OpenMP 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:

- Parallelize code submitted in the sequential delivery using OpenMP.
- Achieve highest speedups possible.

Part 1: Parallelization Strategy and Execution

Focus of Project: Parallelizing the count_species_and_simulate() function. This function processes each cell in our grid, updating values and tracking species occurrences and cell values.

Parallelization Strategies Considered:

- Option 1: Parallelizing all three nested loops using a collapse directive.
- Option 2: Parallelizing only the outermost loop.

Chosen Approach: Parallelizing the outer loop.

- We wound choosing Option 2. This method allows us to divide a complex task into simpler sections. We split our cube grid into equal parts, assigning each to a processor, ensuring balanced work distribution.
- **Implementation:** The cube was divided into segments, each with an equal number of layers, allowing for fair task division among processors.
- **Effectiveness:** This approach worked well. Each segment needed a similar amount of work, eliminating the need for complex load balancing.
- **Decision Against Collapse Directive:** We decided against using the collapse directive. Testing showed this made our system run smoother and about 16 seconds faster in larger tests.
- **Performance Findings:** Not using the collapse directive turned out to be a good move. It kept our data well-organized and made memory use more efficient.

Conclusion: This strategy ensured each processor handled an equal part of the work, prevented memory issues, and overall, sped up our program.

Part 2: Additional Parallelization Details and Decisions

Counting Species Occurrences:

- **Method Used:** For counting species within a generation, we utilized array reduction. This approach was chosen to prevent race conditions, ensuring accuracy in our parallel processing.
- **Memory Consideration:** Considering the array's small size, the memory overhead was minimal, so it didn't affect our performance.

apply_grid_updates() Function:

- **Parallelization Approach:** We followed a similar approach to our main strategy. We only parallelized the outer loop, avoiding the collapse directive based on our earlier findings.
- **Other Parallelized Primitives:**
  - **Primitives:** Beyond these functions, we also implemented parallelization in tasks like memory allocation and destruction.

- **Significance**: Although these tasks are executed only n number of times and do not majorly impact overall performance, parallelizing them contributed to the system's overall efficiency and speed.

Performance Analysis

Execution times:

OMP without Collapse				
Threads	8	4	2	1
test 1:	8.0	7.7	14.9	29.1
test 2:	13.7	13.1	25.5	50.3
test 3:	53.6	52.1	100.9	198.7
test 4:	139.3	131.2	253.6	498.5

OMP with Collapse				
Threads	8	4	2	1
test 1:	8.1	8.9	15.9	30.6
test 2:	14.1	14.2	26.7	53.0
test 3:	55.5	56.0	104.5	204.9
test 4:	146.4	141.8	269.7	518.7

*Grey color is time of Sequential Code

Speedup:

OMP without Collapse			
Threads:	8	4	2
Test 1:	3.64	3.78	1.95
Test 2:	3.67	3.84	1.97
Test 3:	3.71	3.81	1.97
Test 4:	3.58	3.80	1.97

OMP with Collapse			
Threads:	8	4	2
Test 1:	3.78	3.44	1.92
Test 2:	3.76	3.73	1.99
Test 3:	3.69	3.66	1.96
Test 4:	3.54	3.66	1.92

Speedup Analysis:

- As we can see from the tables above, our **speedups were exceptionally good**. We managed to get **extremely close to p speedup**, which signals the great parallelization we managed to achieve, especially when it came to the bigger tests.
- **Important to note** that the results for the **8 threads** shouldn't be taken at face value, since we ran it on a **machine with only 4 cores**. We elected to keep the results on the tables to show the effects of trying to overclock the machine and processors.

Conclusion:

In conclusion, we can observe that, whenever we **double the number of threads** available, the observed **speedup also close to doubles**. This makes sense because, in the submitted solution, **most of the execution is parallelized**. Also, we **don't observe a linear double** of performance because **there are always overheads** when creating and destroying threads and **certain parts** of the code **are not parallelizable**.