Tim Lagnese, Ryan Soley

Professor Kamesh

CMPSC 450

13 April 2017

Homework 4

For this assignment, we worked together on all three questions. Tim took the lead on question 1 and Ryan on questions 2 and 3. We both contributed parts to each question.

1. Response to 1
2. Code for an MPI parallelized Game of Life can be found in the attached file gameoflife.c. The code takes inputs m, the grid dimension (m x m) and k, the number of time steps. It also accepts an optional ‘-p’ flag that prints the game grid at each time step to verify correctness. Raw data mention below can be found in “Table 2: Game of Life” at the end of this document.

For a small number of processors (<= 8), the fraction of time spent in communication is very low (< 1%) regardless of the grid size or number of time steps. This makes sense as the processors do not need to communicate that much when there are not many of them. As we increase the number of processors beyond 8, we see that the fraction of time spent in communication becomes apparent, especially for larger sized grids. We see the fraction of time spent in communication in these scenarios range from ~1% to ~15%. Figure 2.1 shows that for large grid sizes there appears to be a point where the fraction peaks and either levels off or begins to decrease. It seems more likely that it will level off at this point.

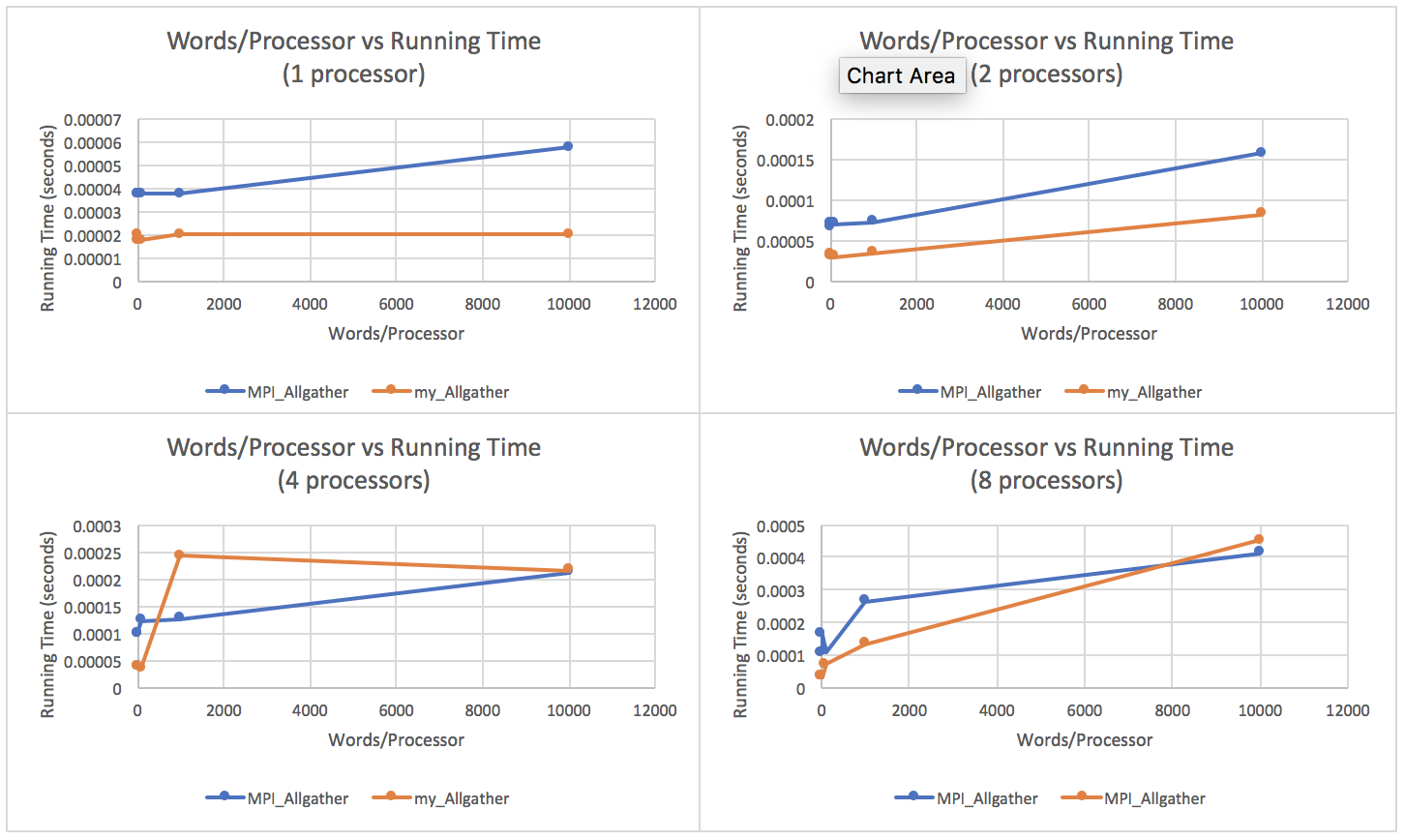
**Figure 2.1.** Processors vs Fraction of Time Spent in Communication for a 10,000 x 10,000 grid

The code appears to scale very well with increasing gird size and process concurrency. Figure 2.2 shows that as we increase the number of processors for a statically sized grid, the execution time keeps getting faster and faster. We can also see in our table that as we increase grid size, our billons-of-cell-updates/second statistic remains largely the same. This shows that as grid size increases, we are still working at the same relative speed which is good for scalability.

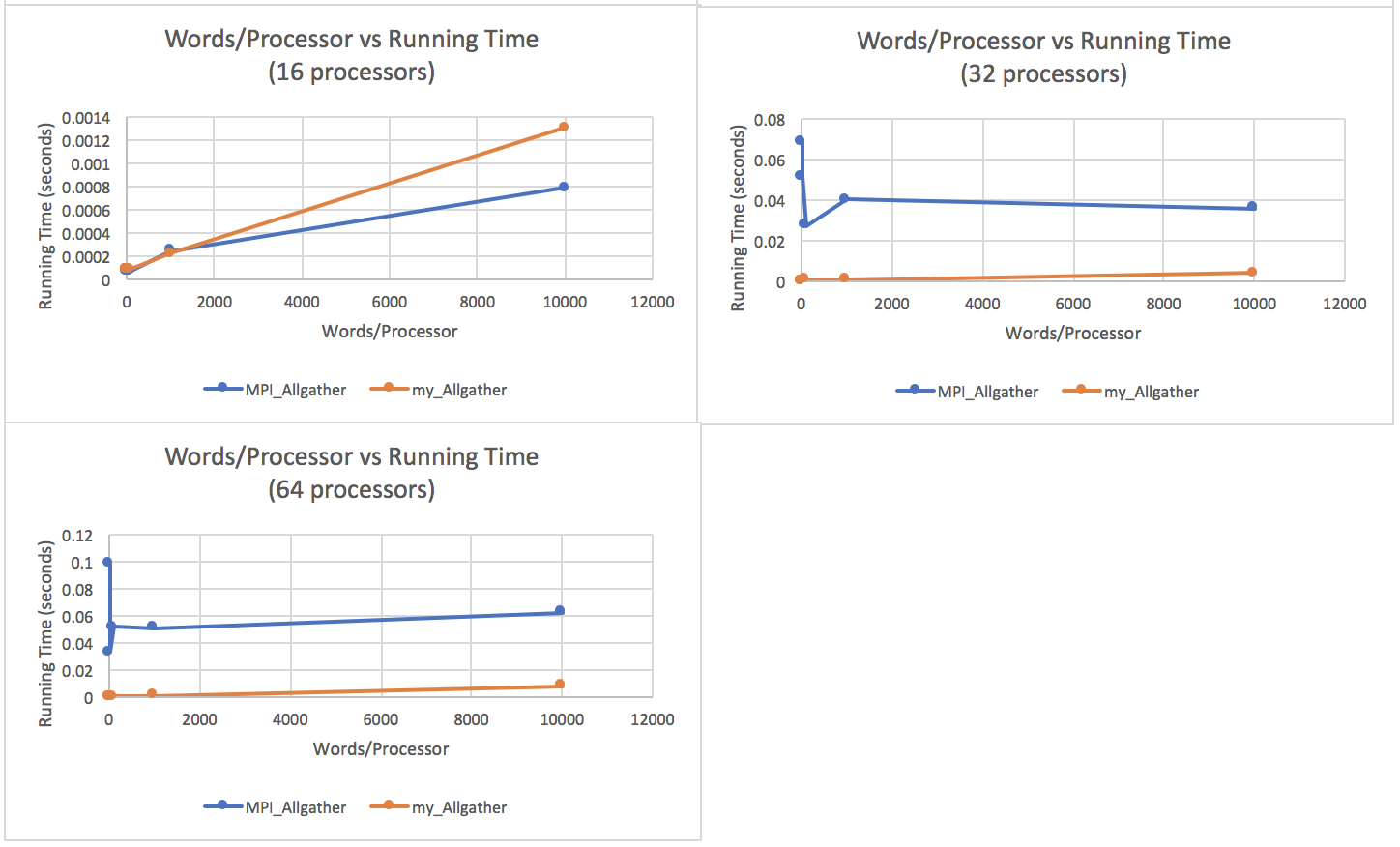
**Figure 2.2.** Processors vs Running Time for a 10,000 x 10,000 grid

1. Code for a recursive doubling based implementation of Allgather can be found in the attached file my\_Allgather.c. The code takes input n, the number of words per processor. The Allgather executes on an array of size n times the number of processors. Raw data mention below can be found in “Table 3: My Allgather” at the end of this document.

Figures 3.1 and 3.2 show seven different graphs of words per processor vs running time, with each one running a different number of processes. Note that for the recursive doubling approach, the number of processors must be a power of two. As we can see based on Figures 3.1 and 3.2, the recursive doubling strategy has varying success against MPI’s default implementation of Allgather. For both a small (<= 2) and large (>= 32) number of processors, the recursive doubling approach was faster than MPI’s implementation. For small sizes, this is likely because recursive doubling (and Allgather in general) does not have to do much work when there are not a lot of processors. For large sizes, recursive doubling is likely much faster because it can really take advantage of processor parallelism with more processors. Recursive doubling’s downside is that when message size becomes too big in relation to the number of processors, the algorithm suffers a slowdown. This can be seen in the graphs for 4, 8 and 16 processors. In each of them, we can see where this ratio of word size to number of processors becomes a problem and MPI’s Allgather performs faster. Presumably, if we carried out the graphs for 32 and 64 processors out further we would eventually find this ratio and see MPI’s implementation become faster than our recursive doubling one.



**Figure 3.1** Words per Processor vs Running Time for 1, 2, 4 and 8 processors



**Figure 3.2** Words per Processor vs Running Time for 16, 32 and 64 processors

**Table 1: Raw Data For 1**

**Table 2: Game of Life**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Processors | Grid Width | Time Steps | Running Time (s) | T\_Comm (s) | T\_Comm/Time | Cell Updates/s (billions) |
| 1 | 5000 | 10 | 0.271 | 0.0000 | 0.000 | 0.922 |
| 1 | 5000 | 100 | 2.711 | 0.0010 | 0.000 | 0.922 |
| 1 | 5000 | 1000 | 27.191 | 0.1000 | 0.000 | 0.919 |
| 1 | 10000 | 10 | 1.141 | 0.0000 | 0.000 | 0.876 |
| 1 | 10000 | 100 | 11.424 | 0.0020 | 0.000 | 0.875 |
| 1 | 10000 | 1000 | 114.170 | 0.0150 | 0.000 | 0.876 |
| 2 | 5000 | 10 | 0.138 | 0.0004 | 0.003 | 1.807 |
| 2 | 5000 | 100 | 1.392 | 0.0030 | 0.002 | 1.796 |
| 2 | 5000 | 1000 | 13.869 | 0.0280 | 0.002 | 1.803 |
| 2 | 10000 | 10 | 0.582 | 0.0006 | 0.001 | 1.717 |
| 2 | 10000 | 100 | 5.834 | 0.0040 | 0.001 | 1.714 |
| 2 | 10000 | 1000 | 58.280 | 0.0360 | 0.001 | 1.716 |
| 4 | 5000 | 10 | 0.081 | 0.0004 | 0.005 | 3.077 |
| 4 | 5000 | 100 | 0.809 | 0.0030 | 0.003 | 3.090 |
| 4 | 5000 | 1000 | 8.066 | 0.0280 | 0.004 | 3.099 |
| 4 | 10000 | 10 | 0.326 | 0.0010 | 0.002 | 3.071 |
| 4 | 10000 | 100 | 3.262 | 0.0040 | 0.001 | 3.066 |
| 4 | 10000 | 1000 | 32.564 | 0.0380 | 0.001 | 3.071 |
| 8 | 5000 | 10 | 0.076 | 0.0005 | 0.006 | 3.281 |
| 8 | 5000 | 100 | 0.758 | 0.0030 | 0.005 | 3.298 |
| 8 | 5000 | 1000 | 7.578 | 0.0370 | 0.005 | 3.299 |
| 8 | 10000 | 10 | 0.300 | 0.0010 | 0.002 | 3.334 |
| 8 | 10000 | 100 | 2.994 | 0.0050 | 0.002 | 3.340 |
| 8 | 10000 | 1000 | 29.969 | 0.0630 | 0.002 | 3.337 |
| 16 | 5000 | 10 | 0.039 | 0.0010 | 0.016 | 6.458 |
| 16 | 5000 | 100 | 0.381 | 0.0070 | 0.019 | 6.547 |
| 16 | 5000 | 1000 | 3.794 | 0.0610 | 0.016 | 6.567 |
| 16 | 10000 | 10 | 0.151 | 0.0010 | 0.006 | 6.616 |
| 16 | 10000 | 100 | 1.505 | 0.0080 | 0.005 | 6.647 |
| 16 | 10000 | 1000 | 15.035 | 0.0960 | 0.006 | 6.651 |
| 32 | 5000 | 10 | 0.020 | 0.0020 | 0.091 | 12.690 |
| 32 | 5000 | 100 | 0.192 | 0.0100 | 0.051 | 12.951 |
| 32 | 5000 | 1000 | 1.922 | 0.2960 | 0.154 | 12.967 |
| 32 | 10000 | 10 | 0.076 | 0.0020 | 0.033 | 13.177 |
| 32 | 10000 | 100 | 0.755 | 0.0470 | 0.062 | 13.209 |
| 32 | 10000 | 1000 | 7.549 | 1.4040 | 0.186 | 13.204 |
| 64 | 5000 | 10 | 0.008 | 0.0010 | 0.150 | 30.891 |
| 64 | 5000 | 100 | 0.071 | 0.0080 | 0.115 | 35.065 |
| 64 | 5000 | 1000 | 0.725 | 0.1650 | 0.228 | 34.359 |
| 64 | 10000 | 10 | 0.038 | 0.0020 | 0.044 | 25.927 |
| 64 | 10000 | 100 | 0.380 | 0.0170 | 0.043 | 26.200 |
| 64 | 10000 | 1000 | 3.805 | 0.5980 | 0.157 | 26.196 |

**Table 3: My Allgather**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Processors | Words/Processor | Array Size | MPI\_Allgather Time (s) | my\_Allgather Time (s) |
| 1 | 1 | 1 | 0.0000375 | 0.0000175 |
| 1 | 10 | 10 | 0.0000375 | 0.0000200 |
| 1 | 100 | 100 | 0.0000375 | 0.0000175 |
| 1 | 1000 | 1000 | 0.0000375 | 0.0000200 |
| 1 | 10000 | 10000 | 0.0000575 | 0.0000200 |
| 2 | 1 | 2 | 0.0000650 | 0.0000325 |
| 2 | 10 | 20 | 0.0000700 | 0.0000300 |
| 2 | 100 | 200 | 0.0000700 | 0.0000300 |
| 2 | 1000 | 2000 | 0.0000725 | 0.0000350 |
| 2 | 10000 | 20000 | 0.0001575 | 0.0000825 |
| 4 | 1 | 4 | 0.0000975 | 0.0000375 |
| 4 | 10 | 40 | 0.0001000 | 0.0000375 |
| 4 | 100 | 400 | 0.0001225 | 0.0000350 |
| 4 | 1000 | 4000 | 0.0001275 | 0.0002425 |
| 4 | 10000 | 40000 | 0.0002125 | 0.0002150 |
| 8 | 1 | 8 | 0.0001075 | 0.0000350 |
| 8 | 10 | 80 | 0.0001625 | 0.0000325 |
| 8 | 100 | 800 | 0.0001100 | 0.0000725 |
| 8 | 1000 | 8000 | 0.0002650 | 0.0001325 |
| 8 | 10000 | 80000 | 0.0004125 | 0.0004525 |
| 16 | 1 | 16 | 0.0000775 | 0.0000850 |
| 16 | 10 | 160 | 0.0000625 | 0.0000800 |
| 16 | 100 | 1600 | 0.0000700 | 0.0000825 |
| 16 | 1000 | 16000 | 0.0002425 | 0.0002125 |
| 16 | 10000 | 160000 | 0.0007800 | 0.0012975 |
| 32 | 1 | 32 | 0.0686575 | 0.0000775 |
| 32 | 10 | 320 | 0.0514325 | 0.0000775 |
| 32 | 100 | 3200 | 0.0270175 | 0.0002425 |
| 32 | 1000 | 32000 | 0.0399225 | 0.0004675 |
| 32 | 10000 | 320000 | 0.0356400 | 0.0036950 |
| 64 | 1 | 64 | 0.0974275 | 0.0001450 |
| 64 | 10 | 640 | 0.0325775 | 0.0001300 |
| 64 | 100 | 6400 | 0.0512025 | 0.0002625 |
| 64 | 1000 | 64000 | 0.0506375 | 0.0007400 |
| 64 | 10000 | 640000 | 0.0622850 | 0.0076575 |