

CSC4005 Assignment 3

119010115 Yongjin, Huang

Execution

Build

```
1 | cd /path/to/project
2 | # 1. On virtual Machine(without cuda)
3 | ./debug.sh
4 | # 2. On Slurm(with cuda)
5 | ./release.sh
```

Run

```
1 | cd /path/to/project/build
2 |
```

Introduction

An N-body simulation approximates the motion of particles, often specifically particles that interact with one another through some types of physical forces. Using this broad definition, the types of particles that can be simulated using n-body methods are quite significant, ranging from celestial bodies to individual atoms in a gas cloud. From here out, we will specialize the conversation to gravitational interactions, where individual particles are defined as a physical celestial body, such as a planet, star, or black hole. A motion of the particles on the bodies themselves is neglected since it is often not interesting to the problem and will hence add an unnecessarily large number of particles to the simulation. N-body simulation has numerous applications in areas such as astrophysics, molecular dynamics, and plasma physics. The simulation proceeds over time steps, each time computing the net force on each body and thereby updating its position and other attributes. If all pairwise forces are computed directly, this requires $O(N^2)$ operations at each time step.

In this assignment, the N-body simulation is in two dimensions. In order to visualize the N-body simulation, each body is modeled as a ball.

Design

In each time step, the simulation is divided into two parts for each body:

1. Part One: Calculate the forces between it and the others, and check whether two bodies collide. Two bodies collide if their distance is less than the radius of the ball.
2. Part Two: Update its position, velocity based on the time elapse and check whether it crosses the boundary.

If there are N bodies, the time complexity of the simulation in each time step is $O(N^2)$

The design of sequential version and parallel version differs a little.

- In the sequential version, in Part One, if two bodies collide, their position and velocity are immediately updated. Thus, Part One computation is halved, and only $\frac{N(N-1)}{2}$ calculation is needed.
- In parallel versions, in Part One, if two bodies collide, the changes in their position and velocity are saved. Thus, their position and velocity are not updated in Part One but in Part Two. The reason for this design is to eliminate the data dependency between bodies.

This slight difference results in different movements of the bodies in visualization.

Implementations

Sequential Implementation

This sequential implementation is given by teaching assistants.

```
1 void update_for_tick(double elapse,
2                     double gravity,
3                     double position_range,
4                     double radius)
5 {
6     ax.assign(size(), 0);
7     ay.assign(size(), 0);
8     // Part One
9     for (size_t i = 0; i < size(); ++i)
10    {
11        for (size_t j = i + 1; j < size(); ++j)
12        {
13            // update both bodies' positions and velocities immediately
14            check_and_update(get_body(i), get_body(j), radius, gravity);
15        }
16    }
17    // Part Two
18    for (size_t i = 0; i < size(); ++i)
19    {
20        get_body(i).update_for_tick(elapse, position_range, radius);
21    }
22 }
```

MPI Implementation

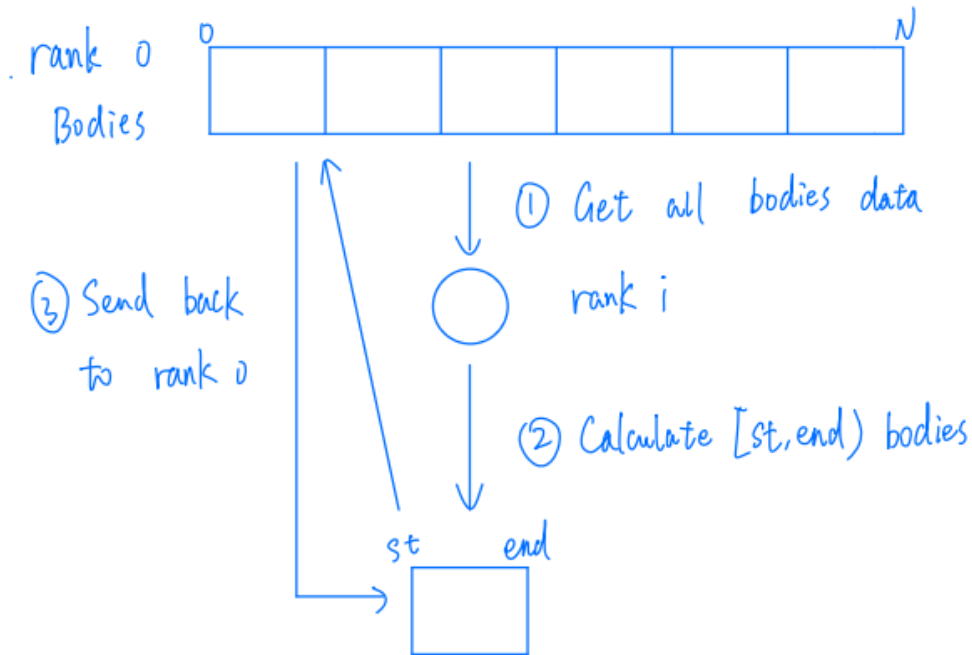


Figure 1: MPI Implementation

The MPI implementation contains the following steps:

1. The root process creates a body pool and broadcast it to every other process.
2. Each process calculates its range of bodies. Bodies are distributed evenly across processes.
3. Each process runs Part One and Part Two.
4. Gather bodies data to the root process.

```

1 void worker(int rank, int world_size)
2 {
3     // Preparation: Broadcast all bodies to all processes
4     // ...
5     // Each process calculate its range of bodies
6     pool.clear_acceleration();
7     int elements_per_process = pool.size() / world_size;
8     size_t st_idx = elements_per_process * rank;
9     size_t end_idx = st_idx + elements_per_process;
10    if (rank == world_size - 1)
11        end_idx = pool.size();
12    // Part One
13    for (size_t i = st_idx; i < end_idx; i++)
14    {
15        for (size_t j = 0; j < pool.size(); j++)
16        {
17            if (i == j)
18                continue;
19            // update body i's position and velocity immediately
20            pool.mpi_check_and_update(pool.get_body(i), pool.get_body(j),
21            local_radius, local_gravity);
22        }
23    }
24    // Part Two
25    for (size_t i = st_idx; i < end_idx; i++)
26    {

```

```

26     pool.get_body(i).update_for_tick(local_elapse, local_space,
    local_radius);
27     }
28     // Gather bodies data to the root process
29     // ...
30 }

```

Pthread Implementation

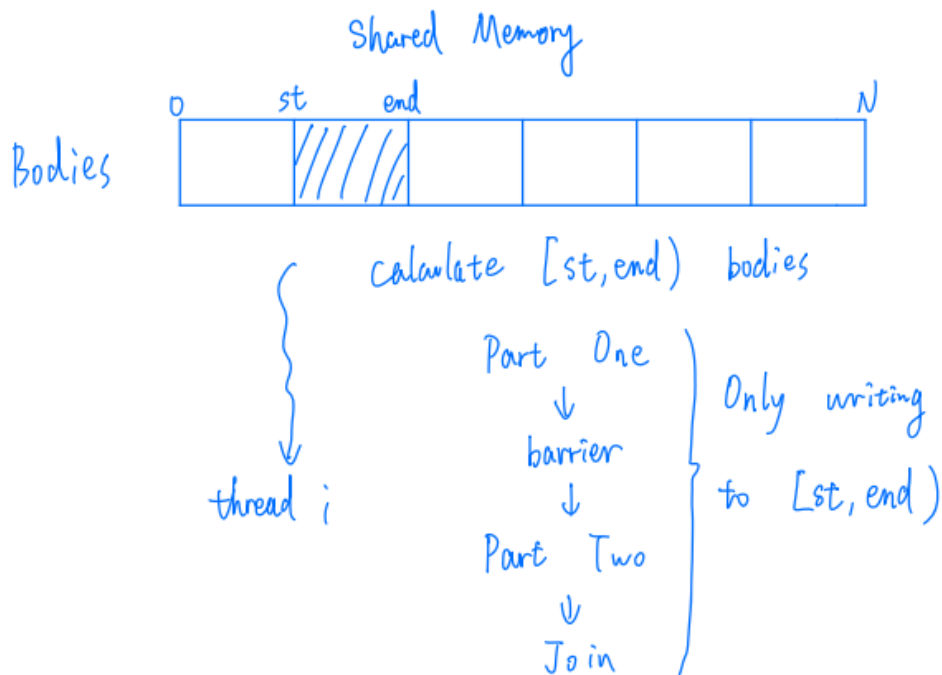


Figure 2: Pthread Implementation

The Pthread implementation contains the following steps:

1. The main thread distributes bodies range evenly and create child threads.
2. Each child thread gets its bodies range.
3. Each child thread runs Part One, and wait until all child threads complete Part One using a barrier. Then run Part Two.

```

1  void schedule(size_t thread_num)
2  {
3      std::vector<pthread_t> threads(thread_num);
4      pthread_barrier_init(&barrier, NULL, thread_num);
5      pool.clear_acceleration();
6      pool.init_delta_vector();
7      size_t idx_per_thread = pool.size() / thread_num;
8      size_t remainder = pool.size() % thread_num;
9      size_t st_idx = 0;
10     std::vector<idx_struct> idx_struct_arr;
11     for (size_t i = 0; i < threads.size(); i++)
12     {
13         size_t end_idx = i < remainder ? st_idx + idx_per_thread + 1 :
    st_idx + idx_per_thread;
14         idx_struct_arr.push_back({st_idx, end_idx});

```

```

15     st_idx = end_idx;
16 }
17 for (size_t i = 0; i < threads.size(); i++)
18 {
19     pthread_create(&threads[i], nullptr, worker, reinterpret_cast<void
*>(&idx_struct_arr[i]));
20 }
21 for (auto &thread : threads)
22 {
23     pthread_join(thread, nullptr);
24 }
25 }
26
27 void *worker(void *data)
28 {
29     struct idx_struct *p = reinterpret_cast<idx_struct *>(data);
30     // Part One
31     for (size_t i = p->st_idx; i < p->end_idx; i++)
32     {
33         for (size_t j = 0; j < pool.size(); ++j)
34         {
35             if (i == j)
36                 continue;
37             // save the changes in body i's position and velocity
38             pool.shared_memory_check_and_update(pool.get_body(i),
pool.get_body(j), radius, gravity);
39         }
40     }
41     pthread_barrier_wait(&barrier);
42     // Part Two
43     for (size_t i = p->st_idx; i < p->end_idx; i++)
44     {
45         // update body i's position and velocity using the saved changes in
Part One
46         pool.get_body(i).update_by_delta_vector();
47         pool.get_body(i).update_for_tick(elapse, space, radius);
48     }
49     return nullptr;
50 }

```

OpenMP Implementation

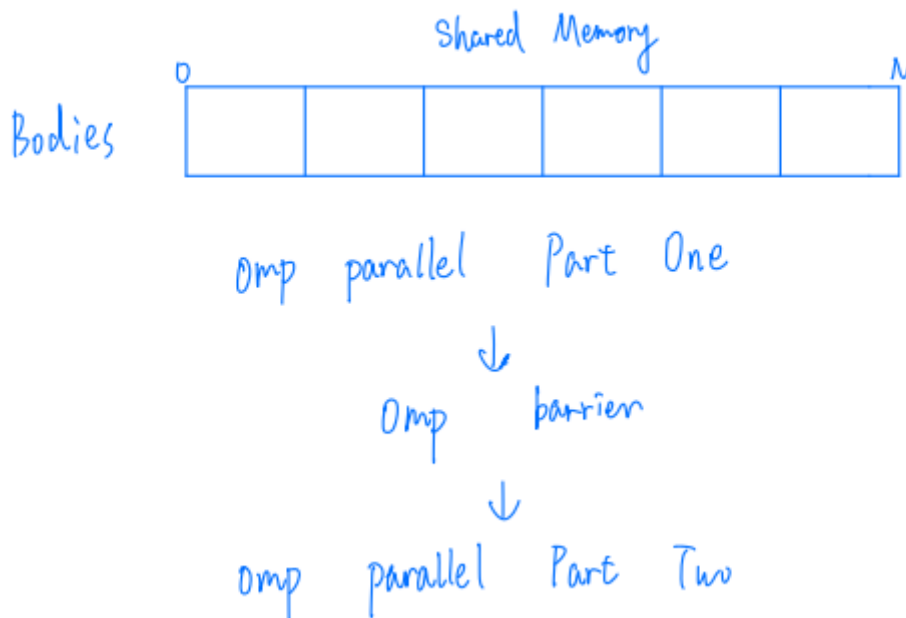


Figure 3: OpenMP Implementation

Similar to Pthread implementation, the OpenMP implementation contains the following steps:

1. Split Part One among working threads, and each working thread gets one or more bodies.
2. Each working thread runs Part One. After all working threads have finished Part One, they start to run Part Two.

```

1  void schedule()
2  {
3      pool.clear_acceleration();
4      pool.init_delta_vector();
5      // Part One
6      #pragma omp parallel for shared(pool)
7          for (size_t i = 0; i < pool.size(); ++i)
8          {
9              for (size_t j = 0; j < pool.size(); ++j)
10             {
11                 if (i == j)
12                     continue;
13                 // save the changes in body i's position and velocity
14                 pool.shared_memory_check_and_update(pool.get_body(i),
15                 pool.get_body(j), radius, gravity);
16             }
17         }
18         // Part Two
19         #pragma omp parallel for shared(pool)
20             for (size_t i = 0; i < pool.size(); ++i)
21             {
22                 // update body i's position and velocity using the saved changes in
23                 // Part One
24                 pool.get_body(i).update_by_delta_vector();
25                 pool.get_body(i).update_for_tick(elapse, space, radius);
26             }
27     }

```

CUDA Implementation

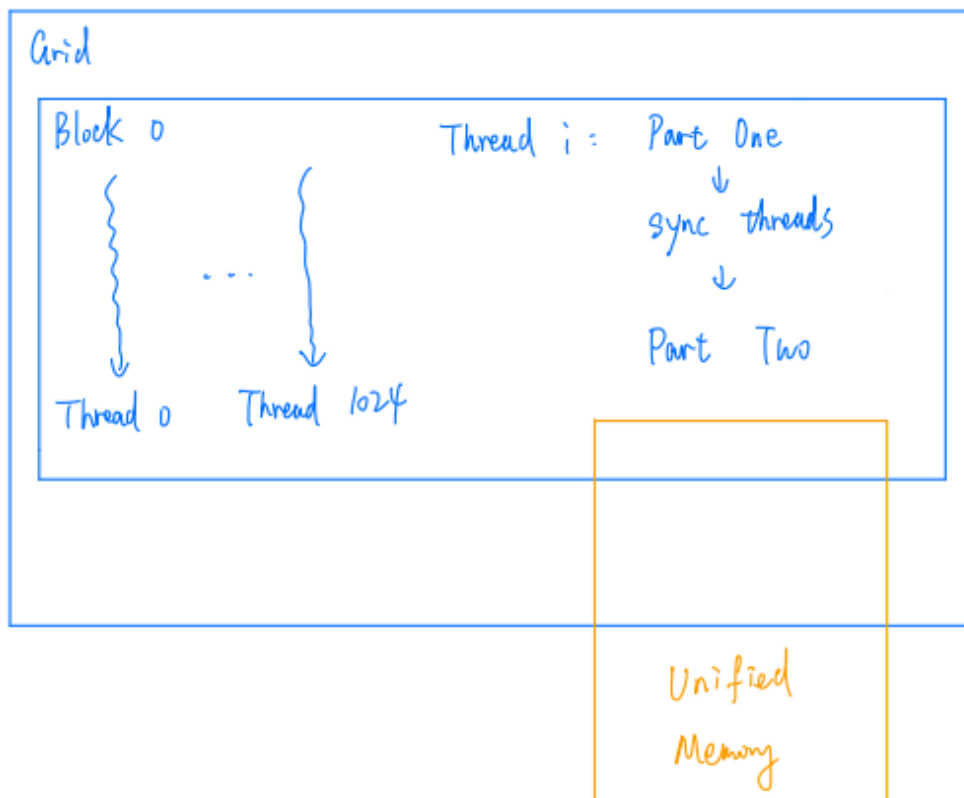


Figure 4: CUDA Implementation

Similar to Pthread implementation, the OpenMP implementation contains the following steps:

1. Create a block with many threads using CUDA.
2. Each thread gets its bodies range.
3. Each thread runs Part One, and wait until all child threads complete Part One. Then run Part Two.

```

1  __device__ __managed__ BodyPool *pool;
2  // worker() is called in this way:
3  int main()
4  {
5      dim3 grid(1);
6      dim3 block(thread_num);
7      worker<<<grid, block>>>();
8  }
9
10 __global__ void worker()
11 {
12     int thread_id = threadIdx.x;
13     int elements_per_thread = pool->size / thread_num;
14     int st_idx = elements_per_thread * thread_id;
15     int end_idx = st_idx + elements_per_thread;
16     if (thread_id == thread_num - 1)
17         end_idx = pool->size;
18     // Part One
19     for (int i = st_idx; i < end_idx; i++)
20     {
21         for (int j = 0; j < pool->size; ++j)
22         {

```

```

23         if (i == j)
24             continue;
25         // save the changes in body i's position and velocity
26         pool->shared_memory_check_and_update(pool->get_body(i), pool-
>get_body(j), radius, gravity);
27     }
28 }
29 __syncthreads();
30 // Part Two
31 for (int i = st_idx; i < end_idx; i++)
32 {
33     // update body i's position and velocity using the saved changes in
Part One
34     pool->get_body(i).update_by_delta_var();
35     pool->get_body(i).update_for_tick(elapse, space, radius);
36 }
37 }

```

Here, I use Unified Memory to simplify the memory copy calls. According to the CUDA documentation, the underlying system manages data access and locality within a CUDA program without the need for explicit memory copy calls. This benefits GPU programming in two primary ways:

- GPU programming is simplified by unifying memory spaces coherently across all GPUs and CPUs in the system and by providing tighter and more straightforward language integration for CUDA programmers.
- Data access speed is maximized by transparently migrating data towards the processor using it.

I overload the operators `new` and `delete` to provide better encapsulation.

```

1  class Managed
2  {
3  public:
4      __host__ void *operator new(size_t len)
5      {
6          void *ptr;
7          cudaMallocManaged(&ptr, len);
8          cudaDeviceSynchronize();
9          return ptr;
10     }
11
12     __host__ void operator delete(void *ptr)
13     {
14         cudaDeviceSynchronize();
15         cudaFree(ptr);
16     }
17 };
18
19 class BodyPool : public Managed{}
20 int main()
21 {
22     pool = new BodyPool(static_cast<size_t>(bodies), space, max_mass);
23     // call kernel
24     delete pool;
25 }

```


Hybrid Implementation

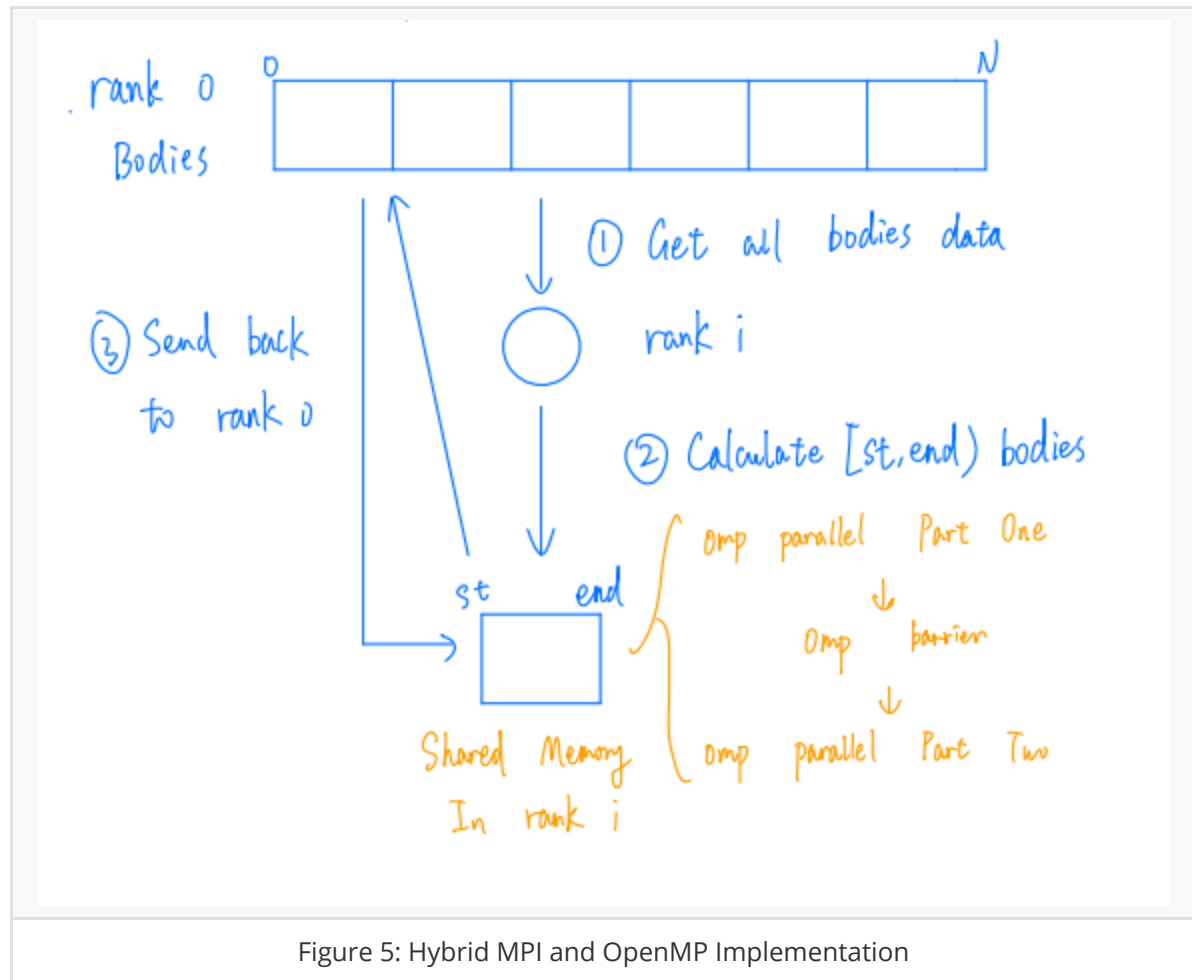


Figure 5: Hybrid MPI and OpenMP Implementation

```

1 void worker(int rank, int world_size)
2 {
3     // Preparation: Broadcast all bodies to all processes
4     // ...
5     // Part One
6     #pragma omp parallel for shared(pool)
7     for (size_t i = st_idx; i < end_idx; i++)
8     {
9         for (size_t j = 0; j < pool.size(); j++)
10        {
11            if (i == j)
12                continue;
13            // update body i's position and velocity immediately
14            pool.mpi_check_and_update(pool.get_body(i), pool.get_body(j),
15            local_radius, local_gravity);
16        }
17    }
18    #pragma omp parallel for shared(pool)
19    // Step 2
20    for (size_t i = st_idx; i < end_idx; i++)
21    {
22        pool.get_body(i).update_for_tick(local_elapse, local_space,
23        local_radius);
24    }
25    // Gather bodies data to the root process
26    // ...

```

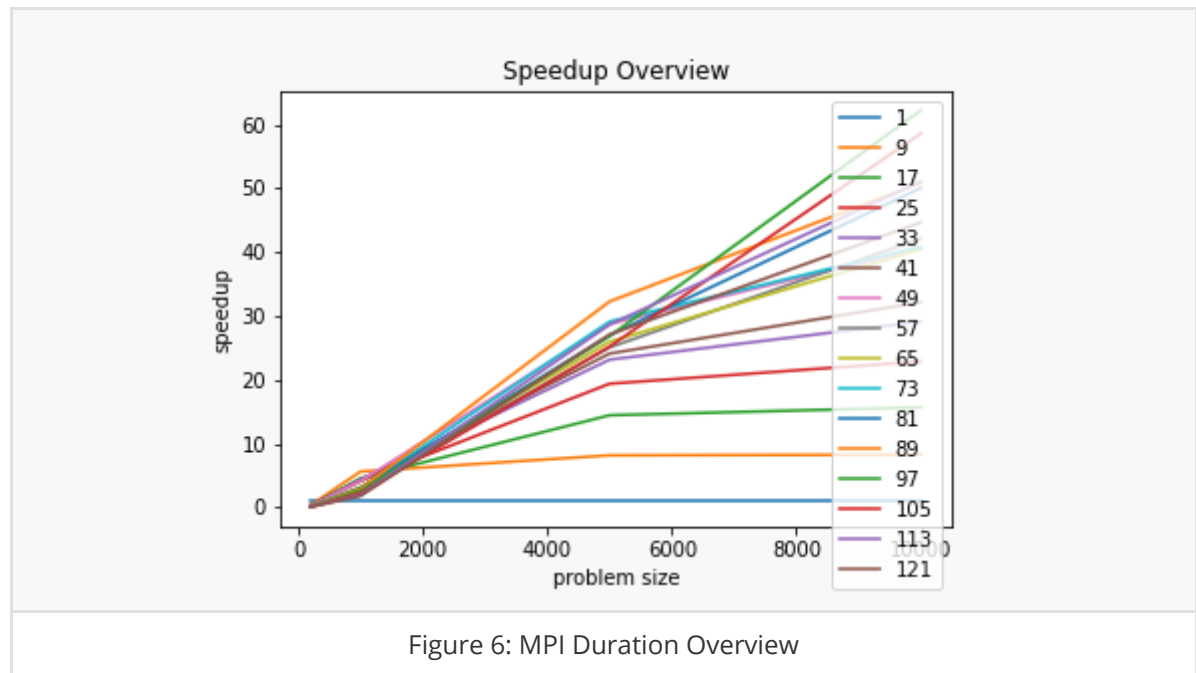
Results

Four different problem sizes ranging from 200, 1000, 5000, and 10000 are tested.

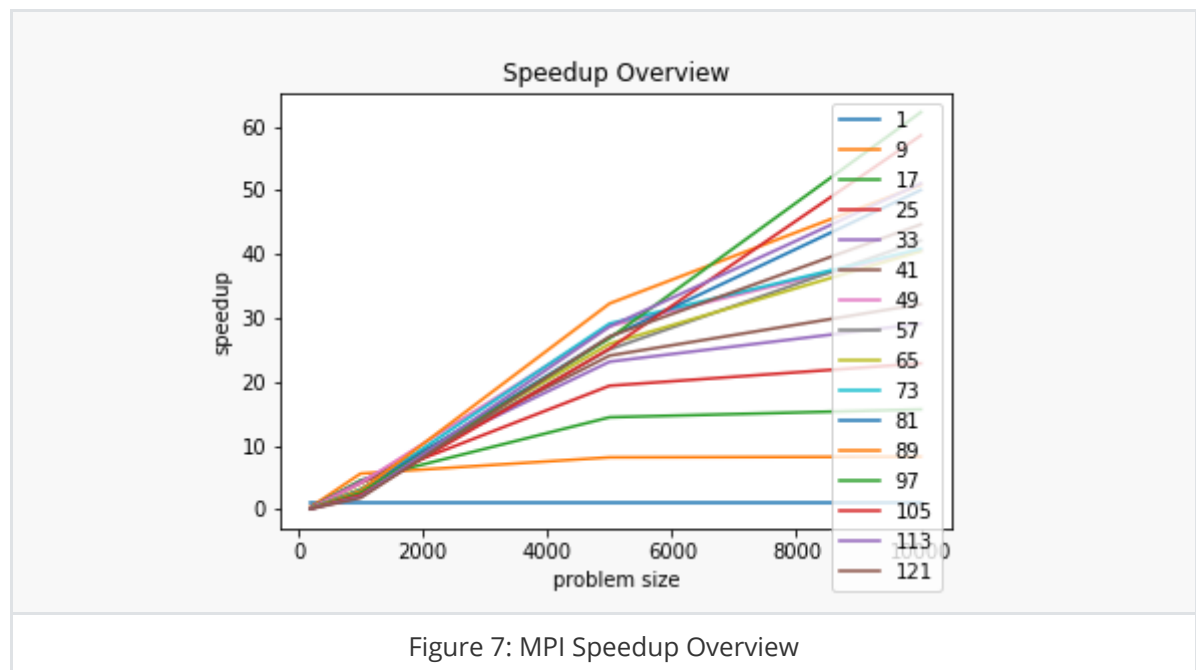
The performance of the program is analyzed from the following three dimensions:

- Duration in nanoseconds(ns) per time step
- Speedup $Speedup_n = \frac{T_1}{T_n}$ where T_1 is the execution time on one process and T_n is the execution time on n processes.

MPI



In Figure 6, the large gap between the blue curve and the other curves shows that the sequential version is much slower than the MPI version. As problem size increases, the gap becomes even larger.



In Figure 7, for those process numbers greater than 1, as problem size increases, the speedup also increases. The larger the process number, the faster the growth (bigger slope).

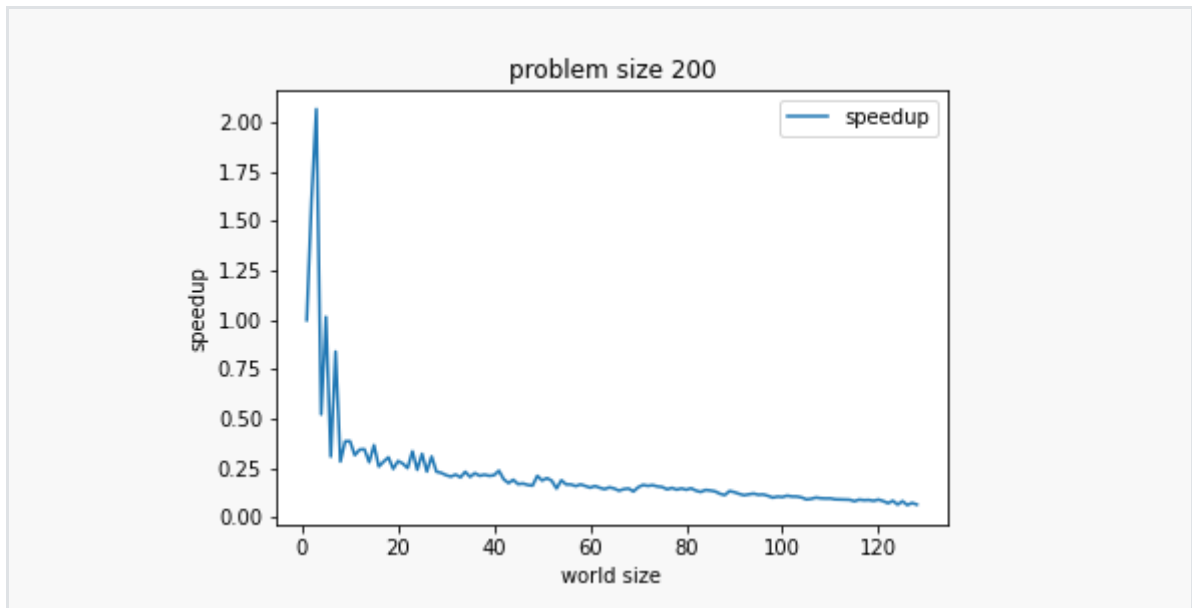


Figure 8: MPI Speedup with Problem Size 200

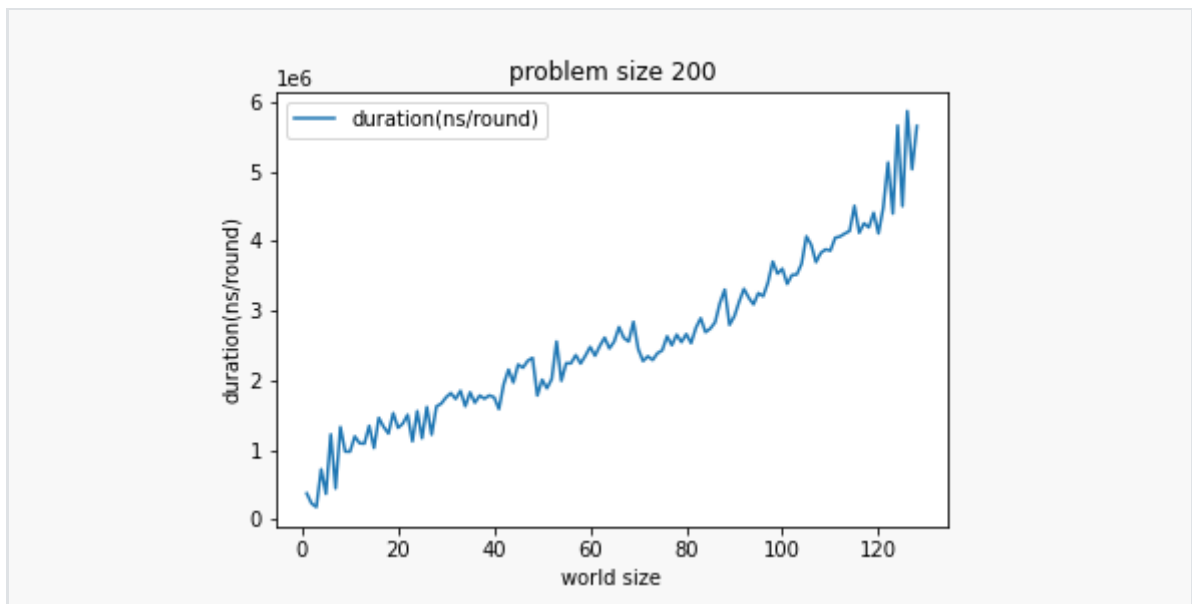


Figure 9: MPI Duration with Problem Size 200

In Figure 8 and Figure 9, as problem size increases, the speedup decreases, and the duration increases. The possible reason is that the computation time of each process is short, and the inter-process communication time becomes the determining factor. Since the duration is so small (less than $6\mu s$), the startup time and the inter-process communication time are relatively large compared to the computation time.

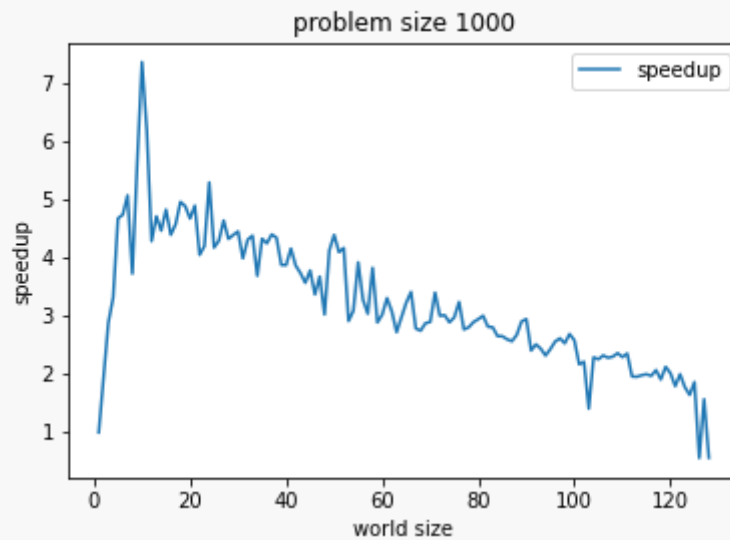


Figure 10: MPI Speedup with Problem Size 1000

In Figure 10, initially, the speedup first increases with the process number. When the process number reaches about 10, the speedup starts to fluctuate up and down in a slow downward trend. I think the reason is the same as above.

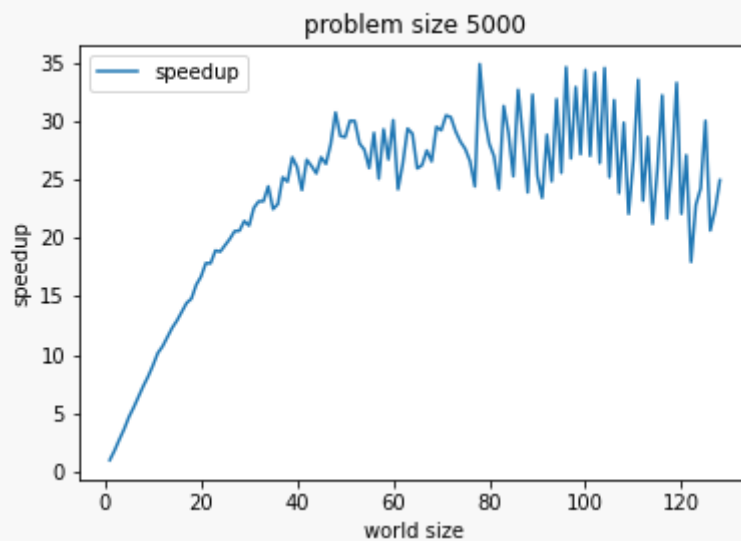
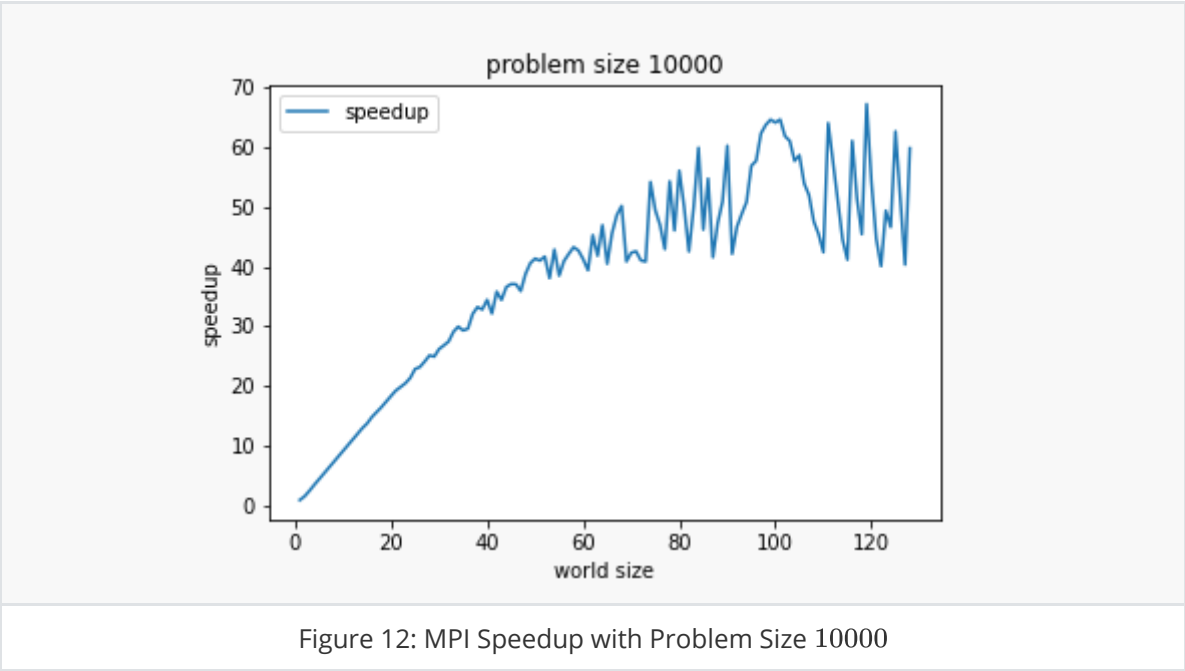




Figure 11: MPI Speedup with Problem Size 5000





In Figure 11 and Figure 12, the speedup keeps increasing with the process number. Compared to Figure 11, Figure 12 has a more significant speedup under the same process number. That's because the computation work is more extensive.

Pthread


For Pthread, the thread number increases from 1 to 32, each time by 1, since Pthread requires shared memory and on a single node, there are only 32 physical threads available.

 Figure 10	 Figure 11
Figure 10: Pthread Duration Overview	Figure 11: MPI Duration Overview Comparison

In Figure 10, the large gap between the blue curve and the orange curve shows that the sequential version is much more slower than the Pthread version. As problem size increases, the gap becomes even larger.
 Compared to MPI version, pthread version shares similar results.

 Figure 12	 Figure 13
Figure 12: Pthread Speedup Overview	Figure 13: MPI Speedup Overview Comparison

In Figure 12, for those thread number greater than 1, as problem size increases, the speedup also increases. The larger the thread number, the faster the growth (bigger slope).
 Compared to MPI version, the slope of the curve and the speedup is much smaller.

 Figure 14	 Figure 15
Figure 14: Pthread Speed Overview	Figure 15: MPI Speed Overview Comparison

In Figure 14, it can be noticed that greater thread number brings faster speed. As the problem size increases, the speed decreases for all thread numbers.
 Compared to the MPI version, the speed is much smaller.

CUDA

For CUDA, it is tested on GTX2080Ti with maximum 1024 threads per block.

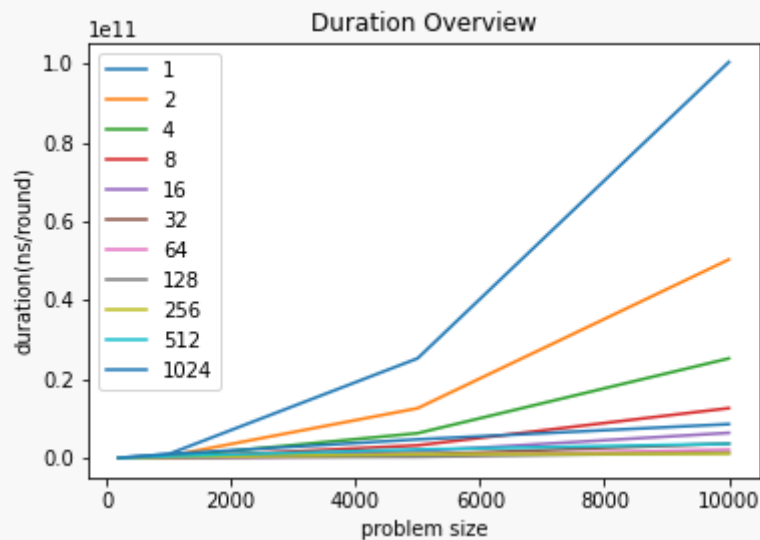


Figure 6: CUDA Duration Overview

In Figure 6, the large gap between the blue curve and the other curves shows that the sequential version is much slower than the CUDA version. As problem size increases, the gap becomes even larger.

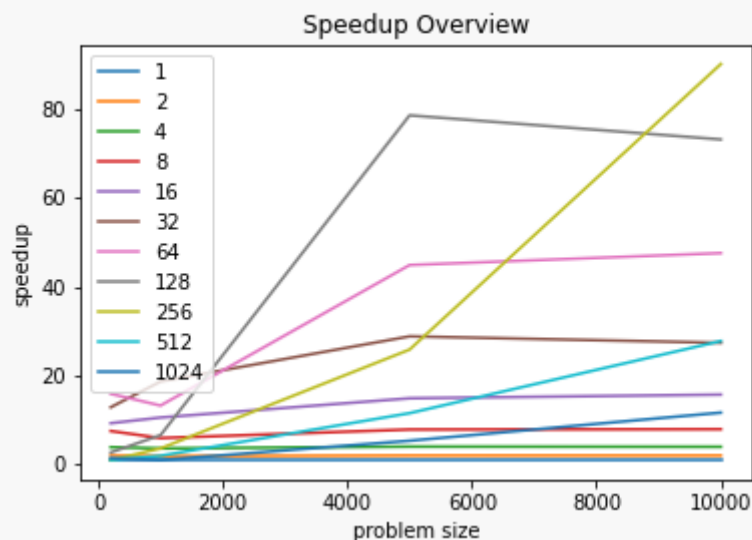


Figure 7: CUDA Speedup Overview

In Figure 7, it can be noticed that the speedup of some thread numbers first increases with problem size then decreases. When the thread number is 256, the speedup always increases with the problem size.

It is interesting to point out that when the thread number is 1024, which is the maximum thread number per block for this NVIDIA GPU, the speedup does not perform well. The possible reason is that CUDA uses a unique architecture called SIMT (Single-Instruction, Multiple-Thread).

- According to the CUDA documentation, The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute

independently. The term warp originates from weaving, the first parallel thread technology. A half-warp is either the first or second half of a warp. A quarter-warp is either the first, second, third, or fourth quarter of a warp.

- I think this can explain the result that when the problem size is large, thread numbers 256, 128, 64, and 32 perform better than other thread numbers.

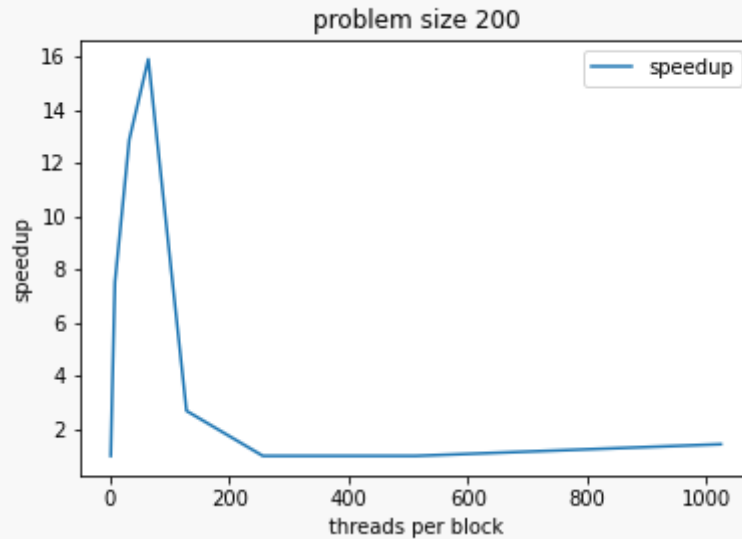


Figure 8: CUDA Speedup with Problem Size 200

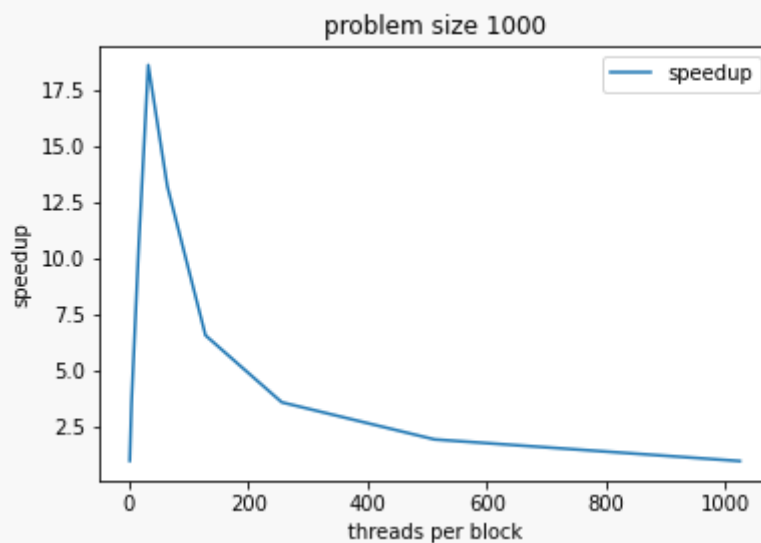


Figure 9: CUDA Speedup with Problem Size 1000

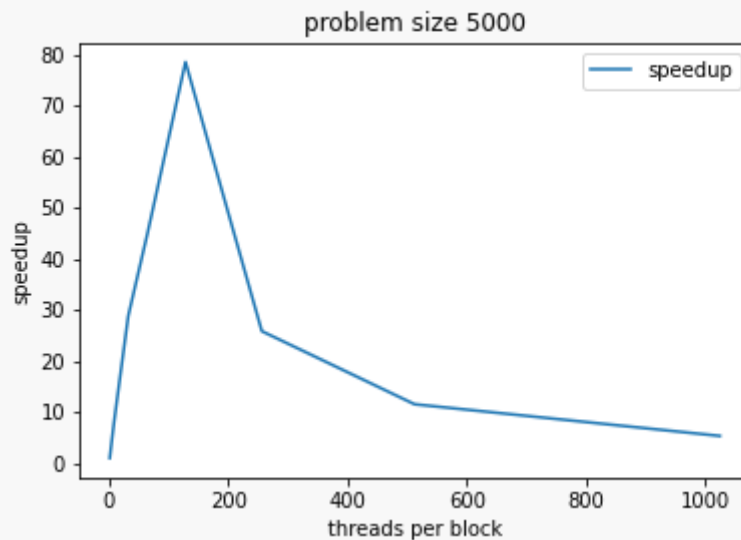


Figure 9: CUDA Speedup with Problem Size 5000

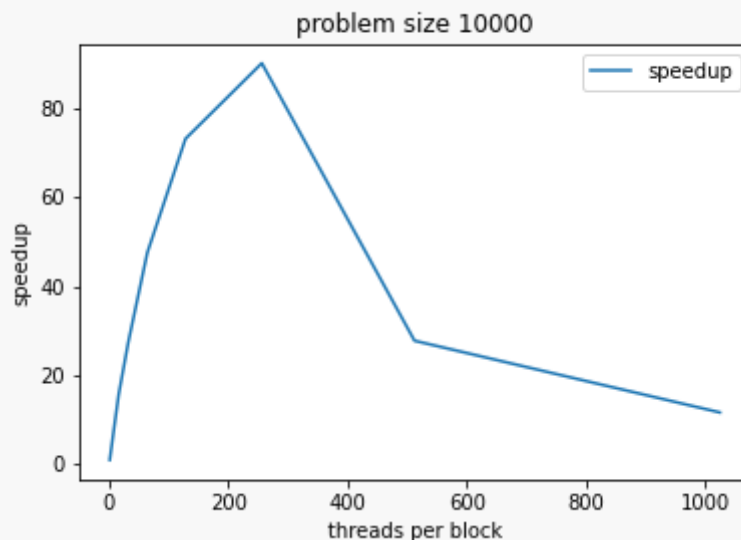


Figure 9: CUDA Speedup with Problem Size 10000

In the above figures, as problem size increases, the speedup first increases, then decreases. The possible reason is that the computation time of each thread is short, and the thread creation time becomes the determining factor. This can be proved by the fact that when the thread number is fixed, as the problem size increases, the speedup get larger.

Conclusion

After working out this assignment, I have learned to write OpenMP, CUDA, and hybrid MPI and OpenMP programs. Different implementations of parallel programming bring different programming experience. It enables me to think deeply about the pros and cons of each implementation. OpenMP is definitely the most easiest one to write. Pthread and CUDA are two shared memory models using CPU and GPU respectively. MPI presents the highest performance.

References

- https://openmp.org/wp-content/uploads/HybridPP_Slides.pdf
- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

- <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture>