the issue. Threads are sharing the same cache and actively working against each other. However from 1 to 2 process the time taken stays about the same for the weak scaling tests and the time is nearly divided by 2 for the strong scaling tests.

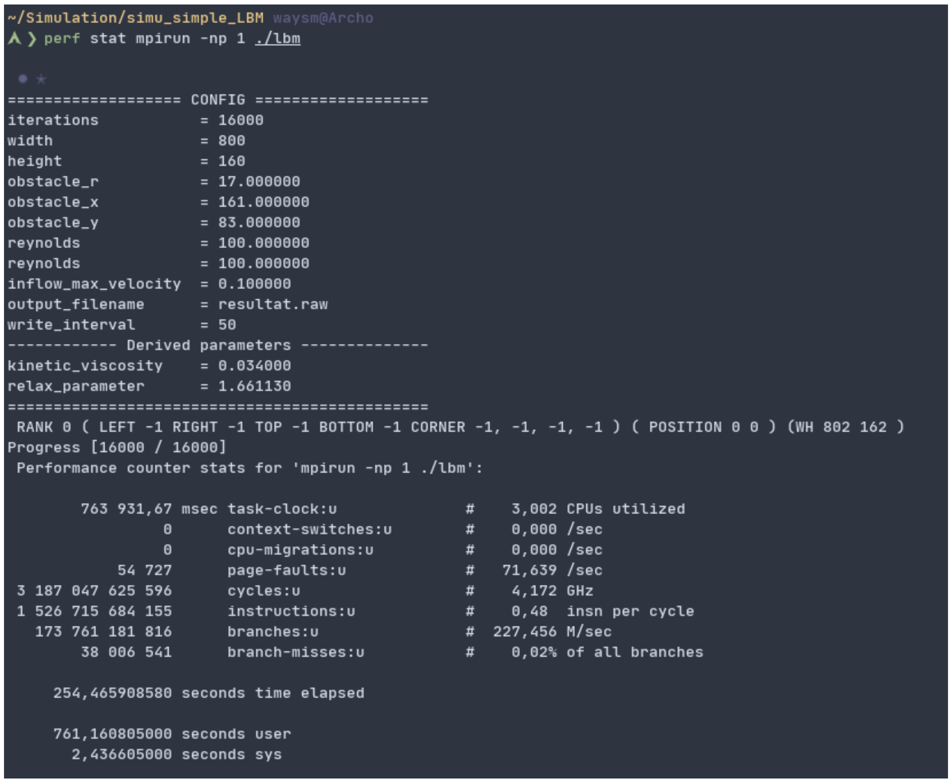
2.6 Things to consider

There is still much that can be done and to improve the current source code I have for my own hardware (like going from 4 to only 1 MPI communication). However, I resolved compiling issue, removed a lot of unnecessary functions and wait time that was bloating the program, there are so much things I wish I had time to do. There are a lot of parameters associated with mpirun / mpiexec that allow us to really control the behavior of each thread to a certain extent. There's also the fact that we could be using non blocking MPI communication function that would allow us to do some of the operations while we're waiting for the information to be received. And while our mesh wasn't massive, the issue of the bandwidth may be of concern in a more real life scale situation where the amount of data we wish to send is just too big. This could also impact heavily the performance if the things we deal with have sizes that are above what's in our L1 cache.

Also, something that wasn't reflected in this report is that in order to be able to gauge the effectiveness of each improvement we would need to mix and match the order in which they are implemented to see how effective each one of them is.

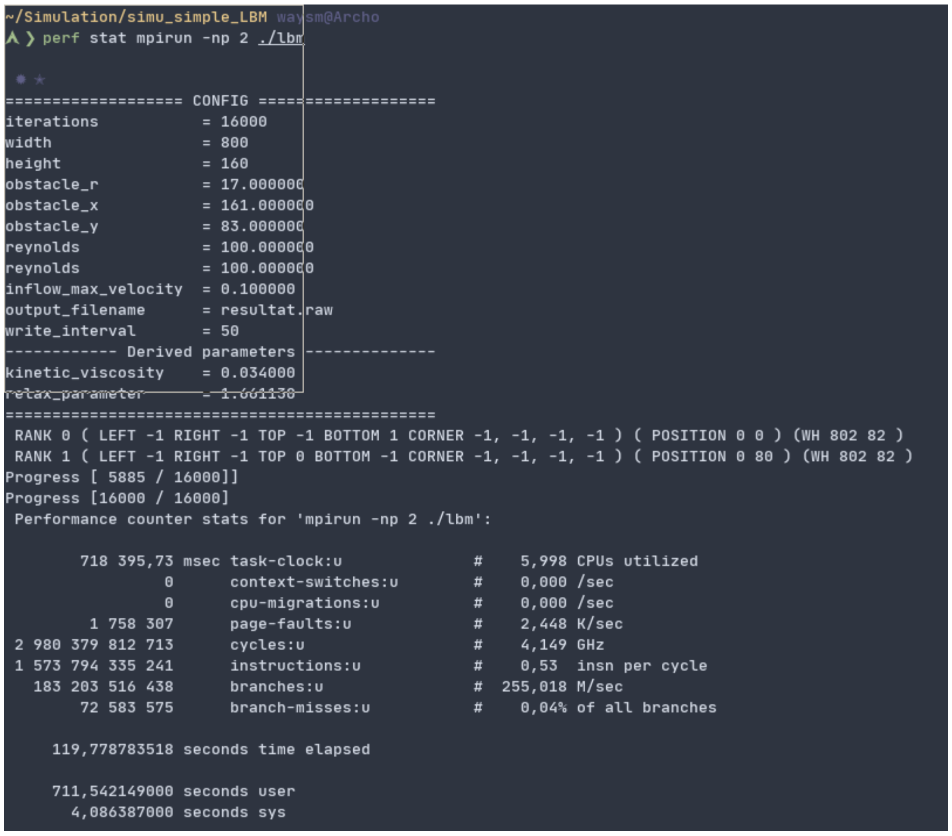
3 Conclusion

As a conclusion I just wanted to compare the perf stats data between 1 MPI and 3 threads and 2 MPI and 3 threads each.



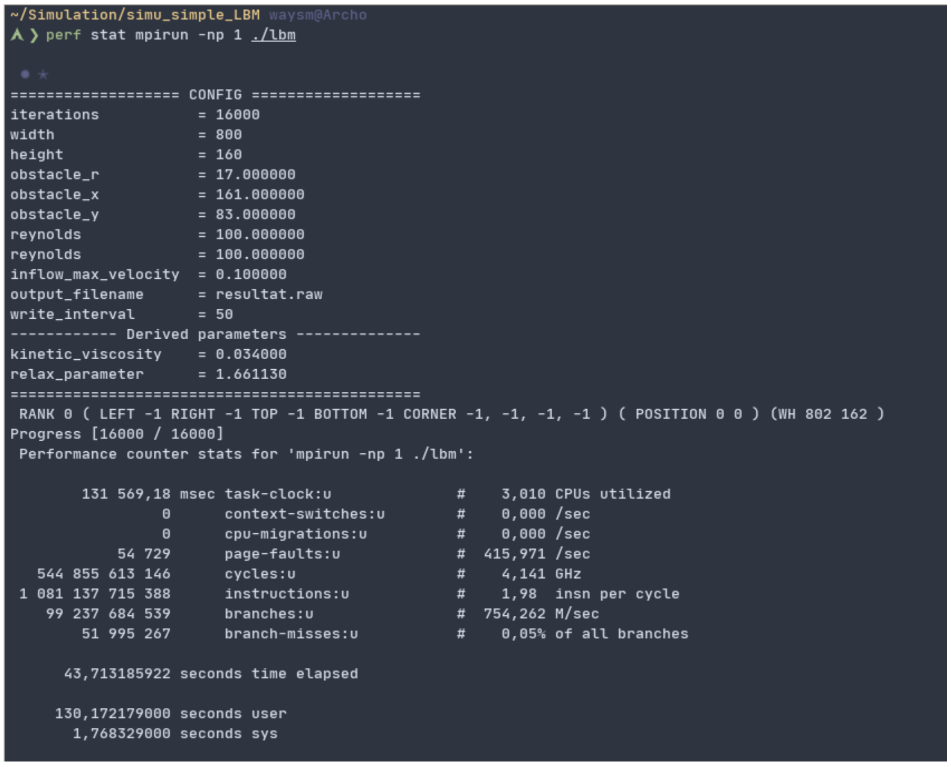
```
~/Simulation/simu_simple_LBM waysm@Archo
❯ perf stat mpirun -np 1 ./lbm

================== CONFIG ==================
iterations           = 16000
width                = 800
height               = 160
obstacle_r           = 17.000000
obstacle_x           = 161.000000
obstacle_y           = 83.000000
reynolds             = 100.000000
reynolds             = 100.000000
inflow_max_velocity  = 0.100000
output_filename      = resultat.raw
write_interval       = 50
------------ Derived parameters --------------
kinetic_viscosity    = 0.034000
relax_parameter      = 1.661130
==============================================
 RANK 0 ( LEFT -1 RIGHT -1 TOP -1 BOTTOM -1 CORNER -1, -1, -1, -1 ) ( POSITION 0 0 ) (WH 802 162 )
Progress [16000 / 16000]
 Performance counter stats for 'mpirun -np 1 ./lbm':

        763 931,67 msec task-clock:u              #    3,002 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
            54 727      page-faults:u             #   71,639 /sec
 3 187 047 625 596      cycles:u                  #    4,172 GHz
 1 526 715 684 155      instructions:u            #    0,48  insn per cycle
   173 761 181 816      branches:u                #  227,456 M/sec
        38 006 541      branch-misses:u           #    0,02% of all branches

     254,465908580 seconds time elapsed

     761,160805000 seconds user
       2,436605000 seconds sys
```

Figure 11: Perf Stats with 1 MPI Process and 3 Threads (Dynamic Scheduling)



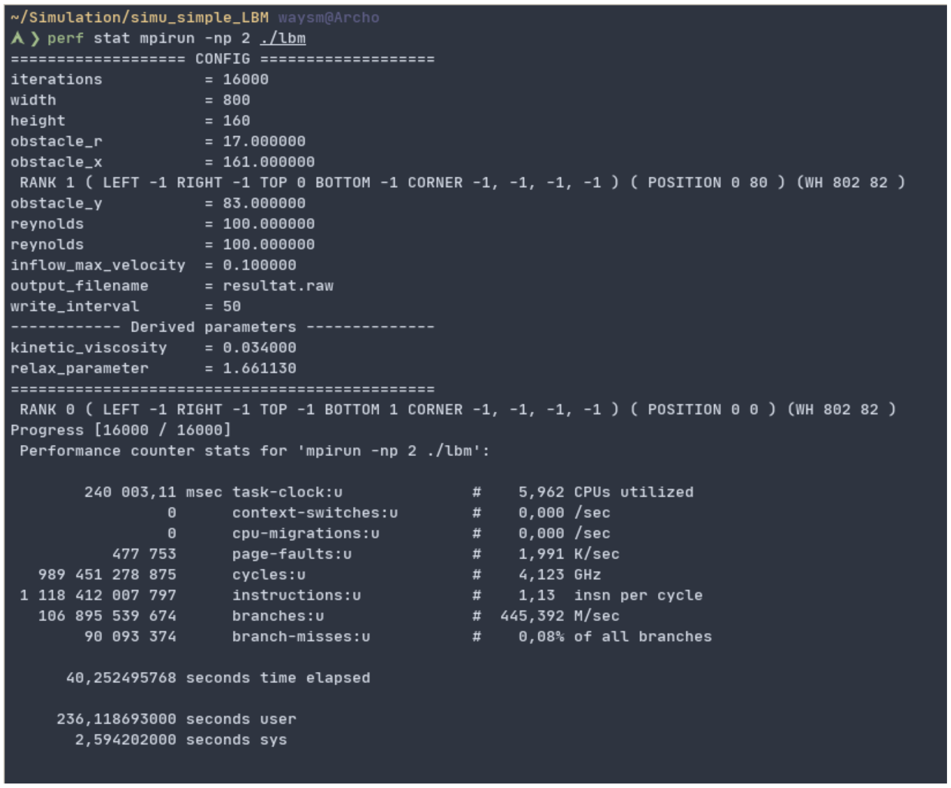
```
~/Simulation/simu_simple_LBM waysm@Archo
λ ❯ perf stat mpirun -np 2 ./lbm

 ● ★
=================== CONFIG ===================
iterations            = 16000
width                 = 800
height                = 160
obstacle_r            = 17.000000
obstacle_x            = 161.000000
obstacle_y            = 83.000000
reynolds              = 100.000000
reynolds              = 100.000000
inflow_max_velocity   = 0.100000
output_filename       = resultat.raw
write_interval        = 50
------------ Derived parameters --------------
kinetic_viscosity     = 0.034000
relax_parameter       = 1.661130
==============================================
 RANK 0 ( LEFT -1 RIGHT -1 TOP -1 BOTTOM 1 CORNER -1, -1, -1, -1 ) ( POSITION 0 0 ) (WH 802 82 )
 RANK 1 ( LEFT -1 RIGHT -1 TOP 0 BOTTOM -1 CORNER -1, -1, -1, -1 ) ( POSITION 0 80 ) (WH 802 82 )
Progress [ 5885 / 16000]]
Progress [16000 / 16000]
 Performance counter stats for 'mpirun -np 2 ./lbm':

        718 395,73 msec task-clock:u              #    5,998 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
         1 758 307      page-faults:u             #    2,448 K/sec
 2 980 379 812 713      cycles:u                  #    4,149 GHz
 1 573 794 335 241      instructions:u            #    0,53  insn per cycle
   183 203 516 438      branches:u                #  255,018 M/sec
        72 583 575      branch-misses:u           #    0,04% of all branches

     119,778783518 seconds time elapsed

     711,542149000 seconds user
       4,086387000 seconds sys
```

Figure 12: Perf Stats with 2 MPI Process and 3 Threads each (Dynamic Scheduling)



```
~/Simulation/simu_simple_LBM waysm@Archo
λ ❯ perf stat mpirun -np 1 ./lbm

 ● ★
=================== CONFIG ===================
iterations            = 16000
width                 = 800
height                = 160
obstacle_r            = 17.000000
obstacle_x            = 161.000000
obstacle_y            = 83.000000
reynolds              = 100.000000
reynolds              = 100.000000
inflow_max_velocity   = 0.100000
output_filename       = resultat.raw
write_interval        = 50
------------ Derived parameters --------------
kinetic_viscosity     = 0.034000
relax_parameter       = 1.661130
==============================================
 RANK 0 ( LEFT -1 RIGHT -1 TOP -1 BOTTOM -1 CORNER -1, -1, -1, -1 ) ( POSITION 0 0 ) (WH 802 162 )
Progress [16000 / 16000]
 Performance counter stats for 'mpirun -np 1 ./lbm':

        131 569,18 msec task-clock:u              #    3,010 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
            54 729      page-faults:u             #  415,971 /sec
   544 855 613 146      cycles:u                  #    4,141 GHz
 1 081 137 715 388      instructions:u            #    1,98  insn per cycle
    99 237 684 539      branches:u                #  754,262 M/sec
        51 995 267      branch-misses:u           #    0,05% of all branches

      43,713185922 seconds time elapsed

     130,172179000 seconds user
       1,768329000 seconds sys
```

Figure 13: Perf Stats with 1 MPI Process and 3 Threads (Static Scheduling)



```
~/Simulation/simu_simple_LBM waysm@Archo
⋀ ❯ perf stat mpirun -np 2 ./lbm
================== CONFIG ==================
iterations           = 16000
width                = 800
height               = 160
obstacle_r           = 17.000000
obstacle_x           = 161.000000
 RANK 1 ( LEFT -1 RIGHT -1 TOP 0 BOTTOM -1 CORNER -1, -1, -1, -1 ) ( POSITION 0 80 ) (WH 802 82 )
obstacle_y           = 83.000000
reynolds             = 100.000000
reynolds             = 100.000000
inflow_max_velocity  = 0.100000
output_filename      = resultat.raw
write_interval       = 50
------------ Derived parameters --------------
kinetic_viscosity    = 0.034000
relax_parameter      = 1.661130
==============================================
 RANK 0 ( LEFT -1 RIGHT -1 TOP -1 BOTTOM 1 CORNER -1, -1, -1, -1 ) ( POSITION 0 0 ) (WH 802 82 )
Progress [16000 / 16000]
 Performance counter stats for 'mpirun -np 2 ./lbm':

        240 003,11 msec task-clock:u              #    5,962 CPUs utilized
                 0      context-switches:u        #    0,000 /sec
                 0      cpu-migrations:u          #    0,000 /sec
           477 753      page-faults:u             #    1,991 K/sec
   989 451 278 875      cycles:u                  #    4,123 GHz
 1 118 412 007 797      instructions:u            #    1,13  insn per cycle
   106 895 539 674      branches:u                #  445,392 M/sec
        90 093 374      branch-misses:u           #    0,08% of all branches

      40,252495768 seconds time elapsed

     236,118693000 seconds user
       2,594202000 seconds sys
```

Figure 14: Perf Stats with 2 MPI Process and 3 Threads each (Static Scheduling)

The last thing I wanted to highlight was that even if the dynamic version is has both type of scaling, the time lost in thread management is huge. In my case, it's better to keep the static scaling.

Increasing the number of process just increases the number of page faults and branch misses explodes. This is a something that could be interesting to study in how to reduce them.