Point to Point Communication and Collective MPI Performance Test

Yitao Shen
Rensselaer Polytechnic Institute
Troy, NY, 12180
sheny20@rpi.edu

Jinqian Jiang
Rensselaer Polytechnic Institute
Troy, NY, 12180
jiangj7@rpi.edu

Chau-Lin Huang
Rensselaer Polytechnic Institute
Troy, NY, 12180
huangc11@rpi.edu

Lorson Blair
Rensselaer Polytechnic Institute
Troy, NY, 12180
blairl@rpi.edu

Christopher D. Carothers
Rensselaer Polytechnic Institute
Troy, NY, 12180
chrisc@cs.rpi.edu

ABSTRACT

Communication among multiple nodes plays a critical role in the performance of High Performance Computing. MPI [?] offers a great number of libraries to maximize and test the communication performance in the parallel computing networks. The message passing has been observed to spend additional time in transfering information from one node to another, and several parallel models have been devised to properly describe the phenomenon, which in terms serve as criteria for future benchmarking. In this work, we build a collective MPI performance test to evaluate the performance among different models. To accomplish this, we have a parallel version of Game of Life program optimized with MPI communication scheme and CUDA for GPU parallelization. As far as the authors are concerned, this work could be the first intend to verify the networking perforamnce of a GPU-aided program in terms of different paralllel models. In terms of hardware, the benchmarking takes place on AiMOS [?], an eight petaflop supercomputer using a heterogenous system architecture built with IBM POWER9 CPUs and NVIDIA GPUs. On the software side, the program relies on IBM Spectrum MPI and NVidia CUDA math library [?].

KEYWORDS

Parallel Computing, High Perforamnce Computing, CUDA, MPI

ACM Reference Format:
Yitao Shen, Jinqian Jiang, Chau-Lin Huang, Lorson Blair, and Christopher D. Carothers. 2020. Point to Point Communication and Collective MPI Performance Test. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In order to make modern parallel computing software efficient and powerful, we need to understand what are the underlying caveats that may slow the process down. There are various models proposed with the endavour to effectively describe the parallel system and yet simple enough to capture the essential factors that affect the perforamnce of the system in substantial degree. Each of the models is no better than the others but only is deemed most useful under suitable contexts. Through the performance analysis with the aid of these mathematical models, we can further find the bottlenecks that could be mitigated to optimize the communication perforamnce of the parallel system.

In this article, we will discuss the main parallel performance models, and through the benchmarking of the parallel version of Game of Life program, we can further utilize these models to evaluate the point-to-point and collective communication performances in the fashion of weak or strong scaling.

As per the summary, in section 2 we will discuss previous studies related to the topic of our discussion in this article. Section 3 we will give a brief introduction about Conwayś Game of Life algorithm and how we modified it into its MPI-enabled parallel version. In section 4 and 5 we will present the hardware and frameworks we utilize to perform the benchmarking. More importantly, in section 6 we will discuss the main performance models to describe parallel systems. The course of our discussion will culminate in the introduction of the performance metrics in section 7 and the experiement results in section 8. Last but not least, we will finalize the article with a brief summary and propose prospective future works in section 9.

2 RELATED WORKS

There are several works discussing the benchmarking of MPI performance on parallel systems using different libraries. Pješivac-Grbović et al. [?] give a thorough overview of the parallel models, and compare the performances on inter-cluster MPI collective operations on two systems. The authors in the mentioned article also demontrate that the gap between the message sendings depends on the number of unique destination nodes.

There are also works whose authors discuss the communication performance on the parallel version of Conway's Game of Life. L. Ma et al. [?] demonstrate the performance boosting by implementing the algorithm into its parallel counterpart. The articleś author implemented the parallel program with OpenMP library, which takes advantage of the multithreading in the CPU rather than seeking time efficiency improvement from multi-node perspective.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

In our work we referenced an open-source framework for implementing network benchmarks presented by T. Hoefler et al. [**?**] This framework seperate communication patterns from communication moddules which allows independently added benchmark types and network protocols. The authors also presented a LogGPS pattern [**?**] which supports measurement of LogP and LogGP models parameters such as latency, overhead and gap per bytes over MPI.

3 GAME OF LIFE AS AN ALGORITHM

The Game of Life as invented by the British mathematician John Horton Conway in 1970 [**?**] is a cellular automaton. The algorithm is a zero-player game and as the game evolves throughout undetermined number of iterations, the outcome is determined by the given initial configuration. The game is taken place on a two-dimensional orthogonal grid of square cells. The cell status is atomic, that is it can only be found as alive or dead. Each cell's status is determined by eight adjacent neighboring cells. At each step in time, any cell with fewer than two live neighbors dies due to under-population. On the contrary, if the cell lives with two or three live neighbors survives to the next step. If there are more than three immediately adjacent cells, the cell perishes due to over-population. Lastly, the cell resucitates with exactly three live neighbors and can be seen as the result of cellular reproduction.

The operations involved in the Game of Life include instantiation of the world configuration, the update of the cells in the world, and swaping the newly updated world with the previous one. The operations not only is possible to implement serially, but also with parallel speed-up mechanism to take advantage of the efficient memory manipulation offered by NVIDIA CUDA math library, or through the de-facto networking of various nodes through MPI libraries when the dimension of the world increases toward dimension of high orders of magnitude. For the message passing optimization, we have to take care of the "ghost" rows at the MPI rank boundaries apart from allocating suitable memory chunk for the rank itself. To achieve such requirement, special care needs to be taken at sending and receiving messages at the boundaries where the current rankś top ghost row meets with the previous rankś last row, and where the bottom ghost row meets the first row of the next rank. In addition, the CUDA kernel also needs to undergo modifications to account for the lack of top and bottom edge ẅorld wrapping̈. Therefore, at the initialization of the world configuration, the placement of elements at the edge of the world should be placed in determined ranks. Once the instantiation of the initial world configuration takes place, the MPI-and-CUDA-enabled Game of Life program opens way for further bencharking.

4 GPU AND CUDA TOOLKIT

According to [**?**], AiMOS uses NVidia Tesla V100 GPUs in conjunction with the compute nodes. The NVIDIA Tesla V100 accelerator contains the Volta GV100 GPU. Volta is the codename for NVidiaś GPU microarchitecture release on December 7, 2017. Volta was NVidiaś first chip to feature Tensor Cores, designed specially to yield higher deep learning performance than regular CUDA cores. [**?**]. The architecture is implemented with TSMCś 12 nm FinFET process.

Tesla V100 delivers 7.8 TFLOPS of double precision floating point (FP64) performance, 15.7 TFLOPS of single precision (FP32) performance, and 125 Tensor TFLOPS based on GPU Boost clock.

In AiMOS cluster, there are 16 nodes each containing four NVidia Tesla V100 GPUs with 16 GiB of memory each. In addition, there are 252 nodes each possessing six of the same accelerators with 32 GiB of memory each [**?**].

The CUDA Toolkit version used in AiMOS is CUDA 9.1 and 10.0 [**?**].

5 MPI

AiMOS adopts MPI Spectrum as its message passing interface library. The MPI Spectrum is developed by IBM as a high-performance, production quality implementation of MPI to accelerate end-to-end communication. MPI Spectrum is based on the open-source MPI library, but is integrated with improved RDMA networking add supports NVIDIA GPUs based on IBM PAMI backend. Another feature is that MPI Spectrum enhaces collective library running blocking and non-blocking algorithms [**?**].

6 MODELS TO DESCRIBE THE PARALLEL SYSTEMS

To verify the communication performance of the GPU-accelerated MPI-aided Game of Life program, different configurations of nodes are set. The collective MPI performance is expressed in terms of the four common netowkring performance models: Hockney [**?**], LogP [**?**], LogGP [**?**], and PLogP [**?**]. The parallel models can be seen as a sequence of proposals towarad establishing the proper description for both point-to-point and collective commnication time consumption under any parallel computing system.

The Hockney model is considered the simplest parallel model of communication. The model assumes that the time taken to send a message is

$$T = \alpha + \beta m$$

where α is the size of the messsage, and β is the inverse of the bandwidth, while m represents the message size. The model is suitable to describe point-to-point communication systems.

The LogP model intends to offer a simple yet more detail view to facilitate the finding of bottlenecks in possible communication latency. The model id described with four parameters: the latency L, overhead o, gap between the sending of messages g, and the nmber of processors or nodes involved in the communication P. The model assumes that only small amount of messages is transferred simultaneously. The time needed to transfer messages between nodes takes

$$T = L + 2o$$

where L is the latency, and o as the overhead.

Since LogP does not monitor transmission of long messages, LogGP further extend such aspect in its description. A new parameter Gap per byte (G) is taken into account, which is defined as the time per byte for a long message [**?**]. Unlike the LogP model which restricts to constant small size messages, LogGP allows sending larger messages. Typically, time taken to transfer a message is:

$$T = L + 2o + (m - 1)G$$

where G is the gap per byte and m is the size of the message.

In the work of T. Kielmann et al. [**?**], parametrized LogP is introduced as a slight extension of LogP and LogGP models that is capable of monitoring the minimal gap between two messages without saturating the network for each message size. In addition to the parameters contained in LogP, additional parameters are included: the sender and receiver overheads, message transfer time and data copying time to and from the network interfaces are included in the latency. Moreover, the gap paraneter is defined as the minimum time interval between consecutive message transmission or reception. The overall time spent in the message transfering can be expressed as:

$$T = L + g(m)$$

where $g(m)$ is the gap per message. The worth pointing out that the sender $o_s(m)$ and receiver $o_r(m)$ overheads depend on the message size.

7 PERFORAMNCE METRICS

As for the LogP, LogGP and PLogP model. We at first intended to get the parameters from AiMOS based on the Netgauge toolkit[**?**] mentioned above. But it turned out that this toolkit doesn't work when we run it on AiMOS. We spent hours trying to fix this problem in order to continue our experiment but we failed to do so. A possible cause for the failure is that the program is out-of-date and isn't compatible with the AiMOS system. We also tried to implement the algorithm referencing the paper related to this toolkit[**?**], but we didn't finish the implementation. We were forced to run the toolkit on our own laptop. The laptop used has CPU: Intel® Core™ i7-8750H CPU @ 2.20GHz × 12 and GPU: GTX 1060 6G. We record the parameters that we mearued under this setting in the following table.

L(ms)	s(bytes)	o_s(ms)	o_r(ms)	g(ms)	G(ms/bytes)
0.2635	1	0.052	0.118	0	0
0.2635	1025	0.123	0.306	0.101	0.000125
0.2635	2049	0.158	0.381	0.107	0.000108
0.2635	3073	0.184	0.484	0.115	0.000095
0.2635	4097	0.644	0.671	0.074	0.000135
0.2635	5121	1.076	1.163	0.012	0.00018
0.2635	6145	1.28	1.182	-0.002	0.000196
0.2635	7169	1.348	1.301	-0.004	0.000191

Table 1: LogP and LogGP parameter measured on Laptop

Here, L stands for latency, s stands for the message size, o_s and o_r stands for two o values available, where o_s should be used with compatible packet size and o_r is relatively imprecise and should be used carefully. g: is the approximate point where the fitted curve crosses the y axis. And G is the slope of fitted g,G curve. Detailed calculation and explanation can be found in the paper[**?**] mentioned above.

To estimate the execution time according to Hockney's model, we used the formula **t(s) = l + s / b**. In the formula, **s** stands for the message size, **l** stands for the latency of the network, **b** stands for the bandwidth of the network.

$$T = \alpha + \beta$$
$$\alpha = l = Latency$$
$$\beta = \frac{s}{b} = \frac{MessageSize}{Bandwidth}$$

Then it came with the quesion "How can we determine the latency and the bandwidth?" In this benchmark model, it assumes that some process A sends a message to process B, while process B sends message back The advantage is that it will not require any synchronized clocks between these two processes. While the disadvantage is that it will presume the communication performance or the costs between two points is totally symmetric.

To determine latency, it reuires to execute the benchmark for itration time equal to zero.

More specifically, to calculate the **total execution time**, it simply subtracts the end timestamp generated after the for loop of iteration by the start timestamp, both of which are generated via the function **MPI_Time()**.

$$TotalExecutionTime = EndTimeStamp - StartTimeStamp$$

The **average executioin time** can be calculated via using the message size timed by MAX_MEASUREMENTS to split the total execution time

$$AverageExecutionTime = \frac{TotalExecutionTime}{MessageSize * MAX\_MEASUREMENTS}$$

The **bandwidth** can be claculated by using message size timed by its data size to be split by its average execution time.

$$Bandwidth = \frac{MessageSize * sizeof(DataType)}{AverageExecutionTime}$$

We evaluate the scalability of our modeling results for both strong scaling and weak scaling. The strong scaling refers to the case where problem size is fixed and number of processing elements increases. Strong scaling can be used as justification for CPU-bound programs.This type of program don't scale up very well and it's hard to get good performance at large process elements count. The strong scaling efficiency is calculated by

$$t_1/(N * t_N) * 100\%$$

git where t_1 represents the time taken to complete one unit of work on a MPI rank and t_N represents the time taken to complete N units of work.

The weak scaling is the case where workload on each processing elements stays fixed and the amount of processing elements increases to increase the total problem size. Weak scaling is jused as justification for programs that are memory or other system resources bound. This type of programs scales up well at large process elemtns count. The weak scaling efficiency is calculated by

$$(t_1/t_N) * 100\%$$

where just as the strong scaling efficiency, t_1 represents the amount of time taken to complete one unit of work and t_N represents the time taken to complete N units of work on N processing elements.

8 PERFORMANCE RESULTS

The experiment results of Hockney model is shown as below. We estimated the running time for different number of MPI ranks. Figure 1 shows the comparison between them.

For this experiment, the message size chanegs from 1 to 1024 by the power of two. And the rank number per node changes from 1 to 6 for up to two nodes. After the testing in different conditioins, it is found that the lantency of the network is almost static which is fixed around 0.004 ms.

Figure 2 shows the result comparison of estimated execution time across different ranks versus message size and number of ranks. Note that the green column in Figure 2(b) represents the result from running on 2 nodes with each nodes having 6 MPI ranks.

Strong scaling result is shown in the figure below.

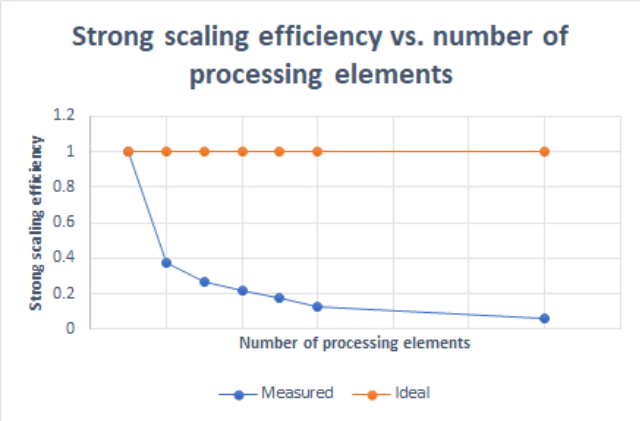


Figure 4: Strong scaling results

It can be observed that the strong scaling results are not very good but this is as expected. The overhead for the algorithm and communication increases as the number of processes increases and takes us to this result. It could be hard to reduce this part and we hadn't figure out how to do so.

And below is the weak scaling results.

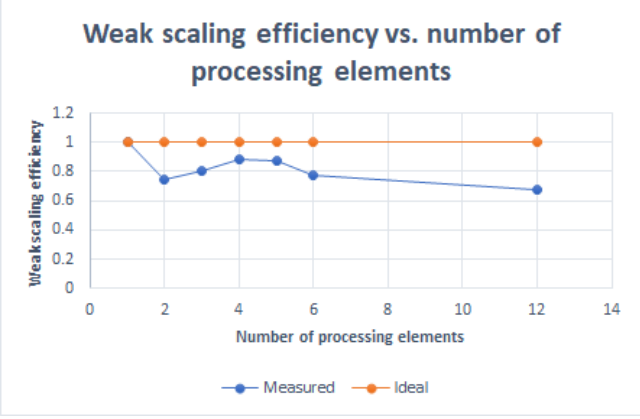


Figure 5: Weak scaling results

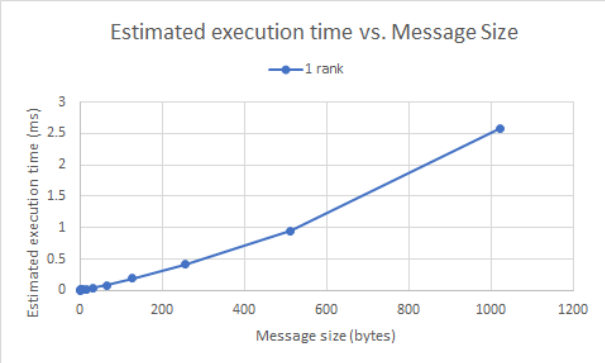
This result is better than the strong scaling result. In the sense that we didn't see significant drop in weak scaling efficiency as that in the strong scaling. Note that here is still a trend of dropping in the weak scaling efficiency.

9 CONCLUSION AND FUTURE WORKS

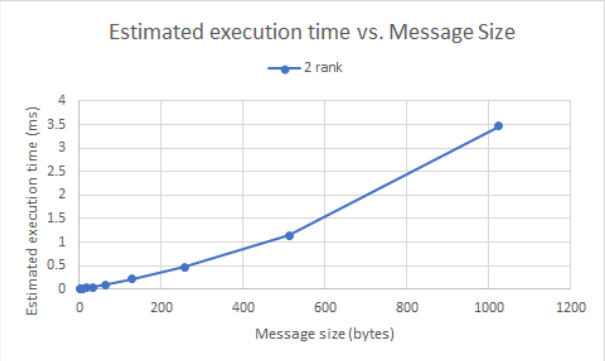
In this work we perform benchmarking on four different parallel models to evaluate the perforamnce of the CUDA-aided Game of Life program. The GPU-accelerated program is integrated with MPI's communication mechanism to effectively manage the memory access as well as transfering among different nodes.

For future works, there are two possible extensions on the program to study how different mechanism may further improve or deteriorate the performance of the current version of Game of Life program. The first possible direction is to incorporate multithreading to the current version to observe how sharing resources on the same node may affect the message communication, and another possible study involves the execution of the same GPU-accelerated program on different topologies such as the Fat Tree [**?**], Dragonfly [**?**] or Slimfly [**?**] networks to investigate the difference in performance described with the mentioned parallel models.

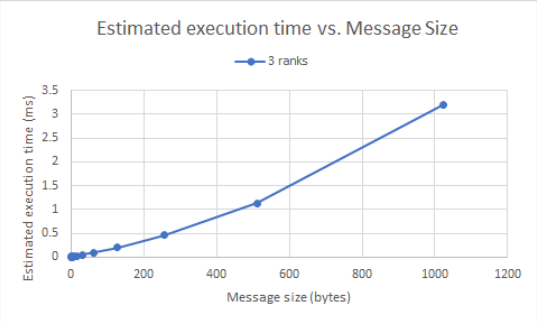
10 ACKNOWLEDGMENTS



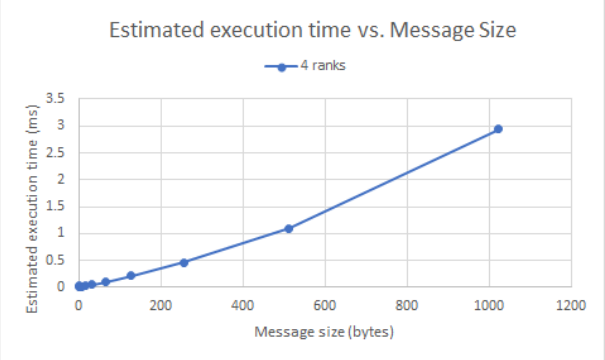
(a) Rank 1



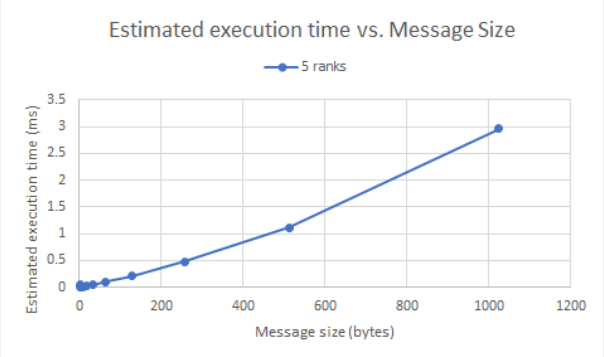
(b) Rank 2



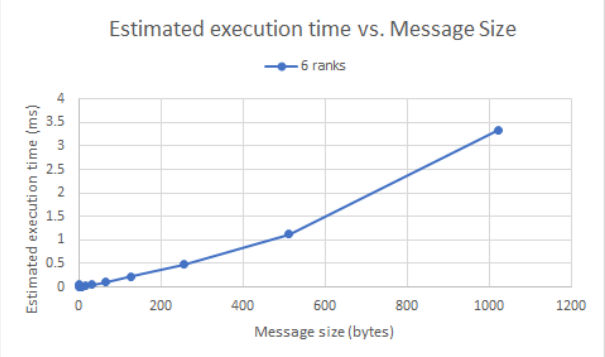
(c) Rank 3



(d) Rank 4

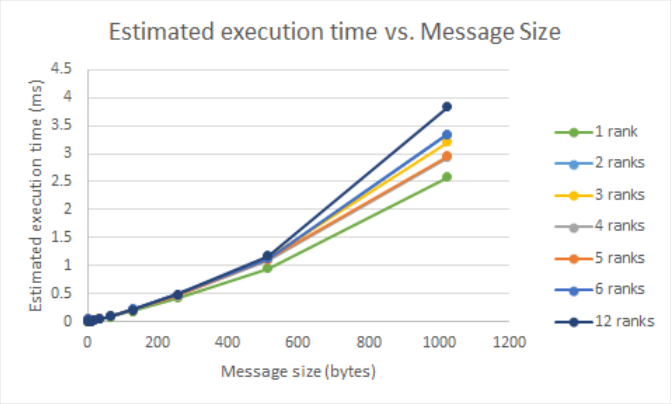


(e) Rank 5

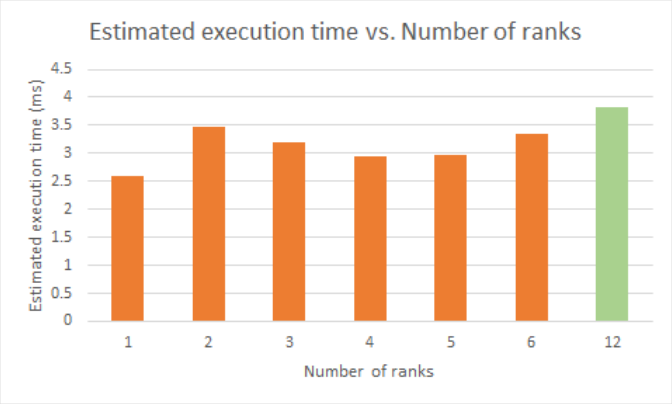


(f) Rank 6

Figure 1: Estimated execution time vs message size along different ranks.



(a) Comparison of estimated execution time vs message size across different ranks.



(b) Estimated execution time in relation to ranks

Figure 2: The relationship between message sizes and number of ranks to estimated execution time.