Programming Assignment 3 Report

Team Members: Chenyu Dai, Fei Ding, Shuge Fan, Lintong Han

1. Implementation

We used the TA's code template as a starter so that we can largely adopt their serial implementation for Conway's Game of Life. The parallel implementation shares the File IO code with the provided serial implementation. The initial split-grid and the final merge-grid operations by the root processor are implemented with the asynchronous point-to-point communication primitive. This is acceptable per TA's instruction because eventually almost all data must leave the root processor. In fact, the scatter/gather operation has no advantage over the point-to-point communication in terms of the message size-dependent μ factor but only in terms of the constant overhead τ (p vs. log p). In practice, the difference may be even more negligible.

We generally followed the guidelines in the instruction document and fulfilled all the requirements. Specifically,

1. The MPI grid topology is created at the beginning and later used for communicating with the diagonal neighbors and exchanging the corner numbers.

2. The row and column communicators are also created at the beginning and later used for communicating with the cardinal neighbors (along the main axes) and exchanging entire rows or columns.

3. Multiple persistent communication channels are set up for each processor with their cardinal and diagonal neighbors just before the generation update loop. This includes calling the *MPI_Send_init* and *MPI_Recv_init* functions. Inside the loop, the *MPI_Start* function is called each iteration for exchanging data; notice this function is inherently asynchronous, and all *MPI_Wait* calls are made at the end of the loop. Multiple *MPI_Request_free* calls occur at the end to close all channels.

4. Asynchronous send and receive calls are used to the maximal extent: the root processor uses asynchronous sends when splitting the grid and asynchronous receives when merging the grid. The other processors do not have to use asynchronous calls, though, because they cannot start until they have received their sub grids and they also have nothing else to do after the iterations are over. Inside the update loop, all processors initiate the asynchronous send calls first, then update the internal grid cells, then receive the edges and corners from the neighbors, and finally update the edges and corners of their own sub grid. Notice the receive calls are synchronous because the immediate step after depends on the data received.

5. Derived datatypes are used whenever necessary. In particular, the column-typed edges and the sub grids (matrices) are sent using the custom MPI vector type. Rows and

corner numbers do not require custom datatypes because they are stored in consecutive memories.

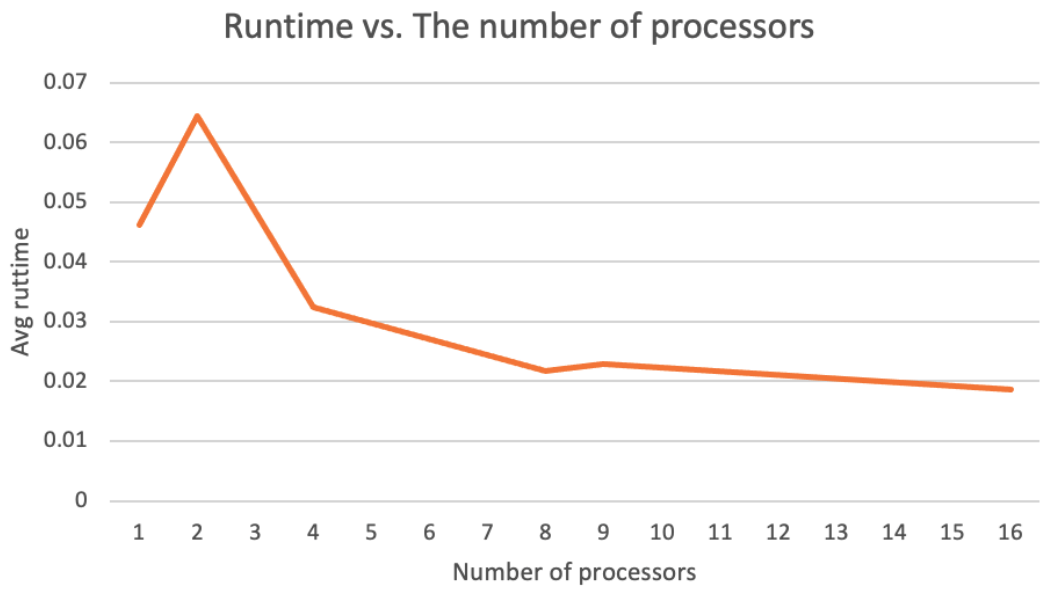
Finally, there is one additional interesting detail about our implementation: we update the sub grid stored in "*local_data*" by writing to a temporary buffer called "*next_local_data*." Then, we do a std::swap (pointers exchanged). This eliminates the need for a full value copy and significantly improves the parallel performance. The drawback is that we must create another set of persistent communication channels, but that's only a constant memory overhead per processor. This technique also does not increase the total amount of communication.

2. Experiments and Analysis

(Note: the original data can be found in Appendix 1)

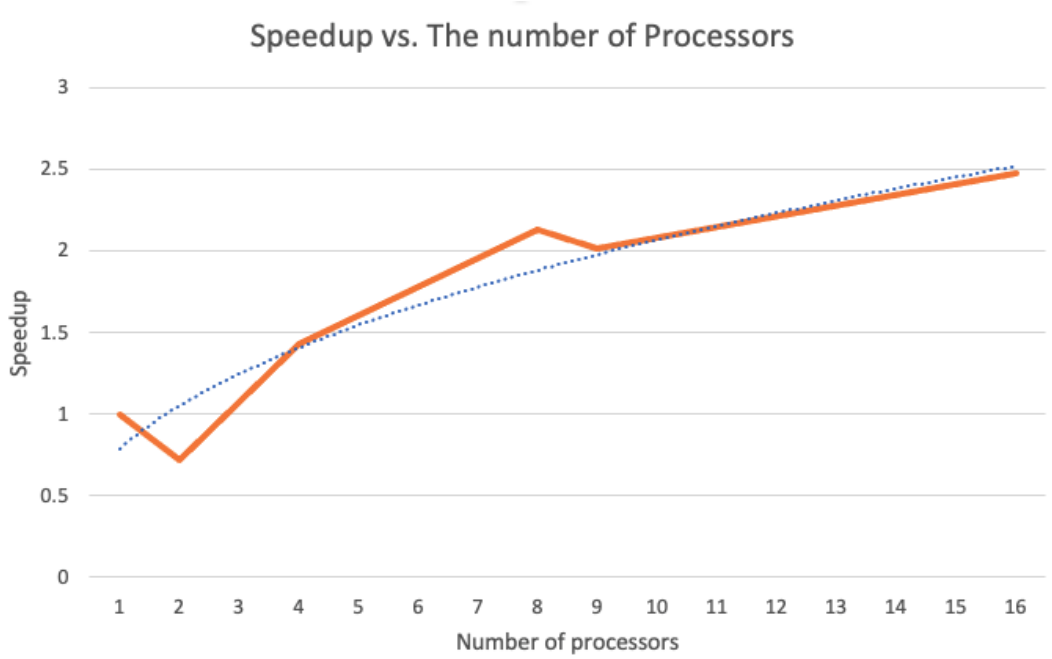
2.1 Varying number of processors

First, we tested various runtime with different number of processors using 2000 x 2000 grid size. The plot is as follows:



As we can see, the average runtime decreases when the number of processors increases. This is expected as the more processors in the system, the faster a solution will be found due to simultaneous computing.

Based on the runtime, we calculated the speedup and included a speedup trend plot with different number of processors below.



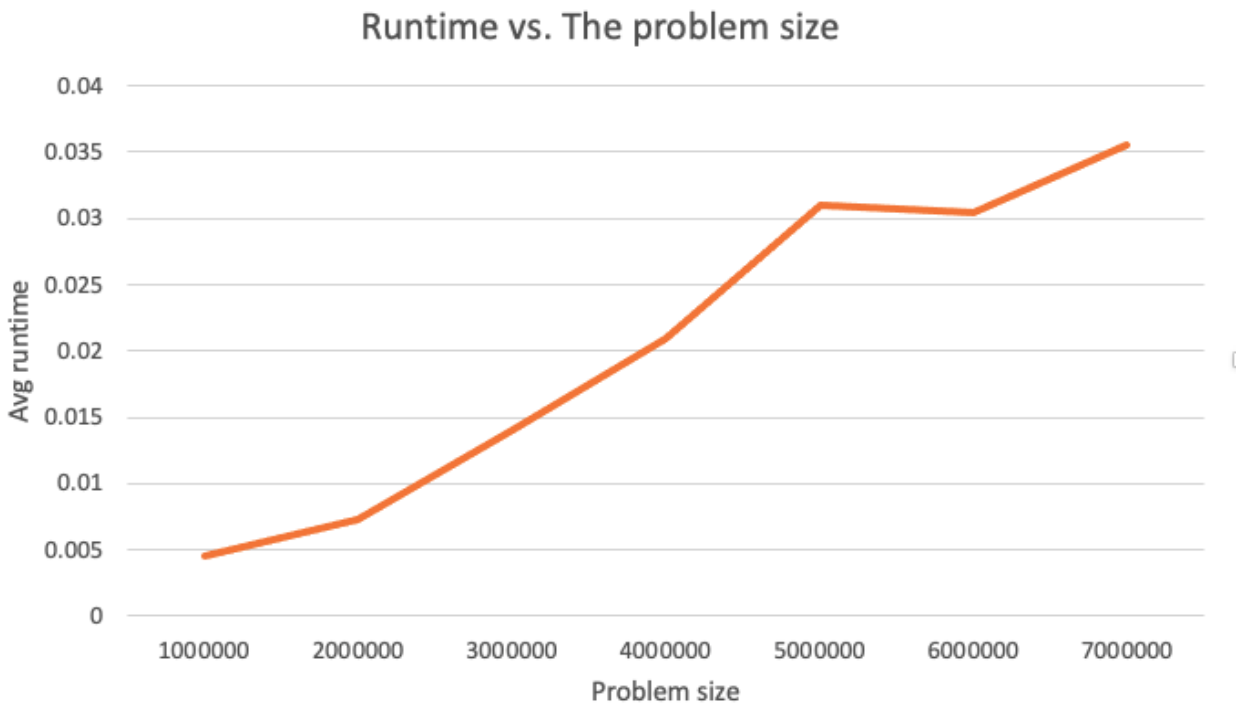
The speedup increases when the number of processors increases. However, we noticed that the speedup is not increasing linearly when the number of processors increases in a linear manner. The analysis is as follows:

First, we assume that the grid size is $m \times n$, and we consider it as single generation update with *p* processors arranged in square mesh. The sequential runtime is O(mn) and the parallel runtime is O($\mu\frac{m+n}{\sqrt{p}}$) for communication and O($\frac{mn}{p}$) for computation. Note that we don't have τ in our communication runtime because we already set up a persistent communication channel ahead of time, considering the edges and corners, the data size per processor is $\frac{2(m+n)}{\sqrt{p}} + 4$ in the order of $\frac{m+n}{\sqrt{p}}$. Since the speedup is $\frac{T(mn,1)}{T(mn,p)}$, we have $\frac{mn}{\mu\frac{m+n}{\sqrt{p}} + \frac{mn}{p}} = \frac{mnp}{mn + \mu(m+n)\sqrt{p}}$. If *m* and *n* are both constant, then the speedup as a function of p should be in the order of O($\sqrt{p}$), which aligns with the trend we observe in our plot.

The imperical speedup function we found in our experiment is 0.58947892 * $\sqrt{p}$ + 0.226921102880953 when the grid size is 2000 * 2000.

2.2 Varying problem(grid) size

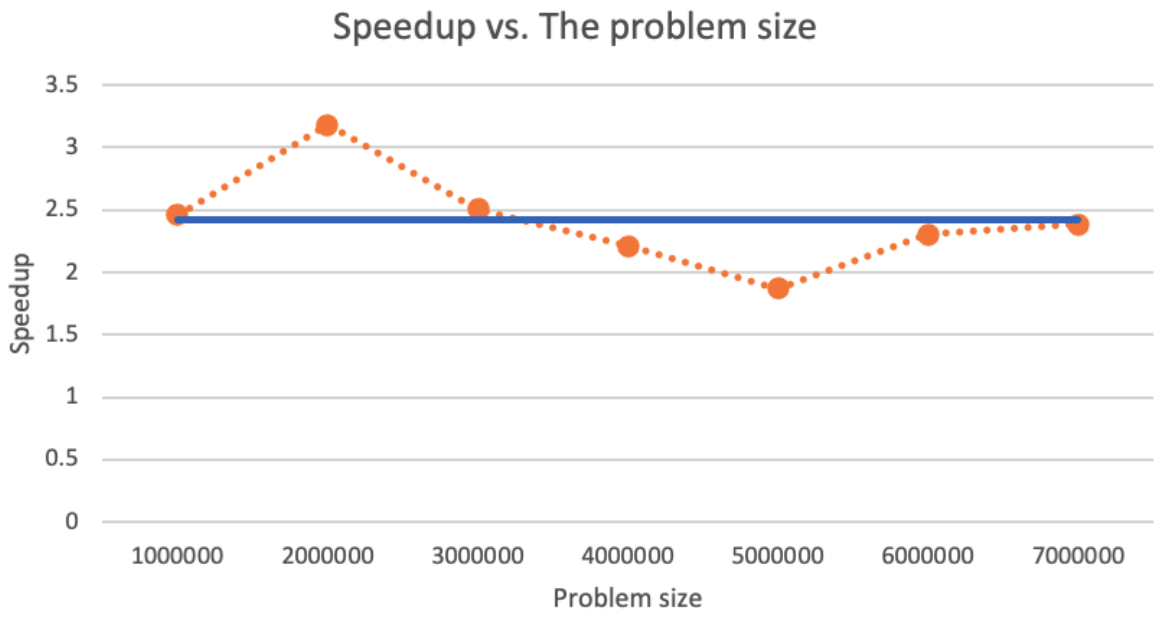
We tested various runtime with different problem sizes using 16 processors. The plot we obtained is as follows:



In general, the average runtime increase linearly when the problem size (in terms of the total amount of elements) increases linearly. This is because the runtime is dominated by the linear factor of *mn* over (*m* + *n*) given that the number of processors (*p*) is fixed.

The imperical runtime function we found in our experiment is $5.5778 * 10^{-9} * n -$ 0.0017741, where *n* is the total number of elements in the grid and *p* is 16.

Similarly, we we calculated the speedup and included a speedup trend plot with different problem sizes below.



The above diagram shows the speedup trend when we increase the problem size when we keep the number of processors fixed. The exact formula estimate of speedup is $O(\frac{mnp}{mn+\mu(m+n)\sqrt{p}})$ as we derived before. Once *m* and *n* increases to a sufficiently large number, the runtime becomes $O(1)$ when *p* is fixed, which satisfies our original expectation. The speedup plot also shows that the speedup factor is likely to be constant with our particular setup. The Imperical speedup constant is 2.471005.

Appendix 1. Experimental Data

# proc	time 1	time 2	time 3	time 4	time 5	avg run time	speedup	sqrt(# proc)
1	0.050431	0.04294	0.049428	0.044521	0.043627	0.0461894	1	1
2	0.06038	0.061419	0.067916	0.061508	0.070781	0.0644008	0.717217799	1.414213562
4	0.030689	0.033533	0.031313	0.034035	0.032152	0.0323444	1.428049369	2
8	0.019092	0.021044	0.024559	0.021724	0.021959	0.0216756	2.130939859	2.828427125
9	0.018887	0.022134	0.021446	0.028717	0.023674	0.0229716	2.010717582	3
16	0.018189	0.017741	0.018335	0.016907	0.022316	0.0186976	2.470338439	4

Figure 1. Runtime & speedup vs. the number of processors when grid size is 2000 x 2000

p size	time 1	time 2	time 3	time 4	time 5	avg run time
1000000	0.004152	0.006912	0.004091	0.004822	0.002915	0.0045784
2000000	0.005666	0.006702	0.006432	0.006355	0.010933	0.0072176
3000000	0.014128	0.011987	0.017591	0.015183	0.01156	0.0140898
4000000	0.021242	0.022994	0.021798	0.019496	0.018977	0.0209014
5000000	0.031282	0.03164	0.032902	0.02931	0.029825	0.0309918
6000000	0.027725	0.030605	0.033007	0.029313	0.031825	0.030495
7000000	0.029974	0.032279	0.036664	0.037495	0.041015	0.0354854

Figure 2. Runtime vs. the problem(grid) size when the number of processors is 16

p size	time 1	time 2	time 3	time 4	time 5	avg run time	speedup	avg speedup
1000000	0.011183	0.010879	0.011446	0.011644	0.011179	0.0112662	2.46072864	2.41710048
2000000	0.023637	0.022212	0.02221	0.024689	0.022163	0.0229822	3.18418865	2.41710048
3000000	0.032819	0.036455	0.035946	0.036685	0.0345	0.035281	2.50400999	2.41710048
4000000	0.044569	0.048703	0.046349	0.044904	0.046532	0.0462114	2.21092367	2.41710048
5000000	0.053718	0.056998	0.059374	0.060084	0.060022	0.0580392	1.87272762	2.41710048
6000000	0.064599	0.067583	0.071692	0.069083	0.077967	0.0701848	2.30151828	2.41710048
7000000	0.080927	0.085765	0.08857	0.080026	0.087983	0.0846542	2.38560647	2.41710048
						avg speeup	2.41710048	

Figure 3. Runtime & speedup vs. the problem(grid) size when the number of processors is 1