CSCI 6360 Parallel Programming and Computing

Final Group Project

Prepared by: Enzhe Lu, Maida Wu, Tao Chen, Guanghan Cai

Table of Content

Abstract

Today's top high-performance parallel computing is used among different industries, including biology, finance, physics, etc. Recently the world-level flue COVID-19 is widely spreading. This project considers to build a COVID-19 infection model and predict the spreading result via parallel computing. The target is comparing a pre-determined performance benchmark with CPU/GPU and MPI hybrid model to see the performance difference on the AiMOS supercomputer system. Our project uses MPI I/O to communicate with the files outside the program. We show the performance and corresponding analysis in the latter part of this report.

1. Introduction

1.1 Purpose and Goal

The project is required to design, deploy, develop, and then experiment with a massive parallel system. Our group applies the CPU with the MPI model, Hybrid CPU/GPU with MPI and CUDA model to compare the performance and make the analysis. The end goal is to make progress towards a parallel computing performance design and analysis that is of high enough quality to be published.

1.2 Project Scope

COVID-19, a world-level coronavirus disease, is now spreading at a staggering rate. Billions of data of COVID-19 is available to predict or measure the coronavirus disease. Using high-performance parallel computing to deal with the substantial volume data in a supercomputer is the topic of the project. Our group plans to create a new infection model as the theme, applying the CPU, GPU, and MPI technologies to test the performance of paralleling computing in different circumstances, including various MPI ranking and GPU nodes. Set benchmark to make comparisons of performance based on running time and cycle count. Analyze the algorithms and reasons behind the different performances.

1.3 Project Assumptions and Constraints

Some several assumptions and constraints are of importance for the project and our team members.

1.3.1 Assumptions

- Our team is expected to gain enough knowledge about parallel computing algorithms and the practical skills behind it.
- Our team members work together to complete the project, even though in exceptional condition, with working remotely.
- Our team can move at a quick pace through encounters problems.
- Team members do best to keep a creative and corporative atmosphere.
- The running environment will be the AiMOS supercomputer at RPI.

1.4.1 Constraints

- The human resources are limited as four.
- Due to the nature of the project and its dependability on already existing solutions and technologies, third party software and already available solutions will be used in the project as needed.

1.4 Deliverables

The high-performance parallel computing model includes design patterns, deployment, development, and experiment. The high completed research report includes the model mentioned above, corresponding performance graphs and records, analysis, and algorithms behind it.

1.5 Schedule

Project Milestone	Project Artifact	Due Date
Project Start	Determine team members	04/10/2020
Phase 0: Assignment of Tasks	Assignment of tasks	04/11/2020
Phase 1: Draft of Project Report	Project report creation	04/13/2020
Phase 2: Spreading Model Implementation	Build basic spreading model with C and CUDA	04/16/2020
Phase 3: MPI I/O Implement	Build MPI I/O system with C	04/22/2020
Phase 4: Combination Code	Combine code in phase 2 and 3	04/23/2020
Phase 5: Model Experiments	Test models via changing of parameters and record result	04/25/2020
Phase 6: Performance analysis	Analyze based on the result and draw comparison graphs	04/28/2020
Phase 7: Report Completion	Fill conclusions into report and make cosmetic edition to finish	05/03/2020

Table 1. Project Schedule

1.6 Tasks Assignment

Assignment	Team Members
Basic Spreading Model	Enzhe Lu
MPI I/O Communication Model	Tao Chen, Maida Wu
Report and Analytic Graphs	Guanghan Cai

Table 2. Task Assignment

2. Model and Algorithm Design

2.1 Spreading Model

Technically, it's a fresh model with the specific spreading rule we created for the COVID-19. We'll show the model by two parts: basic introduction and the spreading rule.

2.1.1 Model Introduction

We create several 2-D matrices as different states. Every matric represents a single state. The cells inside the state are the residents. For a person at a cell, it has 4 conditions which are represented by an int: value 0 means infected with the COVID-19 infection; value 1 means healthy one; value 2 represents recovered from the COVID-19

infection and values 3 implies died due to COVID-19 infection. For every person in every iteration, we'll scan the eight nearby person conditions and apply the spreading rule to determine the value of the central person. For every state shares the same boundary with another, we'll use ghost lines to mimic this situation so that every state can get the information in the boundary of a nearby one. When passing the states to GPU, we'll transfer 2-D to 1-D and layer them as one long row and use the mathematic rules to make offset the index problem.

2.1.2 Spreading Rule

Initially, the value of a person should be one of four conditions: 0, 1. For every iteration, the system gets the conditions of nearby eight persons. If the value of the person is 0, it will plus 10, which means 10% more aggravation. Every infected neighbor will increase 10 value for the central person. So if a person has eight infection neighbors, his value will add 80 in one iteration. For the person who has the 100 value, there are two possible results: 5% to die and 95% to recover. To mimic the possibility, we use pseudorandom. Create a number belongs to [0,99] if it's less than 5, the person dies, and the value of him will be 3. Otherwise, the value becomes 2, and he gets the immunity for COVID-19, which means he will never get infected again.

2.1.3 Model Assumptions

- Every state has the same population.
- We know the number of states.
- We know the population representation of states is a square (e.g., 32*32).
- The people in ghost columns of the most left state and the most right state are supposed to be infected, whose values are 0.
- For the position doesn't have nearby eight cells, we use value 1 to offset the cell that is missing, which means the person is healthy.
- The distribution of states is horizontal. But due to the *MPI_Isend* can only pass the row not column, we'll use matrix transpose for the state matrix to keep the horizontal rule.

2.2 Parallel Algorithm

We'll apply CUDA, MPI, and MPI I/O. For CUDA, we'll apply memories at GPU by *cudaMallocManaged* function. We'll apply a long row then store the data in every state one by one in order so that there should be a 1-D array in GPU memory. So we can use parallel computation by using multiple blocks and nodes at the same time. For MPI, every state is a rank. To share the same boundary, we use ghost columns at the left and right of the state. So different rank passes the ghost columns to the next rank; people at the next state can scan surrounding people and determine the next stage. The thing is in our assumption, the first (the most left) and the last (the most right) states don't share the boundary, which means the states are not a loop. I think this assumption is realistic; people who live in the east won't infect the people who live in the west if they don't move at the social distance circumstance. For MPI I/O, first, we use *MPI_File_open* to open the files contains state infection information. Then we create a vast shared space to store the states' information and assign space for every rank; in other words, every state. Next, we use *MPI_File_read* to read information from the files. Then create a similar huge shared space to store the result and assign space for every rank as we read. To output the result to file we get after iterations, we call *MPI_File_write* to write them into the external file and finish the whole MPI I/O process. In other words, we'll read from several files including the state matrix and write to one shared file for various ranks as one huge matrix which combines results for all states.

2.3 CPU via MPI

The only file is a c file, and it needs to compile by *mpicc* command. For general CPU via MPI model, we keep the ghost columns to mimic boundaries. For this group, we basically use MPI rand and MPI I/O for this group. First of all, we prepare the txt file with state information outside the program. Then use the *MPI_File* system to read and store information into our prepared huge virtual space. Then extract information from the space, converting it as a 1-D array and store it in the global variant. For every rank, we pass the ghost rows as boundaries for the next rank. So every rank has a passing situation. Not only rank 0 passes everything to other ranks, then passing everything back. We apply the spreading rule mentioned above and do a specific iteration, which is predetermined in the program argument. After we get the result, similarly, we'll store the result into a vast virtual space which is independent with

the previous one. Then output things in space to outside receiving the file. To run this model, we need to apply nodes in the AiMOS supercomputer system to make sure the CPU and MPI ranks can parallelly running for the program.

2.4 Hybrid CPU/GPU via MPI

The files are c and cu files, and we'll compile them by *mpicc* and *nvcc* together. The MPI ranks and I/O ideas are the same as the above group. But after we extract the data from reading vast space, data are passed into GPU memory while converting it to the 1-D array. Then the CUDA part will parallel compute the result via iterations. We also run this group in AiMOS supercomputer system to make sure there are enough GPUs. We're able to test this by changing the amounts of nodes and GPUs.

2.5 Comparison Design

Basically, we have three type parameters to change: amounts of nodes, amount of ranks, and the file size. Technically, we use time and cycle count as our evaluation criteria. There are two groups: CPU via MPI and hybrid CPU/GPU with both MPI and CUDA, changing the parameters and get the performance. For the performance result, firstly we set the benchmark and then experiment both strong scaling performance and weak scaling performance for the hybrid group. In addition, for general parallel performance test of CPU via MPI group, we plan to test several situations:

Parameter	World Size	Iteration	Node
	6	10	1
	32	70	1

Table 3. Parallel Performance Test for CPU&MPI Group

For general parallel performance test of CPU/GPU via MPI group, we have following situations:

Parameter	World Size	Rank	Thread	Iteration	Node
	6	6	16	10	1
	32	32	32	70	1

Table 4. Parallel Performance Test for CPU/GPU&MPI Group

The idea to test the 70 iteration is because we assume finally a person has 95% possibility to recover from the COVID-19. We can see finally how many people can survive from this disease under our model.

For the strong and weak scaling part, we choose world size 256*256, ranks 32, thread 32, iteration 70 as the constant value. The node amounts of strong scaling will vary from 1 to 4 to get the strong scaling test. The world size will vary as 256*256, 512*512, 1024*1024 and 2048*2048.

We also calculate the performance of "cells updates per millisecond" rate. For example, a world of 1024*1024 that runs for 1024 iterations will have 1024^3 cell updates. So the "cells updates per millisecond" is 1024^3 divided by the total execution time in millisecond for that configuration.

2.5.1 Benchmark

Measuring and reporting performance of parallel computing constitutes the basis for the scientific advancement of high-performance computing. Using a benchmark is a technical method to measure the performance in the parallel computation performance evaluation. In addition to distilling best practices from existing work, we propose to use the benchmark to perform a statistically sound analysis of the project. We plan to use two parts as measuring attributes: running time and cycle count. For a running time, we plan to record the MPI I/O read time, MPI I/O write time, and total running time. As mentioned in the part of the parallel algorithms, we use MPI I/O read from several files and write all results to one shared file. The single read and write time are separate due to different algorithms. For a total running time, it reflects not only the MPI rank communication time, I/O time, but also the time of parallel computing in the GPU part. For the cycle count, we use cycle counter available in *POWER9*, which is a family of superscalar, multithreading, symmetric multiprocessors based on the Power ISA. Underlying inline machine code is provided in the project handout. Technically, it can be used to count any running time mentioned above. Actually,

it'll be used to get the parallel computing time in GPU. Our group plans to use the worst performance as our benchmark so that we can observe every progress while we change the value of parameters. Based on this reason, we'll use the CPU via MPI group as our benchmark to compare with other types of hybrid group (different value of parameters in hybrid group). In terms of time, it should be the longest one. But to obey the rule that "better is larger" for performing clear graphs, we plan to represent this by the amounts of files that can be tackled with at a predetermined period. For example, for the same piece of data, benchmark A needs 10 seconds, and another program B needs only 5 minutes. In the graphs, we'll first set like 1 minute. Then show at 1 minute, the benchmark A is able to deal with 6 same pieces of and the B can deal with 12 files. So that better numbers mean better performance in our graphs, making everything clear enough for a glance.

2.5.2 Strong Scaling Performance

For both two groups, we'll test the strong scaling performance. In this case, the problem size, which is the size of states remains the same and CPU/GPU node increases. We plan to use three world sizes: 32*32, 128*128 and 1024*1024. In our testing environment, every node can contain maximum 6 GPUs and we can apply at most 2 nodes every time, which means there will be 1 to 12 nodes when we run the program. In strong scaling performance experiment, the program is considered to scale linearly if the speedup, here we'll use running time and cycle count to represent is equal to the number of processing nodes used. We keep the 256*256 as world size and change the node amounts as 1, 2, 3, 4.

2.5.3 Weak Scaling Performance

For both two groups, we'll also test the weak scaling performance. In this case, the problem size, which is size of states increases the same and CPU/GPU node increases. In weak scaling performance experiments, the program speedup remains the same while problem size and the number of processing nodes used are linearly increased at the same time.

3. Experiment Result Analysis

3.1 CPU via MPI Group

For the CPU via MPI group, we have the following result for the general parallel computing performance varied via world size, rank and iteration:

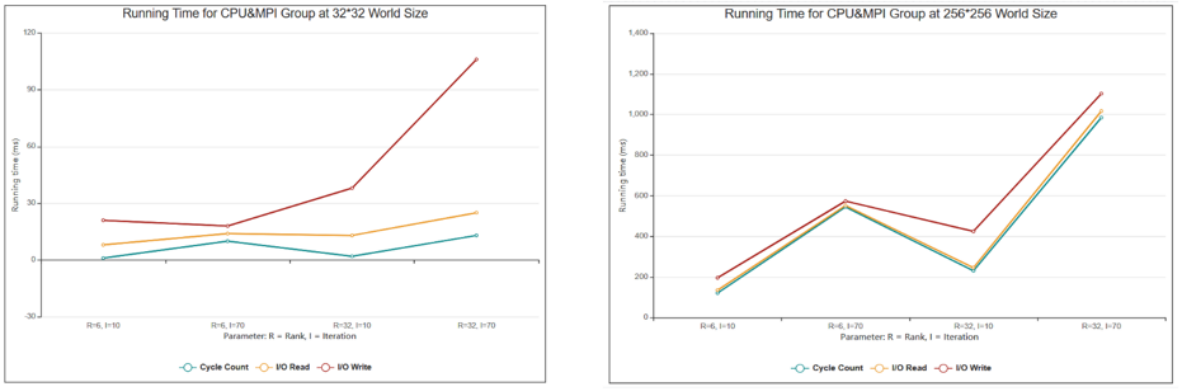

Figure 1. Running Time for CPU&MPI Group at World Size

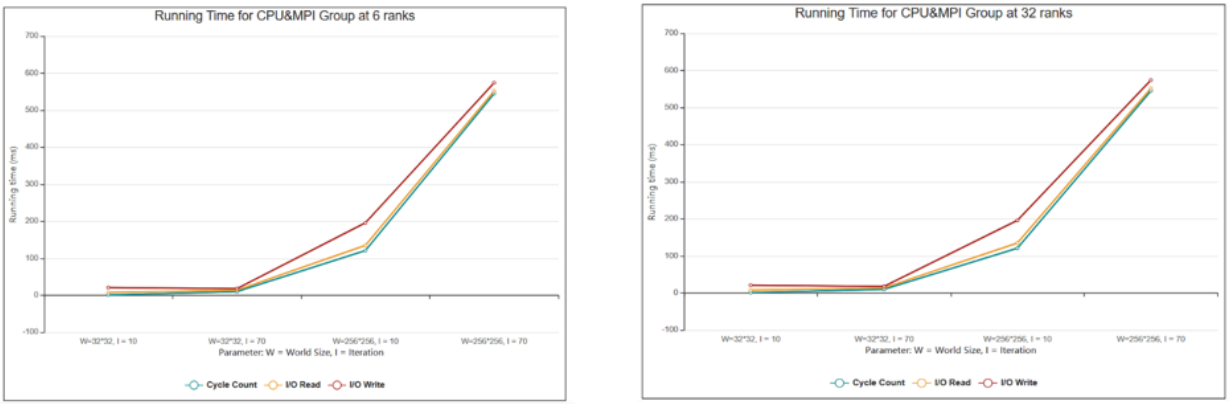

Figure 2. Running Time for CPU&MPI Group at Rank

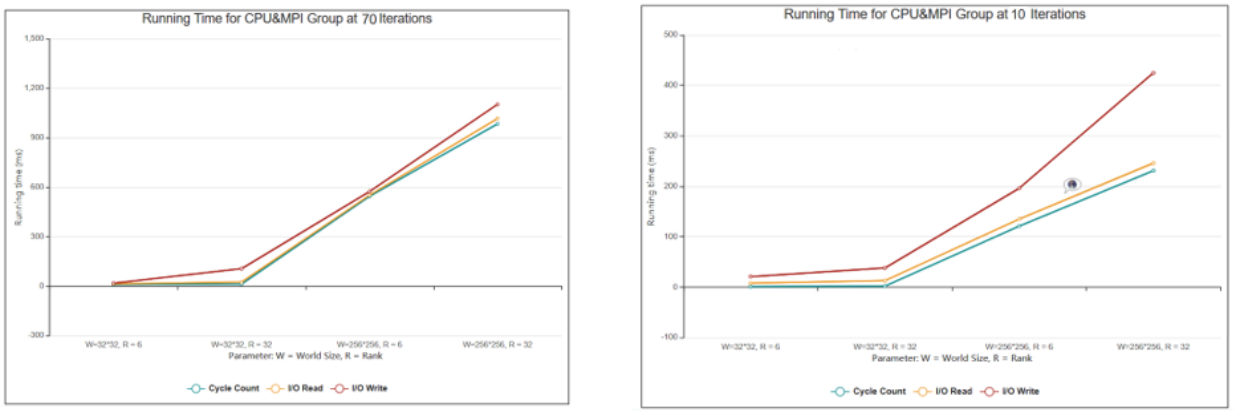

Figure 3. Running Time for CPU&MPI Group at Iteration

From Figure 1, larger world size cost more time for all three type of criteria. From Figure 2, 6 ranks and 32 ranks doesn't make much impact for the running time, it's because that the rank, which is the paralleled reached in our program. The more ranks don't influence the time results, whatever the amounts are, the total running time should be a unit time. From Figure 3, we can observe that 70 iteration has more running time than 10 iteration at every type. Overall, the cycle count, which is computing time for our program often gets the longest part over three criteria. We believe while the world size increasing, the cycle count makes more and more part in the total running time.

3.2 Hybrid CPU/GPU via MPI Group

From Figure 1, the larger world size cost more time for all three types of criteria. From Figure 2, 6 ranks and 32 ranks do not make much impact on the running time. It is because that the rank, which is the paralleled reached in our program. The more ranks do not influence the time results, whatever the amounts are, the total running time should be a unit time. From Figure 3, we can observe that 70 iteration has more running time than 10 iterations at every type. Overall, the cycle count, which is computing time for our program, often gets the longest part over three criteria. We believe that while the world size is increasing, the cycle count makes more and more part in the total

running time.

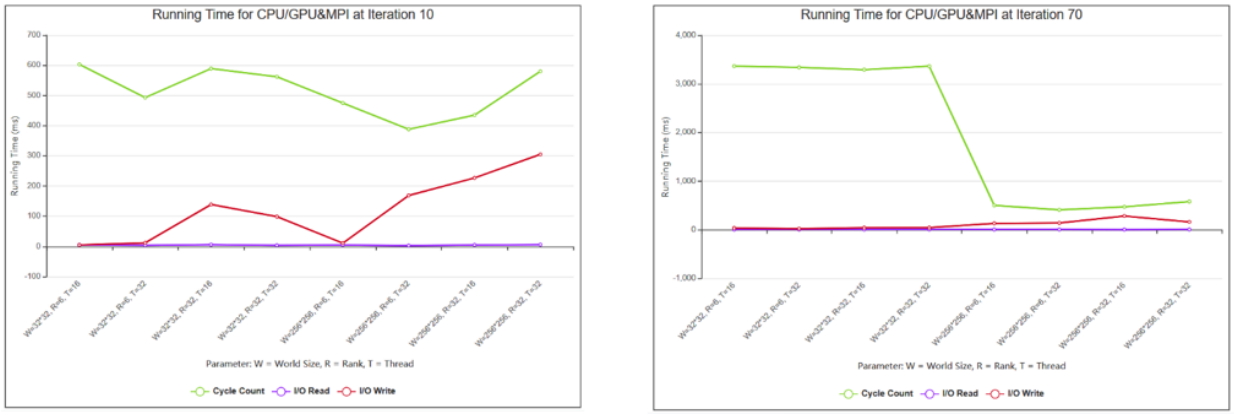

Figure 4. Running Time for CPU/GPU&MPI Group at Iteration

From Figure 4, when iteration is small, like 10, the larger world size makes more running time. But when iteration is large, like 70 at right of Figure 4, small world size makes more running time, which is totally reverse with previous one. The reason is the different features of computing in GPU and CPU, we'll talk about it at implications part.

3.3 Comparison

3.1.1 General Parallel Performance

We choose the fixed configuration world size 256*256, thread 64 and iteration 70 to test the performance of CPU ranks are 6, GPU ranks are 6, 18 and 24.

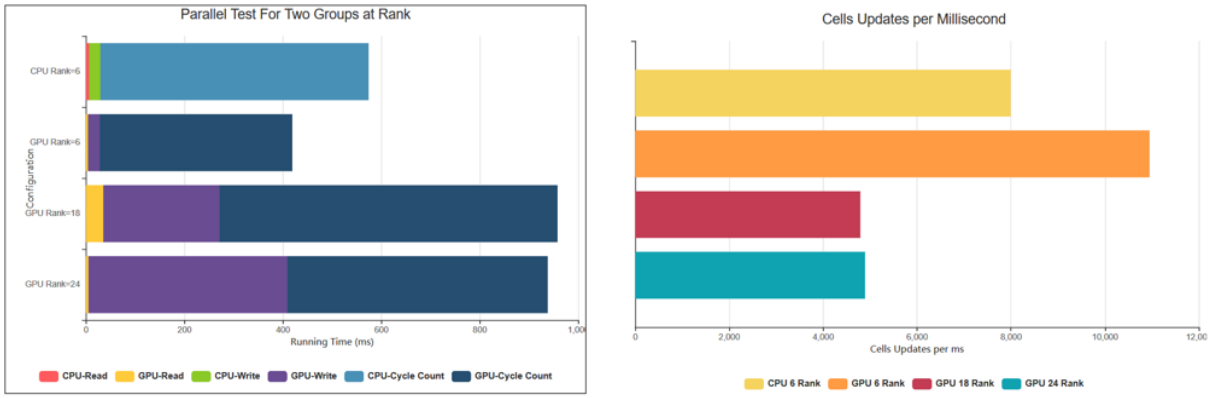

Figure 5. General Parallel Performance

From the left figure in Figure 5, we can see it's not necessary that cycle count time varied via ranks. But when the ranks increase, GPU MPI I/O write time definitely become more. At the same rank number 6, the total running time of CPU via MPI group is more than Hybrid CPU/GPU and MPI group. For cells updates per millisecond rate at right part of Figure 5, more ranks make the update less. But if under same configuration, hybrid group still beats the CPU via MPI group.

3.1.2 Strong Scaling Performance

Configuration is 256*256 world size, 70 iteration, 32 ranks for both groups, and 32 thread for hybrid group. We vary the amounts of nodes as 1, 2 and 4.

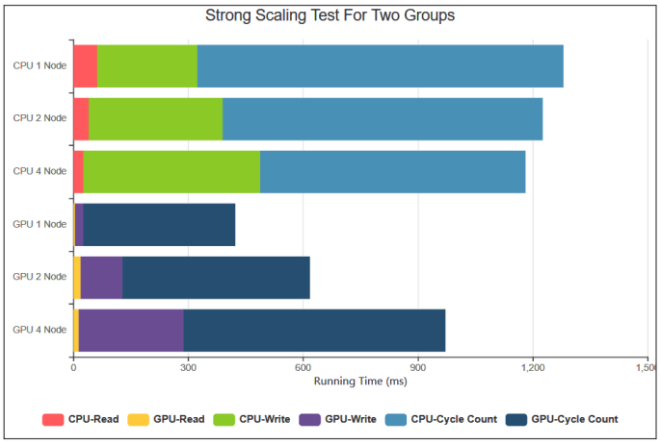

Figure 5. Strong Scaling Test for Two Groups

As shown in the figure above, the first three bars represent the performance of CPU via MPI and the last three bars represent the performance of hybrid CPU/GPU on 256*256 world with different number of nodes. Basically, the average runtime of CPU is higher than the runtime of GPU. The main differences come from the time spent on computing data and writing a file. The reason that CPU via MPI spends more time on writing a file than hybrid CPU/GPU may be that ranks spend more time on computing, and computing takes too much CPU memory. Therefore, when a rank would like to write a file after it completes its computing, it has to spend more time because of the limited memory. The reason why CPU via MPI spends more time on computing than hybrid CPU/GPU is that, each rank of hybrid CPU/GPU uses a parallel way to handle the data on different blocks in GPUs, while each rank of CPU via MPI handles data in a sequential way. Despite that, since the number of ranks is fixed in this figure, the larger the number of nodes, the more the number of states, and the larger the world. However, the number of nodes for CPU via MPI doesn't actually affect the performance since no matter how large the world is, the program always has more ranks to handle the data sequentially. But for hybrid CPU/GPU, the runtime on file writing increases when the number of nodes increases. The reason is that, the number of ranks that is used to write the file increases, so, the communication among ranks also increases which increases the time for a rank to find a place to write the content. So, when the number of nodes increases, the overall runtime for hybrid CPU/GPU also increases.

3.1.3 Weak Scaling Performance

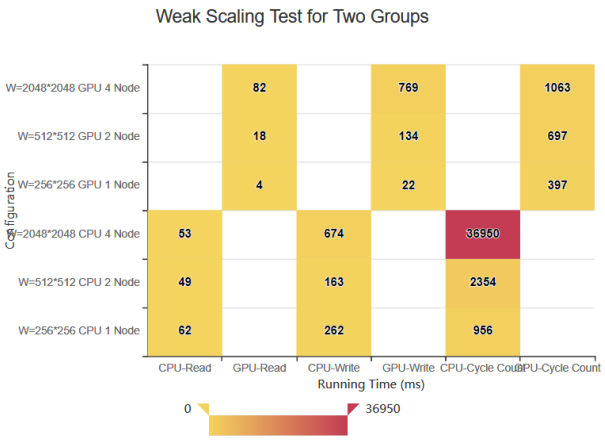

Figure 6. Weak Scaling Test for Two Groups

The Figure 6 is bubble graphs, it's because the one of the values is too large to represent in other graphs. The above figure shows the overall performance for weak scaling test. For this test, the size of world is changed for several

times. And since the size of world increases, the number of ranks that needed to handle the data also increases which also due to the increase of the number of nodes. Basically, the difference between file reading and writing for CPU via MPI and hybrid CPU/GPU is not very obvious, because they all use MPI I/O to do file reading and writing and the number of nodes for each similar test is the same. However, CPU via MPI spends more time on computing than hybrid CPU/GPU because CPU via MPI uses a sequential way to do computation, while hybrid CPU/GPU uses a parallel way.

3.4 Implications

Observe and analyze the result above, and we'll have several implications for parallel computing and MPI parts of our program. First, the ranks do not influence the running time. It is because the MPI I/O read the states as parallelly reading, in on dimension of time, the total running time for several ranks should be a unit running time. Second, in MPI I/O write part, write action costs more time than MPI I/O read. It is because write is supposed to output stuff into a single shared file. The process to assign personal space for every rank cost much more time. So it turns out the MPI read is much faster than MPI write. Third, compute performance/cycle count part is pretty straightforward, more ranks will make the communication time longer because single rank needs to pass ghost line to the next one. Besides, for CPU&MPI group, the larger world size, more computing time; for CPU/GPU&MPI group, the larger world size, less computing time. We believe that the algorithms behind this phenomenon are that for little volume is adverse for GPU computing, which means the GPU performance has anticorrelation with data volume.

3.5 Practical Meaning

Based on the result after applied spreading rule, our model also has the practical meaning for real world.

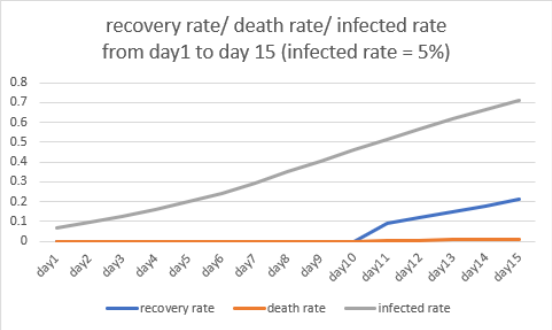

Figure 7(left). propagation rate = 5%

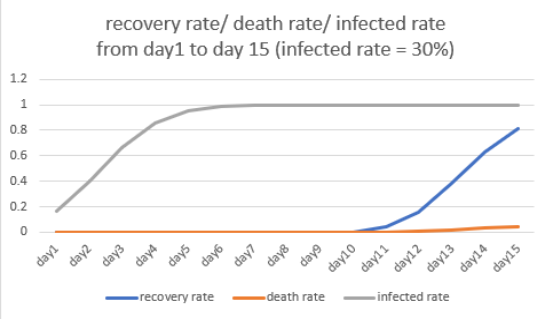

Figure 8. propagation rate = 10%

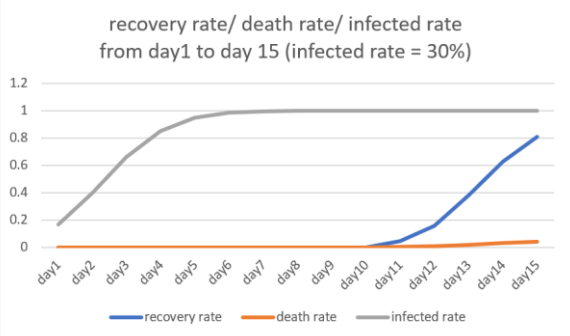

Figure 9. propagation rate = 30%

As we can see from the above graphs, our group calculate the recovery rate/ death rate and infected rate from day1 to day15. When the propagation rate is equal to 5%, the infection rate growth curve is relatively flat, with an average

daily increase of 5%. However, if the propagation rate is equal to 30%, the infected rate reaches 95% within 5 days, which will cause a very serious consequence.

We assume the infected people will transform to death or recovery situation after ten days, so the death rate is 0 in the first ten days. In other words, there's no transformation happens in the first ten days.

We also need to put emphasis on the changes of recovery rate and death rate. As we can see from the graph3, in day15 all of residents in the state are infected. The recovery rate is 81% and death rate is 4%, which mostly matches our setting that the death rate is only 5%. However, in real life, if medical resources are insufficient, the mortality rate may increase greatly.

	day1	day2	day3	day4	day5	day6	day7	day8	day9	day10	day11	day12	day13	day14	day15
recover rate	0	0	0	0	0	0	0	0	0	0	0.093	0.117	0.146	0.176	0.21
death rate	0	0	0	0	0	0	0	0	0	0	0.00476	0.0057	0.0073	0.0086	0.01
infect rate	0.0697	0.094	0.123	0.157	0.198	0.244	0.294	0.349	0.404	0.459	0.516	0.567	0.617	0.665	0.71
	day1	day2	day3	day4	day5	day6	day7	day8	day9	day10	day11	day12	day13	day14	day15
recover rate	0	0	0	0	0	0	0	0	0	0	0.052	0.089	0.144	0.216	0.306
death rate	0	0	0	0	0	0	0	0	0	0	0.0026	0.0045	0.0073	0.011	0.016
infect rate	0.089	0.146	0.2238	0.3186	0.426	0.537	0.643	0.736	0.8138	0.875	0.919	0.951	0.971	0.984	0.991
	day1	day2	day3	day4	day5	day6	day7	day8	day9	day10	day11	day12	day13	day14	day15
recover rate	0	0	0	0	0	0	0	0	0	0	0.047	0.1569	0.38	0.63	0.811
death rate	0	0	0	0	0	0	0	0	0	0	0.0026	0.0084	0.019	0.032	0.04
infect rate	0.165	0.4	0.66	0.854	0.952	0.987	0.997	1	1	1	1	1	1	1	1

Table 3. Detailed Recovery Rate/ Death Rate/ Infected Rate from Day1 to Day15

4. Conclusion and Future Work

4.1 Conclusion

As we discussed in the analysis part, we have several conclusions after going through the whole project: 1) more world size makes longer running time for CPU, but not necessary for GPU. It's because the feature of GPU that it's easy to work for huge data and easy calculate while difficult to work for little data. 2) More ranks will impact the speed of both MPI I/O write and computing. 3) Number of ranks is pointless for MPI I/O read part.

4.2 Future Work

Due to the limited development time for the project, our feature work will focus on following aspects: Set the different population for various state, which is corresponding the truth in our real world; Consider the population mobility, even though we apply the social distance, there is still population move between states; For the position doesn't have nearby eight cells, we'll consider a different way to deal with instead of assuming value 1 for them; Make the distribution both horizontal and vertical so that it's corresponding to the truth that every state has adjacent states.

5. Acknowledge

We're thankful to Professor Christopher D. Carothers, TA Neha Keshan, and TA Angelo Luna for the help in understanding the performance characteristics and techniques of the AiMOS supercomputer system. This project used the resource of RPI AiMOS supercomputer system for testing our model and recording the results.

6. Reference

[1] lang-sc-2009. Samuel Lang, Philip Carns, Robert Latham, Robert Ross, Kevin Harms and William "I/O Performance Challenges at Leadership Scale" pp. 3-4.,
[2] Jens Glaser, Trung Dac Nguyen, Joshua A.Anderson, Pak Lui, Filippo Spiga, Jaime A. Millan, David C. Morse Sharon C. Glotzer "Strong scaling of general-purpose molecular dynamics simulations on GPUs" pp 2-3, 4-8.,
[3] "Measuring Parallel Scaling Performance". Retrieved from https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance
[4] IEEE, 1003.1-1988 INT/1992 Edition, IEEE Standard for Software Project Management Plans (Std 1058a-1998). New York, NY: IEEE,1998
[5] P. Wong and R. der Wijngaart, "NAS parallel benchmarks I/O version 2.4," no. NAS-03-002, 2003.
[6] W. Yu, S. Oral, J. Vetter, and R. Barrett, "Efficiency evaluation of Cray XT parallel I/O stack," in Cray Users Group Meeting, 2007.
[7] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in Proceedings of Supercomputing, November 2008.
[8] E. Gonsiorowski, C. Carothers, and C. Tropper. Modeling large scale circuits using massively parallel discrete-event simulation. In Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on, pages 127{133, Aug.
[9] J. Gonzalez-Dominguez, S. Ramos, J. Tourino, and B. Schmidt, "Parallel Pairwise Epistasis Detection on Heterogeneous Computing Architectures," IEEE Transactions on Parallel and Distributed Systems, 2016, http://ieeexplore.ieee.org/document/7165657/.
[10] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium, April 2009.
[11] H. Schunkert, I. R. K¨onig, S. Kathiresan, M. P. Reilly, T. L. Assimes, H. Holm, M. Preuss, A. F. Stewart, M. Barbalic, C. Gieger et al., "Large-scale association analysis identifies 13 new susceptibility loci for coronary artery disease," Nature genetics, vol. 43, no. 4, p. 333, 2011.
[12] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. E_cient optimistic parallel simulations using reverse computation. ACM Trans. Model. Comput. Simul., 9(3):224{253, July 1999.
[13] W. Dally. Virtual-channel ow control. Parallel and Distributed Systems, IEEE Transactions on, 3(2):194{205, Mar 1992.
[14] G. Kathareios, C. Minkenberg, B. Prisacari, G. Rodriguez, and T. Hoeer. Cost-E_ective Diameter-Two Topologies: Analysis and Evaluation. Nov. 2015. Accepted at IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC15).
[15] NVIDIA. Summit and sierra supercomputers: An inside look at the u.s. department of energy's new pre-exascale systems. Technical report, NVIDIA, November 2014.
[16] A. Ching, A. Choudhary, W. K. Liao, R. Ross, and W. Gropp, "Evaluating structured I/O methods for parallel file systems," International Journal of High Performance Computing and Networking, vol. 2, pp. 133–145, 2004.
[17] A. Phanishayee, E. Krevat, V. Vasudevan, D. Andersen, G. Ganger, G. Gibson, and S. Seshan, "Measurement and analysis of TCP throughput collapse in cluster-based storage systems," Carnegie Mellon University, Tech. Rep. CMU-PDL-07-105, September 2007.
[18] J. Borrill, L. Oliker, J. Shalf, and H. Shan, "Investigation of leading HPC I/O performance using a scientific-application derived benchmark," in Proceedings of Supercomputing, November 2007.
[19] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops Tensor Framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, 2013, pp. 813–824,
[20] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the. IEEE, 1992, pp. 120–127, http://ieeexplore.ieee.org/document/234898/.
[21] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In Proceedings of the 22nd ACM International Conference on Supercomputing, pages 298–301, June 2008.
[22] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing, pages 1–12, New York, NY, USA, 2007. ACM.