|epcc|

Parallel Design Patterns Coursework

B177252

April 9, 2021

1 Introduction

This report describes the details of design and implementation of the parallel shipping model using geometric decomposition pattern and explores the performance and scaling of the code. A reusable framework is used and is successfully separated from the program specific code which uses the framework to simulate the model. All experiments of performance and scalabilty test are discussed in detail.

2 Implementation

2.1 Use of geometric decomposition pattern

In this problem, we use geometric decomposition pattern on two parts. The first one is simulation functionality and the second part is route planner functionality.

We start with route planner, which is easier to implement. First, we need to find the data to split up. Obviously, the data is the $*route$ pointer in the structure datatype $specific\_route$. In the serial program, $*route$ is allocated a memory of the configuration size, which is $size\_x \times size\_y$ in the function $generate\_route$. We use 1D decomposition and make the size of $*route$ to be $(local\_nx + 2) \times (size\_y + 2)$, in which $local\_nx$ is evenly distributed across all processes and extra 2 are allocated for halos.(If it can not be evenly distributed, the extra size would be allocated one by one to the first several processes). We need to perform halo swap of the boundary values in the function $calculate\_routes$ after getting the $route\_index$ in order for the convenience of the function $getNextCell$, which determines the next cells to move to for ships. The details of $getNextcell$ will be discussed later.

Then we parallelise the simulation part. Same as the first part, we first need to find the data. The data is $cell\_struct$. We use the same decomposition strategy to decompose $cell\_struct$. The part to parallelise the simulation is in the function $updateMovement$, in which the ships in the boundary cells might move to the neighboring UEs by invoking the function $getNextCell$. Therefore, we need to find a way to transfer the ships in the boundary cells to the neighboring UEs. However, there is another situation where no ship needs be transferred between UEs. To solve this problem, we use two buffers to save the ships to be sent. One is sent to the previous neighboring UE and another is sent to the next neighboring UE. We maintain a parameter to record the number of ships to be sent for each iteration. If the number is zero, it will only send a number zero and will not send any ships. Otherwise it will send both the number of ships and the ships buffer. Moreover, we also need to send the locations of cells for the receivers to enable the receivers to update the ships. Please note that the sends are all non-blocking using MPI_Isend. As for the receivers, we use two buffers to receive the message. Before receiving, we first need to receive the message recording the number of ships. If the number is zero, there is no need to receive anything. Otherwise, the two buffers would

be used to receive the ships. We also need to allocate another buffer to receive the location to update the ships in the sub_domain. We use blocking receive MPI_Recv to receive the messages.

2.2 Optimization of getNextCell

In the serial code, the function $getNextCell$ traverses each grid of the route map to find the next cell. This function can be optimized. According to the algorithm for finding the optimal route, the only possible next cell which the ship will move to is one of the eight cells surrounding the current cell. Therefore, there is no need to traverse all the grids of the route map. Instead, we only need to traverse the surrounding cells. In the previous section, we mentioned that we perform halo swap of boundary values for the convenience of $getNextCell$. The reason is that for the boundary cells for an UE, it can be easier to get the boundary cells of the neighboring UEs in order to calculate the next cell to move to after swapping halos.

2.3 Reusable framework

Based on the 1D geometric decomposition of the serial program, we expect to extend the program into a more reusable framework. The philosophy is to separate the problem specific code and the framework which implements the geometric decomposition without problem specific code. As we discussed before, we use geometric decomposition on both simulation and route planner. We design a framework for each of these two parts.

Language C enables us to define a variable pointing to the address of a function, which is called function pointer. For both parts, we use function pointers as the foundation to build our frameworks.

As for the route planner, the major function is $calculate\_routes$. This function traverses all the permutation of ports and uses two ports as the starting and target ports to generate the corresponding route each time. We perform halo swap by invoking the function $perform\_halo\_swap$ each time when there is a route index. The function $generate\_route$ is the core function for calculating the route map. This function is a problem specific function which can be used as the function pointer parameter in $calculate\_routes$. We also encapsulate all the other problem specific codes related to route planning as the function $run\_route\_planner$. The details of code is listed below.

```
1 #if ROUTE_PLANNER_TO_USE == 0
2   run_route_planner(simulation_configuration, local_nx, myrank, size,
      basex, generate_route);
3 #endif
4 static void run_route_planner(struct simulation_configuration_struct
    simulation_configuration, int local_nx, int myrank, int size, int
    basex, int (* generate_route_strategy)(int,int,int,int))
```

```
{
  initialise_routemap(&simulation_configuration, local_nx, myrank,
    size, basex);
  initialiseSimulationSupport();
  MPI_Barrier(MPI_COMM_WORLD);
  double time1 = MPI_Wtime();
  calculate_routes(&simulation_configuration,generate_route_strategy)
    ;
  MPI_Barrier(MPI_COMM_WORLD);
  double time2 = MPI_Wtime();
  if (myrank == 0)
  {
    printf("The time of route planning is %g\n", time2 - time1);
  }
}
void calculate_routes(struct simulation_configuration_struct *
    simulation_configuration,int (* generate_route_strategy)(int,int,
    int,int))
{
  for (int i = 0; i < simulation_configuration->number_ports; i++)
  {
    for (int j = 0; j < simulation_configuration->number_ports; j++)
    {
      if (i != j)
      {
        int route_index = generate_route_strategy(
    simulation_configuration->ports[i].x, simulation_configuration->
    ports[i].y,                          simulation_configuration->ports
    [j].x, simulation_configuration->ports[j].y);
        if (route_index == -1)
        {
          fprintf(stderr, "Error, can not plan a route between points
     X=%d,Y=%d and X=%d,Y=%d\n",
                  simulation_configuration->ports[i].x,
    simulation_configuration->ports[i].y,
                  simulation_configuration->ports[j].x,
    simulation_configuration->ports[j].y);
        }
        else
        {
          perform_halo_swap(myrank, size, local_nx, size_y,
    mem_size_y, routes[route_index].route);
          simulation_configuration->ports[i].target_route_indexes[j]
    = route_index;
        }
      }
    }
  }
}
```

Listing 1: framework for route planner

As for the simulation part, the idea is the same as before. We encapsulate $initialiseDomain$, $updateProperties$, $getNextCell$ and $findFreeShipIndex$ as the function pointers

because these four functions are all problem specific codes. $updateMovement$ is the core function where we simulate the shipping model and use lots of MPI sends and receives. This function contains one problem specific function which is $getNextCell$, which is used as the function pointer parameter. The details of framework are shown below.

```
#if SIMULATION_TO_USE == 0
  run_simulation(&simulation_configuration, init_simulation,
    initialiseDomain, updateProperties, getNextCell, findFreeShipIndex
    , finalise_simulation);
#endif

static void run_simulation(struct simulation_configuration_struct *
    simulation_configuration, void (*init_simulation)(int, int), void
    (*initialise_domain_strategy)(struct
    simulation_configuration_struct *), void (*
    update_properties_strategy)(struct simulation_configuration_struct
     *),
void(*get_next_cell_strategy)(int,int,int,int*,int*),int(*
    find_fresh_index_strategy)(struct cell_struct*), void (*
    finalise_simulation)())
{
  int mem_size_x = local_nx + 2;
  int mem_size_y = ny + 2;
  init_simulation(mem_size_x, mem_size_y);
  MPI_Barrier(MPI_COMM_WORLD);
  double time1 = MPI_Wtime();
  initialise_domain_strategy(simulation_configuration);
  int hours = 0;
  for (int i = 0; i < simulation_configuration->number_timesteps; i
    ++)
  {
    update_properties_strategy(simulation_configuration);
    updateMovement(simulation_configuration,get_next_cell_strategy,
    find_fresh_index_strategy);
    if (i % simulation_configuration->reportStatsEvery == 0)
      reportGeneralStatistics(simulation_configuration, hours);
    hours += simulation_configuration->dt; // Update the simulation
    hours by dt which is the number of hours per timestep
  }
  MPI_Barrier(MPI_COMM_WORLD);
  double time2 = MPI_Wtime();
  if (myrank == 0)
  {
    printf("The time of simulation is %g\n", time2 - time1);
  }
  reportFinalInformation(simulation_configuration);
  finalise_simulation();
}
```

Listing 2: framework for simulation

3 Performance and scalability test

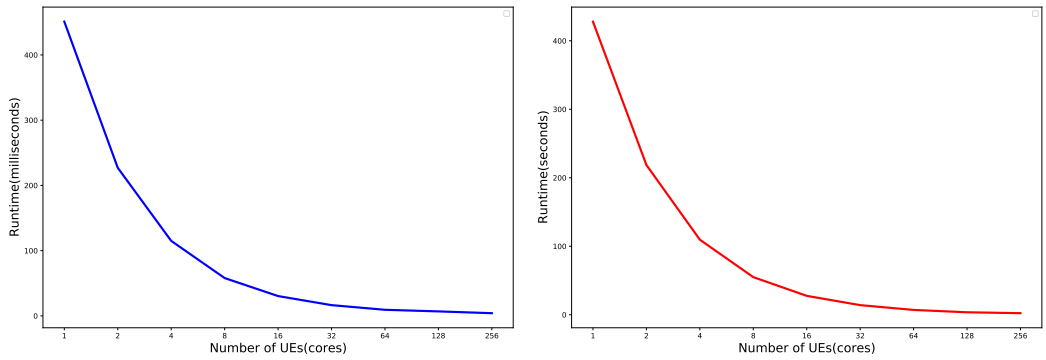
In this section, we perform experiments on the performance and scalability test of our parallel program. Experiments are performed on Cirrus using compute nodes. Performance test includes test of runtime, speedup and parallel efficiency. Scalability test includes strong scaling and weak scaling.

3.1 Performance test

In performance test, we set the global problem size to be 1024×1024. The reason for this setting is that we need to test the time of serial code, which means too large problem size would take enormous running time. However, too small problem size comes with a high inaccuracy. We test the parallel version using 1, 2, 4, 8, 16, 32, 64, 128, 256 processes. We record the runtime for both route planner and simualation parts. The results are shown in table 1. p refers to the number of processes. Time1 and Time2 refer to the runtime of route planner and simulation respectively. The plots of runtime versus number of processes, speedup versus number of processes and parallel efficiency versus number of processes are shown in figure 1, 2 and 3 respectively.

Execution time versus number of processes									
p	1	2	4	8	16	32	64	128	256
Time1(ms)	451.132	227.230	115.210	58.732	30.482	17.127	9.280	7.101	4.012
Time2(s)	427.914	218.673	109.610	54.941	27.594	14.047	7.037	3.616	2.281

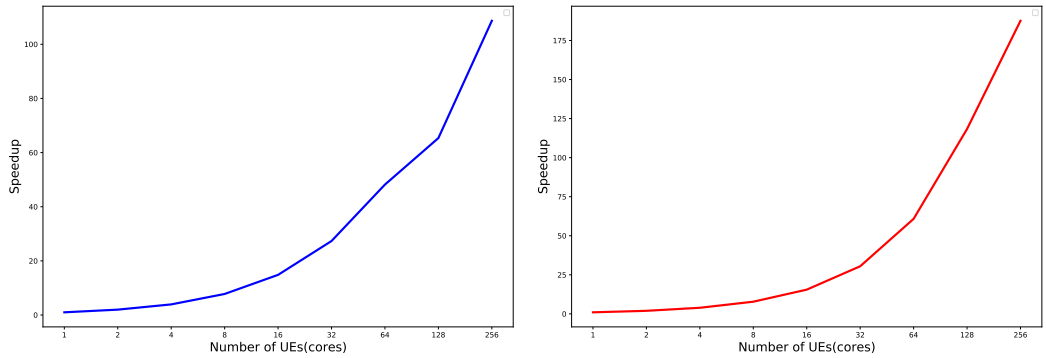
Table 1: Execution time versus number of processes



(a) Execution time(ms) of route planner versus number of processes

(b) Execution time(s) of simulation versus number of processes

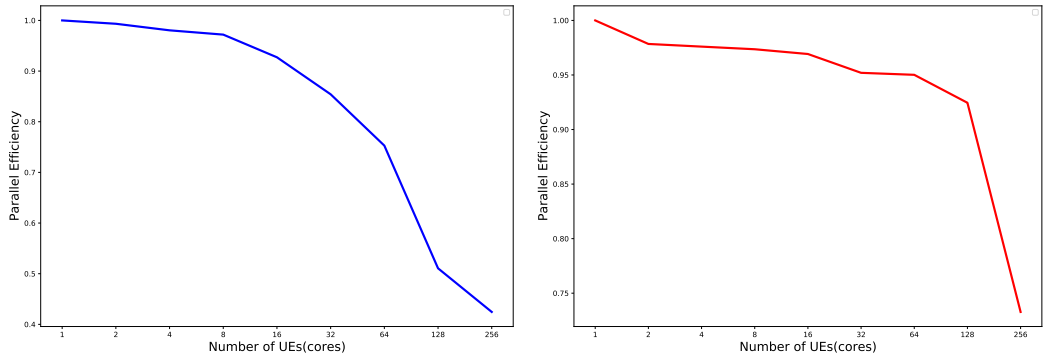
Figure 1: Execution time versus number of processes



(a) Speedup of route planner versus number of processes

(b) Speedup of simulation versus number of processes

Figure 2: Speedup versus number of processes



(a) Parallel efficiency of route planner versus number of processes

(b) Parallel efficiency of simulation versus number of processes

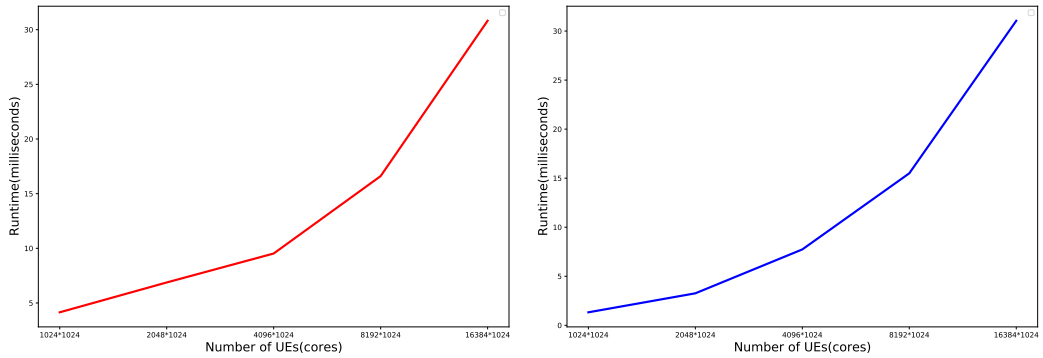
Figure 3: Parallel efficiency versus number of processes

3.2 Runtime versus problem size

We are also interested in how runtime varies with problem size. We set the number of processes to be 256 and vary the problem size to be 1024×1024, 2048×1024, 4096×1024, 8192×1024 and 16384×1024 respectively. The results are shown in table 2 and figure 4.

Execution time versus problem size					
size	1024×1024	2048×1024	4096×1024	8192×1024	16384×1024
Time1(ms)	4.147	6.877	9.525	16.605	30.812
Time2(s)	1.325	3.262	7.737	15.497	31.047

Table 2: Execution time versus problem size



(a) Execution time(ms) of route planner versus problem size

(b) Execution time(s) of simulation versus problem size

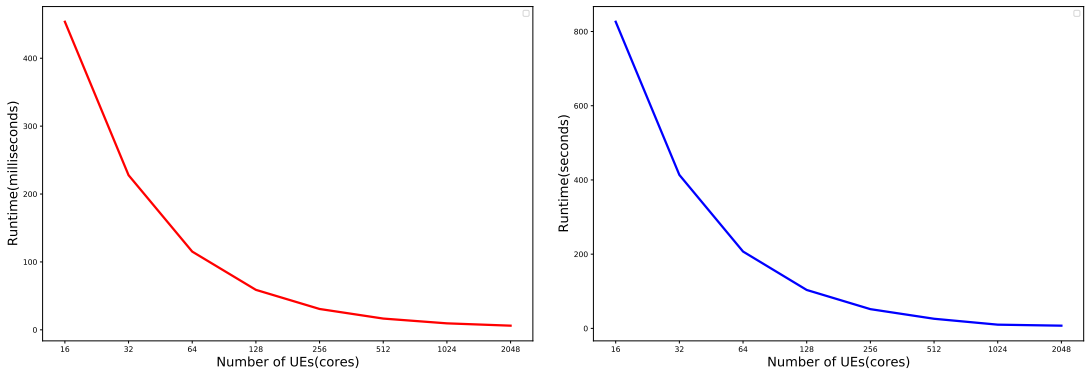
Figure 4: Execution time versus problem size

3.3 Scalability test

Scalability test includes strong and weak scaling test. In strong scaling test, the global problem size should be fixed. We set the global problem size to be 16384×1024, which is very large. The reason for this large size setting is that we expect to see how the scalability of our program is using a large number of processes such as 1024 even 2048 processes. We experiment on 16, 32, 64, 128, 256, 512, 1024, 2048 processes. Same as the performance test, we record the runtime for both route planner and simulation parts. The results are shown in table 3. Time1 and Time2 refer to the runtime of route planner and simulation respectively. The plots of strong scaling are shown in figure 5. In weak scaling test, the local problem size is fixed. We set the local problem size to be 32768 and the global problem size to vary with the number of processes. Results are shown in table 4. The plots of weak scaling are shown in figure 6.

Strong scaling test		
Number of processes	Time1(ms)	Time2(s)
16	453.742	826.29
32	227.77	413.578
64	115.218	207.099
128	58.872	103.550
256	30.703	51.831
512	16.574	25.920
1024	9.535	13.024
2048	6.119	7.3082

Table 3: Strong scaling test



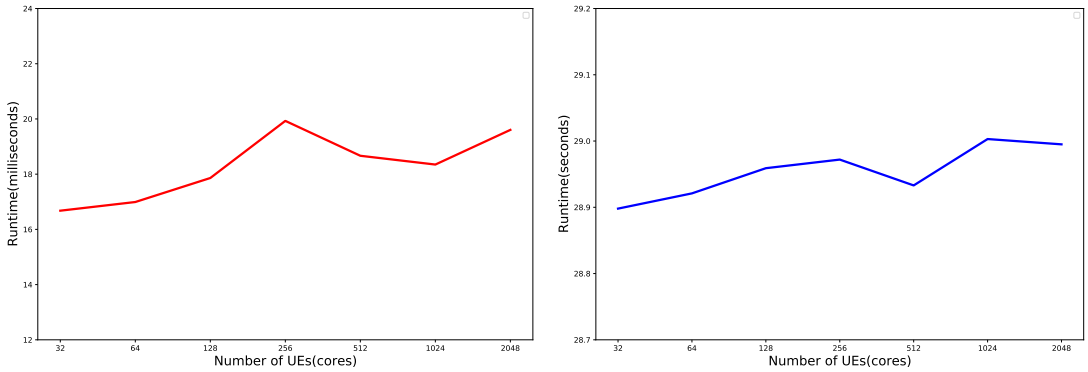
(a) Execution time(ms) of route planner versus number of processes

(b) Execution time(s) of simulation versus number of processes

Figure 5: Strong scaling

Weak scaling test				
Number of processes	SizeX	SizeY	Time1(ms)	Time2(s)
32	1024	1024	16.677	28.898
64	2048	1024	16.991	28.922
128	2048	2048	17.861	28.959
256	4096	2048	19.930	28.972
512	4096	4096	18.666	28.933
1024	8192	4096	18.348	29.002
2048	8192	8192	19.604	28.995

Table 4: Weak scaling test



(a) Execution time(ms) of route planner versus number of processes

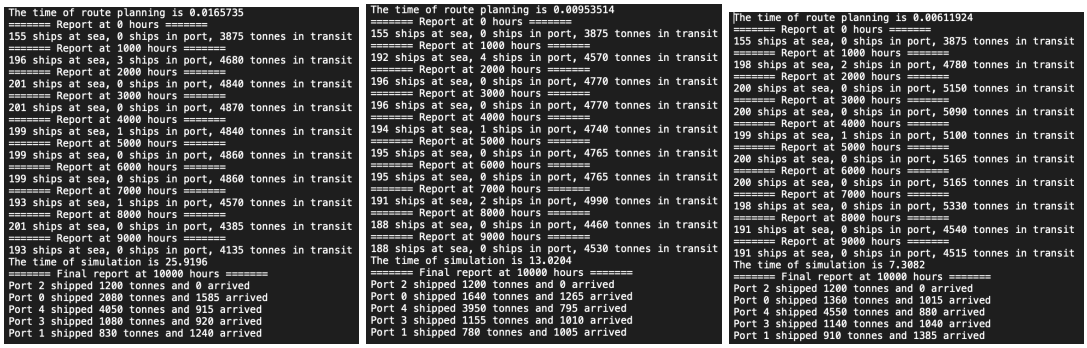
(b) Execution time(s) of simulation versus number of processes

Figure 6: Weak scaling

3.4 Example outputs

In this section, we show several examples of output results of simulation. We use the experiment in strong scaling test. The number of processes is 512, 1024, 2048 respec-

tively. The results are shown in figure 7.



(a) Output(number of processes is 512) (b) Output(number of processes is 1024) (c) Output(number of processes is 2048)

Figure 7: Weak scaling

4 Discussion and Analysis

4.1 Discussion of performance and scalability test

In this section, we discuss and analyze in depth about the performance test and scalability test.

In the performance test, we can see that the speedup increases with the number of processes from figure 2. We can use Amdahl's law to quantify this relationship. Amdahl's law is presented as

$$T_p = \alpha T_s + \frac{(1-\alpha)T_s}{P}$$

T_p, T_s, α and P represents the execution time of parallel code, execution time of serial code, proportion of serial part and number of processes respectively. Speedup can be represented as

$$S_p = \frac{T_s}{T_p} = \frac{P}{\alpha P + (1-\alpha)}$$

In this problem, we mainly focus on the parallelised part. Therefore, the time for serial part is negligible compared to the time for parallel part, which means that α is close to zero. As a result, the speedup would be close to P theoretically. We can check from our experiment that the speedup is actually close to the number of processes used, which matches our assumption. As for the parallel efficiency, it can be represented as

$$E_p = \frac{S_p}{P} = \frac{T_s}{PT_p} = \frac{1}{\alpha P + (1-\alpha)}$$

The range of α is between 0 and 1. Suppose α is fixed, theoretically, parallel efficiency would decrease as number of processes increases. The range of parallel efficiency would be beteween $\frac{1}{P}$ and 1 depending on α. We can see from our experimental

result in figure 3 that parallel efficiency decreases with number of processes, which proves our assumption.

In the scalability test, as for strong scaling, the result also shows that runtime decreases with number of processes. The relationship can be represented by Amdahl's law. If α is close to zero, T_p is close to $\frac{T_s}{P}$, which is proved by our experiments. As for weak scaling, we can see that the runtime remains almost unchanged. To explain this, we first need to see how problem size affects the runtime. From figure 4, we can see that runtime increases proportionally with problem size. This is easy to understand because the problem size determines the runtime. To better illustrate this, for the serial code, we suppose L_0 to be the initial problem size, L to be the current problem size and T_0 to be the runtime of serial code when problem size is L_0. Current runtime of serial code T_s can be represented as

$$T_s = \frac{L}{L_0} T_0$$

Because the local problem size is fixed in weak scaling, we define

$$\frac{L}{P} = \beta$$

, in which β is a constant number. Therefore, if we apply these equations to the Amdahl's law, we can get

$$T_p = \alpha \frac{L}{L_0} T_0 + \frac{(1-\alpha)\frac{L}{L_0} T_0}{P} = \alpha \frac{\beta P}{L_0} T_0 + \frac{(1-\alpha)\frac{\beta P}{L_0} T_0}{P} = \alpha \frac{\beta P}{L_0} T_0 + \frac{(1-\alpha)\beta T_0}{L_0}$$

We can find that $\frac{(1-\alpha)\beta T_0}{L_0}$ is a constant number and the part that enables T_p to change is $\alpha \frac{\beta P}{L_0} T_0$. However, in our problem, α is close to zero. Therefore, T_p is close to $\frac{\beta T_0}{L_0}$, which equals $\frac{\beta T_s}{L}$. This can explain our experimental results in figure 6.

4.2 Guarantee of correctness

As the program uses random number to simulate the process, it's difficult to prove the correctness. Our method is to display the status of route map and ships for each step of simulation using the given functions. The thing that we focus includes whether the transportation of boundary ships is successful and whether route planning is correct after parallelism. But most importantly, the key thing to guarantee the correctness is to use proper MPI sends and receives. We use $MPI\_Waitall$ at the end of $updateMovement$ in order to guarantee all the messages are successfully received. Same method is used in route planner.

5 Conclusions

To conclude, we successfully apply geometric decomposition pattern to the shipping model for both route planner and simulation parts. We use framework which is sepa-

rated from problem specific code. We perform performance test and scalability test. In performance test, we test the runtime, speedup and parallel efficiency. We also explore how runtime varies with problem size. Scalabilty test includes strong scaling and weak scaling. We test both of them. Finally, we use Amdahl's law to discuss and analyze the experimental results in depth and dicuss the guarantee of success.

However, more improvement can be expected in future work. For example, we can try 2D decomposition due to its flexibility.