MESSAGE-PASSING IMPLEMENTATION OF THE “GAME OF LIFE” PROGRAM

Homework 4

By

Colin Sanders

CS 581 High Performance Computing
November 4, 2021

Problem Specification

The goals of the previous three homework assignments were to design and implement the “Game of Life” program, test the program, both serial and multithreaded using the OpenMP library, for functionality and correctness, measure the performance of the program while optimizing to improve performance, and then further analyze performance by testing the implementation on the Dense Memory Cluster hosted by the Alabama Supercomputer Authority. This homework requires a rewrite of the previous Game of Life implementations in order to leverage the message-passing library, OpenMPI (Open Message Passing Interface).

The objectives of this homework are:

1. design and implement an efficient message-passing version of the Game of Life program (developed in Homework-1) using the Message Passing Interface (MPI), meeting the following requirements:
    a. take the problem size, max number of iterations, number of processes, and an output directory to write the final state of the board as command line arguments
    b. use dynamic memory allocation to create the two arrays with ghost cells
    c. check if there is a change in the board state between iterations and exit when there is no change after printing which iteration the program is exiting
    d. write the state of the final board to an output file
    e. use one-dimensional data distribution (row-wise distribution of the array) to partition the board across multiple processes
    f. use the MPI_Sendrecv function exchange data between different processes and use the ghost cells within each process to store the data received from the neighboring process
    g. do not use the scatter and gather operations within the generational iteration loop, although they can be used to distribute the initial board and gather the final board

2. test the program for functionality and correctness for different numbers of processes
3. execute the program on the ASC cluster for 1, 2, 4, 8, 10, 16, and 20 processes and compute the average time taken across three runs for each thread count, each using the matrix size 5000x5000 with 5000 maximum iterations
4. compute the speedup and efficiency of the message-passing version, plotting each of these for each number of processes specified above
5. analyze the performance of the message-passing version in comparison to the serial and multithreaded (OpenMP) versions, and note any potential areas for optimization
6. specifically for graduate students, design and implement a separate version of the Game of Life program that uses non-blocking point-to-point MPI calls to exchange data between different processes
    a. compare the performance of the non-blocking implementation to that of the blocking version and the multithreaded version
    b. compute and plot the speedup and efficiency of the non-blocking implementation

Program Design

While implementing the OpenMPI multithreaded version of the Game of Life program was fairly trivial (as trivial as is adding a couple of C pragmas and synchronizing a few pieces of data between processes), the message-passing interface requires a decent amount of design as the paradigm must change since each process has its own complete set of memory. Beginning with the serial version of my implementation and a few improvements that I had gathered from the multithreaded implementation; my first step was determining which process was going to generate and distribute the initial game board. As is the case with many MPI programs, process 0 became designated as the root process. This root process validates the command line arguments and typecasts where necessary, allocates the memory for the two instances of the game board (the current state and the previous state), initializes with random data, and prepares to distribute the validated command line arguments to each of the processes. One design choice that I made was to broadcast the validated arguments to each process rather than performing a redundant char-to-integer execution for each argument. This reduces clock cycles of the program executed, at a trade off of increases network communication, but since our processes are running on local networks or one machine, I felt this choice was justified.

After the distribution of the arguments and generation of the gameboard, the root process-specific logic ends. Each process then calculates two arrays: sendCounts[] and displacement[]. These arrays are necessary to determine how many rows each process should get, and where in the original game board these rows start for each process, required by the MPI_Scatterv calls that will have the root process distributing the game board initial state. One choice I made is how rows are distributed when they do not divide evenly among the processes. Every process will get the integer division of the number of rows by the number of processes, and then the last process will receive any leftover rows calculated by the modulo of the number of rows by the number of processes. Another key moment is the allocation of arrays that represent the “ghost” cells, as the distribution of the original array does not automatically assign ghost cells to each of the smaller chunks distributed to each process. Each process requires four arrays to represent ghost cells: Two that represent a process’s local top and bottom rows, to send to its neighbor processes directly above and below in the game board, and two more arrays that represent the rows a

process will receive from its neighbors at the start of each iteration necessary to calculate the new top and bottom rows for a process.

Now that the ghost cell arrays have been allocated for each process and each process has received its portion of the game board, the generational loop can begin. Each generation begins with each process populating its ghost cell communication arrays. Next, the inter-process communication occurs – processes begin sending their ghost cells to their neighbors and receiving their neighbors ghost cells simultaneously using the MPI_Sendrecv functions. Then similarly, each process sends its top row to its neighbor above. In order to avoid deadlock or race conditions and overly complex logic on the MPI_Sendrecv calls, the edge processes send their edges to the process wrapped around to the other size with the data cleared later to keep the SumOfNeighbors calculations correct. After this, each process has their neighbors rows as their "ghost cells" to calculate the new generation of the board.

This is when the implementation starts looking very similar to the previous versions, with a few minor differences. The rows iterated over are now intentionally restricted to each process's row count, instead of having OpenMP divide the for loop behind the scenes. The SumOfNeighbors calculation for a given cell is similar to the previous implementations as well, with a few key differences. Now, we need to use the ghost cells from our neighbors when a cell is located on the top or bottom row a process's local game board, so there is conditional logic for this to make sure the sum is calculated correctly without out of bounds errors, and the new local board for each process is fully calculated with each process keeping track of whether it has changes. After this, an MPI_Allreduce is called, which makes all process's block until this point, then a synchronizes reduce operation is called and the result is distributed to each process. This is done with a sum reduce operation to combine all of the change flags, and then distribute whether there was a global change to each process. If there was a global change, each process can break from this loop and the program can clean up and finish or continue until the maximum generations count is reached otherwise. After the end of the generation loop, the root process gathers each process's local game board back into the whole game board array and prints this to file before doing garbage collection.

The non-blocking version of the program has a few key differences. Rather than each process waiting for each of its neighbors' top and bottom rows before beginning the new generation cell calculations, it simply initiates the send and receive requests before advancing. It then starts processing cells, starting with the cells it knows it can calculate without the neighbor rows – all cells not located on the local board's first or last rows. After doing all of these calculations, each process initiates an MPI_Waitall for each of its requests, guaranteeing that the processes receive the neighbors rows before they advance if they have not already. Then, the process calculates the data for the rows it could not before, ultimately reducing the time spent waiting for the inter-process communication. Performance will be explored below.

Testing Plan

As in each of the previous implementations, testing this program fully and accurately require both correctness and performance to be tested. In the previous assignments, I created separate "test bed" programs that will execute my Game of Life program and analyze the results. I

extended the test bed program to have support for MPI and tested all of my original test cases with the 7 different thread counts listed above, confirming consistency and correctness across different numbers of threads. As before, these tests include “still-lifes”, which are stable after 1 generation, “oscillators”, which will indefinitely cycle through a small number of states without stabilizing, and a sanity test based on the R-pentomino that confirms the program is working correctly across many iterations.

After the correctness test cases, performance was then analyzed among process counts and whether the program was using blocking or non-blocking send and receive calls. Three random executions of size 5000x5000, 5000 maximum iterations, with different process counts (1, 2, 4, 8, 10, 16, 20) were executed on the ASC cluster using OpenMPI batch jobs and the average time was marked below. Speedup was calculated by dividing the single-process time by the n-process time for each case, and efficiency was calculated by diving speedup by the number of processes used.

Test Cases

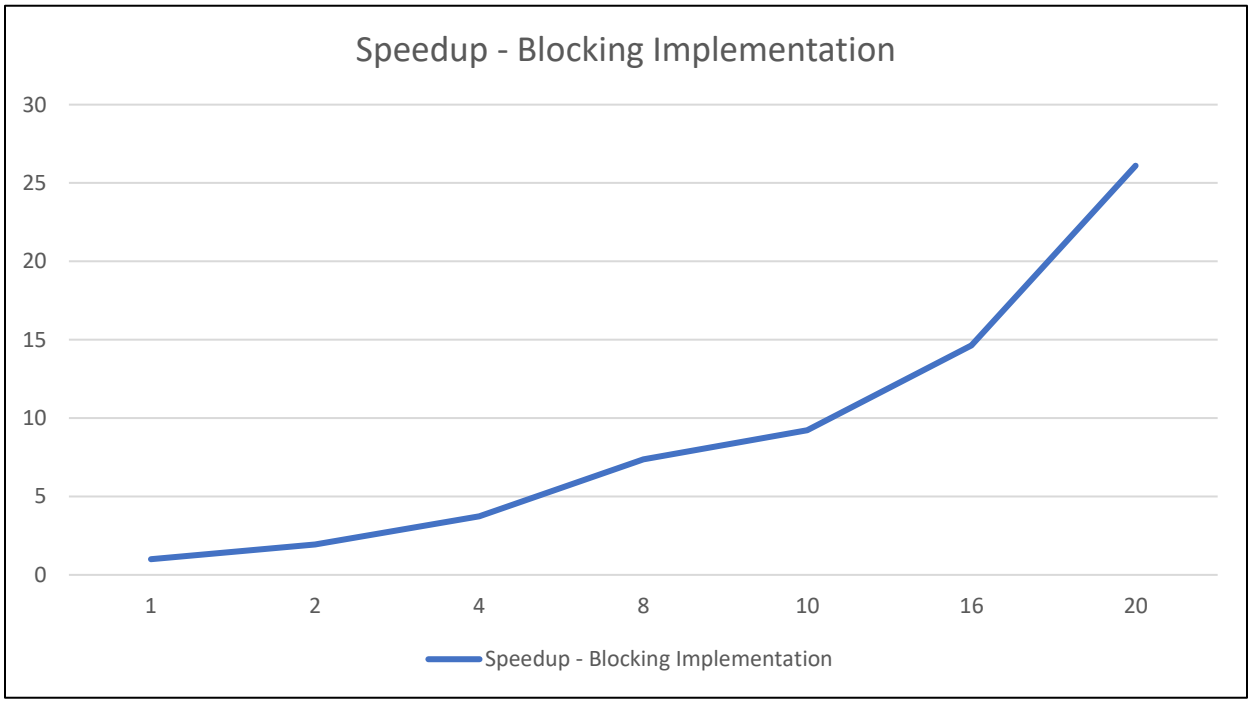
Test Case Number	Input Values	Generations Before Stable
Correctness 1 – Block	0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0	1 Generation
Correctness 2 – Boat	0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0	1 Generation
Correctness 3 – Tub	0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0	1 Generation
Correctness 4 – Blinker	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0	1000 Generations with MAX_GENERATIONS = 1000

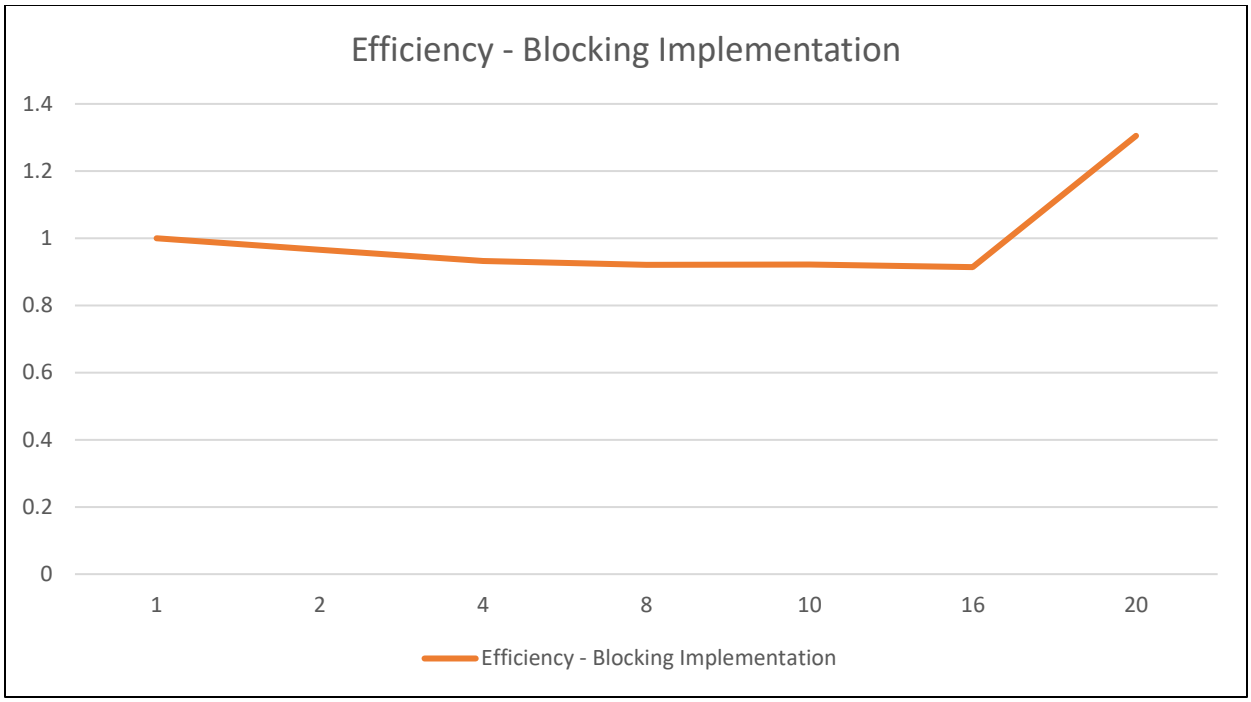
Correctness 5 – Beacon	0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0	1000 Generations with MAX_GENERATIONS = 1000
Sanity – R Pentomino	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,	Initially took 20 Generations and was consistent through 100 attempts.

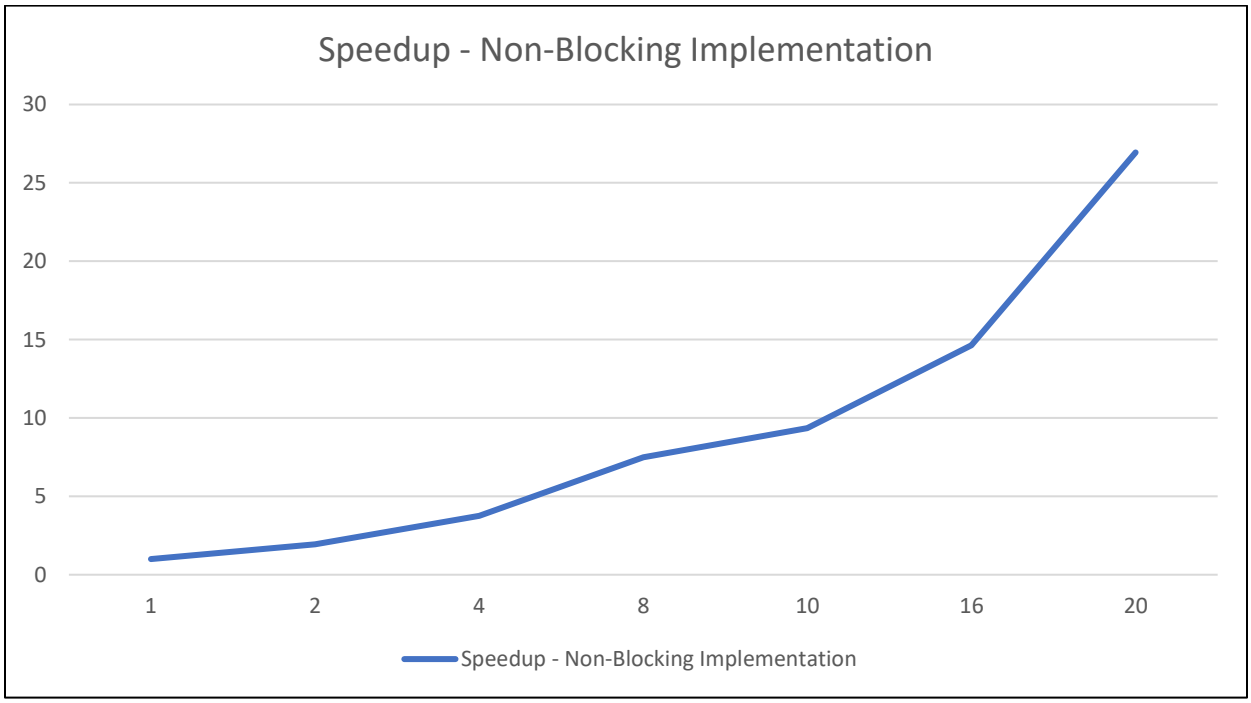
Performance, 5000x5000 Problem Size, 5000 Maximum Generations

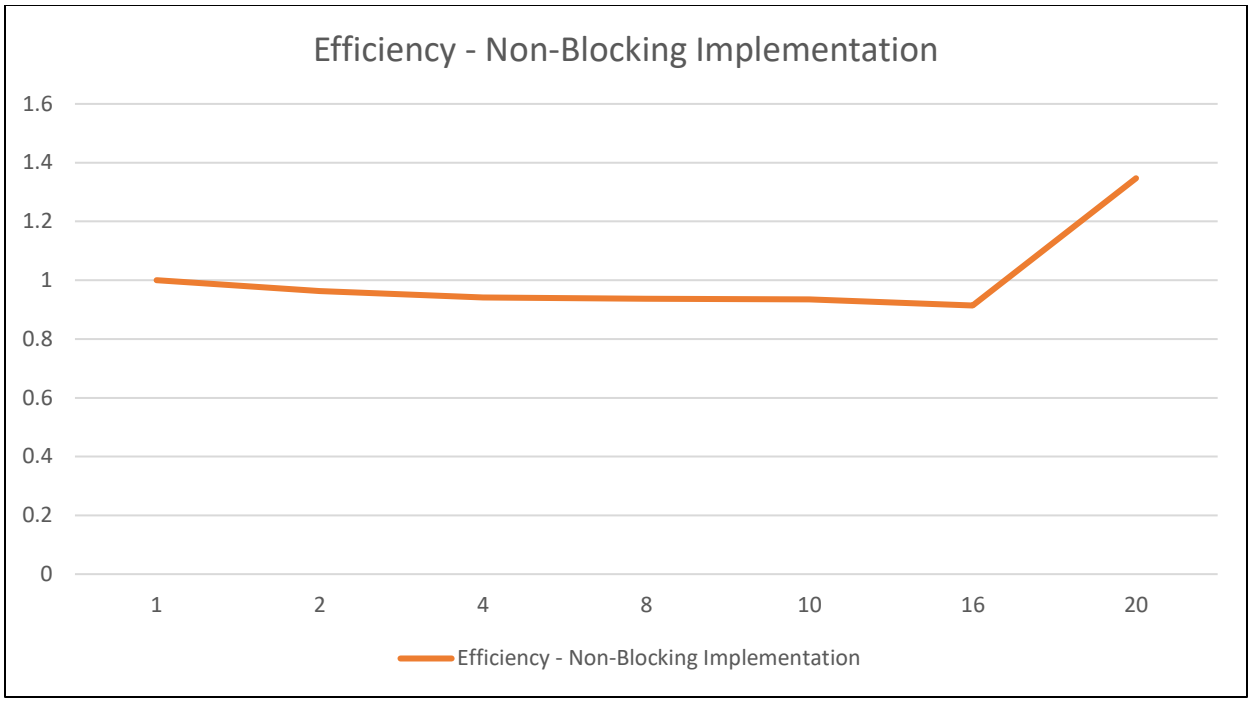
Process Count	Time Taken ASC Cluster (blocking)	Speedup (blocking)	Efficiency (blocking)	Time Taken ASC Cluster (non-blocking)	Speedup (non-blocking)	Efficiency (non-blocking)
1	557.2 s	1.000	1.000	503.8 s	1.000	1.000
2	288.5 s	1.931	0.966	261.7 s	1.930	0.963
4	149.4 s	3.730	0.932	133.9 s	3.763	0.941
8	75.61 s	7.369	0.921	67.23 s	7.494	0.937

10	60.42s	9.222	0.922	53.86 s	9.354	0.935
16	38.09 s	14.629	0.914	34.44 s	14.630	0.914
20	21.35 s	26.104	1.305	18.70 s	26.94	1.347




Efficiency - Blocking Implementation
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
4
8
10
16
20
Efficiency - Blocking Implementation


Speedup - Non-Blocking Implementation
30
25
20
15
10
5
0
1
2
4
8
10
16
20
Speedup - Non-Blocking Implementation



Analysis and Conclusions

Compared to the serial version, the single process message-passing implementation is significantly slower as is expected with the amount of data movement and overhead associated with the MPI program. However, the speedup values almost directly scaling with the process count demonstrate that the message passing implementation does a great job splitting the workload amongst processes. The OpenMP implementation from the previous homework outperforms this version by on average about 25% when using the Intel compiler, but the MPI version outperforms the GCC compiled OpenMP program. The overhead of generating additional processes and sending data between processes throughout the program is more significant compared to spinning up threads which makes up a significant portion of the reasons for the slightly decreased performance in the MPI version.

The non-blocking version of the MPI Game of Life performs about 10% better in every process count, especially including increased efficiency in the 10 and higher process counts. This aligns with the implementation, as less time is spent on processes idling serially while waiting for other processes to send or receive their data and work can begin on other cells while the process-to-process communication occurs asynchronously. Outside of the performance boost, the speedup and efficiency plots show that the implementations scale very similarly.

One peculiar data point in the data gathered was captured in both the blocking and non-blocking implementations for when the process count was equal to 20. Both of these versions when ran with 20 processes had greater than 1.00 efficiency, meaning that adding processes increased the performance of the non-parallel segments of the program. A similar situation occurred in the OpenMP implementation of the simulation, and after digging in, I noticed similar DMC node lists for this executions compared to the lower process and thread tasks, meaning that almost certainly the node lists such as DMC44 (OpenMP program execution, thread count = 16) and

DMC78 (MPI program execution, process count = 20) have higher performing CPUs than the node lists such as DMC3, DMC4, and DMC5 which were used more commonly in the lower count process and thread tasks.

The goals of the assignment were successfully achieved: both efficient blocking and non-blocking message-passing versions of the "Game of Life" simulation were implemented in C, tested with previously shown correctness and functional tests across multiple process counts, and performance was analyzed and refined through the above performance tests. Careful design went into how edge cells should be distributed and how initial and final game boards should be scattered and gathered. Additionally, a separate driver program was created as a test bed in order to more effectively test the cases indicated above, along with job scripts to queue up different MPI executions on the DMC cluster to measure performance.

References

Wikipedia – Conway's Game Of Life – Examples of Patterns (Test Case Inspiration)
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life#Examples_of_patterns

MPI_Sendrecv man page
https://www.open-mpi.org/doc/v4.0/man3/MPI_Sendrecv.3.php

MPI Collective Communications: Scatter vs. Scatterv – Cornell University
https://cvw.cac.cornell.edu/mpicc/scatterv

Point to Point Communication Routines: Non-blocking Message Passing Routines – LLNL Tutorials
https://hpc-tutorials.llnl.gov/mpi/non_blocking/

Other than these references, I also used both the textbook and Dr. Bangalore's code to learn how to use the MPI functions.

Github Repository: https://github.com/CCSanders/CS581_HW4