Parallelization optimization techniques

Optimization of a C implementation of a Lattice Boltzmann method

Author :
Mr Gabriel Dos Santos

Supervisors :
Mr Julien Jaeger
Mr Hugo Taboada

université PARIS-SACLAY

M1 High Performance Computing and Simulation

May 1, 2022

Contents

Chapter 1

Introduction

1.1 Goals of the project

The aim of this project is to optimize a numerical simulation of *Computational Fluid Dynamics* (CFD) and to make it as scalable as possible. The base code is voluntarily "unoptomized" and contains errors that will need to be fixed through the use of debugging tools like GDB or Valgrind. Afterwards, the goal will be to identify and remove the different contentions in the program in order to achieve peak performance. To that extent, we shall use tracing and profiling tools such as MAQAO, Tau or Interpol. These will help up modify the base implementation and obtain a hybrid code that uses both MPI and multi-threading (e.g. through automatic parallelization with OpenMP).

1.2 Reminder on the problematic

The provided code aims at simulating a KÁRMÁN vortex street. We consider a fluid flowing down a 2D tube in which we place a round-shaped obstacle. This simulates, in principle, the setup we would have in a wind tunnel. As for the resolution, we use the *Lattice Boltzmann Method* (LBM), a relatively recent technique which consists in replacing the NAVIER-STOKES equations with a discretization of the BOLTZMANN equation in order to simulate the complex behavior of fluids using streaming and collision (relaxation) processes. This method is particularly well-suited for use in High Performance Computing (HPC) as it is highly parallelizable, which efficiently leverages the architecture of modern multi-core CPUs (*Central Processing Units*) and GPGPUs (*General Purpose Graphical Processing Units*).

1.3 Overview of the hardware and the tools used

In order to extract the most performance possible out of the simulation, it is important to know the hardware on which it is run. Moreover, there are many useful tools at our disposal that help us locate the "hotspots" in our programs. Whether they are functions or loops, it is valuable to know where they are and what is their cost, so that we can optimize them and get out the maximal available performance. The MPI implementation used also plays an important role, in that regard, choosing the right one is key. We will show that performance may vary dramatically depending on that choice.

1.3.1 Hardware

Regarding the hardware used for this project, almost every measurements shown in this report have been done on a laptop (see specs below). However, for an overview of the optimizations done to the code, some benchmarks have been run on the OB1 cluster, on an Intel *Knights Mill* (KNM) CPU (specs available below as well).

Here is an excerpt of the laptop's specifications:

```
$ lscpu
```

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  8
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
    Thread(s) per core:  2
    Core(s) per socket:  4
    Socket(s):           1
    CPU(s) scaling MHz:  33%
    CPU max MHz:         4200.0000
    CPU min MHz:         400.0000
Caches (sum of all):
  L1d:                   192 KiB (4 instances)
  L1i:                   128 KiB (4 instances)
  L2:                    5 MiB (4 instances)
  L3:                    8 MiB (1 instance)
NUMA:
  NUMA node(s):          1
  NUMA node0 CPU(s):     0-7
```

The flags for this CPU have been omitted here for reasons of space, however we would like to mention that Intel's AVX512 extension to the x86-64 ISA is available on this machine. This is important as some optimizations rely on the latter to make use of intrinsics and ensure complete vectorization of critical code sections.

The Intel KNM in the OB1 cluster also has access to the AVX512 extension, as such the optimizations done on the laptop will also transfer and apply here. You can find the CPU specifications below:

```
$ lscpu
```

```
Architecture:                     x86_64
  CPU op-mode(s):                 32-bit, 64-bit
  Byte Order:                     Little Endian
  Address sizes:                  46 bits physical, 48 bits virtual
CPU(s):                           288
Vendor ID:                        GenuineIntel
  Model name:                     Intel(R) Genuine Intel(R) CPU 0000 @
   ↪ 1.40GHz
    CPU family:                   6
    Model:                        133
    Thread(s) per core:           4
    Core(s) per socket:           72
    Socket(s):                    1
    Stepping:                     0
    CPU MHz:                      1163.970
    CPU max MHz:                  1500.0000
    CPU min MHz:                  1000.0000
Caches:
  L1d cache:                      2.3 MiB
  L1i cache:                      2.3 MiB
  L2 cache:                       36 MiB
NUMA:
  NUMA node(s):                   4
  NUMA node0 CPU(s):              0-17,  72-89,   144-161, 216-233
  NUMA node1 CPU(s):              18-35, 90-107,  162-179, 234-251
  NUMA node2 CPU(s):              36-53, 108-125, 180-197, 252-269
  NUMA node3 CPU(s):              54-71, 126-143, 198-215, 270-287
```

1.3.2 Tools

In order to optimize the simulation as much as possible, we have used a few debugging, profiling and tracing tools which we will shortly present here.

Debugging tools:

- The GNU debugger (`gdb`) was used to locate and fix the errors present in the base code.
- The memory debugging tool Valgrind was used to check the memory usage of the base implementation and fix potential memory leaks.

Profiling/tracing tools:

- The Interpol interposition library was used to trace and profile MPI calls. I am personally developing this tool as part of my M1 yearly project under the supervision of Mr Jean-Baptiste Besnard. Although it is primarily focusing on MPI's non-blocking routines, it proved to be exceptionally helpful at optimizing and factorizing the communications performed by the simulation. Its usage will be further discuss in the following chapters, however you can find its source code here: `https://github.com/async-mpi-benchmarks/interpol`.
- The *Modular Assembly Quality Analyzer and Optimizer* (MAQAO) framework, developped at UVSQ's LI-PaRAD laboratory, was used to evaluate the performance of the already optimized code in order to fine tune its behavior and achieve maximum performance.
- The `perf` profiler was used in conjonction with MAQAO to further enhance the simulation's performance.

1.3.3 MPI implementation

As MPI is just a standard, it is important to wisely choose the implementation to use. To that extent, three different MPI libraries have been tested on the LBM simulation:

- OpenMPI v4.1.2;
- MPICH v4.0.2;
- MPC v4.1.0.

For each of the above, benchmarks have been done in order to determine which one yields the better results.

Chapter 2

Development of a method for performance evaluation

Before making any optimizations, we tried to define a methodology to evaluate the performance of the simulation.

At each step of the optimization process, we start by determining where there is a slowdown in the code. Then, we try to find a way to patch it by implementing the solution. Finally, we benchmark the changes and interpret the results we get. Measuring the improvements includes evaluating both the strong and weak scalability of the code, trying it with various MPI implementations and verifying that the results are still correct.

To that end, some modifications had to be done to the base code in order to evaluate its performance. We also wrote various scripts, which we will present hereafter, that aim at assessing the performance gains and verify the correctness of our results.

2.1 Performance measurements *intra*-simulation

First, we had to find a way to efficiently and accurately measure how long the simulations takes to be completed. With this in mind, we decided to use the `clock_gettime` function from the `time.h` header, with the `CLOCK_MONOTONIC_RAW` ID. This clock has two convenient properties: on the one hand it is very precise (up to the nanosecond scale), on the other hand it is a separated hardware clock from the CPU's time-stamp counter (which we can get the value of thanks to the `rdtsc` instruction). This means that the clock is not affected by frequency variations not by adjustments from the Network Time Protocol (NTP).

The listing below demonstrates how we can measure the time it takes to execute a loop:

Filename: `loop_bench.c`

```c
#include <time.h>

int main() {
    struct timespec before, after;

    clock_gettime(CLOCK_MONOTONIC_RAW, &before);
    for (size_t i = 0; i < SIZE_MAX; i++) {
        // Put some code to benchmark here...
    }
    clock_gettime(CLOCK_MONOTONIC_RAW, &after);

    // `elapsed` stores the time in second that has passed between
    // the two `clock_gettime` calls.
    double elapsed = (after.tv_sec - before.tv_sec) +
                     (after.tv_nsec - before.tv_nsec) / 1e9;
}
```

The strategy implemented in the actual LBM simulation is to measure both the complete execution of the main loop (in `main.c`) and how long each iteration takes. This allows us to have a better insight on the simulation as we can track both its complete execution time, as well as how quickly each iteration is performed. We intentionally left the initialization and cleanup parts out of the benchmark, as they become negligible when we perform enough iterations.

Every MPI rank computes both the time to complete the whole loop, as well as an average time per iteration. At the end of the simulation, we perform a reduction on the master rank for both of these timings (using `MPI_Reduce` with the `MPI_SUM` operation), and average it depending on the number of MPI processes. This gives us the following output when running the code:

```
$ mpiexec -n 2 target/lbm

iterations = 160
# Rest of the configuration...
Rank 0: local average loop latency: 8.857610ms
Rank 0: local simulation latency:   1.419133194s
Rank 1: local average loop latency: 8.869306ms
Rank 1: local simulation latency:   1.419126143s

Global average loop latency:        8.863458ms
Global simulation latency:          1.419129669s
```

It is also worth mentioning that the inner-loop timing *does count* the saving step, where all the MPI processes send their data to the master rank, which dumps the buffered data to a file. These I/O operations are costly and affect the time measured. However, the user can override this behavior and not measure any in-loop I/O by compiling with the `-DNO_IO` flag. For convenience, this is included in the redesigned Makefile with the `DEF` variable, as you can see in the following example:

```
$ make run DEF=-DNO_IO

Rank 0: local average loop latency: 7.364646ms   (I/Os not measured)
Rank 0: local simulation latency:   1.184387623s
Rank 1: local average loop latency: 7.378001ms   (I/Os not measured)
Rank 1: local simulation latency:   1.184385106s

Global average loop latency:        7.371324ms   (I/Os not measured)
Global simulation latency:          1.184386364s
```

2.2 Checksum verification

To ensure that each modification of the code does not impact the result of the simulation, we wrote a small bash script (`checksum.sh`) that verifies for each frame contained in the `results.raw` file that we have the correct checksum compared to the reference given in the base code. Although this is small script, it also checks that the configuration of the simulation is similar (i.e. same width/height and number of frames) and will print a warning if these don't match. This handy script can be executed through the `make check` command, provided in every Makefile of the project. It will compare the local `results.raw` file to the reference in the base code directory.

Example output when checking the simulation results of the most optimized version against the reference one:

```
lbm/v6-fine_tuning $ make check

warning: simulations have different frames (base is 32 and
         provided simulation is 320).
info: checking only first 32 frames.
==> Verifying checksum for results.raw... ok
```

Note: *The reader is advised to check the real output in a terminal, it looks much better than on this report.*

2.3 Benchmarks automatization

In order to gain as much as possible when evaluating the performance of changes done to the code, we wrote a bash script (`bench.sh`) that can perform both *weak* and *strong scalability* benchmarks on the modified code. Using a gnuplot script, it automatically generates a histogram plot with the recorded values of the simulation runs. This makes it easy to measure the code's "new" scalability after each change in the implementation of the LBM.

The strategy here was twofold. For the strong scaling we start by running the simulation on a single process, then we double that number at each subsequent run until we reach the limit of available cores on the machine. This limit is computed from the output of the `lscpu` command, by multiplying the number of logical cores per socket by the number of NUMA nodes available on the machine.

For the weak scaling, we start from a scaled down version of the simulation (dimension-wise) and double one of the dimensions for each subsequent run (see example script output below). The number of MPI processes/OpenMP threads is fixed for each benchmarks to measure only how the size of the problem affects the performance.

As with the checksum, the Makefiles provide a `make check` command that runs the scripts for the user. For ease of use, it is possible to define the mode in which to benchmark the code, the MPI run command, as well as the flags to use when running the simulation.

An example of its use with the MPC implementation of MPI (which relies on different run commands and flags than OpenMPI or MPICH):

```
lbm/v6-fine_tuning $ make bench MODE=weak MPICMD=mpcrun
FLAGS="-p=4 -n=4"
```

```
==> Starting weak scalability benchmark on target/lbm...
Running with dimensions 400x80... done
Running with dimensions 400x160... done
Running with dimensions 800x160... done
Running with dimensions 800x320... done
==> Generating weak scalability histogram plot in
↪   plots/results_weak.png...
[+] /home/gabrl/lbm/v6-fine_tuning/benchmarks/bench_weak.dat
[+] /home/gabrl/lbm/v6-fine_tuning/plots/results_weak.png

Finished running benchmarks in 3.71s.
```

2.4 Results interpretation

This leads us to the final step of the performance evaluation. Thanks to the generated plots, it is easy to determine how the simulation scales as we increase the number of processes (*strong* scalability) or the size of the problem (*weak* scalability). We can simply compare the time taken for each run and then plot it as a graph comparing the ideal scalability (2x) compared to the one measured.

The figure 2.1 below is an example output graph from the benchmark we just run with the `make check` command in the above section. It is visible that the time taken to perform the simulation roughly increases twofolds as we double the size of the problem.

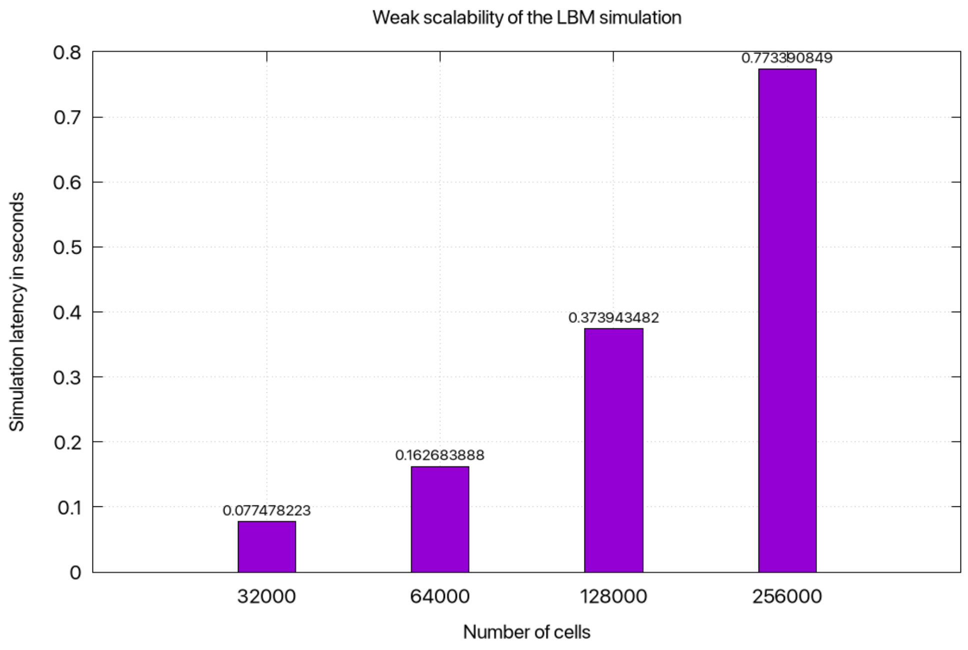


Figure 2.1: Plot generated by the weak scalability benchmark in the previous code snippet

Chapter 3

Base code corrections

As there was quite a large number of corrections to be made on the base code, we felt like it deserved an entire section dedicated to all the changes done to it. We will start by explaining the thinking behind the directory reorganization, which required a rewrite of the Makefile. Then, we will take a look at how the code was formatted to make it easier to read and to modify, applying good programming practices where possible. Afterwards, we will deep dive in the process of correcting the errors present in the base implementation, notably how they were debugged and fixed. Finally, we will evaluate the performance of the base code and look for potential contentions that are preventing performance improvements in the program.

3.1 Directory reorganization and Makefile rewrite

To make the directory structure clearer, we created two directories: `src/` where we put all the `.c` source files, and `include/`, where we put the header files (`.h`). This needed a Makefile rewrite so that we can compile the simulation correctly. To avoid having all the object files artifacts needed for the compilation in the root of the project, we compile them away in a separate folder (`target/deps/`) while the binaries for the LBM and the `display` helper program go in the already created `target/` directory. This structure is duplicated each time we make an optimization. Although it uses more disc space, it also makes it a lot easier to follow the evolution of the implementation and benchmark them side by side.

Moreover, the Makefile gives us the possibility to define more commands (as we saw earlier) to gain more flexibility, e.g. when we want to launch scripts. These are all located in a separate folder, as they are universal for all the optimized versions of the simulation.

The available commands are:

- `make build`: compiles both the simulation and the helper `display` program;
- `make run`: runs the simulation with the default MPI command and flags (`mpiexec -n 2`);
- `make check`: runs the `checksum.sh` script using the simulation's output against the reference results in the `base` directory;
- `make gif`: generates the gif of the simulation and opens it in a browser (**brave** by default but can be overridden by specifying the `BROWSER` variable, e.g. `make gif BROWSER=firefox`).
- `make bench`: runs the benchmark script in a mode specified by the user through the `MODE` variable (set to ”strong” by default);
- `make trace`: profiles the simulation using the Interpol library and generates a report about the collected traces.

Overall, these changes leave us with a much simpler code-base structure, even when the code as been compiled, as you can see in the listing below:

```
$ tree

Permissions Name
drwxr-xr-x  .
drwxr-xr-x  |-- include
.rw-r--r--  |  |-- *.h # Header files
drwxr-xr-x  |-- src
.rw-r--r--  |  |-- *.c
drwxr-xr-x  |-- target
drwxr-xr-x  |  |-- deps
.rw-r--r--  |  |  |-- *.o # Object files
.rwxr-xr-x  |  |-- display
.rwxr-xr-x  |  |-- lbm
.rw-r--r--  |-- Makefile
.rw-r--r--  |-- config.txt
```

3.2 Formatting of the base code

The next step was to format the code to make it more readable and easier to fix. To that end, we used a tool provided by clang, `clang-format`. The latter allows us to very easily to format an entire code base to the same style (indentation, spacing, maximum number of characters per line, etc). This, in turn, makes the code much more manageable to work with and is also a good practice when developing a project with multiple people.

Other miscellaneous changes include removing variable declarations at the start of the functions (notably loop counters) and adding the `const` keyword where possible. The thinking here is that code, especially C, is meant to be read from top to bottom. Thus, while having all the variable declarations in one place makes it easy to find them, it forces the programmer to keep their values in mind and often requires scrolling back up to the top when we forget them. Another point can be made regarding loop counters, as having these shared throughout the whole function is not good practice, even more so when doing parallel programming. A counter should *always* be part of the loop declaration and it is generally a good idea to give it the `size_t` type, which is always positive and, for example, prevents indexing an array using a negative index.

As this step required reading through the code, it was really helpful in understanding its structure and how each element fits with the others. This work also allowed us to spot oddities upstream so that we can fix them later, when they become the main contention to the simulation's performance.

3.3 Removal of errors

Once the code was restructured and ready to go, we were able to start fixing the errors. In this next section, we will explain every step of that process, so that we end up with a simulation that runs and that can be benchmarked.

3.3.1 Compilation errors

We started by fixing the compilation errors. The first one happened at the linking phase of the compilation because of multiple declarations of the same variables. These were declared in the `include/lbm_phys.h` header, a file which was included in multiple source files, thus generating the linking error. The fix was simply to add the `extern` keyword, as shown in the listing below:

Filename: `include/lbm_phys.h`

```c
extern int const opposite_of[DIRECTIONS];
extern double const equil_weight[DIRECTIONS];
extern Vector const direction_matrix[DIRECTIONS];
```

The second error was because of two declarations of a main function: one in the `src/main.c` file, the other in the `src/usleep.c` file. As the latter only consisted in an empty main function, we removed the file entirely.

3.3.2 Segmentation fault

After that, we had a code that could compile correctly, without any warnings. However, when run, the program panicked because of a segmentation fault. To fix that, we used the GNU debugger, GDB. The following listing gives us the debugger's output when runnign the simulation:

```
$ mpiexec -n 1 gdb target/lbm

Thread 1 "target/lbm" received signal SIGSEGV, Segmentation fault.
  in setup_init_state_global_poiseuille_profile at lbm_init.c:85
85 |    Mesh_get_cell(mesh, i, j)[k] =
86 |        compute_equilibrium_profile(v, density, k);
(gdb) backtrace
#0  in setup_init_state_global_poiseuille_profile at lbm_init.c:85
#1  in setup_init_state at lbm_init.c:155
#2  in main at main.c:159
```

Thanks to GDB, we were able to track where the error came from. Looking at the code and especially where the `mesh` variable was initialized, it was easy to spot that the line allocating the cells was commented out. We uncommented that line and the error check underneath, also removing a `NULL` affectation at the same time. This gave us the following, fixed code:

Filename: src/lbm_struct.c

```c
void Mesh_init(Mesh* mesh, uint32_t width, uint32_t height)
{
    // Setup parameters
    mesh->width = width;
    mesh->height = height;

    // Allocate memory for cells
    mesh->cells = malloc(width * height * DIRECTIONS * sizeof(double));
    if (mesh->cells == NULL) {
        perror("malloc");
        abort();
    }
}
```

3.3.3 Removal of the slowdown (*sleep*)

At this point, the simulation ran properly but was extremely slow, at around 1 iterations per second. Thanks to the formatting step we had done earlier, we had spotted a suspicious macro that was called at the end of each communications exchanges between the MPI processes:

Filename” src/lbm_comm.c

```
void lbm_comm_ghost_exchange(lbm_comm_t* mesh, Mesh* mesh_to_process)
{
    // Rest of the function...

    // Wait for I/Os to finish, VERY important, do not remove.
    FLUSH_INOUT();
}
```

Jumping to its definition, we get the following:

Filename: include/lbm_config.h

```
#define concat(a,b,c,d,e) a##b##c##d##e
#define __FLUSH_INOUT__ concat(s,l,e,e,p)(1)
#define FLUSH_INOUT() __FLUSH_INOUT__
```

This macro uses the C preprocessor to concatenate the letters ”s”, ”l”, ”e”, ”e”, ”p” together, thus inserting a call to the `sleep` function, causing the one second slowdown at each iteration. Of course, we removed the call to this macro completely from the code.

We decided not to benchmark the base code without fixing this error first, as we would have needed to run it with very few iterations. This would not have been an interesting comparison and was thus left out.

3.3.4 *Deadlock* at the end of the execution

Now that the code was running much faster, we could try to run a complete simulation to check its output. However, when reaching the last iteration, the program stopped and was stuck in a loop. Investing this, it became clear that upon closing the file, the master rank made a call to `MPI_Barrier`. However, as no other MPI processes could make this function call, rank 0 was stuck in a deadlock. The barrier was to removed from the `close_file` function and moved back to `main` to ensure that every process has finished its simulation loop before closing the file.

Filename: src/main.c

```c
int main(int argc, char* argv[argc + 1])
{
    // Initialization and rest of function...

   // Time steps
    for (ssize_t i = 1; i < ITERATIONS; i++) {
        // Rest of loop...

        // Save step
        if (i % WRITE_STEP_INTERVAL == 0 &&
            lbm_gbl_config.output_filename != NULL) {
            save_frame_all_domain(fp, &mesh, &temp_render);
        }
    }

    MPI_Barrier(MPI_COMM_WORLD); // Inserted here
    if (rank == RANK_MASTER && fp != NULL) {
        close_file(fp); // Removed from this function
    }
}
```

3.4 Performance and scalability evaluations

Now that the code compiled and ran without any issues, we were finally able to benchmark it and evaluate its scalability. To that end, we first used the `checksum.sh` to verify that we were still computing the same results as the provided reference (which is only 200 frames compared to the 320 we get with the default configuration parameters):

```
$ make check

warning: simulations have different frames (base is 200 and
         provided simulation is 320).
info: checking only first 200 frames.
==> Verifying checksum for results.raw... ok
```

The output being correct, we then measured the code's performance and scalability. For these first sets of benchmarks, we chose the OpenMPI implementation as it was surprisingly the fastest out of the three mentioned in the introduction. This allowed us to reduce the time necessary to run the benchmarks and also gave us the "best" measures we could hope for from the base code. We started by evaluating the strong scalability on the laptop, using a single MPI process as our reference and doubling it for each subsequent run, up to 4 processes. To avoid getting warnings from OpenMPI not finding CUDA libraries, we need to pass an option, which is facilitated by the Makefile's provided variables:

```
lbm/base $ make bench MODE="strong" MPICMD="mpirun"
FLAGS="--mca opal_warn_on_missing_libcuda 0"
```

```
==> Starting strong scalability benchmark on target/lbm...
Running with 1 processes... done
Running with 2 processes... done
Running with 4 processes... done
==> Generating strong scalability histogram plot in
↪   plots/results_strong.png...
[+] lbm/base/benchmarks/bench_strong.dat
[+] lbm/base/plots/results_strong.png

Finished running benchmarks in 420.69s.
```

The `bench.sh` script generates the following graph, plotting the time taken for each run of the simulation depending on the number of processes:

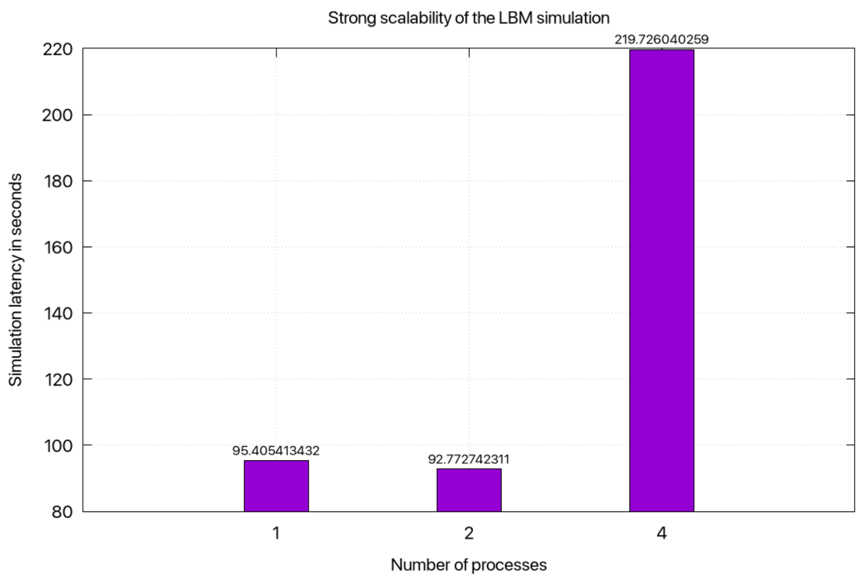


Figure 3.1: Latency of the simulation over the number of MPI processes

Figure 3.1 shows us that the simulation does not scale well at all. We only get a 2.84% speedup with two processes compared to only one. The performance with the full four processes is even worth, being 2.3 times slower than on a single rank.

Then, we need to evaluate how the program performs as we increase the dimensions of the problem, that is to say its width and height. We keep a fixed number of iterations (at 16000) and a fixed number of MPI processes. For each run, we double one of the dimensions of the simulation, the goal being to observe that we take around twice as long to complete it.

```
lbm/base $ make bench MODE="weak" MPICMD="mpirun" FLAGS="--mca
opal_warn_on_missing_libcuda 0"
```

```
==> Starting weak scalability benchmark on target/lbm...
Running with dimensions 800x160... done
Running with dimensions 800x320... done
Running with dimensions 1600x320... done
Running with dimensions 1600x640... done
==> Generating weak scalability histogram plot in
↪   plots/results_weak.png...
[+] /home/gabrl/uni/m1/s2/top/project/base/benchmarks/bench_weak.dat
[+] /home/gabrl/uni/m1/s2/top/project/base/plots/results_weak.png

Finished running benchmarks in 1250.13s.
```

We get the following output graph:

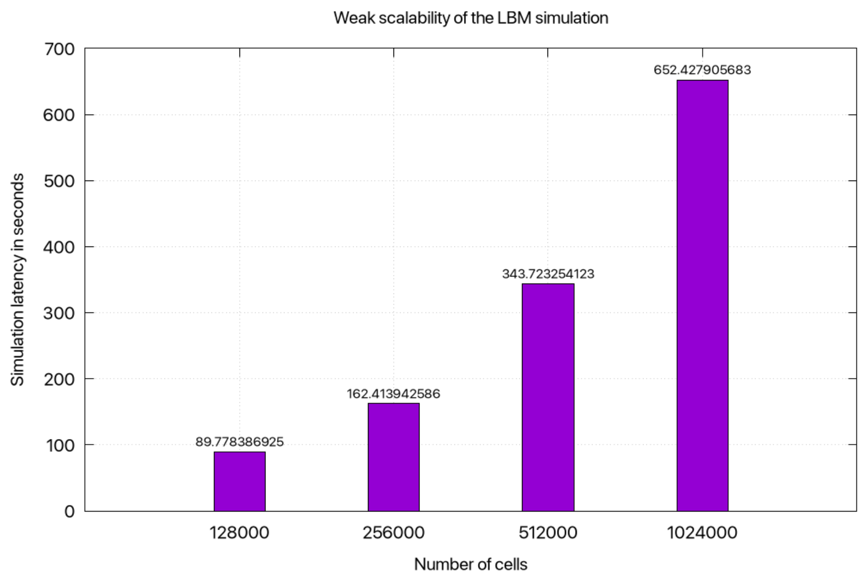


Figure 3.2: Latency of the simulation over the number of cells to compute

Compared to the strong scalability, the weak one is much better as we take around 1.941 times longer each time we double the dimensions of the problem.

Chapter 4

Optimizing the simulation

Now that we have evaluated the performance of the base simulation code, we can focus on optimizing it. In the following section, we will present the various optimizations done to the code, starting by analyzing where does the slowdown come from, how can we fix it, and what is the performance impact of the patch. At each step of the optimization process, we will verify that our code still runs as expected and gives out the correct result.

4.1 Useless code elimination

Before making any significant changes, we removed any useless code that was still present. Here is an exhaustive list of the code sections that were eliminated:

- In `src/main.c:main`: the `printf` function displaying the progress of the simulation's loop;
- In `src/lbm_comm.c:helper_get_rank_id`: the `else if` block was merged with the first condition and the `else` block was removed;
- In `src/lbm_comm.c:lbm_comm_ghost_exchange`: the duplicated left to right phase.

These changes to the base code are available in the `v1-no_useless_code` directory.

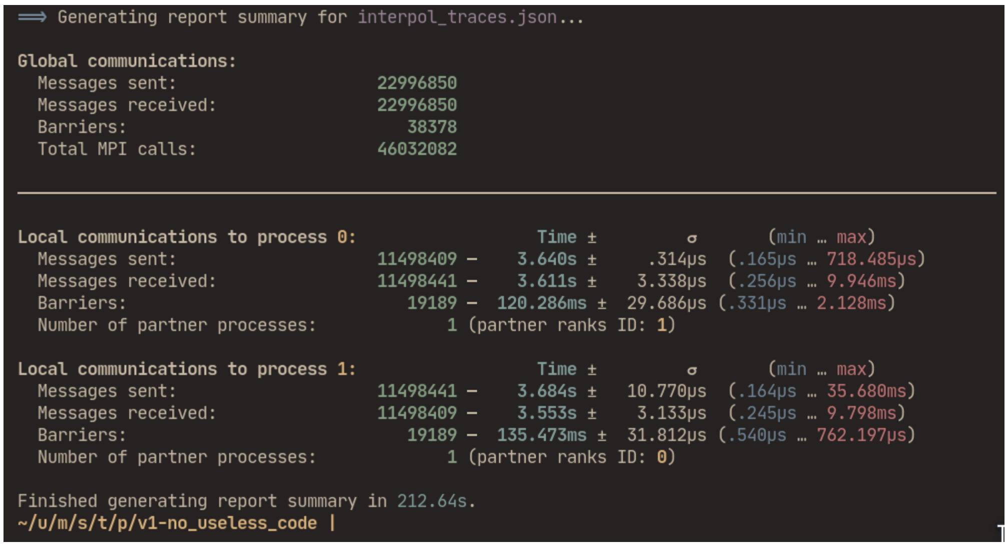
4.1.1 Removal of useless `MPI_Barrier` calls

In order to inspect the communications made by the MPI processes, we used to Interpol library. As mentioned in the introduction, this is a project in which I am involved and have contributed to the interposition of the MPI calls. We will start by making a brief overview of how the Interpol tool operates.

Interpol is designed as a lightweight MPI profiler, written primarily in Rust. This allows it to reach performance levels comparable to what we could achieve with C or C++, while also offering memory and thread safety out of the box. Interpol will record every calls to MPI routines that it has redefined. As such, it is able to support any MPI implementation that strictly follows the standard.

When a call to MPI is made, Interpol intercepts and registers information about it. More crucially, it tracks the time taken to perform the operation. On a more specific level, e.g. for `MPI_Send`, the tool will also keep track of the number of bytes sent and to which rank. At the end of the traced program's execution, Interpol will output a JSON file containing all the recorded MPI events. As this tool is still in development, we wrote a bash script (`interpol_report.sh`) that extracts valuable information out of the JSON output.

Using the Makefile's `trace` command, we ran Interpol on the base code, with two MPI processes and only 1600 iterations. We then obtained the following report summary:



```
⟹ Generating report summary for interpol_traces.json...

Global communications:
  Messages sent:                 22996850
  Messages received:             22996850
  Barriers:                         38378
  Total MPI calls:               46032082

──────────────────────────────────────────────────────────────────────────────────

Local communications to process 0:                Time ±         σ       (min … max)
  Messages sent:                 11498409 –     3.640s ±      .314μs  (.165μs … 718.485μs)
  Messages received:             11498441 –     3.611s ±     3.338μs  (.256μs … 9.946ms)
  Barriers:                         19189 –  120.286ms ±  29.686μs (.331μs … 2.128ms)
  Number of partner processes:          1 (partner ranks ID: 1)

Local communications to process 1:                Time ±         σ       (min … max)
  Messages sent:                 11498441 –     3.684s ±   10.770μs  (.164μs … 35.680ms)
  Messages received:             11498409 –     3.553s ±     3.133μs  (.245μs … 9.798ms)
  Barriers:                         19189 –  135.473ms ±  31.812μs (.540μs … 762.197μs)
  Number of partner processes:          1 (partner ranks ID: 0)

Finished generating report summary in 212.64s.
~/u/m/s/t/p/v1-no_useless_code |
```

Figure 4.1: Interpol's report summary on the execution of the base code

It is quite obvious that the simulation is making a lot of calls to MPI, over 46 million in this case. As we're running MPI on a laptop with a single socket, this means that the data sent and received by each process has to be copied directly in memory. This in turn puts a huge load on the CPU and slows down the code drastically. We can also notice that both processes spend a lot of time inside calls to `MPI_Barrier`, which means that they are constantly waiting on each other. However, as we are using MPI's blocking routines, these calls are useless and can be removed completely.

Every call to `MPI_Barrier` has thus been removed from the code, apart from the one after the main loop. However, we can go much further than that by factorizing the MPI communications, which we will see in the next section.

4.1.2 Factorizing MPI communications

As mentioned just before, we have a lot of messages being exchanged by the processes in the simulation (over 20 million per process in the example above). When looking at the code, we can notice that the functions `lbm_comm_sync_ghosts_horizontal` and `lbm_comm_sync_ghosts_vertical` are exchanging data in a very suspicious way. Indeed, instead of sending a buffer with all the updated elements at once, they are performing the communications one element at a time. This can be easily fixed for the horizontal communications as we have access to the `Mesh_get_col` function, which allows us to retrieve an entire column of the mesh at once. The refactored code for this function is the following, we load/unload all the items being exchanged directly in the mesh to process with a single call to `MPI_Send` or `MPI_Recv`, depending on the communication type:

Filename: `src/lbm_comm.c`

```
void lbm_comm_sync_ghosts_horizontal(Mesh* mesh_to_process,
                                     lbm_comm_type_t comm_type,
                                     int target_rank, uint32_t x)
{
  switch (comm_type) {
    case COMM_SEND:
      MPI_Send(Mesh_get_col(mesh_to_process, x), DIRECTIONS,
               MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD);
      break;
      case COMM_RECV:
        MPI_Recv(Mesh_get_col(mesh_to_process, x), DIRECTIONS, MPI_DOUBLE,
                 target_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        break;
    }
}
```

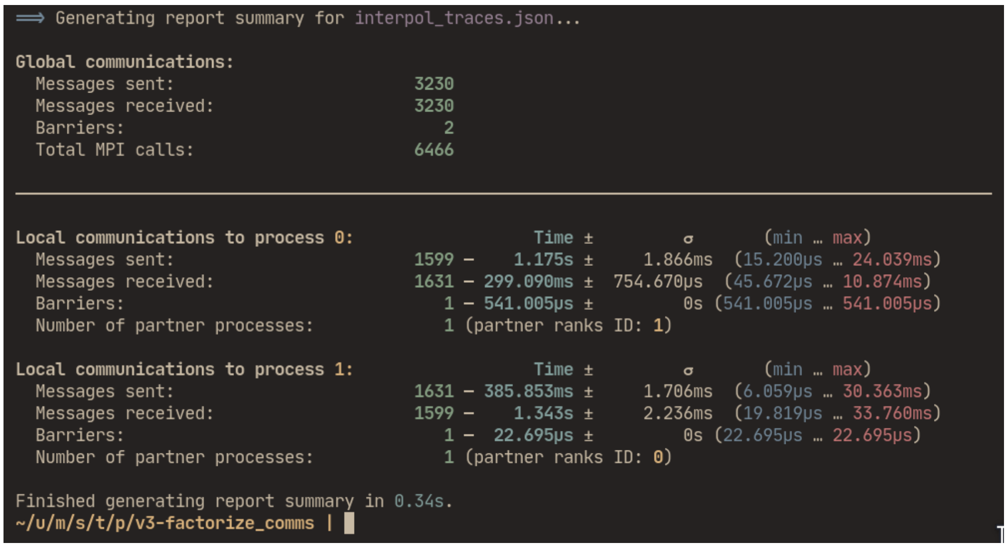
Some sections of the previous function have been omitted for space reasons.

For the vertical version of the exchange, it is a little bit more complicated as we do not have a function such as `Mesh_get_row` that would return an entire row of the mesh. However, we have at our disposal a `lbm_comm_t mesh` which conveniently has a pre-allocated buffer that we are able to use to exchange the ghost cells with a single MPI communication. We must load/unload this buffer to get all of the elements that are being transferred but this saves from having to make multiple calls to MPI routines. The refactored code can be found below:

Filename: `src/lbm_comm.c`

```c
void lbm_comm_sync_ghosts_vertical(lbm_comm_t* mesh,
                                   Mesh* mesh_to_process,
                                   lbm_comm_type_t comm_type,
                                   int target_rank,
                                   uint32_t y)
{
    MPI_Status status;
    switch (comm_type) {
        case COMM_SEND:
            for (size_t x = 1; x < mesh_to_process->width - 2; x++) {
                for (size_t k = 0; k < DIRECTIONS; k++) {
                    mesh->buffer[(x - 1) * DIRECTIONS + k] =
                        Mesh_get_cell(mesh_to_process, x, y)[k];
                }
            }
            MPI_Send(mesh->buffer, DIRECTIONS * (mesh_to_process ->width -
            ↪   2), MPI_DOUBLE, target_rank, 0,
                     MPI_COMM_WORLD);
            break;
        case COMM_RECV:
            MPI_Recv(mesh->buffer, DIRECTIONS * (mesh_to_process->width -
            ↪   2),
                     MPI_DOUBLE, target_rank, 0, MPI_COMM_WORLD, &status);
            for (size_t x = 1; x < mesh_to_process->width - 2; x++) {
                for (size_t k = 0; k < DIRECTIONS; k++) {
                    Mesh_get_cell(mesh_to_process, x, y)[k] =
                        mesh->buffer[(x - 1) * DIRECTIONS + k];
                }
            }
            break;
        default:
            fatal("unknown type of communication");
    }
}
```

After these major optimization changes, we can re-run Interpol on the simulation (same parameters as before) and we get the following report:



```
⟹ Generating report summary for interpol_traces.json...

Global communications:
  Messages sent:                         3230
  Messages received:                     3230
  Barriers:                                 2
  Total MPI calls:                       6466

────────────────────────────────────────────────────────────────────────────────────────────────

Local communications to process 0:                  Time ±          σ       (min … max)
  Messages sent:                         1599 -     1.175s ±    1.866ms  (15.200µs … 24.039ms)
  Messages received:                     1631 - 299.090ms ± 754.670µs  (45.672µs … 10.874ms)
  Barriers:                                 1 - 541.005µs ±         0s (541.005µs … 541.005µs)
  Number of partner processes:              1 (partner ranks ID: 1)

Local communications to process 1:                  Time ±          σ       (min … max)
  Messages sent:                         1631 - 385.853ms ±    1.706ms  (6.059µs … 30.363ms)
  Messages received:                     1599 -     1.343s ±    2.236ms  (19.819µs … 33.760ms)
  Barriers:                                 1 -  22.695µs ±         0s (22.695µs … 22.695µs)
  Number of partner processes:              1 (partner ranks ID: 0)

Finished generating report summary in 0.34s.
~/u/m/s/t/p/v3-factorize_comms | █
```

Figure 4.2: Interpol report summary after MPI optimizations

We have decreased the number of calls to MPI routines by a whopping 7000+ factor, which made a huge impact on the execution time of the simulation. We can now run the strong scalability benchmark on the optimized code using MPICH and we get the following plot:

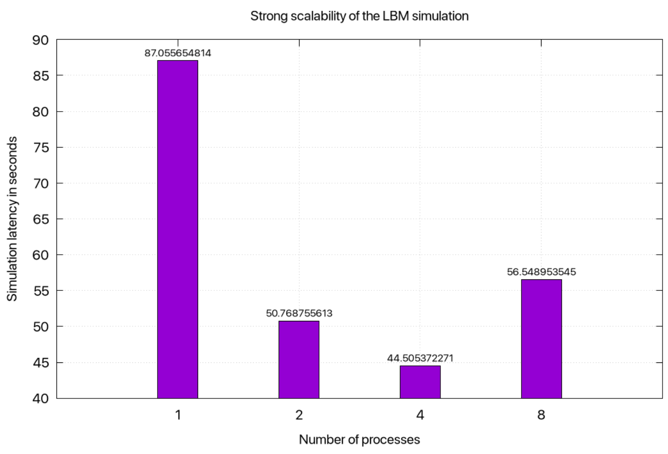


Figure 4.3: Latency of the simulation over the number of processes

The histogram in figure 4.3 shows that the simulation on a single rank is now a little bit faster than previously, by about 15 seconds. Although the strong scalability is now much better, it is far from perfect has we only have a 1.714 speedup between one and two MPI processes. This number crumbles as we increase up to 4 processes and only get 1.14 increase in computation speed. Finally, as we go up to 8 processes, which twice as much as the number of physical cores on the machine, the simulation latency rises up again and is slower than with only 2 processes.

On the other hand, the weak scalability stays efficient as we still have around a 2x increase of the latency as we double the size of the problem, as shown in the graph 4.4 below:

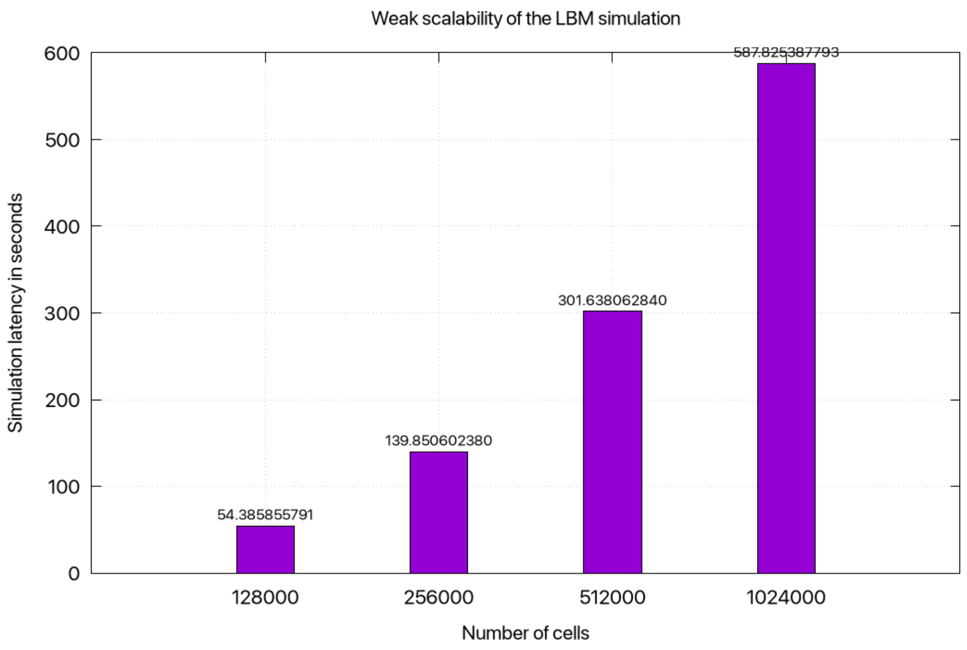


Figure 4.4: Latency of the simulation over the number of cells

4.2 Changing the domain decomposition

The next optimization we are going to tackle is the domain decomposition. Indeed, in the base code, the domain is split horizontally in equal parts for each MPI process. This means that each rank has to send as many ghosts cells to its neighboors as they are cells in the width of the simulation. A more efficient way to split the domain would be to do it vertically. Conveniently, the code to perform just that was commented out in the `lbm_comm_init` function:

Filename: `src/lbm_comm.c`

```c
void lbm_comm_init(lbm_comm_t* mesh_comm, int rank, int comm_size,
                   uint32_t width, uint32_t height)
{
    // Compute splitting
    int nb_x = lbm_helper_pgcd(comm_size, height);
    int nb_y = comm_size / nb_x;

    assert(nb_x * nb_y == comm_size);
    if (height % nb_y != 0) {
        fatal("Can't get a 2D cut for current problem size and number of "
              "processes.");
    }

    // Rest of the function...
}
```

Benchmarking the strong and weak scalability, we can get the following plots:

Graph 4.5 shows that we have not gained any significant performance out of this change, which was a bit surprising. On a small note, we can see that the latency of the simulation with 8 processes is marginally better than before, Overall, the strong scalability does not improve with this optimization.

Regarding the weak scalability in figure 4.6, we observe the same tendency as before, with an average latency doubling each time we double the dimensions of the simulation.

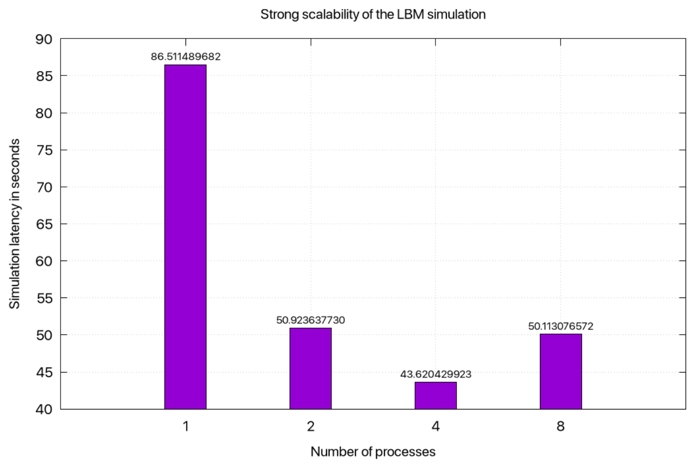


Figure 4.5: Latency of the simulation over the number of processes

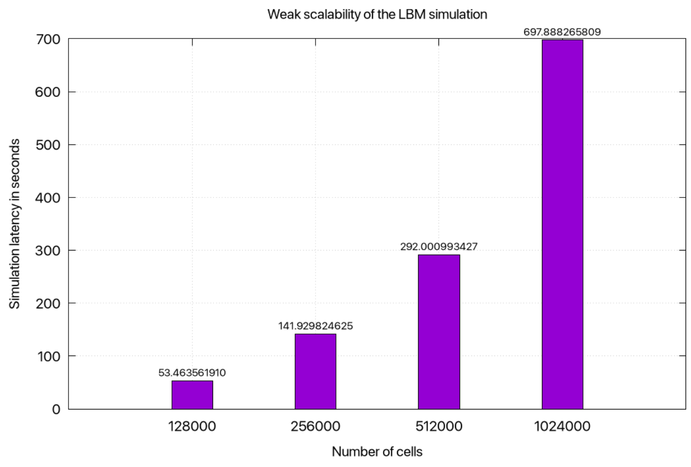


Figure 4.6: Latency of the simulation over the number of cells

4.3 Automatic parallelization with OpenMP

We reach the most interesting part of the optimization process here, as we incorporate OpenMP into the code-base. This will give us a hybrid MPI/OpenMP implementation that will hopefully dramatically increases the performance of the code. To enable multi-threading with MPI, we had to change its initialization for `MPI_Init_thread`.

Filename: `src/main.c`

```c
int main(int argc, char* argv[argc + 1])
{
    int provided;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    // Rest of the function...
}
```

Afterwards, we simply had to add *pragmas* on top of the loops we wanted to parallelize. Using the MAQAO framework and the perf profiling tool, it came out that the most expensive functions were `collision` and `propagation`, in the `src/lbm_phys.c` file. Thus we added *pragma* macros annotating the outer loop to automatically parallelize it with OpenMP.

Here, we used the *collapse* clause to indicate to OpenMP that these are perfectly nested loop. After some trial and error with the schedule, we ended picking *static* as it gave better results than the *dynamic* and *guided* clauses. The best results were yielded using 2 OMP threads.

Filename: `src/lbm_phys.c`

```c
void collision(Mesh* mesh_out, const Mesh* mesh_in)
{
    assert(mesh_in->width == mesh_out->width);
    assert(mesh_in->height == mesh_out->height);

    // Loop on all inner cells
    #pragma omp parallel for collapse(2) schedule(static)
    for (size_t j = 1; j < mesh_in->height - 1; j++) {
        for (size_t i = 1; i < mesh_in->width - 1; i++) {
            compute_cell_collision(Mesh_get_cell(mesh_out, i, j),
            ↪  Mesh_get_cell(mesh_in, i, j));
        }
    }
}

void propagation(Mesh* mesh_out, Mesh const* mesh_in)
{
    // Loop on all cells
    #pragma omp parallel for collapse(3) schedule(static)
    for (size_t j = 0; j < mesh_out->height; j++) {
        for (size_t i = 0; i < mesh_out->width; i++) {
            // For all direction
            for (size_t k = 0; k < DIRECTIONS; k++) {
                // Compute destination point
                ssize_t ii = (i + direction_matrix[k][0]);
                ssize_t jj = (j + direction_matrix[k][1]);
                // Propagate to neighboor nodes
                if ((ii >= 0 && ii < mesh_out->width) &&
                    (jj >= 0 && jj < mesh_out->height))
                {
                    Mesh_get_cell(mesh_out, ii, jj)[k] =
                        Mesh_get_cell(mesh_in, i, j)[k];
                }
            }
        }
    }
}
```

Here are the scalability benchmarks results:
Although the latency decreased across the board the strong scalability was worse than before. On the other hand, the weak scalability stayed as it was in the previous implementations.

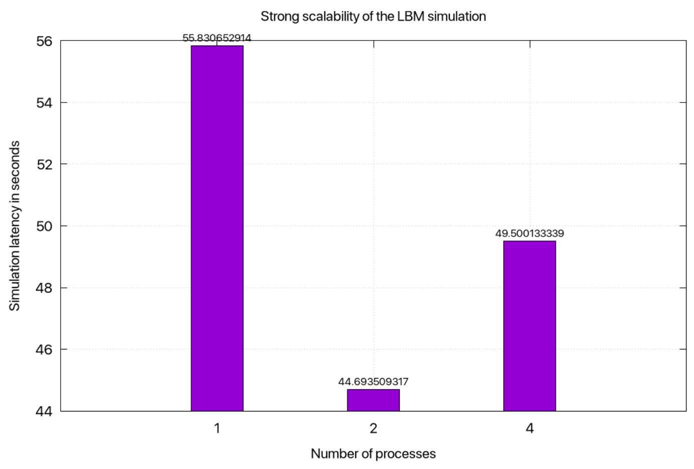


Figure 4.7: Latency of the simulation over the number of processes

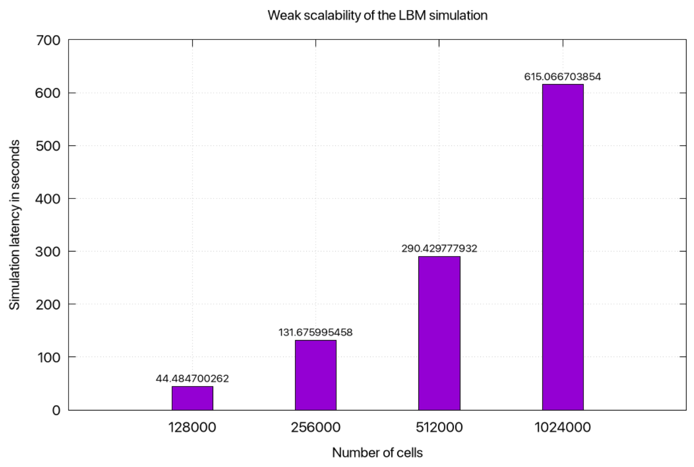


Figure 4.8: Latency of the simulation over the number of cells

4.4 Fine-tuning, vectorization and more...

For the last bits of optimization, we went down to the core by fine-tuning the compiler's flags, MPI and OpenMP's settings, manually vectorizing functions using AVX512 intrinsics, simplifying memory accesses by changing arrays of structures (AoS) to structure of arrays (SoA) and rearranged loops to better exploit the cache locality of data.

To start off, we added the following optimization flags to the compiler:

Filename: `lbm/v6-fine_tuning/Makefile`

```
OFLAGS := -march=native -mtune=native -mavx2 -Ofast -ffast-math
↪  -funsafe-math-optimizations -finline-functions -funroll-loops
↪  -floop-interchange -fpeel-loops -ftree-vectorize -ftree-loop-vectorize
↪  -fomit-frame-pointer -flto
```

We then removed the *collapse* clauses off of the *pragma omp* macros as these seem to reduce performance more than the increased it.

We also changed the `get_cell_density` and `get_cell_velocity` functions. We entirely removed the loop by unrolling and vectorizing it using AVX512 intrinsics. The previous versions of these functions were slighted optimized and kept to ensure compatibility on machines which don't support the AVX512 extension.

Filename: `lbm_phys.c`

```
#ifdef __AVX512F__
inline double get_cell_density(lbm_mesh_cell_t const cell) {
    __m512d vcell = _mm512_loadu_pd(cell);
    double res = _mm512_reduce_add_pd(vcell);
    res += cell[DIRECTIONS - 1];
    return res;
}

inline void get_cell_velocity(Vector v, lbm_mesh_cell_t const cell,
                              double const cell_density) {
    __m512d dir_a = _mm512_loadu_pd(direction_a + 1);
    __m512d dir_b = _mm512_loadu_pd(direction_b + 1);
    __m512d vcell = _mm512_loadu_pd(cell + 1);
    v[0] = _mm512_reduce_add_pd(_mm512_mul_pd(vcell, dir_a)) /
    ↪  cell_density;
    v[1] = _mm512_reduce_add_pd(_mm512_mul_pd(vcell, dir_b)) /
    ↪  cell_density;
}
```

Finally, we interchanged the loops in the `collision` and `propagation` functions in order to better exploit the CPU's cache.

Filename: lbm_phys.c

```c
void collision(Mesh* mesh_out, const Mesh* mesh_in) {
    #pragma omp parallel {
        #pragma omp for schedule(static)
        for (size_t i = 1; i < mesh_in->width - 1; i++) {
            for (size_t j = 1; j < mesh_in->height - 1; j++) {
                compute_cell_collision(// omitted );
            }
        }
    }
}

void propagation(Mesh* mesh_out, Mesh const* mesh_in) {
    #pragma omp parallel {
        #pragma omp for schedule(static)
        for (size_t i = 0; i < mesh_out->width; i++) {
            for (size_t k = 0; k < DIRECTIONS; k++) {
                double dir_a = direction_a[k];
                double dir_b = direction_b[k];
                for (size_t j = 0; j < mesh_out->height; j++) {
                    // Compute destination point ...
                }
            }
        }
    }
}
```

Final strong scalability plot: The weak scalability stayed the same although

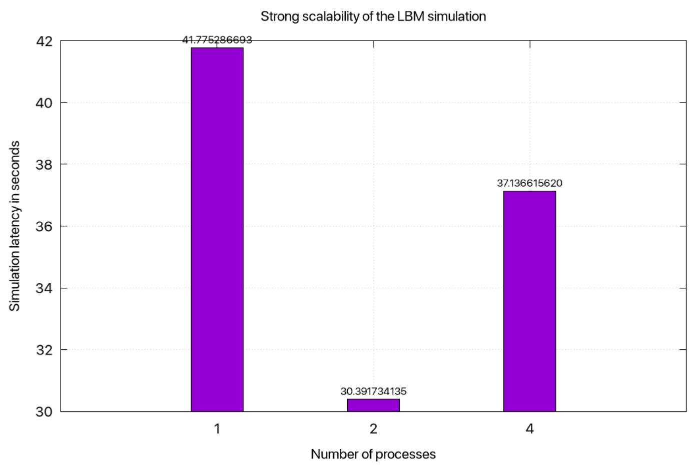
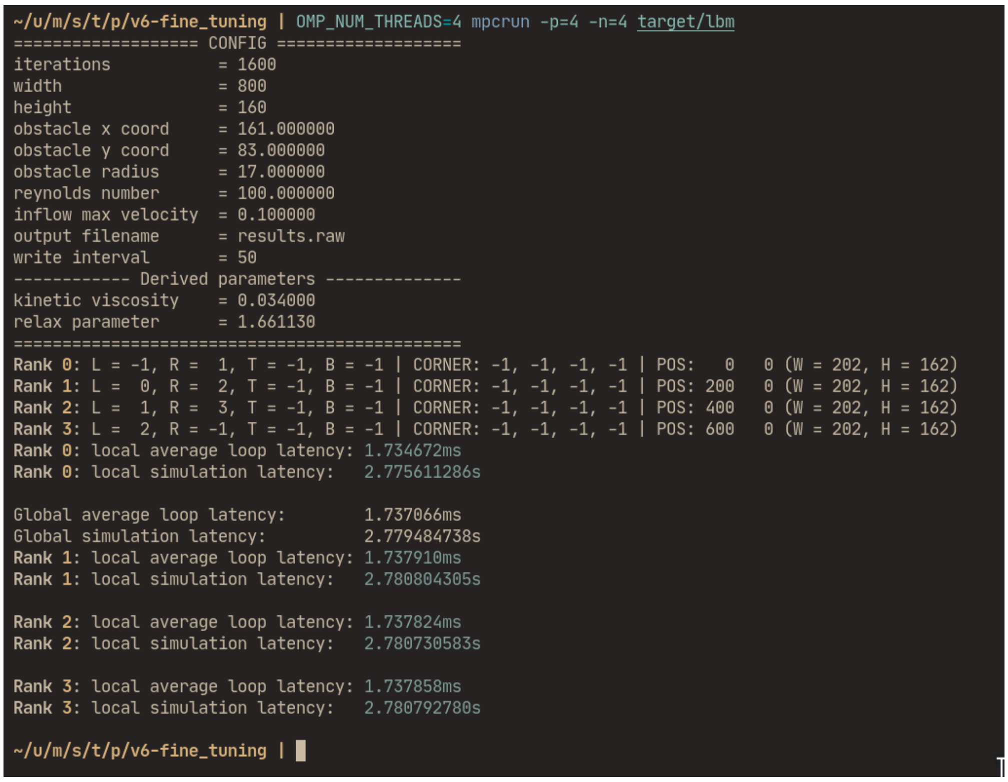


Figure 4.9: Latency of the simulation over the number of processes

the compute time decreased.

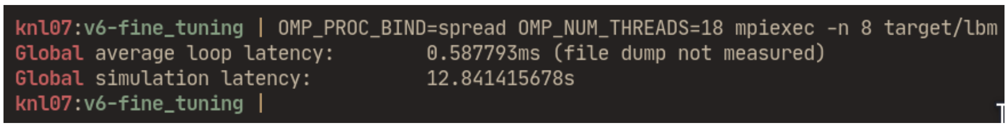
Note: *The best performance we achieved on the laptop was using MPC with 4 UNIX processes and 4 OMP threads, as shown below on 1600 iterations, dimensions: 800x160.*



```
~/u/m/s/t/p/v6-fine_tuning | OMP_NUM_THREADS=4 mpcrun -p=4 -n=4 target/lbm
=================== CONFIG ===================
iterations           = 1600
width                = 800
height               = 160
obstacle x coord     = 161.000000
obstacle y coord     = 83.000000
obstacle radius      = 17.000000
reynolds number      = 100.000000
inflow max velocity  = 0.100000
output filename      = results.raw
write interval       = 50
------------ Derived parameters -------------
kinetic viscosity    = 0.034000
relax parameter      = 1.661130
==============================================
Rank 0: L = -1, R =  1, T = -1, B = -1 | CORNER: -1, -1, -1, -1 | POS:   0   0 (W = 202, H = 162)
Rank 1: L =  0, R =  2, T = -1, B = -1 | CORNER: -1, -1, -1, -1 | POS: 200   0 (W = 202, H = 162)
Rank 2: L =  1, R =  3, T = -1, B = -1 | CORNER: -1, -1, -1, -1 | POS: 400   0 (W = 202, H = 162)
Rank 3: L =  2, R = -1, T = -1, B = -1 | CORNER: -1, -1, -1, -1 | POS: 600   0 (W = 202, H = 162)
Rank 0: local average loop latency: 1.734672ms
Rank 0: local simulation latency:   2.775611286s

Global average loop latency:        1.737066ms
Global simulation latency:          2.779484738s
Rank 1: local average loop latency: 1.737910ms
Rank 1: local simulation latency:   2.780804305s

Rank 2: local average loop latency: 1.737824ms
Rank 2: local simulation latency:   2.780730583s

Rank 3: local average loop latency: 1.737858ms
Rank 3: local simulation latency:   2.780792780s

~/u/m/s/t/p/v6-fine_tuning |
```

Figure 4.10: Best performance on the laptop CPU using MPC

A final run on the KNM of the OB1 cluster gave us the following performance statistics:



```
knl07:v6-fine_tuning | OMP_PROC_BIND=spread OMP_NUM_THREADS=18 mpiexec -n 8 target/lbm
Global average loop latency:        0.587793ms (file dump not measured)
Global simulation latency:          12.841415678s
knl07:v6-fine_tuning |
```

Figure 4.11: Best performance on the Intel KNM of the OB1 cluster