MPI report on Game of life

Parallel and Distributed Computing – PDC

Tiago Vaz - 96913

Erik Mišenčík - 1111466

Miguel Rocha - 1110916

Group 14

Project Description

"The Life Game" is a 3D cube where each cell, alive or dead, changes based on neighbor cells. Rules determine cell survival or death across generations, with up to 9 species affecting outcomes. The game tracks population changes over time, starting from a randomly generated setup.

Goals:

- Improve upon the parallelized code submitted in the OpenMP delivery using MPI and OpenMP hybrid.
- Achieve highest speedups and better scalability.

Part 1: Parallelization Strategy and Execution

Focus of Project: In this delivery we first had to re-construct the program to account for 2 main things. The first was the specific request that none of the processes were allowed to hold the entirety of the cube. The second was how to divide the problem and ensure correct communication between the different processes.

Parallelization Strategies:

The main struggle for this delivery was on how to distribute the cube between the different processes, what each process should have access to and what needs to be transmitted between the different processes.

- We wound up choosing to "cut" the cube in planes and distributing said planes. This deconstruction allows us to divide our cube grid into equal parts, ensuring balanced work distribution.

- **Implementation:** The cube was divided into slices, each with an equal number of planes, allowing for fair task division among processors. The number of layers per process corresponds to n/p + 2. This is due to the necessity of the plane prior to the first "work" plane and of the plane after the last "work" plane to compute the values correctly. As such, at the end of each generation, we send the first working plane to the process prior to the current process and send the last working plane to the next process. Finally, we have an MPI_Reduce at the end of each generation to send their species counter to the process p – 1. We send it to p – 1, because, in the case that n isn't divisible by p, we distribute the extra planes in order, so p – 1 will always either have the same or less amount of work as the other processes.
- **Effectiveness:** This approach worked well on a large scale. Each segment needed a similar amount of work, eliminating the need for complex load balancing, however the overheads were quite significant, so in smaller tests with many processes, the speedups aren't as good.
- **Performance Findings:** We found that, on average the speedups fell off as we scaled the problem with processes but didn't increase the size of the problem. With the 64 and 32 process version having a significantly lower speedup.

Conclusion: This strategy ensured each process handled an equal part of the work. We also used a pure MPI and a hybrid MPI and OpenMP to see the speedups we could achieve. For the Hybrid version, we used 4 threads per process.

Part 2: Additional Parallelization Details and Decisions

Counting Species Occurrences:

- **Method Used:** We counted the species during the generation in the "work" portion of each process and then we utilized an MPI array reduction to group the results.

apply_grid_updates() Function:

- **Parallelization Approach:** We followed a similar approach to our OpenMP strategy, with being careful to only update the values that concerned the “work” planes of each process and not the 2 extra ones that are required.

count_species_and_simulate () Function:

- **Communication optimization:** to leverage the power of asynchronous communication, since each process must always send the first and last updated planes, we calculated them first, and send the result to each correspondent process, asynchronously, while continuing the update of the other cube planes.

Performance Analysis

Execution times:

MPI (No OpenMP)					
Processes:	64	32	16	8	1
Test 1:	2.5	2.2	3.6	6.7	29.1
Test 2:	1.7	2.9	5.9	11.0	50.3
Test 3:	5.7	11.0	21.8	43.4	198.7
Test 4:	14.3	27.7	57.6	109.9	498.5

MPI (Hybrid)					
Processes:	16	8	4	2	1
Test 1:	1.5	2.2	3.7	6.8	29.1
Test 2:	1.7	3.1	5.8	11.2	50.3
Test 3:	5.9	11.2	22.1	43.8	198.7
Test 4:	14.3	28.1	55.1	109.5	498.5

*Column in gray are the values for serial version

Speedup:

MPI (No OpenMP)				
Processes:	64	32	16	8
Test 1:	11.6	13.2	8.0	4.3
Test 2:	29.6	17.3	8.5	4.6
Test 3:	34.8	18.1	9.1	4.6
Test 4:	34.9	18.0	8.7	4.5

MPI (Hybrid)				
Processes:	16	8	4	2
Test 1:	19.4	13.2	8.0	4.3
Test 2:	29.6	17.3	8.5	4.6
Test 3:	34.8	18.1	9.1	4.6
Test 4:	34.9	17.7	9.0	4.5

Speedup Analysis:

- As we can see from the tables above, our **speedups were quite good**. Both the **solo MPI and the Hybrid** versions had extremely **similar speedups**, which makes sense considering they effectively used the **same number of physical cores**.
- The **main difference** between the 2 approaches is **test 1**, for the maximum number of cores, which makes sense, since **initializing 64 processes** has a much **larger overhead** than **initializing 16 processes with 4 threads** in each. However, for the rest of the tests the speedups are extremely similar.
- Finally, although our **baseline is lower than with solo OpenMP**, given the fact that we achieve, on average, around **p/2 speedup** (p * 2 in the case of the Hybrid version) it is **highly scalable**, as we can see from the fact that every time we **double p**, our **speedup almost doubles**, as well.

Conclusion:

In conclusion, we can observe that the **speedups are rather good**. The larger overheads from **initializing MPI and the communication** between them lead to **smaller speedups**, however the rate at which the speedups increase seem to be on par to what is expected. Although not perfect, this shows that the **program is scalable and performs better as the problem grows**. The **overheads also become increasingly smaller** as the problem grows, since the 2 additional planes we're copying will become a smaller portion of the overall cube as n increases. It is also safe to assume that the **network bandwidth for communication is the biggest bottleneck** in our program, given the fact that we're transmitting n * n planes twice per process.