Report of TOP Project

Bouton Nicolas

April 2021

Contents

1 Debugging

1.1 Multiple definition

For beginnig the code have a link problem at the compile time because we define 3 variables in **lbm_phys.h** and we redefine them in **lbm_phys.h**. Then we have a multiple definition.

```
[148] [sholde@ground simu_simple_LBM] (master) (6m32s) $ make
mpicc -Wall -g -c -o main.o main.c
mpicc -Wall -g -c -o lbm_phys.o lbm_phys.c
mpicc -Wall -g -c -o lbm_init.o lbm_init.c
mpicc -Wall -g -c -o lbm_struct.o lbm_struct.c
mpicc -Wall -g -c -o lbm_comm.o lbm_comm.c
mpicc -Wall -g -c -o lbm_config.o lbm_config.c
mpicc -Wall -g -o lbm main.o lbm_phys.o lbm_init.o lbm_struct.o lbm_comm.o lbm_con
/usr/bin/ld : lbm_phys.o:/home/sholde/dev/master/M1/S2/hm/top_project/simu_simple_
/usr/bin/ld : lbm_phys.o:/home/sholde/dev/master/M1/S2/hm/top_project/simu_simple_
/usr/bin/ld : lbm_phys.o:/home/sholde/dev/master/M1/S2/hm/top_project/simu_simple_
/usr/bin/ld : lbm_init.o:/home/sholde/dev/master/M1/S2/hm/top_project/simu_simple_
/usr/bin/ld : lbm_init.o:/home/sholde/dev/master/M1/S2/hm/top_project/simu_simple_
/usr/bin/ld : lbm_init.o:/home/sholde/dev/master/M1/S2/hm/top_project/simu_simple_
collect2: erreur: ld a retourné le statut de sortie 1
make: *** [Makefile:24 : lbm] Erreur 1
```

To fix the problem we just need to mark these variable extern in header file to indicate they are define in another file.

```
/********************* CONSTS *********************/
extern const int opposite_of[DIRECTIONS];
extern const double equil_weight[DIRECTIONS];
extern const Vector direction_matrix[DIRECTIONS];
```

1.2 Program doesn't give back and abort sometimes

When we execute the code now, with the command `mpirun -np 512 --oversubscribe ./lbm`, thr program doesn't give back.
I decide to run **gdb** and try to identify where we block. And **gdb** say me that I have a **segfault** in the function **setup_init_state_global_poiseuille_profile** in file **lbm_init.c** at line 85.

This line contain the result of a call fonction. But as the **segfault** is in this function, it is not in the call funtion. Therefore it is the affectaion the error and either the array are not allocate or we go out of the limit of our array. When I reading the code I saw in the init function that malloc call was commented. I decomment it and the error is corrected.

1.3 Program doesn't give back and abort sometimes 2

I re-run **gdb** and it says me that I have a **segfault** when I call the **libc**. It is true because an allocation via **malloc** failed (printed in stdout).

```
malloc: Permission denied
```

The problem was that I don't realize we set address of the last problem to **NULL** after we allocate it. I deleted the line.

```
mesh->cell = NULL;
```

1.4 Run the program

Then I run the program with a reduce number of iteration (16) and without **MPI**. The program finish without crashing.
I decide to run it with **MPI** but it doesn't give back again and doesn't print on standart output anything. I decide to inspect the code and find an error of communication maybe.
The problem is that on the **main** function, we autorize only the **RANK_MASTER** process to execute the function **close_file**. But in this function we have a **MPI** barrier with **MPI_COMM_WORLD** that wait all process.
And the problem is that others threads haven't a **MPI** barrier. Thus the **RANK_MASTER** wait infinitely.
To patch this, I remove the line of **MPI** barrier in **close_file** function.

1.5 Valgrind

I decide to run **valgrind** now, but it seems to be normal. We haven't **possbilby lost** or **suppressed** bytes. But we have **833 bytes definitely lost**.

```
==29527== HEAP SUMMARY:
==29527==     in use at exit: 83,271 bytes in 50 blocks
==29527==   total heap usage: 21,807 allocs, 21,757 frees, 33,251,263 bytes alloca
==29527==
==29527== LEAK SUMMARY:
==29527==    definitely lost: 833 bytes in 12 blocks
==29527==    indirectly lost: 599 bytes in 20 blocks
==29527==      possibly lost: 0 bytes in 0 blocks
==29527==    still reachable: 81,839 bytes in 18 blocks
==29527==         suppressed: 0 bytes in 0 blocks
```

Therfore I will check the code to see where memory is not release. But everything seems ok, I zap this part.

2 Original Code

2.1 Result

Fist of all, I want to execute the code and generate a **gif** to see if it working. This is the last frame (frame 4 of 4):

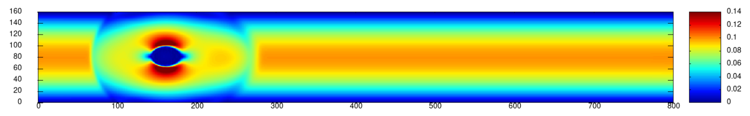

Figure 1: Last frame of Origin Code

It is not like the image in the subject. So I will investigate the code and see what is wrong. But for the moment I will zap this part.

2.2 GIF, Image and script

Also I have only **4** frame when I run your script. And an error occur:

```
gnuplot> splot "< ./display --gnuplot result.raw 4" u 1:2:4
                                                           ^
         line 0: All points x value undefined
```

But the srcipt generate the **gif**

2.3 Checksum Script

I make a script which call **display** binary with the 2 file and the number of frame in parameter to compare their checksum.

2.4 Communication

2.4.1 Explaination

Fistr of all I reduce the number of iteration from **1600** to **16** and the number of processus **MPI** from **512** to **4**. Because it was too long to test.
I am workeing on a personnal project of **MPI profiler** and for a number of **4** process and the following configuration:

```
=================== CONFIG ===================
iterations           = 10
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
==============================================
```

I obtain that:

```
================================================================================
================================ MPI PROFILER ==================================
================================================================================
GLOBAL SUMMARY:
        388317 message send
        388317 message recv
        436 barrier passed

LOCAL SUMMARY (Process 0):
        64719 message send [ 1 ]
        64722 message recv [ 1 2 3 ]
        109 barrier passed

LOCAL SUMMARY (Process 1):
        129439 message send [ 0 2 ]
        129438 message recv [ 0 2 ]
        109 barrier passed

LOCAL SUMMARY (Process 2):
        129439 message send [ 0 1 3 ]
        129438 message recv [ 1 3 ]
        109 barrier passed

LOCAL SUMMARY (Process 3):
        64720 message send [ 0 2 ]
        64719 message recv [ 2 ]
        109 barrier passed

ERROR SUMMARY:
        No error
```

For the moment, it is just an interposition library that interpose **MPI_Send**, **MPI_Recv** and **MPI_Barrier**.
We can see that we have a lot of communication, and a lot of barrier. We can see that also in the code bacause we loop on **MPI_Send**, **MPI_Recv** and **MPI_Barrier** call.
For the communication, if we consider that all process send their info to master process (process 0) for print information. We can see that the communication for **4** MPI process are the following:

- each process communicate with his neighbors

The code of my **MPI Profiler** will be added to my github this week. For the moment we need to preload manually the library with **LD_PRELOAD**.

https://github.com/Sholde/mprof
Here with **8** processus to confirm the communication scheme:

```
===============================================================================
================================ MPI PROFILER =================================
===============================================================================
GLOBAL SUMMARY:
        906073 message send
        906073 message recv
        872 barrier passed

LOCAL SUMMARY (Process 0):
        64719 message send [ 1 ]
        64726 message recv [ 1 2 3 4 5 6 7 ]
        109 barrier passed

LOCAL SUMMARY (Process 1):
        129439 message send [ 0 2 ]
        129438 message recv [ 0 2 ]
        109 barrier passed

LOCAL SUMMARY (Process 2):
        129439 message send [ 0 1 3 ]
        129438 message recv [ 1 3 ]
        109 barrier passed

LOCAL SUMMARY (Process 3):
        129439 message send [ 0 2 4 ]
        129438 message recv [ 2 4 ]
        109 barrier passed

LOCAL SUMMARY (Process 4):
        129439 message send [ 0 3 5 ]
        129438 message recv [ 3 5 ]
        109 barrier passed

LOCAL SUMMARY (Process 5):
        129439 message send [ 0 4 6 ]
        129438 message recv [ 4 6 ]
        109 barrier passed

LOCAL SUMMARY (Process 6):
        129439 message send [ 0 5 7 ]
        129438 message recv [ 5 7 ]
        109 barrier passed

LOCAL SUMMARY (Process 7):
        64720 message send [ 0 6 ]
        64719 message recv [ 6 ]
        109 barrier passed

ERROR SUMMARY:
        No error
```

So it is not the scheme describe in the subject were we have a cube, and we can exchange in diagonally, and vertically. Here we exchange only vertically.
But on the code, we exchange horizontally and diagonally, so I don't know why we don't exchange. Maybe the initialisation was not correct.

2.4.2 Scheme

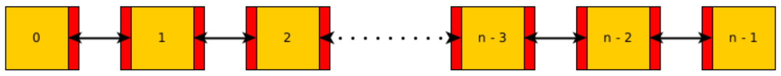

Figure 2: Scheme of Original Code

2.4.3 Problem

First of all, this is the bad scheme (not the scheme of Original Code), with this type of communication, we cut the array vertically, and as we see in course, the cache line of processor is not optimized. Because cut the adjoin of array.

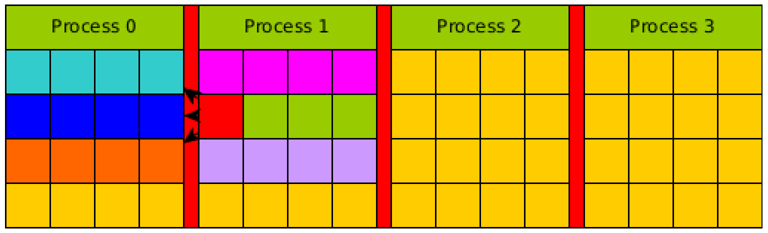

Figure 3: Problem of Original Code

There are 2 problem with this scheme, first we can see that we need 3 communication to get values and they cab be in 3 different cache line. Normally in the code we get one by one all the element of column, so we make **height** communication for one phase.
But we can optimize that with this scheme, we have normally all tha array in the cache line of the processor and it is a reduce number of cacheline. Also, me need only 2 communication because array is contigous.

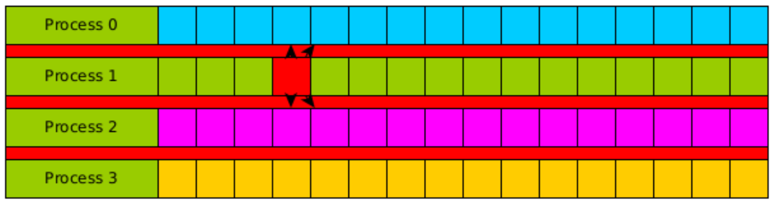

Figure 4: Solution of Original Code

And I think that we can reduce enormous the number of communication with this scheme. As you will see below with my tool, we have a lot of communication.
The Original code uses this scheme but uses simple Send of **1** element. Insted of Send an array of contigous element.

2.5 Scalability of Original Code

I added few lines to compute the times of the code with **MPI_Wtime**. And I print only the time of rank master. I choose to compute the time of the loop in **main.c** because is here we work.

There are **2 ways** to determine the scability:

- **strong scaling:** when we increase the number of processus by **2**, we reduce the time by **2**
- **weak scaling:** when we increase the quantity of data by **2** and the number of processus by **2**, the time is the same

2.5.1 Strong scaling

```
# 1 process
Elapsed time in second(s): 9.617342

# 2 processes
Elapsed time in second(s): 9.463310

# 4 processes
Elapsed time in second(s): 9.271398
```

We can see when we increase the number of processus by **2**, we don't reduce the time by **2**. So we don't scale.

2.5.2 Weak scaling

To evaluate the **weak scaling** I decide to begin with default config:

```
iterations          = 10
width               = 800
height              = 160
#obstacle_r          =
#obstacle_x          =
#obstacle_y          =
reynolds            = 100
inflow_max_velocity  = 0.100000
inflow_max_velocity  = 0.100000
output_filename      = resultat.raw
write_interval       = 50
```

And I will multiply the height by **2** when I increase the number of processus by **2**, because if I do that I multiply the quantity of data by **2**, if I

multiply also the width by **2**, then I multiply the quantity of data by **4** and it is not that we want.
So for **1 process** height equal **160**, for **2 processes** height equal **320** and for **4 processes** height equal **740**.

```
# 1 process
Elapsed time in second(s): 9.764508

# 2 processes
Elapsed time in second(s): 9.662370

# 4 processes
Elapsed time in second(s): 9.840443
```

We scale because when we increase the number of processus by 2 and the quantity of data by 2, we take the same time.

3 Optimization

3.1 Introduction

I run the original code with the config file **original_config.txt** (with 160 iteration to have **4** frame and be able to compare the checksum when I will optimize the code). The result is on **original_code.raw**.
And I run with **4** process **MPI** (because I have only **4** process and **512** is not accept by **MPI** when I **oversubscribe**).
The time for original code is: **165.417102 seconds**.

3.2 Barrier MPI

3.2.1 Identification

Like you can see above in the report, my **MPI profile** detect a lot of **barrier**.

3.2.2 Justification

But in another course, we said us that **MPI_Barrier** are useless because we are on a distributed memory (not shared). But only for print we can keep them, for keep an readable output.

I don't need other justification to delete all of them (almost).

3.2.3 Evaluation

I run the code with the same config but the time stay the same, it is **165** seconds (~2 min 45).
I check the checksum with my script and it is the same so it don't change the behaviour.

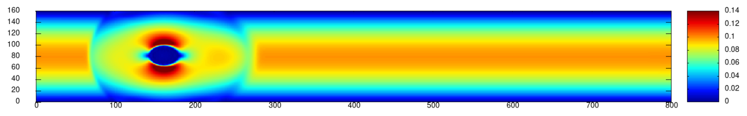

Figure 5: Without MPI Barrier

I decide to not evaluate the scability because it is not barrier that cause the **slow-down**. (normally it is the main problem but not here)
Therefore this optimization doesn't affect the code for the moment.

3.3 FLUSH

3.3.1 Identification

When I remove **MPI_Barrier** in **lbm_comm.c**, I see that we have **flush** function, which normally refer to **output**. But I didn't know this function, so I run my alias **search** (it is just an alias of `grep -re word *`):

```
[148] [sholde@ground simu_simple_LBM] (master)* (13.3s) $ search FLUSH
grep: display: binary file matches
grep: lbm: binary file matches
lbm_comm.c:  FLUSH_INOUT();
lbm_config.h:#define __FLUSH_INOUT__ concat(s,l,e,e,p)(1)
lbm_config.h:#define FLUSH_INOUT() __FLUSH_INOUT__
```

We can see that is in reality a macro which define a sleep ^^. (you are not kind ^^)

3.3.2 Justification

I think I don't need to justify that a sleep is very unless on a code. So let's remove it. (I just remove the call).

3.3.3 Evaluation

The program take only **4.372996 seconds** now instead of **165**. The checksum is the same.
Therefore it was **the main problem of the code**.

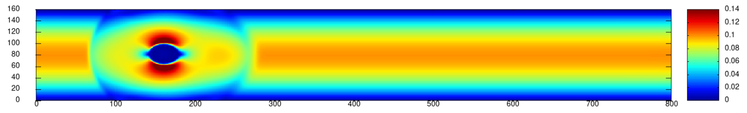

Figure 6: Without Sleep

We can now re-evaluate the scalability. I put bach the number of iteration at **16000** now.

1. Strong Scaling

   ```
   # 1 process
   Elapsed time in second(s): 94.997312

   # 2 processes
   Elapsed time in second(s): 58.412028

   # 4 processes
   Elapsed time in second(s): 48.756189
   ```

   We don't reduce the time by **2** each step, but we have a speedup, so we can say that we scale half.

   Maybe we need to remove **fprintf** in the next step because **io** is too long compared to the compute.

2. Weak Scaling

   Like above, we multiply the height by **2** each time me increase the number of processus by **2**.

```
# 1 process
Elapsed time in second(s): 94.997312

# 2 processes
Elapsed time in second(s): 107.737355

# 4 processes
Elapsed time in second(s): 134.550160
```

Here we can see that we don't have the same time. It is the reflect of **strong scaling**.

3.3.4 Conclusion

This optimization is good but we can still optimize the code to have a good scaling.
I keep 160 iteration because original code was run with that and we need the same number of iteration to compare the checksum.

3.4 Useless Communication

3.4.1 Identification

With my **profiler tool**, I ensure that diagonal and horizontal communication don't work and don't affect the code because we have the same number of communication.
I also relevate that we do **2** horizaontal communication of right to left phase.

1. With diagonal and horizontal

```
========================================================================
============================== MPI PROFILER ============================
========================================================================
==mprof== GLOBAL SUMMARY:
==mprof==          message sent: 6,860,226 - waiting 4.824620 sec in total
==mprof==          message recv: 6,860,226 - waiting 5.032828 sec in total
==mprof==     barrier(s) passed: 1 - waiting 0.021257 sec in total
```

```
==mprof==
==mprof== LOCAL SUMMARY (Process 0):
==mprof==          message sent: 1,143,369 - waiting 1.270680 sec (max: 0.017
==mprof==          message recv: 1,143,381 - waiting 1.163514 sec (max: 0.028
==mprof==     barrier(s) passed: 1 - waiting 0.000002 sec (max: 0.000002 sec)
==mprof==       list(s) sent to: 1
==mprof==     list(s) recv from: 1 2 3
==mprof==
==mprof== LOCAL SUMMARY (Process 1):
==mprof==          message sent: 2,286,742 - waiting 1.365050 sec (max: 0.029
==mprof==          message recv: 2,286,738 - waiting 0.980989 sec (max: 0.008
==mprof==     barrier(s) passed: 1 - waiting 0.010554 sec (max: 0.010554 sec)
==mprof==       list(s) sent to: 0 2
==mprof==     list(s) recv from: 0 2
==mprof==
==mprof== LOCAL SUMMARY (Process 2):
==mprof==          message sent: 2,286,742 - waiting 1.431860 sec (max: 0.023
==mprof==          message recv: 2,286,738 - waiting 1.097262 sec (max: 0.018
==mprof==     barrier(s) passed: 1 - waiting 0.007246 sec (max: 0.007246 sec)
==mprof==       list(s) sent to: 0 1 3
==mprof==     list(s) recv from: 1 3
==mprof==
==mprof== LOCAL SUMMARY (Process 3):
==mprof==          message sent: 1,143,373 - waiting 0.757031 sec (max: 0.021
==mprof==          message recv: 1,143,369 - waiting 1.791063 sec (max: 0.041
==mprof==     barrier(s) passed: 1 - waiting 0.003454 sec (max: 0.003454 sec)
==mprof==       list(s) sent to: 0 2
==mprof==     list(s) recv from: 2
==mprof==
==mprof== ERROR SUMMARY:
==mprof==          No errors
```

2. Without diagonal and horizontal

```
==================================================================================
================================ MPI PROFILER ====================================
==================================================================================
==mprof== GLOBAL SUMMARY:
```

```
==mprof==          message sent: 6,860,226 - waiting 5.298880 sec in total
==mprof==          message recv: 6,860,226 - waiting 4.093540 sec in total
==mprof==     barrier(s) passed: 1 - waiting 0.022888 sec in total
==mprof==
==mprof== LOCAL SUMMARY (Process 0):
==mprof==          message sent: 1,143,369 - waiting 1.292504 sec (max: 0.014
==mprof==          message recv: 1,143,381 - waiting 1.126989 sec (max: 0.023
==mprof==     barrier(s) passed: 1 - waiting 0.000003 sec (max: 0.000003 sec)
==mprof==       list(s) sent to: 1
==mprof==     list(s) recv from: 1 2 3
==mprof==
==mprof== LOCAL SUMMARY (Process 1):
==mprof==          message sent: 2,286,742 - waiting 1.210447 sec (max: 0.008
==mprof==          message recv: 2,286,738 - waiting 1.211880 sec (max: 0.014
==mprof==     barrier(s) passed: 1 - waiting 0.011559 sec (max: 0.011559 sec)
==mprof==       list(s) sent to: 0 2
==mprof==     list(s) recv from: 0 2
==mprof==
==mprof== LOCAL SUMMARY (Process 2):
==mprof==          message sent: 2,286,742 - waiting 1.690453 sec (max: 0.017
==mprof==          message recv: 2,286,738 - waiting 0.758160 sec (max: 0.004
==mprof==     barrier(s) passed: 1 - waiting 0.007670 sec (max: 0.007670 sec)
==mprof==       list(s) sent to: 0 1 3
==mprof==     list(s) recv from: 1 3
==mprof==
==mprof== LOCAL SUMMARY (Process 3):
==mprof==          message sent: 1,143,373 - waiting 1.105475 sec (max: 0.017
==mprof==          message recv: 1,143,369 - waiting 0.996512 sec (max: 0.019
==mprof==     barrier(s) passed: 1 - waiting 0.003657 sec (max: 0.003657 sec)
==mprof==       list(s) sent to: 0 2
==mprof==     list(s) recv from: 2
==mprof==
==mprof== ERROR SUMMARY:
==mprof==          No errors
```

3.4.2 Justification

I don't know if it is normal but they don't impact code. So I comment diagonal and horizontal communication because it is useless for the original code.
For the duplicate communication, I removed the second because it is useless.

3.4.3 Evaluation

I put back the number of iteration to **160** because **1600** was too long even if last optimization.

1. Strong Scaling

```
# 1 process
Elapsed time in second(s): 9.484721

# 2 processes
Elapsed time in second(s): 5.716905

# 4 processes
Elapsed time in second(s): 4.816284
```

2. Weak Scaling

```
# 1 process
Elapsed time in second(s): 9.484721

# 2 processes
Elapsed time in second(s): 10.496072

# 4 processes
Elapsed time in second(s): 13.787685
```

3.4.4 Conclusion

It is relatively the same result that last optimization. So this optimization has not effect. I am not surprised because we do not the communication (I see that with my tool).

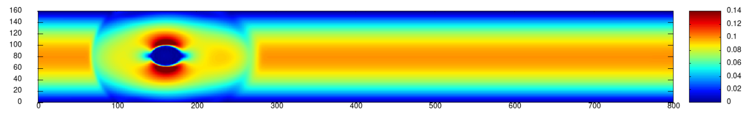

Figure 7: Without Diagonal and Horizontal Communication

3.5 Factorize Communication

3.5.1 Identification

We make vertical communication but like I said above we make it one by one element. It is not optimize because we can send and receive an array instead of an element.
But for the moment I just factorize with DIMENSION step to get each dimension in the same time because I don't know how give all the line directly. As DIMENSION is a factor of the index on Mesh_get_cell I can do that.
Also, the loop for receive contained yet an array of DIMENSION but I d'ont know why it was here because we loop on DIMENSION with **k** variable. When I factorize I test with an array of DIMENSION and an array of DIMENSION * DIMENSION to receive but the checksum was the same. So I just keep an array of DIMENSION to receive.
It is on the function which get cell of ghosts.

3.5.2 Justification

I justify above with a scheme. We can sand an array that reduce the number of communication.

3.5.3 Evaluation

Fist of all, look what my **profiler** give us. For **160** iteration we had **6,860,226** send and **6,860,226** receive. So **6,860,226** communication because send and receive are in pair.
Now when we send and receive an array we have:

```
================================================================================
================================= MPI PROFILER =================================
================================================================================
==mprof== GLOBAL SUMMARY:
==mprof==          message sent: 762,258 - waiting 1.479853 sec in total
==mprof==          message recv: 762,258 - waiting 1.913188 sec in total
==mprof==     barrier(s) passed: 1 - waiting 0.022368 sec in total
==mprof==
==mprof== LOCAL SUMMARY (Process 0):
==mprof==          message sent: 127,041 - waiting 0.833233 sec (max: 0.022397 sec
==mprof==          message recv: 127,053 - waiting 0.208592 sec (max: 0.021491 sec
==mprof==     barrier(s) passed: 1 - waiting 0.000007 sec (max: 0.000007 sec)
==mprof==       list(s) sent to: 1
==mprof==     list(s) recv from: 1 2 3
==mprof==
==mprof== LOCAL SUMMARY (Process 1):
==mprof==          message sent: 254,086 - waiting 0.261738 sec (max: 0.021010 sec
==mprof==          message recv: 254,082 - waiting 0.171416 sec (max: 0.012849 sec
==mprof==     barrier(s) passed: 1 - waiting 0.011410 sec (max: 0.011410 sec)
==mprof==       list(s) sent to: 0 2
==mprof==     list(s) recv from: 0 2
==mprof==
==mprof== LOCAL SUMMARY (Process 2):
==mprof==          message sent: 254,086 - waiting 0.259406 sec (max: 0.012548 sec
==mprof==          message recv: 254,082 - waiting 0.660349 sec (max: 0.018675 sec
==mprof==     barrier(s) passed: 1 - waiting 0.007384 sec (max: 0.007384 sec)
==mprof==       list(s) sent to: 0 1 3
==mprof==     list(s) recv from: 1 3
==mprof==
==mprof== LOCAL SUMMARY (Process 3):
==mprof==          message sent: 127,045 - waiting 0.125476 sec (max: 0.024310 sec
==mprof==          message recv: 127,041 - waiting 0.872831 sec (max: 0.024739 sec
==mprof==     barrier(s) passed: 1 - waiting 0.003568 sec (max: 0.003568 sec)
==mprof==       list(s) sent to: 0 2
==mprof==     list(s) recv from: 2
==mprof==
==mprof== ERROR SUMMARY:
==mprof==          No errors
```

A few minute of compute and we can see that **6,860,226** divide by **762,258** equal **9** (not exactly but I don't know why). It was predictable because we factorize the number of communication by **9** (because DIMENSION = 9). Of course I test the cheksum, ant it is the same.

1. Strong Scaling

   Still with **160** iteration.

   ```
   # 1 process
   Elapsed time in second(s): 9.082287

   # 2 processes
   Elapsed time in second(s): 4.815402

   # 4 processes
   Elapsed time in second(s): 3.135502
   ```

   It is good, we almost go 2 times faster and we gain times since the last optimization.

2. Weak Scaling

   ```
   # 1 process
   Elapsed time in second(s): 9.082287

   # 2 processes
   Elapsed time in second(s): 9.703516

   # 4 processes
   Elapsed time in second(s): 12.141861
   ```

   It is not very good, mostly when we increase the number of processus from 2 to 4 (with the quantity of data also).

   So for the weak scaling, we don't scale.

3.5.4 Conclusion

This optimization work, mostly for **strong scaling** and I think it can we increase if we factorize even more.

3.6 gprof

3.6.1 Identification

It is time to use **gprof**. I add **-pg** in Makefile and after running it, gprof give me:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 19.94      0.49     0.49                             propagation
 18.31      0.94     0.45                             compute_equilibrium_profile
 16.68      1.35     0.41                             get_vect_norme_2
 15.46      1.73     0.38                             compute_cell_collision
 13.83      2.07     0.34                             get_cell_velocity
  6.92      2.24     0.17                             get_cell_density
  5.90      2.39     0.15                             Mesh_get_cell
  1.22      2.42     0.03                             special_cells
  1.02      2.44     0.03                             lbm_cell_type_t_get_cell
  0.81      2.46     0.02                             main
```

We can see that we have **5** functions that take all of out appliction time. Therefore if we want to optimize the code, we must optimize this function, maybe by adding parallelization with OpenMP but we must to be able to parallelize these function in totality or almost, esle we will take the **Ahmdal's law**.
But I don't have time to do that because for a good hybrid implementation of MPI/OpenMP we neeed to change all the structure. If we keep this structure we will adding **omp pragma** on each function but we will **create** and **destroy** thread each time for each call for each loop. And it is not worse I think.
The good way is to re-decompose each array of each **MPI process** on all thread.

3.7 Compilation Optimization

3.7.1 Identification

Everybody know that compiler allow us to optimize ower code with flags.

3.7.2 Justification

I decide to use these flags:

```
OFLAGS=-Ofast -march=native -funroll-loops -finline-functions
```

- `-Ofast` for all basic optimization that bring compiler
- `-march=native` to enable vectorization depending on our system (SSE, AVX and AVX512 if it is supported)
- `-funroll-loops` for unroll loops and use vectorization
- `-finline-functions` bring by default with `-Ofast` but I really want that compiler inline function (because function call have a big impact)

3.7.3 Evaluation

Always with **160** iteration and checksum passed.

1. Strong Scaling

   ```
   # 1 process
   Elapsed time in second(s): 1.628340

   # 2 processes
   Elapsed time in second(s): 1.106629

   # 4 processes
   Elapsed time in second(s): 0.945746
   ```

   We are faster that the last optimization. But we don't very scale.

2. Weak Scaling

```
# 1 process
Elapsed time in second(s): 1.628340

# 2 processes
Elapsed time in second(s): 2.141724

# 4 processes
Elapsed time in second(s): 3.814287
```

We litterraly don't scale.

3.7.4 Conclusion

It is a big optimization. Don't forget compiler optimization flags.

3.8 Remove printf in loop

3.8.1 Identification

We have a printf in main loop that print **each step**.

3.8.2 Justification

We can remove it because it just jor debugging but we already debug. At the end of loop we have a write in the file but we keep it because we want the result.

3.8.3 Evaluation

Always with **160** iteration.

1. Strong Scaling

```
# 1 process
Elapsed time in second(s): 1.749387

# 2 processes
Elapsed time in second(s): 1.020297
```

```
# 4 processes
Elapsed time in second(s): 0.720448
```

We reduce a little bit the time.

2. Weak Scaling

```
# 1 process
Elapsed time in second(s): 1.749387

# 2 processes
Elapsed time in second(s): 2.016925

# 4 processes
Elapsed time in second(s): 2.717607
```

We scale better than the last optimization but it is not already very good. We take 1 more second when we increase the number of processus by **4** and the quantity of data by **4**.

3.9 taskset when running

3.9.1 Identification

No identification.

3.9.2 Justification

Taskset a processus can be usefull to avoid that OS change our processus of core. Of course, if a processus change of core in running we lose performance.
REMINDER: `taskset -c 0,1,2,3 mpirun -np 4 ./lbm`

3.9.3 Evaluation

Always with **160** iteration.

1. Strong Scaling

```
# 1 process
Elapsed time in second(s): 1.759057

# 2 processes
Elapsed time in second(s): 1.009991

# 4 processes
Elapsed time in second(s): 0.705297
```

We reduce a very little bit the time.

2. Weak Scaling

```
# 1 process
Elapsed time in second(s): 1.759057

# 2 processes
Elapsed time in second(s): 2.007785

# 4 processes
Elapsed time in second(s): 2.724929
```

As last optimizatin.

3.9.4 Conclusion

Taskset our processus don't really affect the time. I think if we increase the number of iteration, we will see a little difference.

4 Observation and Important Rules

4.1 First touch

First touch is important, because it is at this moment we **allocate** the memory (not with malloc that prepare the first touch). We do that in **lbm_init.c** when we affect value of our array.
But here in the code, we have the fisrt touch. So it is good.

4.2 Others

There are several rules to respect for developping a parallel application to increase is scalability:

- execute the code a first time to see what is going on
- debug the program with different tools (gdb, valgring, ...)
- read the code to find if calls are good (i.e. FLUSH_INOUT)
- evaluate the scalability of original code and compare it each stop of optimization
- parallelize the code with thread (pthread, openMP) or processus (MPI) or both (hybrid MPI/openMP)
- reduce the number of communication (by sending array for MPI)
- try to respect memory locality, that is the data msut be in the core that compute with
- use cahe line of processor, with boucle index (in C we iterate by line) and with MPI call (using array of contigous memory)
- remove unnecessary output (printf to debug)
- adding compilation optimiazation flags
- taskset out processus or threads on available core
- check all step of optimization if we have the same behaviour than original code