CSC4005 Report 3

Nbody Simulation: pthread, MPI, OpenMP, CUDA & MPI+OpenMP Implementations

Shu Xin
118020362@link.cuhk.edu.cn
Instructor: Chung Yehching

School of Data Science,
Computer Engineering & Engineering

November 17, 2021

Abstract

In this project, I implement parallelized Nbody simulation versions using four parallel frameworks, which are pthread, MPI, OpenMP and CUDA.

After analyzing which parts of the sequential program are suitable for parallelization, I find that among these four APIs, pthread and OpenMP and similar to each other, while MPI and CUDA are similar to each other.

However, OpenMP is easier to use and more flexible than pthread, while MPI is easier to use than CUDA since CUDA has multiple memory hierarchies which needs to be handled by programmers.

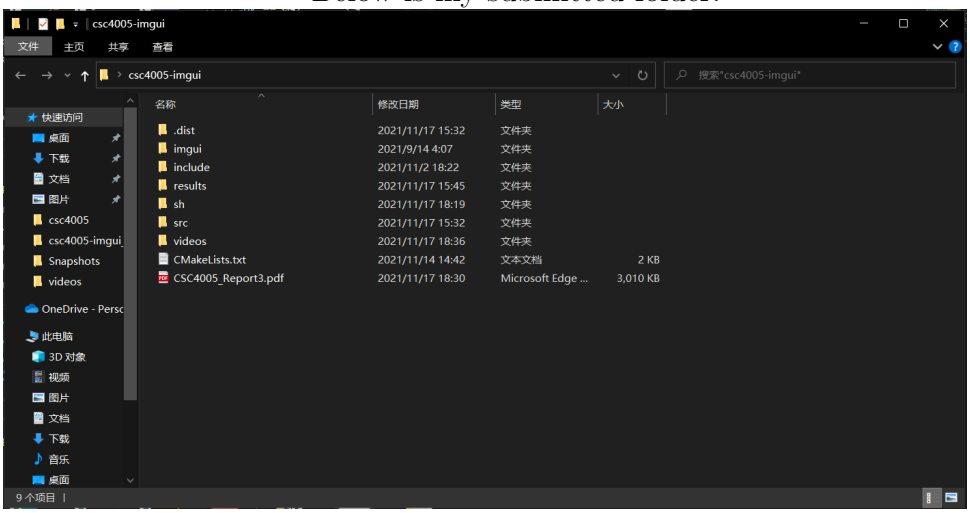
Since the parallelized part for MPI and OpenMP is different, we can combine these two and use them to parallelize different parts of the program. Therefore, I write an extra MPI+OpenMP implementation of this problem.

In this report, same as the previous two, I will explain my program, and then compare and analyze the parallized versions under different configurations.

Contents

1 Content of my Submitted folder

Below is my submitted folder.



- src
  It contains all .cpp files. In case you want to run any of them, make sure to change the file name to main.cpp or main.cu.
- include
  It contains all header files.
- results
  It contains all the running results of the experiments.
- sh
  It contains all the .sh files that you can use to run my program. They will be introduced in *How to run my program?* section.
- videos
  It contains videos of my programs for different versions.
- CMakeLists.txt
- Report

2 Introduction

2.1 What is Nbody Simulation?

In physics and astronomy, an N-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity. Direct N-body simulations are used to study the dynamical evolution of star clusters.

2.2 How can Nbody Simulation problem be parallelized?

What Nbody Simulation problrm differs from Odd-Even-Sort and Mandelbrot Set is that it has data dependency.

Since each of the N-bodies needs to know the gravitational effect of the other bodies, it is important for every processor to know the entire dataset.

There are some algorithms such as *Barnes Hut* and *Fast Multipole Method*, which make use of a tree data structure where nodes only interact with their nearest neighbors at each level of the tree, and splits the tree between the set of processes at a sufficient depth, then having them cooperate only at the highest levels.

However, in my project, due to the data dependency problem and different characteristics of parallelization framework, I choose to parallelize part of the sequential version rather than parallelized the whole version.

Since it is required that each processorthread has the right to change the whole data set, **Data Race** problem is easy to occur. This also needs to be solved.

2.3 Sequential and Parallel Complexity Analysis

2.3.1 Sequential

The overall gravitational Nbody Simulation can be described by the following algorithm:

```
for(t=0;t<tmax ;t++){
/* for each time period */
    for(i=0;i<N;i++){
    /* for each body */
        F=Force routine(i) ;
        /*compute force on ith body */
        v[i]new = v[i]+F * dt / m; /* compute new velocity and
        x[i]new=x[i]+v[i]new *dt;/*new position (leap-frog)*/
    }
}
for(i=0;i<nmax ;i++){
x[i]=x[i]new ;
v[1]=v[i]new ;
}
```

Listing 1: Sequential_pseudocode

The computation complexity for sequential case is O(N^2) since for each body it needs to calculate all other bodies' gravity on it.

2.3.2 Parallel

The computation complexity for Parallel case is also O(N^2) since it has data dependency. For each body, it needs to compute other N-1 bodies' gravity on it. It requires that each processor / thread to calculate every other body.

3 How to run my program?

3.1 CMakeLists.txt

Since my programs use four parallel frameworks, I add the corresponding packages to *CMakeLists.txt*, shown like below, also included in my submitted folder, so that you can run all programs using the same *CMakeLists.txt*.

```
cmake_minimum_required(VERSION 3.2)
project(csc4005_imgui CUDA CXX)
set(CMAKE_CUDA_STANDARD 14)
set(CMAKE_CXX_STANDARD 17)

find_package(SDL2 REQUIRED)
find_package(Freetype REQUIRED)
find_package(MPI REQUIRED)
find_package(Threads REQUIRED)
find_package(OpenMP REQUIRED)
set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
set (CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${
    OpenMP_EXE_LINKER_FLAGS}")
set(CMAKE_CXX_STANDARD 20)
# source scl_source enable devtoolset-10
# CC=gcc CXX=g++ cmake ..
# make -j12
# srun ./csc4005_imgui

set(CMAKE_CUDA_FLAGS "${CMAKE_CUDA_FLAGS} -ccbin=/opt/rh/devtoolset-10/
    root/usr/bin/gcc -gencode=arch=compute_75,code=[sm_75,compute_75]")
set(OpenGL_GL_PREFERENCE "GLVND")
find_package(OpenGL REQUIRED)

include_directories(
        include
        imgui
        imgui/backends
        ${SDL2_INCLUDE_DIRS}
        ${FREETYPE_INCLUDE_DIRS}
        ${MPI_CXX_INCLUDE_DIRS})

file(GLOB IMGUI_SRC
        imgui/*.cpp
        imgui/backends/imgui_impl_sdl.cpp
        imgui/backends/imgui_impl_opengl2.cpp
        imgui/misc/freetype/imgui_freetype.cpp
        imgui/misc/cpp/imgui_stdlib.cpp
        )



add_library(core STATIC ${IMGUI_SRC})
file(GLOB CSC4005_PROJECT_SRC src/*.cpp src/*.c src/main.cu)
```

```
add_executable(csc4005_imgui ${CSC4005_PROJECT_SRC})


set_target_properties(
        csc4005_imgui
        PROPERTIES
        CUDA_SEPARABLE_COMPILATION ON)


get_filename_component(FONT_PATH imgui/misc/fonts/DroidSans.ttf
    ABSOLUTE)
target_link_libraries(core PUBLIC
        Freetype::Freetype SDL2::SDL2 OpenGL::GL ${CMAKE_DL_LIBS}
    Threads::Threads ${MPI_CXX_LIBRARIES})
target_link_libraries(csc4005_imgui core)
target_compile_definitions(core PUBLIC -DImDrawIdx=unsigned)
target_compile_definitions(csc4005_imgui PRIVATE -DFONT_PATH=\"${
    FONT_PATH}\")
if (WIN32)
    target_link_options(csc4005_imgui BEFORE PRIVATE -lmingw32 -
    lSDL2main -lSDL2 -mwindows)
endif()
```

Listing 2: CMakeLists.txt

3.2 Sequential

Sequential version of the program is given in the form of template. Shown below is the bash file *run_seq.sh* to run it. You can use the command *sbatch run_seq.sh* or allocate an interactive section by *salloc -Nx -nx -tx*, then *sh run.sh*. Either one is ok.

```
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf build
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug
cmake --build . -j4
./csc4005_imgui
```

Listing 3: run_seq.sh

In case you want to test the sequential version under different body numbers, I provide a .sh file that can generate the bash files that satisfy the requirement.

Basically, the logic is to first compile the program under /home/118020362/csc4005-imgui/ and move it to /pvfmsnt/118020362/csc4005-imgui/.

To use this file, you just type in *sh create_runs_seq.sh*. The running resulst are stored in /pvfsmnt/118020362/csc4005-imgui/seq/.

```
#!/bin/bash
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf seq
mkdir seq
for body_num in 200 400 800
```

```
do
    settings="#!/bin/bash\n#SBATCH --account=csc4005\n#SBATCH --
    partition=debug\n#SBATCH --qos=normal\n#SBATCH --ntasks=32\n#SBATCH
    --nodes=1\n"
    command_part1="xvfb-run -a "
    command_part2="./csc4005_imgui y > /pvfsmnt/118020362/csc4005-imgui
    /seq/"
    cd /home/118020362/csc4005-imgui/src/
    sed -i '16c static int bodies = '${body_num}';' main.cpp

    cd ..
    rm -rf build
    mkdir build
    cd build
    cmake .. -DCMAKE_BUILD_TYPE=Debug
    cmake --build . -j4
    cp -r /home/118020362/csc4005-imgui/build/ /pvfsmnt/118020362/
    csc4005-imgui/
    cd /pvfsmnt/118020362/csc4005-imgui/
    command_part_0="cd /pvfsmnt/118020362/csc4005-imgui/build/"
    sh_name=$(printf run_seq_body%03d.sh ${body_num})
    result_name=$(printf seq_body%03d.txt ${body_num})
    touch "$sh_name"
    echo -e $settings > "$sh_name"
    echo -e $command_part_0 >> "$sh_name"
    command_line="${command_part1}${command_part2}${result_name}"
    echo $command_line >> "$sh_name"
    sbatch "$sh_name"

done
```

Listing 4: create_runs_seq.sh

3.3 pthread

You can run the pthread version using the command *sbatch run_pthread.sh* or allocate an interactive section by *salloc -Nx -nx -tx*, then *sh run_pthread.sh*. Either one is ok.

```
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf build
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug
cmake --build . -j4
./csc4005_imgui

```

Listing 5: run_pthread.sh

In case you want to test the pthread version under different body numbers and thread numbers, I provide a .sh file that can generate the bash files that satisfy the requirement.

Basically, the logic is to first compile the program under /home/118020362/csc4005-imgui/ and move it to /pvfmsnt/118020362/csc4005-imgui/.

To use this file, you just type in *sh create_runs_pthread.sh*. The running results are stored in /pvfsmnt/118020362/csc4005-imgui/pthread/.

```
#!/bin/bash
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf pthread
mkdir pthread
for body_num in 200 400 800
do
    for thread_num in 1 50 100 200
    do
        settings="#!/bin/bash\n#SBATCH --account=csc4005\n#SBATCH --partition=debug\n#SBATCH --qos=normal\n#SBATCH --ntasks=32\n#SBATCH --nodes=1\n"
        command_part1="xvfb-run -a "
        command_part2="./csc4005_imgui y > /pvfsmnt/118020362/csc4005-imgui/pthread/"
        cd /home/118020362/csc4005-imgui/src/
        sed -i '11c static int bodies = '${body_num}';' main.cpp
        sed -i '8c #define NUM_THREADS '${thread_num}'' main.cpp
        cd ..
        rm -rf build
        mkdir build
        cd build
        cmake .. -DCMAKE_BUILD_TYPE=Debug
        cmake --build . -j4
        cp -r /home/118020362/csc4005-imgui/build/ /pvfsmnt/118020362/csc4005-imgui/
        cd /pvfsmnt/118020362/csc4005-imgui/
        command_part_0="cd /pvfsmnt/118020362/csc4005-imgui/build/"
        sh_name=$(printf run_pthread_t%03d_body%03d.sh ${thread_num} ${body_num})
        result_name=$(printf pthread_t%03d_body%03d.txt ${thread_num} ${body_num})
        touch "$sh_name"
        echo -e $settings > "$sh_name"
        echo -e $command_part_0 >> "$sh_name"
        command_line="${command_part1}${command_part2}${result_name}"
        echo $command_line >> "$sh_name"
        sbatch "$sh_name"
    done
done
```

Listing 6: create_runs_pthread.sh

3.4 OpenMP

You can run the OpenMP version using the command *sbatch run_openmp.sh* or allocate an interactive section by *salloc -Nx -nx -tx*, then *sh run_openmp.sh*. Either one is ok.

```
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf build
```

```
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug
cmake --build . -j4
./csc4005_imgui

```

Listing 7: run_openmp.sh

In case you want to test the openmp version under different body numbers and thread numbers, I provide a .sh file that can generate the bash files that satisfy the requirement.

Basically, the logic is to first compile the program under /home/118020362/csc4005-imgui/ and move it to /pvfmsnt/118020362/csc4005-imgui/.

To use this file, you just type in *sh create_runs_openmp.sh*. The running results are stored in /pvfsmnt/118020362/csc4005-imgui/openmp/.

```
#!/bin/bash
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf openmp
mkdir openmp
for body_num in 200 400 800
do
    for thread_num in 1 50 100 200
    do
        settings="#!/bin/bash\n#SBATCH --account=csc4005\n#SBATCH --partition=debug\n#SBATCH --qos=normal\n#SBATCH --ntasks=32\n#SBATCH --nodes=1\n"
        command_part1="xvfb-run -a "
        command_part2="./csc4005_imgui y > /pvfsmnt/118020362/csc4005-imgui/openmp/"
        cd /home/118020362/csc4005-imgui/src/
        sed -i '19c static int bodies = '${body_num}';' main.cpp
        sed -i '8c #define NUM_THREADS '${thread_num}'' main.cpp
        cd ..
        rm -rf build
        mkdir build
        cd build
        cmake .. -DCMAKE_BUILD_TYPE=Debug
        cmake --build . -j4
        cp -r /home/118020362/csc4005-imgui/build/ /pvfsmnt/118020362/csc4005-imgui/
        cd /pvfsmnt/118020362/csc4005-imgui/
        command_part_0="cd /pvfsmnt/118020362/csc4005-imgui/build/"
        sh_name=$(printf run_openmp_t%03d_body%03d.sh ${thread_num} ${body_num})
        result_name=$(printf openmp_t%03d_body%03d.txt ${thread_num} ${body_num})
        touch "$sh_name"
        echo -e $settings > "$sh_name"
        echo -e $command_part_0 >> "$sh_name"
        command_line="${command_part1}${command_part2}${result_name}"
        echo $command_line >> "$sh_name"
        sbatch "$sh_name"
    done
```

```
done
```

Listing 8: create_runs_openmp.sh

3.5 MPI

You can run the MPI version using the command *sbatch run_MPI.sh* or allocate an interactive section by *salloc -Nx -nx -tx*, then *sh run_MPI.sh*. Either one is ok.

```
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf build
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug
cmake --build . -j4
# you can adjust process number here
mpirun -np 4 csc4005_imgui

```

Listing 9: run_MPI.sh

In case you want to test the MPI version under different body numbers and process numbers, I provide a .sh file that can generate the bash files that satisfy the requirement.

Basically, the logic is to first compile the program under /home/118020362/csc4005-imgui/ and move it to /pvfmsnt/118020362/csc4005-imgui/.

To use this file, you just type in *sh create_runs_MPI.sh*. The running results are stored in /pvfsmnt/118020362/csc4005-imgui/MPI/.

```
#!/bin/bash
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf MPI
mkdir MPI
for body_num in 200 400 800
do
    for proc_num in 1 64 96 128
    do
        settings="#!/bin/bash\n#SBATCH --account=csc4005\n#SBATCH --partition=debug\n#SBATCH --qos=normal\n#SBATCH --ntasks=128\n#SBATCH --nodes=4\n"
        command_part1="xvfb-run -a mpirun -n "
        command_part2=" ./csc4005_imgui y > /pvfsmnt/118020362/csc4005-imgui/MPI/"
        cd /home/118020362/csc4005-imgui/src/
        sed -i '22c static int bodies = '${body_num}';' main.cpp

        cd ..
        rm -rf build
        mkdir build
        cd build
        cmake .. -DCMAKE_BUILD_TYPE=Debug
        cmake --build . -j4
        cp -r /home/118020362/csc4005-imgui/build/ /pvfsmnt/118020362/csc4005-imgui/
```

```
22          cd /pvfsmnt/118020362/csc4005-imgui/
23          command_part_0="cd /pvfsmnt/118020362/csc4005-imgui/build/"
24          sh_name=$(printf run_MPI_p%03d_body%03d.sh ${proc_num} ${
    body_num})
25          result_name=$(printf MPI_p%03d_body%03d.txt ${proc_num} ${
    body_num})
26          touch "$sh_name"
27          echo -e $settings > "$sh_name"
28          echo -e $command_part_0 >> "$sh_name"
29          command_line="${command_part1}${proc_num}${command_part2}${
    result_name}"
30          echo $command_line >> "$sh_name"
31          sbatch "$sh_name"
32      done
33  done
```

Listing 10: create_runs_MPI.sh

3.6 OpemMP+MPI

You can run the MPI version using the command *sbatch run_MPIopenmp.sh* or allocate an interactive section by *salloc -Nx -nx -tx*, then *sh run_MPIopenmp.sh*. Either one is ok.

```
1 cd /pvfsmnt/118020362/csc4005-imgui/
2 rm -rf build
3 mkdir build
4 cd build
5 cmake .. -DCMAKE_BUILD_TYPE=Debug
6 cmake --build . -j4
7 # you can adjust process number here
8 mpirun -np 4 csc4005_imgui
9
```

Listing 11: run_MPIopenmp.sh

In case you want to test the MPIomp version under different body numbers, thread numbers and process numbers, I provide a .sh file that can generate the bash files that satisfy the requirement.

Basically, the logic is to first compile the program under /home/118020362/csc4005-imgui/ and move it to /pvfmsnt/118020362/csc4005-imgui/.

To use this file, you just type in *sh create_runs_MPIomp.sh*. The running results are stored in /pvfsmnt/118020362/csc4005-imgui/MPIomp/.

```
1 #!/bin/bash
2 cd /pvfsmnt/118020362/csc4005-imgui/
3 rm -rf MPIomp
4 mkdir MPIomp
5 for body_num in 200 400 800
6 do
7     for proc_num in 1 64 96 128
8     do
```

```
9         settings="#!/bin/bash\n#SBATCH --account=csc4005\n#SBATCH --
    partition=debug\n#SBATCH --qos=normal\n#SBATCH --ntasks=128\n#SBATCH
     --nodes=4\n"
10        command_part1="xvfb-run -a mpirun -n "
11        command_part2=" ./csc4005_imgui y > /pvfsmnt/118020362/csc4005-
    imgui/MPI/"
12        cd /home/118020362/csc4005-imgui/src/
13        sed -i '24c static int bodies = '${body_num}';' main.cpp
14        thread_num=1
15        if [ $proc_num -eq 1 ]; then
16            thread_num=1
17        elif [ $proc_num -eq 64 ]; then
18            thread_num=50
19        elif [ $proc_num -eq 96 ]; then
20            thread_num=100
21        elif [$proc_num -eq 128 ]; then
22            thread_num=200
23        fi
24        sed -i '8c #define NUM_THREADS '${thread_num}'' main.cpp
25        cd ..
26        rm -rf build
27        mkdir build
28        cd build
29        cmake .. -DCMAKE_BUILD_TYPE=Debug
30        cmake --build . -j4
31        cp -r /home/118020362/csc4005-imgui/build/ /pvfsmnt/118020362/
    csc4005-imgui/
32        cd /pvfsmnt/118020362/csc4005-imgui/
33        command_part_0="cd /pvfsmnt/118020362/csc4005-imgui/build/"
34        sh_name=$(printf run_MPIomp_p%03d_t%03d_body%03d.sh ${proc_num}
     ${thread_num} ${body_num})
35        result_name=$(printf MPIomp_p%03d_t%03d_body%03d.txt ${proc_num
    } ${thread_num} ${body_num})
36        touch "$sh_name"
37        echo -e $settings > "$sh_name"
38        echo -e $command_part_0 >> "$sh_name"
39        command_line="${command_part1}${proc_num}${command_part2}${
    result_name}"
40        echo $command_line >> "$sh_name"
41        sbatch "$sh_name"
42    done
43 done
```

Listing 12: create_runs_MPIomp.sh

3.7 CUDA

You can run the CUDA version using the command *sbatch run_CUDA.sh* or allocate an interactive section by *salloc -Nx -nx -tx*, then *sh run_CUDA.sh*. Either one is ok.

```
1 cd /pvfsmnt/118020362/csc4005-imgui/
2 rm -rf build
3 mkdir build
```

```
cd build
source scl_source enable devtoolset-10
CC=g++ CXX=g++ cmake ..
make -j4
# you can use salloc to srun csc4005_imgui seperately from the
    compilation
salloc -N1 -n2 -t3
srun csc4005_imgui

```

Listing 13: run_CUDA.sh

In case you want to test the CUDA version under different body numbers, thread numbers and block numbers, I provide a .sh file that can generate the bash files that satisfy the requirement.

Basically, the logic is to first compile the program under /home/118020362/csc4005-imgui/ and move it to /pvfmsnt/118020362/csc4005-imgui/.

To use this file, you just type in *sh create_runs_cuda.sh*. The running results are stored in /pvfsmnt/118020362/csc4005-imgui/cuda/.

```
#!/bin/bash
cd /pvfsmnt/118020362/csc4005-imgui/
rm -rf cuda
mkdir cuda
for body_num in 200 400 800
do
    for block_num in 1 5 10 20
    do
        settings="#!/bin/bash\n#SBATCH --account=csc4005\n#SBATCH --
    partition=debug\n#SBATCH --qos=normal\n#SBATCH --ntasks=128\n#SBATCH
     --nodes=4\n"
        command_part1="xvfb-run -a srun "
        command_part2=" ./csc4005_imgui y > /pvfsmnt/118020362/csc4005-
    imgui/cuda/"
        cd /home/118020362/csc4005-imgui/src/
        sed -i '106c static int bodies = '${body_num}';' main.cu
        sed -i '94c grid_size.x = '${block_num}';' main.cu
        thread_num=1
        if [ $block_num -eq 1 ]; then
            thread_num=1
        else
            thread_num=10
        fi
        sed -i '99c block_size.x = '${thread_num}';' main.cu
        cd ..
        rm -rf build
        mkdir build
        cd build
        source scl_source enable devtoolset-10
        CC=g++ CXX=g++ cmake ..
        make -j4
        cp -r /home/118020362/csc4005-imgui/build/ /pvfsmnt/118020362/
    csc4005-imgui/
        cd /pvfsmnt/118020362/csc4005-imgui/
```

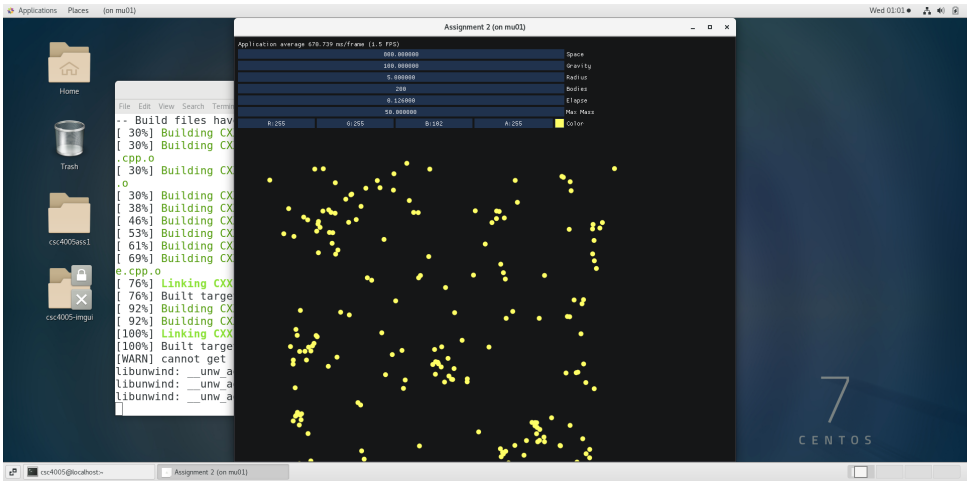
```
31          command_part_0="cd /pvfsmnt/118020362/csc4005-imgui/build/"
32          sh_name=$(printf run_cuda_b%02d_t%02d_body%03d.sh ${block_num}
    ${thread_num} ${body_num})
33          result_name=$(printf cuda_b%02d_t_%02d_body%03d.txt ${block_num
    } ${thread_num} ${body_num})
34          touch "$sh_name"
35          echo -e $settings > "$sh_name"
36          echo -e $command_part_0 >> "$sh_name"
37          command_line="${command_part1}${proc_num}${command_part2}${
    result_name}"
38          echo $command_line >> "$sh_name"
39          sbatch "$sh_name"
40      done
41  done
```

Listing 14: create_runs_cuda.sh

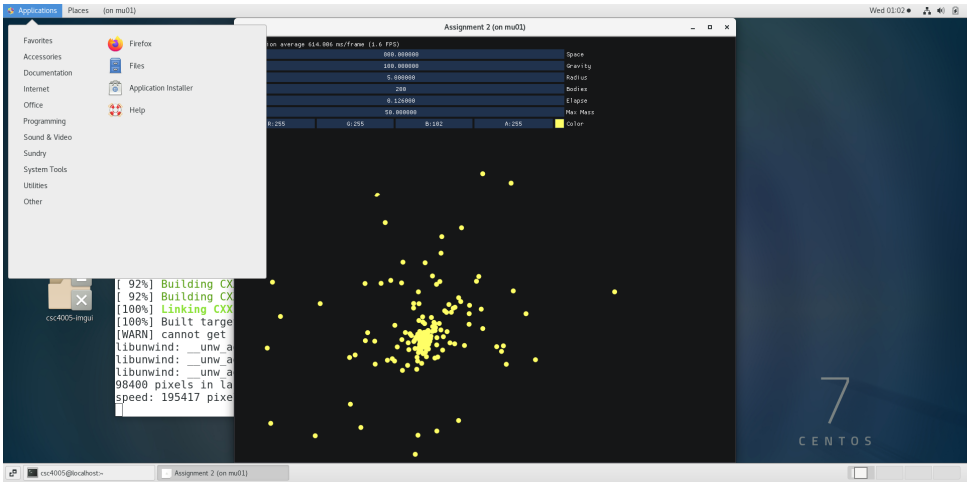
4 Screenshot of result

Shown below are the screenshots of results. The videos are attached into my submitting folder.

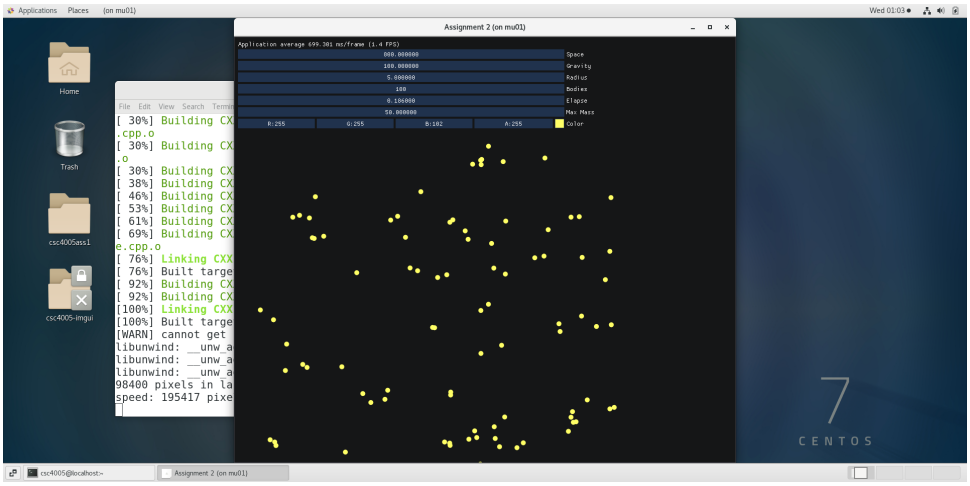
4.1 Sequential



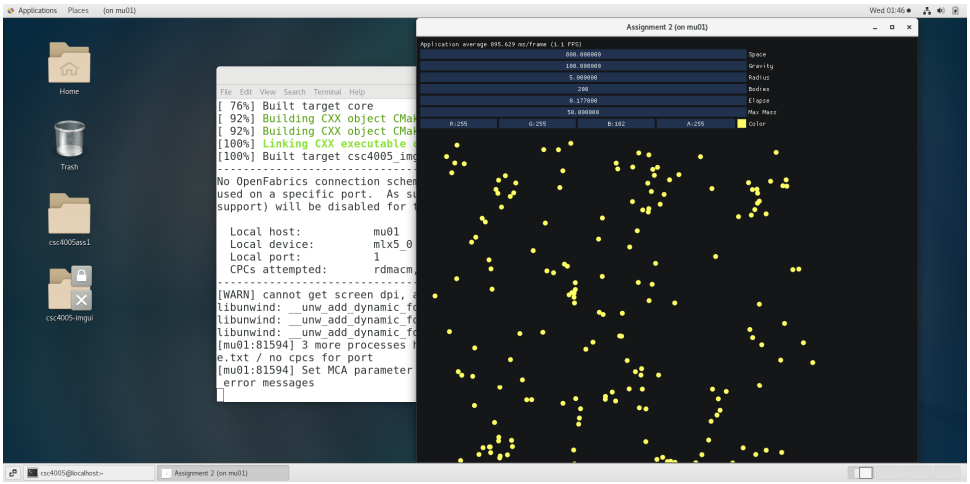
4.2 pthread



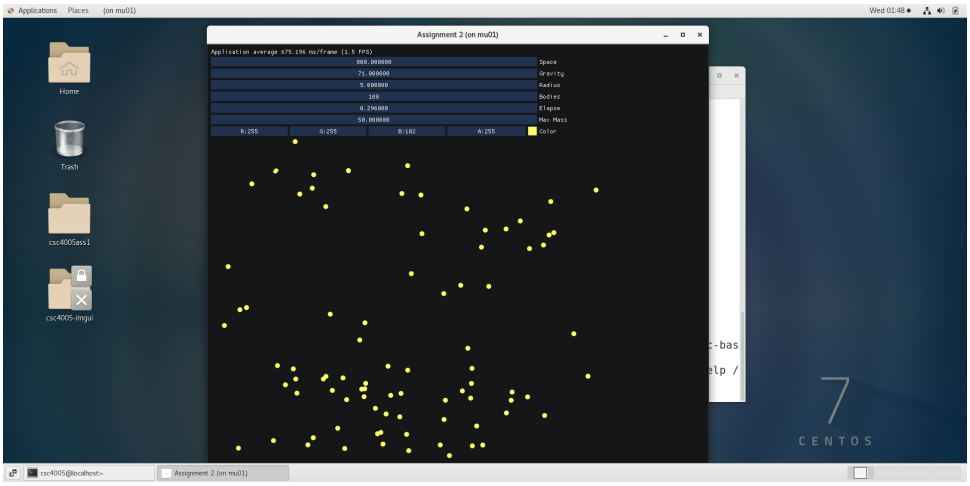
4.3 OpenMP



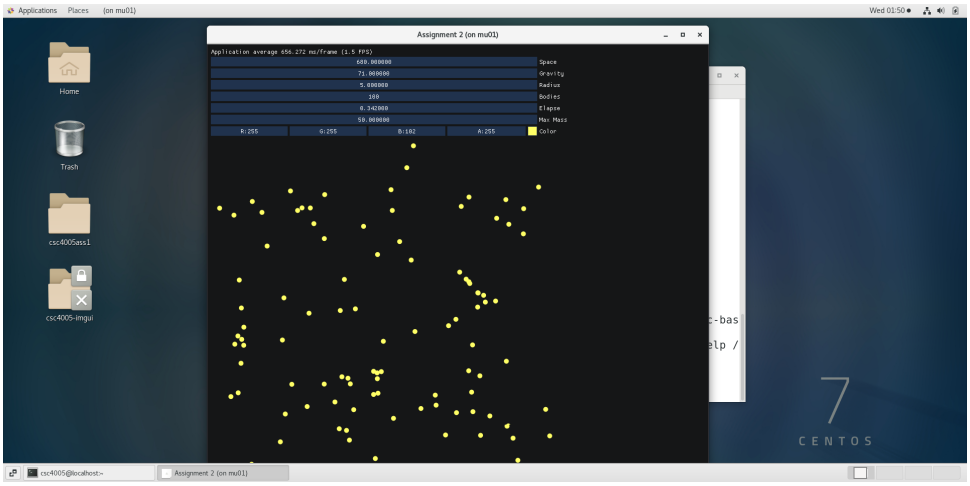
4.4 MPI



4.5 OpemMP+MPI



4.6 CUDA



5 Design of My Programs

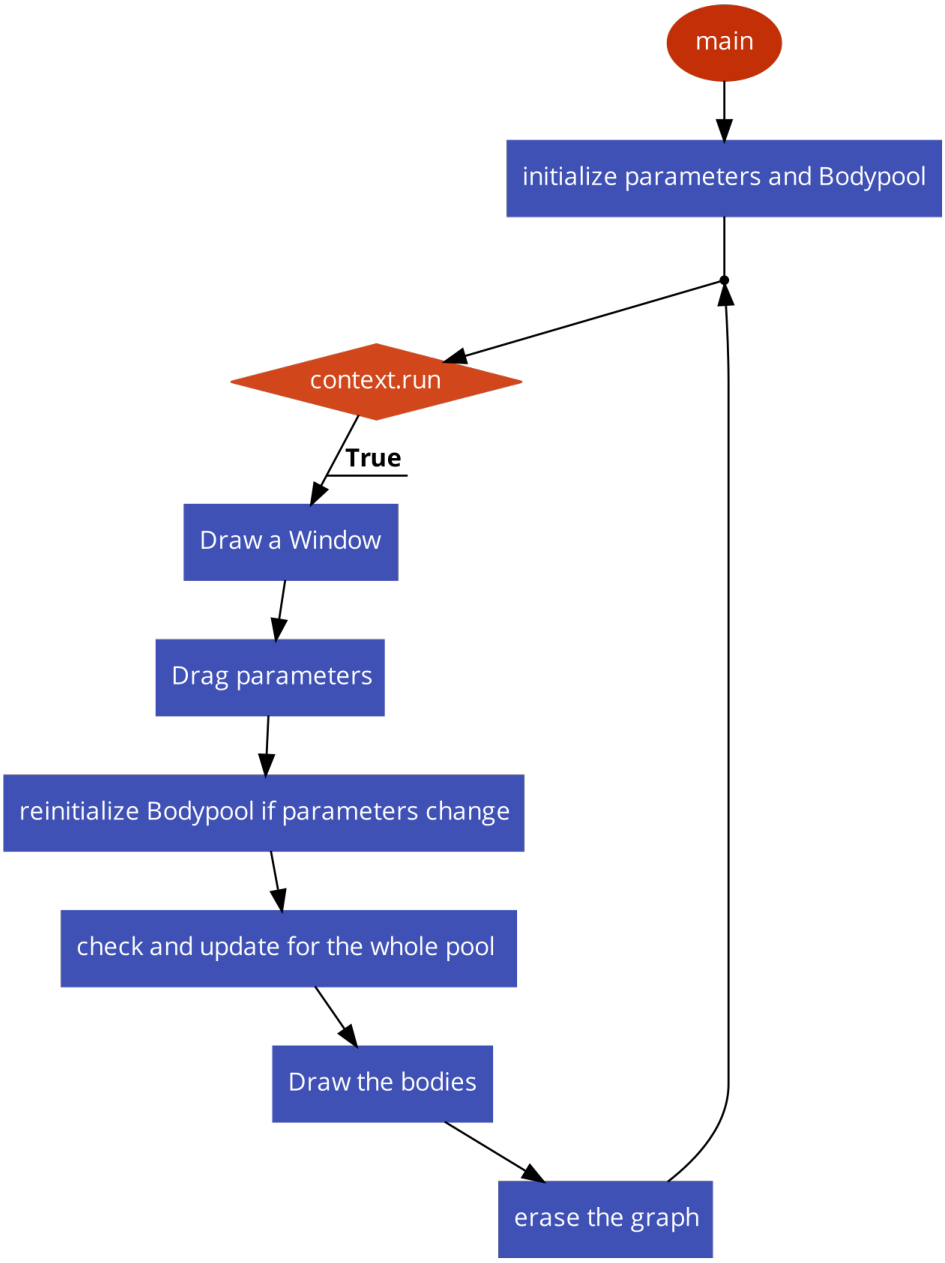
5.1 Sequential

5.1.1 Description

The sequential version is given to us as a template. It can be divided into several parts.

- Two classes: Bodypool and Body
  Bodypool is for storing and modifying the x, y, m, vx, vy, ax, ay of all bodies as a whole. Body is a subclass of Bodypool and handle update from one body's perspective.
- Bodypool.check_and_update()
  This is a class function belongs to Bodypool, which is to handle both collision case and non-collision case. When two bodies are collided, this function reset these two bodies' positions and velocities to the opposite directions. When two bodies are not collided, this function update each body's acceleration induced by the gravity force of another body.
- Body.handle_wall_collision()
  This function handles the case when a body is colliding with the wall. It resets the body's velocity and position to the opposite directions.
- Body.update_for_tick()
  This function updates velocity and position of each body according to the calculated accelerations. It also calls handle_wall_collision() to handle the wall collision.
- Bodypool.update_for_tick()
  This function calls check_and_update() for each body, and finally calls Body.update_for_tick() for each body. Done for one iteration!

5.1.2 Flow Chart



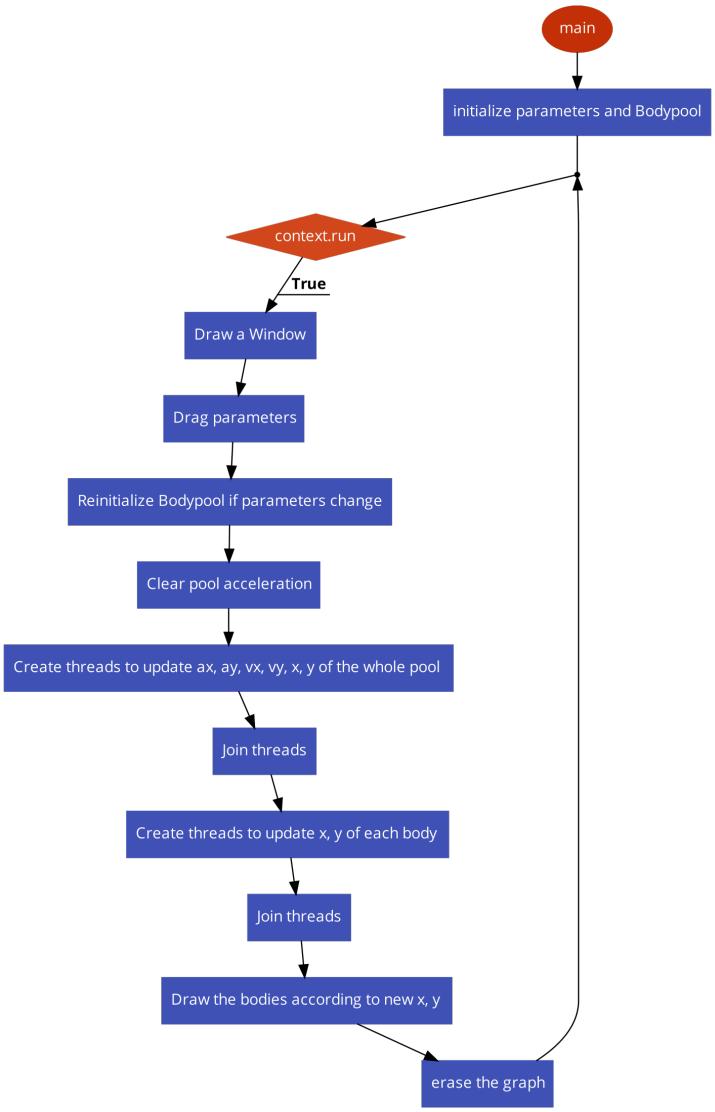
5.2 pthread

5.2.1 Description

Since pthread has shared memory among each thread, the whole process can be parallelized using pthread. However, it should be noticed that only after the Bodypool.check_and_update has done, can the bodies begin update their own vx, vy, x, y,

which is the function pool.update_for_tick. Therefore, Bodypool.update_for_tick is required to seperate apart.

5.2.2 Flow Chart

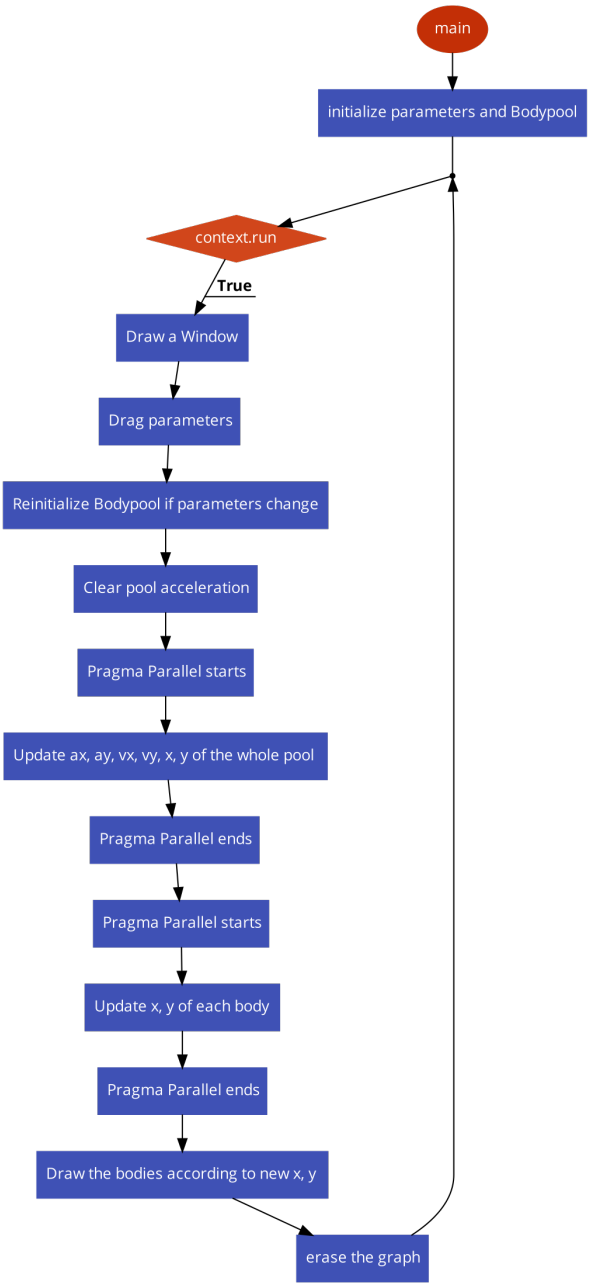


5.3 OpenMP

5.3.1 Description

It is the similar case for OpenMP with the pthread.

5.3.2 Flow Chart



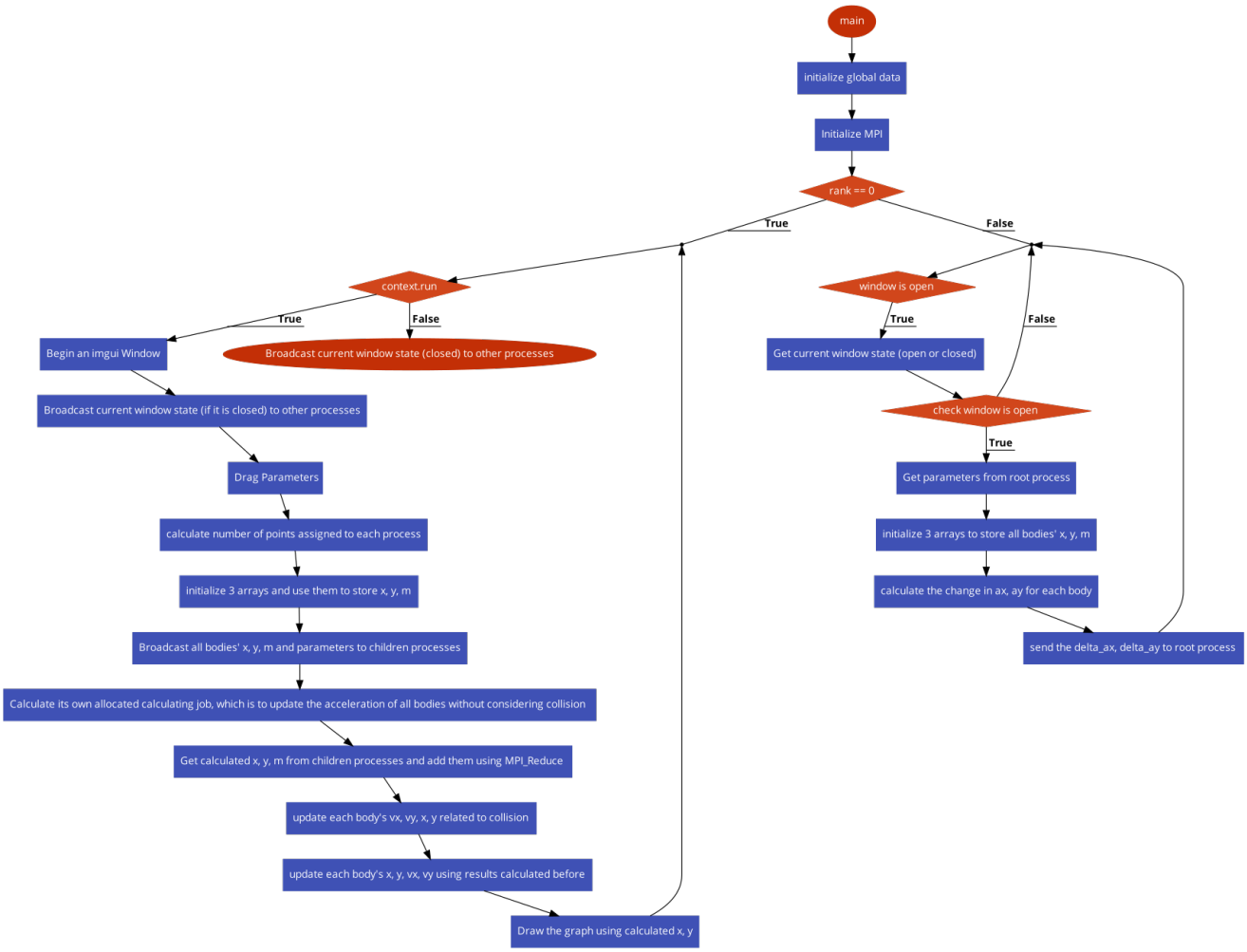
5.4 MPI

5.4.1 Description

Since MPI use Message Passing mechanism to pass information among processes, there is no shared memory among different processes. Therefore, when two bodies,such as a,b, are colliding, and the process which is in charge of a has modified b's vx, vy, x, y. However, it is impossible for the process which is in charge of b to see the modification

and it will modify both a and b's vx, vy, x, y for one more time. This is certainly not we want. Therefore, for MPI, the parallelization can only restricted to the ax, ay calculation with no collision part.

5.4.2 Flow Chart

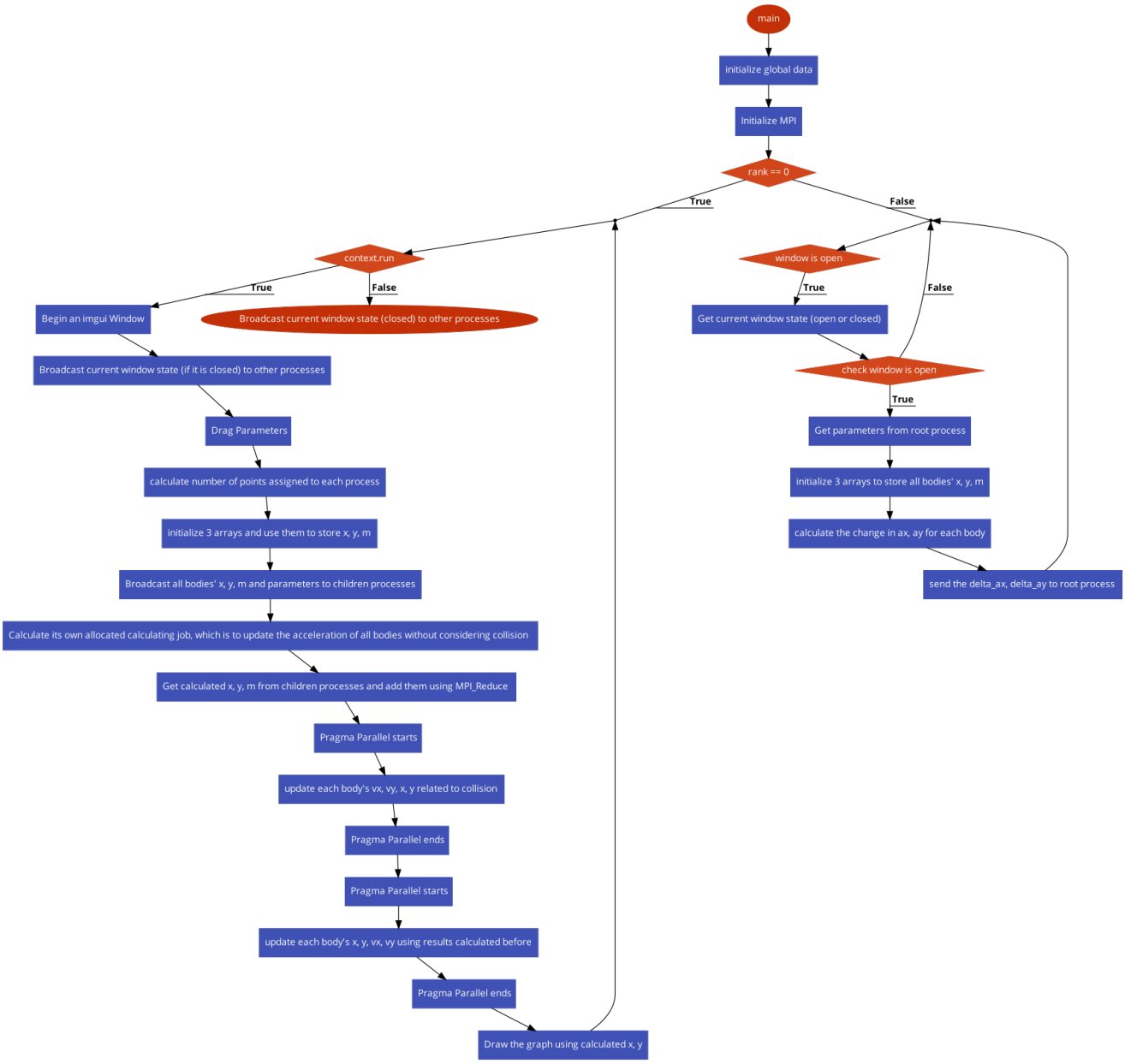


5.5 OpemMP+MPI

5.5.1 Description

As I have said in 4.4.1, it is impossible for MPI to parallelize the part handling collision. Also, for the individual update part, although it can be parallelized by MPI, it is not convenient to scatter the updated ax, ay, vx, vy, x and y to multiple processes and do the update. It is better to do in a sequential way if the program does not have a shared memory. Now, OpenMP is combined to overcome these dilemmas. It provides shared memory so that the part handling collision and updating individual body can be parallelized.

5.5.2 Flow Chart

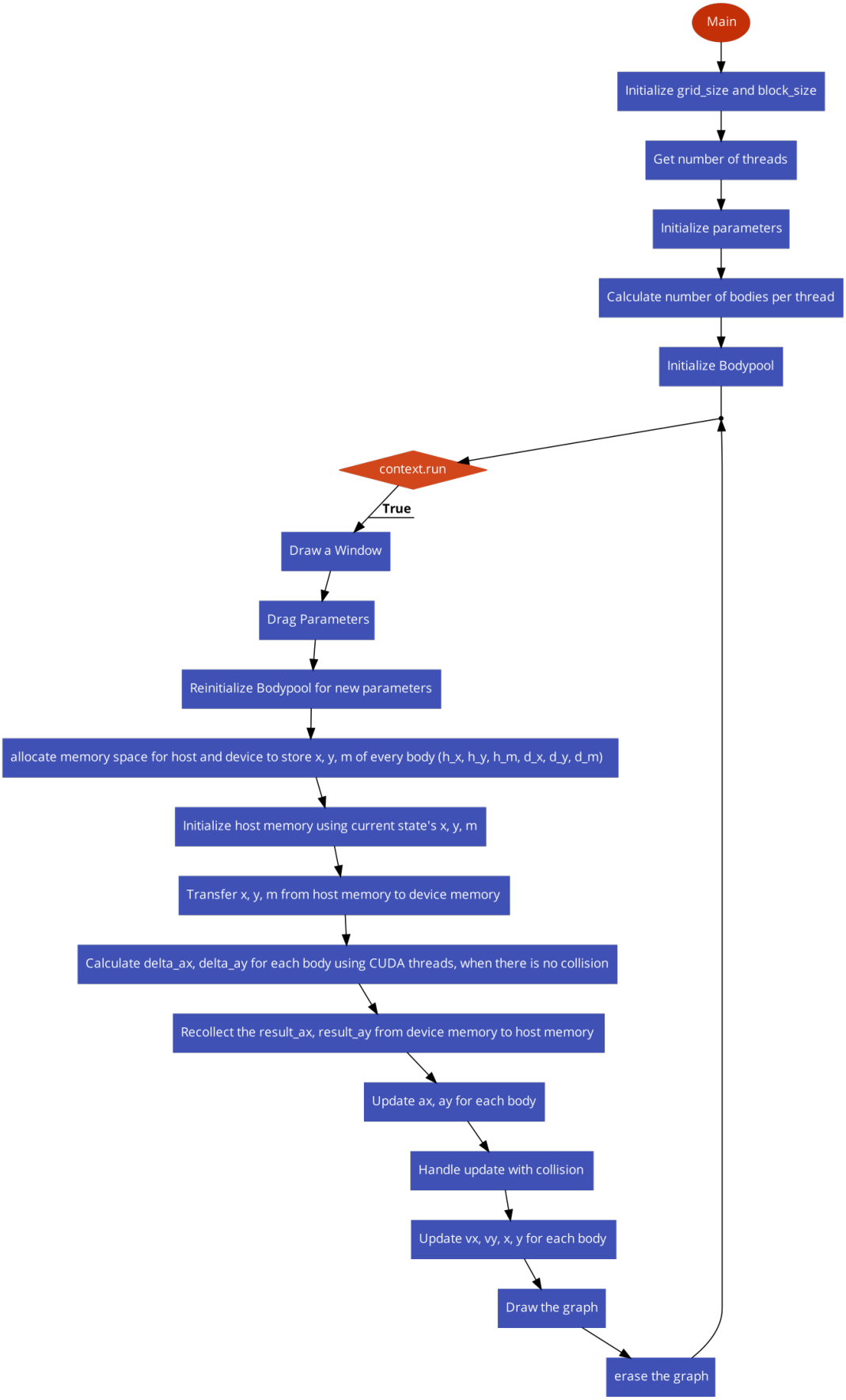


5.6 CUDA

5.6.1 Description

The part that can be parallelized by CUDA is similar with MPI since cudaMalloc() is allocating global memory which can be seen by all grids, blocks and threads. Therefore, it is very important for CUDA to handle data race.

5.6.2 Flow Chart



6 Experiment Settings and Graphs

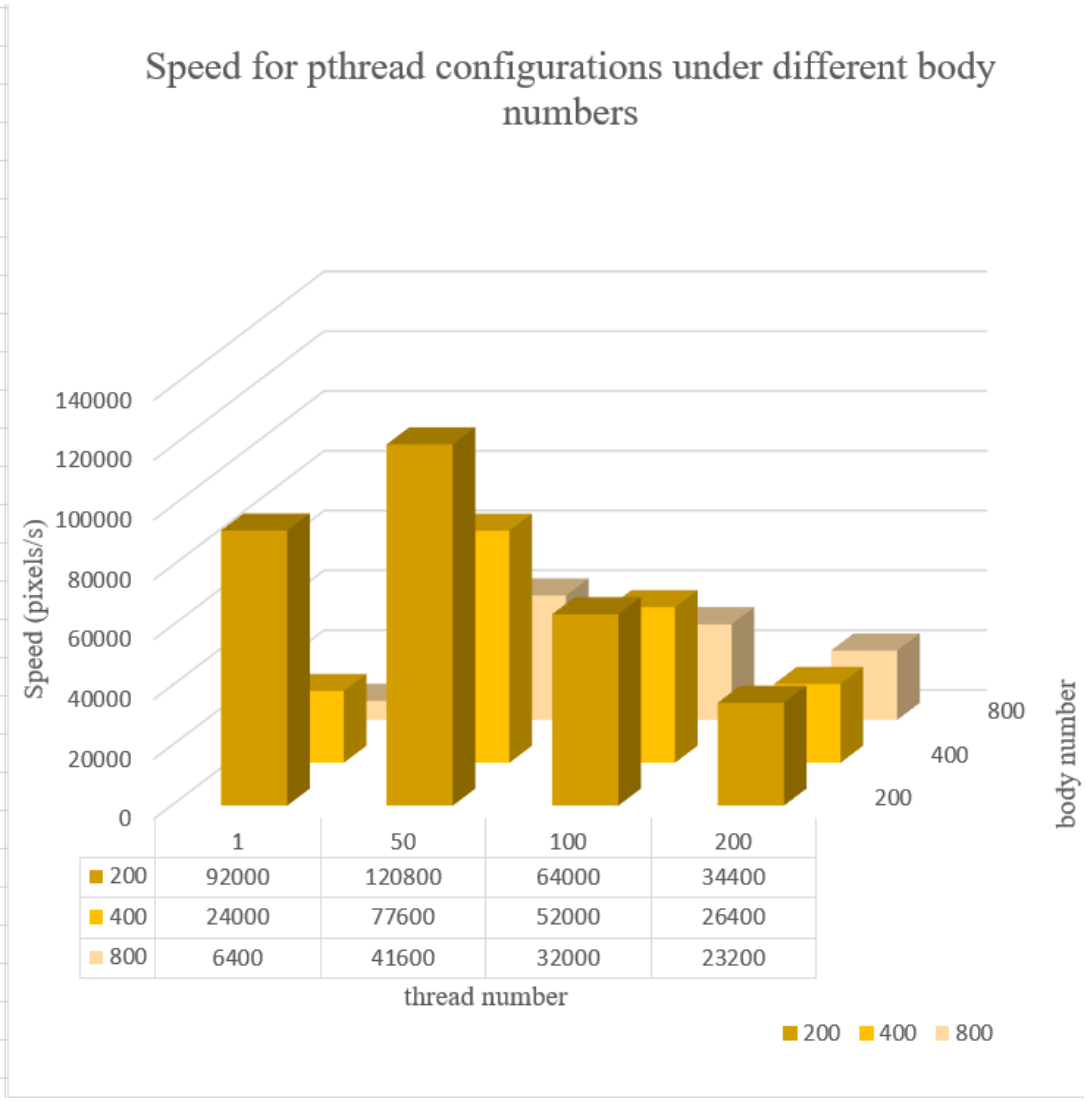
6.1 Description

Basically, I calculate speeds for different configurations under different body numbers, which are 200, 400 and 800.

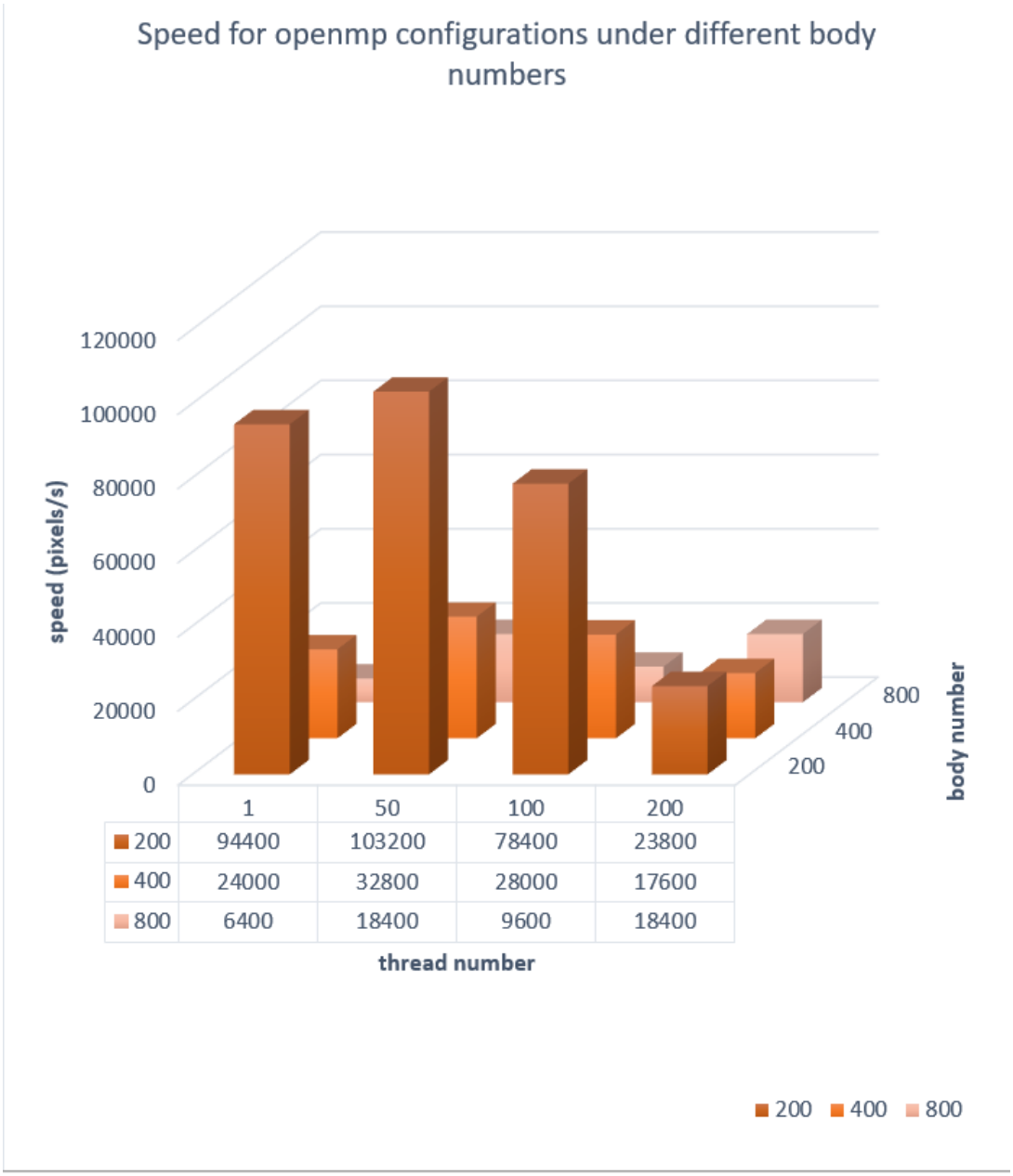
- For pthread, the configurations are 1, 50, 100, 200 thread numbers.
- For OpenMP, the configurations are 1, 50, 100, 200 thread numbers.
- For MPI, the configurations are 1, 64, 96, 128 process numbers.
- For CUDA, the configurations are 1, 50, 100, 200 thread numbers, the block number and thread number per block are b1t1, b5t10, b10t10, b20t10, respectively.
- For MPI+OpenMP, the configurations are p1t1, p64t50, p96t100, p128t200, where p is process number and t is thread number.

In this way, the tested thread number and process number are roughly the same, which are convenient for performance comparison.

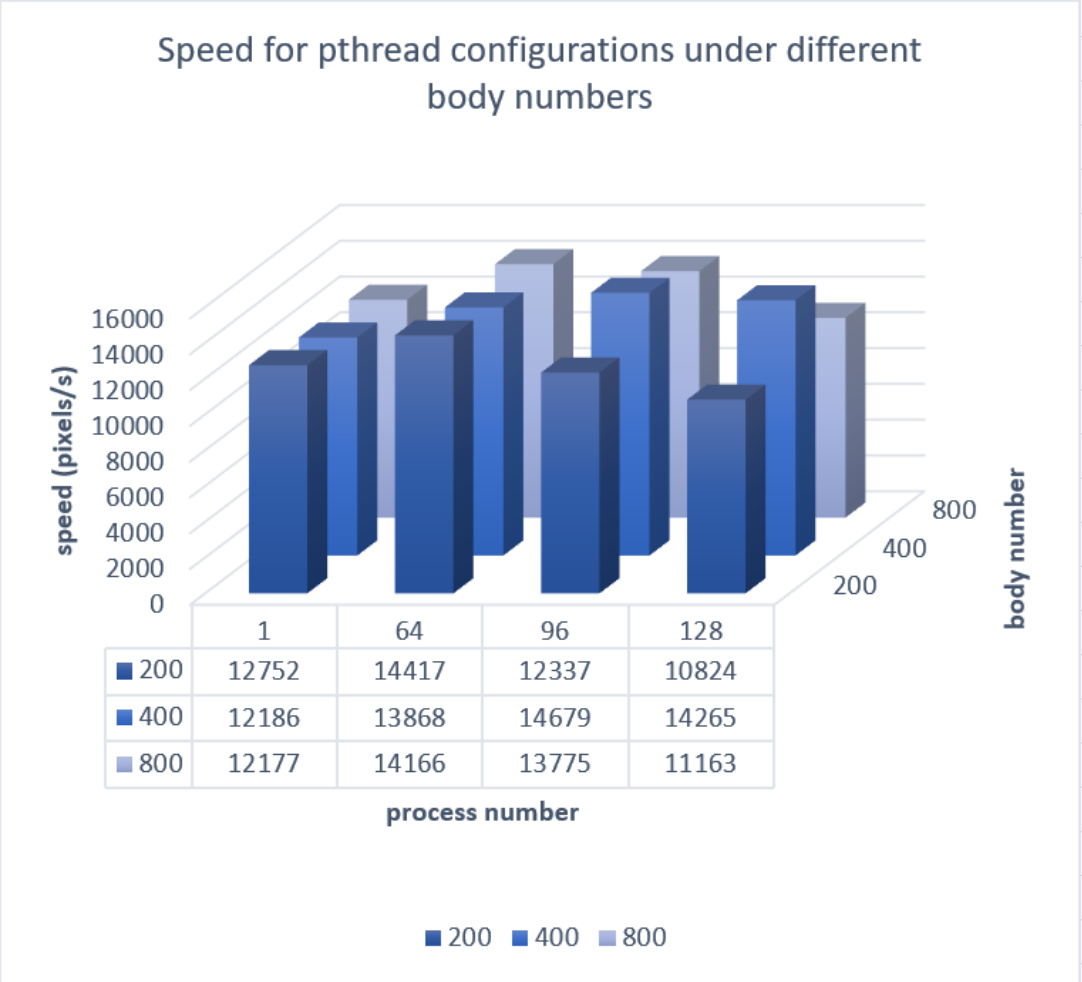
6.2 pthread


Speed for pthread configurations under different body numbers
Speed (pixels/s)
140000
120000
100000
80000
60000
40000
20000
0
200
400
800
body number
1
50
100
200
200 92000 120800 64000 34400
400 24000 77600 52000 26400
800 6400 41600 32000 23200
thread number
200 400 800

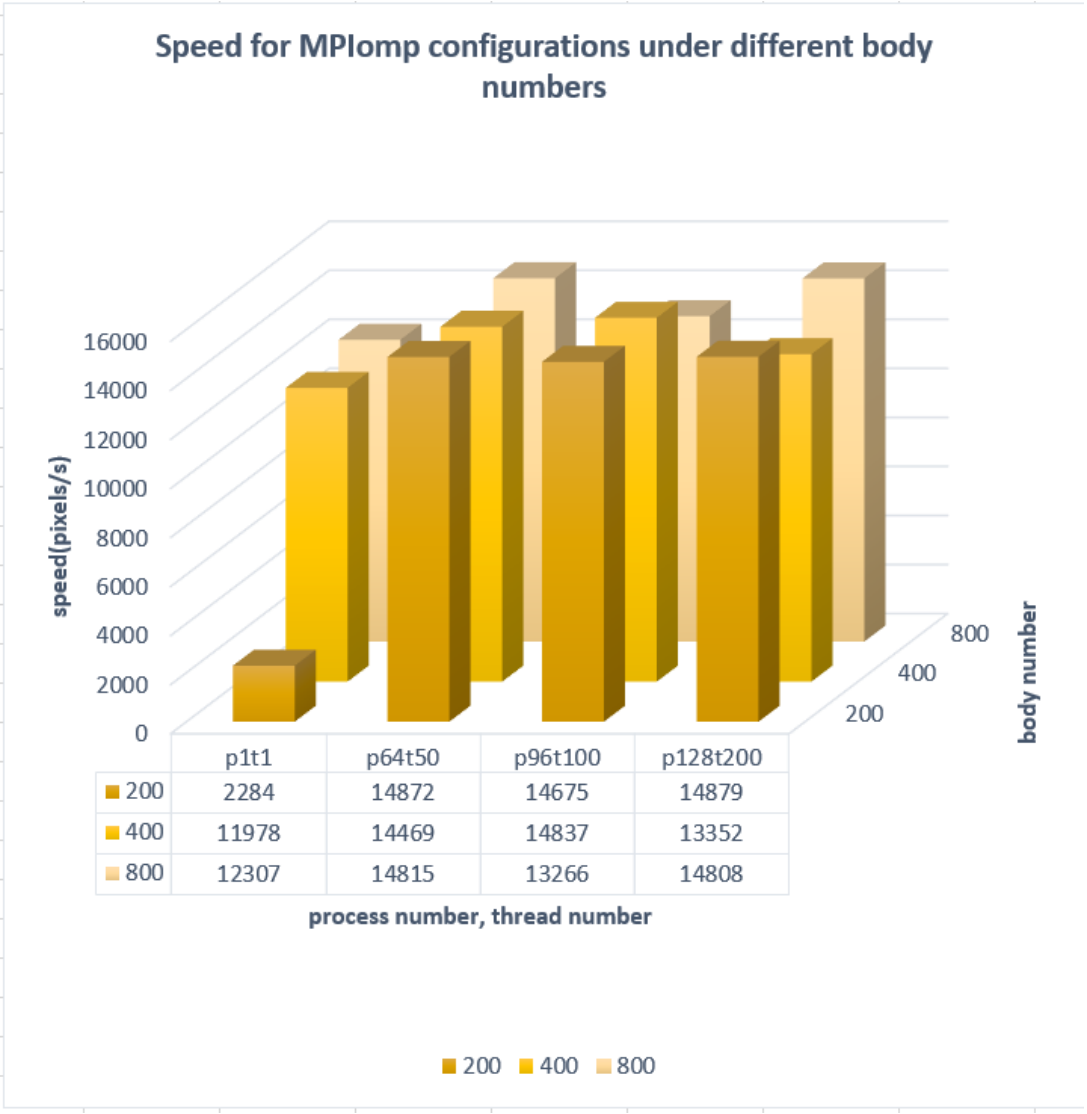
6.3 OpenMP


Speed for openmp configurations under different body numbers
speed (pixels/s)
120000
100000
80000
60000
40000
20000
0
body number
800
400
200
thread number
1
50
100
200
200 94400 103200 78400 23800
400 24000 32800 28000 17600
800 6400 18400 9600 18400
200
400
800

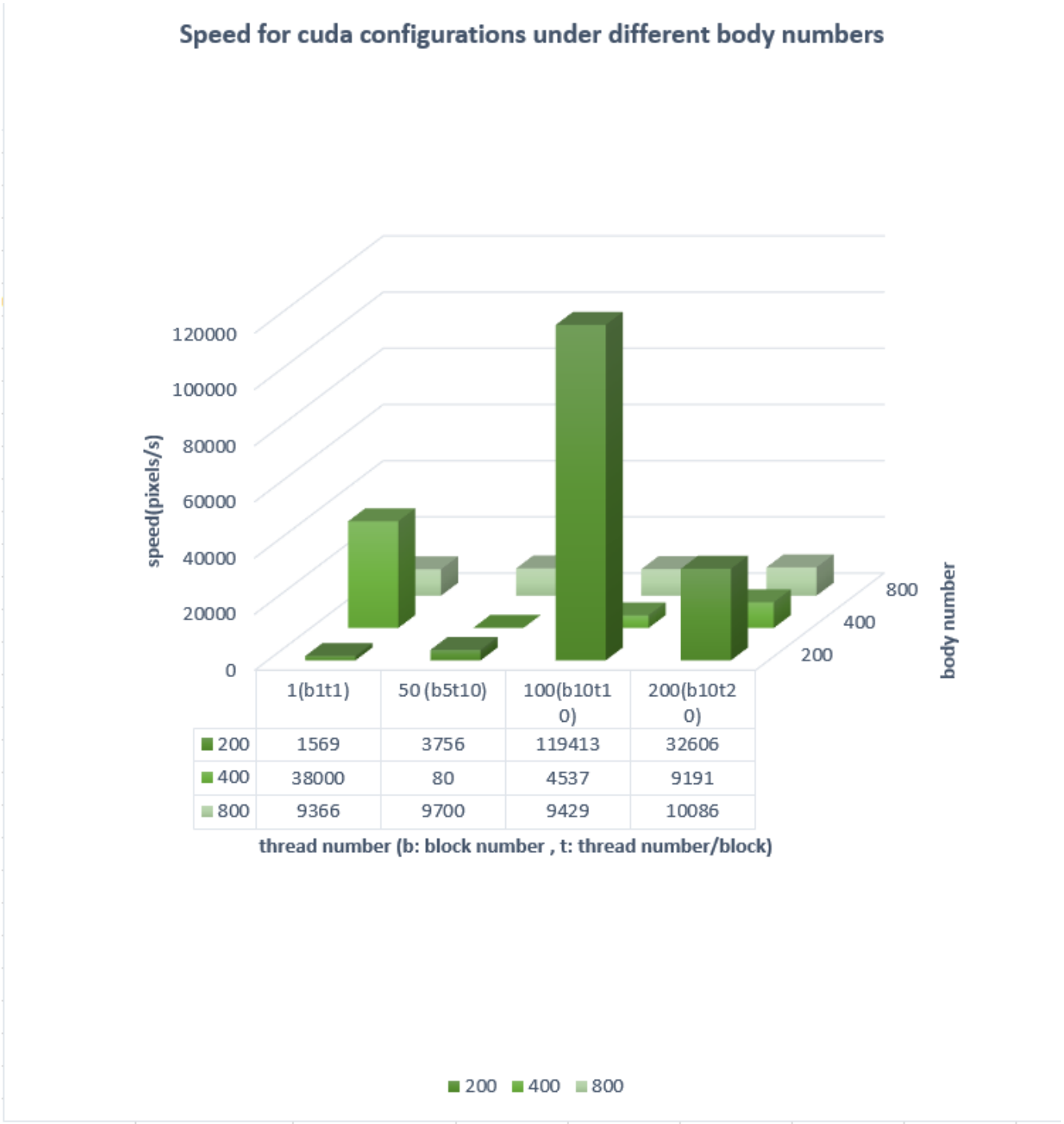
6.4 MPI


Speed for pthread configurations under different body numbers
speed (pixels/s)
16000
14000
12000
10000
8000
6000
4000
2000
0
1
64
96
128
200 12752 14417 12337 10824
400 12186 13868 14679 14265
800 12177 14166 13775 11163
process number
200
400
800
body number
200 400 800

6.5 MPI+OpenMP



6.6 CUDA


Speed for cuda configurations under different body numbers
speed(pixels/s)
120000
100000
80000
60000
40000
20000
0
200
400
800
body number
1(b1t1)
50 (b5t10)
100(b10t10)
200(b10t20)
200 1569 3756 119413 32606
400 38000 80 4537 9191
800 9366 9700 9429 10086
thread number (b: block number , t: thread number/block)
200 400 800

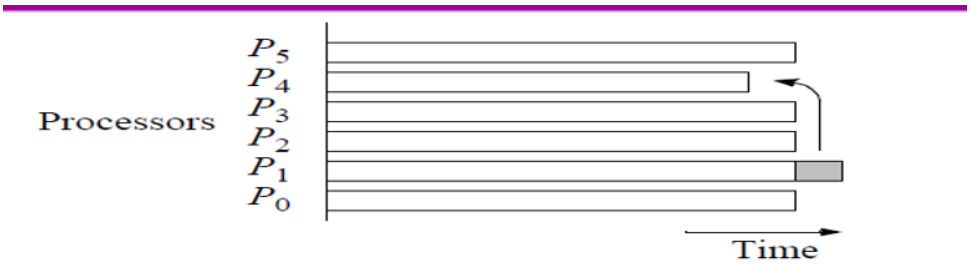
7 Performance Analysis

7.1 Load Balancing

First thing I want to mention is the load balancing problem. Since the algorithm of the Nbody simulation is like this:

- Calculate delta_ax, delta_ay, delta_vx, delta_vy, delta_x, delta_y for the first point. At the same time, every point behind the first point are updated on their delta_ax, delta_ay, delta_vx, delta_vy, delta_x, delta
- Calculate delta_ax, delta_ay, delta_vx, delta_vy, delta_x, delta_y for the second point. At the same time, every point behind the second point are updated on their delta_ax, delta_ay, delta_vx, delta_vy, delta_x, delta_y.
- continue the same process for the third, fourth, ... points.

Therefore, the workload for each point is not evenly distributed. The workload for calculating the first point is the largest, while the workload for calculating the last point is the smallest. Therefore, just as the figure shows, the imperfect load distribution will lead to increased execution time.



(a) Imperfect load balancing leading to increased execution time

7.2 pthread

- Looking Vertically
  It is logic to see that as the body number increases, the speeds for all settings are decreasing.
- Looking Horizontally
  From the figure we can know that when thread number is 50, the speed up is the largest for all body numbers. What's more, as the number of thread number increases to 100 and 200, the speed up is less than 1. This may be due to the high overhead once thread number is over the optimal one.

7.3 OpenMP

For OpenMP,

- The changing patterns looking from horizontally and vertically are both similar with the pthread.
- What's more, the average speeds for each different configurations are roughly the same with the corresponding one in pthread. The performance of OpenMP is roughly the same with pthread, either for the changing pattern or the abolute average speeds.

7.4 MPI

For MPI,

- Whether looking horizontally or vertically, the speed is not changing so much. It seems counter-intuitive. However, it may be due to the imperfect load balancing mentioned at 6.1. Since as the workload is increasing and number of process is increasing, the workload assigned to the first process is always the heaviest. Therefore, the speed for each configuration is determined by the slowest first process. The workload for the first process is of little change as both the body number and process number are increasing.
- The average speed for MPI each configuration is larger than pthread and OpenMP versions.

7.5 MPI+OpenMP

- For MPI+OpenMP, the changing pattern is similar with that of MPI. It can be easily explained by that the performance of MPI+OpenMP is largely determined by the MPI part.
- However, it can be easily found that the average speed value for each configuration is larger than that of MPI.

7.6 CUDA

- For CUDA, the changing pattern is quite weird. It is logic to see that as the body number is increasing, the speed is decreasing for 2 configurations. However, the overall changing patterns are so strange.
- This may be due to that CUDA only has shared memory among one block. Therefore, as the block number is large, the speed up gained by the increasing thread number is compensated by the communication overhead between different blocks.
- The average speed value for each configuration is obviously smaller than any other four versions.

7.7 Summary

- Average speed value for each configuration: MPI+OpenMP > MPI > OpenMP = pthread > CUDA

- Different parallelization frameworks have different memory handling designs and message passing mechanisms.

8 Conclusion

In this project, I implement parallelized Nbody simulation versions using four parallel frameworks, which are pthread, MPI, OpenMP and CUDA.

After analyzing which parts of the sequential program are suitable for parallelization, I find that among these four APIs, pthread and OpenMP and similar to each other, while MPI and CUDA are similar to each other.

However, OpenMP is easier to use and more flexible than pthread, while MPI is easier to use than CUDA since CUDA has multiple memory hierarchies which needs to be handled by programmers.

After analyzing the change of speed under different workloads and configurations, I get the speed ranking of the five versions.

It can be seen that different parallelization frameworks have different memory handling designs and message passing mechanisms.