UWA-CITS5507

HIGH PERFORMANCE COMPUTING

PROJECT 1 – REPORT 2023

AUTHORS:
PRITAM SUWAL SHRESTHA (23771397)
RASPREET KHANUJA (23308425)

ABSTRACT

This report presents the results of experimentation aimed at evaluating the execution time of a simulation program designed to replicate fish school behaviour on the Setonix platform. The experiments focused on both sequential and parallel versions of the program. The primary objectives were to analyse speedup, efficiency, and the impact of various parameters, including the number of threads and fishes, as well as different thread scheduling methods.

The analysis of results reveals challenges in achieving significant speedup, indicating the presence of parallelization overhead. The choice of thread scheduling methods did not lead to substantial improvements in execution time. The report underscores the importance of considering factors beyond scheduling, such as the inherent sequential nature of certain simulation steps and potential bottlenecks.

Overall, the experiments contribute to a deeper understanding of parallel computing in the context of simulation and highlight areas for further investigation and optimization.

TABLE OF CONTENTS

Experiments with Sequential Program

Purpose: Explore the time taken to execute the sequential program with different combinations of NUM_FISHES and NUM_SIMULATION_STEPS on Setonix.

Experimentation Scenarios

1. Baseline Scenario

Description: Establish a baseline performance for program with large input values.

- `NUM_FISHES`: 100
- `NUM_SIMULATION_STEPS`: 1000

Time Taken: 0.0094 seconds

Observations: With 100 fish and 1000 simulation steps, the program executed very quickly, taking only 0.0094 seconds.

2. Scaling NUM_FISHES

Description: Evaluate the impact of varying the number of fish while keeping the number of simulation steps constant.

- Keeping NUM_SIMULATION_STEPS` constant: 1000
- Varying `NUM_FISHES`

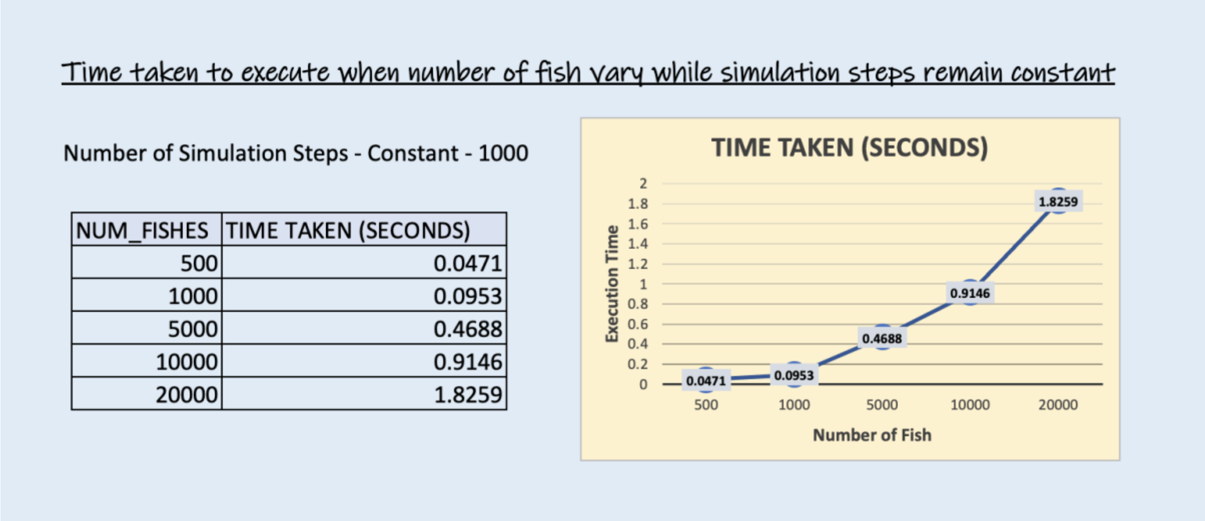
i. 500
Time Taken: 0.0471 seconds.

ii. 1000
Time Taken: 0.0953 seconds.

iii. 5000
Time Taken: 0.4688 seconds.

iv. 10000
Time Taken: 0.9146 seconds.

v. 20000
Time Taken: 1.8259 seconds.



Time taken to execute when number of fish vary while simulation steps remain constant

Number of Simulation Steps - Constant - 1000

NUM_FISHES	TIME TAKEN (SECONDS)
500	0.0471
1000	0.0953
5000	0.4688
10000	0.9146
20000	1.8259

Observations:

- As we increased the number of fish from 500 to 20000, the execution time increased from 0.0471 seconds to 1.8259 seconds.
- This indicates that increasing the number of fish led to a linear increase in execution time. The program had to simulate the behaviour of more fish, which required more computation time.

3. Scaling NUM_SIMULATION_STEPS

Description: Assess how program performance changes with different simulation steps while keeping number of fish constant.

- Keeping `NUM_FISHES` constant: 1000
- Varying `NUM_SIMULATION_STEPS`

i. 500
Time Taken: 0.0493 seconds.

ii. 1000

Time Taken: 0.0928 seconds.

iii. 5000

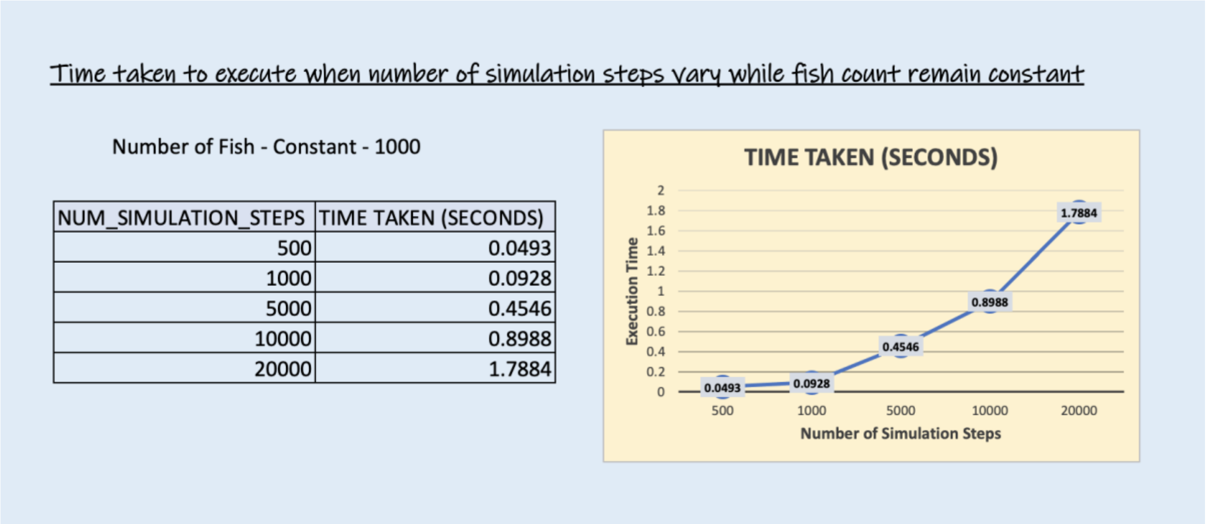
Time Taken: 0.4546 seconds.

iv. 10000

Time Taken: 0.8988 seconds.

v. 20000

Time Taken: 1.7884 seconds.



Time taken to execute when number of simulation steps vary while fish count remain constant

Number of Fish - Constant - 1000

NUM_SIMULATION_STEPS	TIME TAKEN (SECONDS)
500	0.0493
1000	0.0928
5000	0.4546
10000	0.8988
20000	1.7884

Observations:

- As we increased the number of simulation steps from 500 to 20000, the execution time increased from 0.0493 seconds to 1.7884 seconds.
- Like scenario 2, this shows a linear relationship between the number of simulation steps and execution time. Larger simulations require more time to complete.

4. High-Load Scenarios

Description: Test the program under high-load conditions.

- `NUM_FISHES`: 10,000
- `NUM_SIMULATION_STEPS`: 100,000

Time Taken: 1.4345 minutes (86.0707 seconds)

- `NUM_FISHES`: 100,000
  - `NUM_SIMULATION_STEPS`: 15,000

Time Taken: 2.234 minutes (134.0402 seconds)

- `NUM_FISHES`: 500,000
  - `NUM_SIMULATION_STEPS`: 10,000

Time Taken: 7.5288 (451.7292 seconds)

- `NUM_FISHES`: 1000,000
  - `NUM_SIMULATION_STEPS`: 3,000

Time Taken: 4.9156 minutes (294.9383 seconds)

- `NUM_FISHES`: 900,000
  - `NUM_SIMULATION_STEPS`: 2,000

Time Taken: 2.7357 minutes (164.1436 seconds)

Observations:

- In this high-load scenario, we tested your program under extreme conditions with a large number of fish and larger number of simulation steps.
- The execution time increased significantly to 7.5288 (451.7292 seconds), especially when number of simulation steps were high.

5. Exploratory Scenarios

Purpose: Explore the program's behaviour with different combinations of number of fish and simulation steps.

-Vary `NUM_FISHES`: 100
-Vary `NUM_SIMULATION_STEPS`: 1000
Time Taken: 0.0108 seconds.

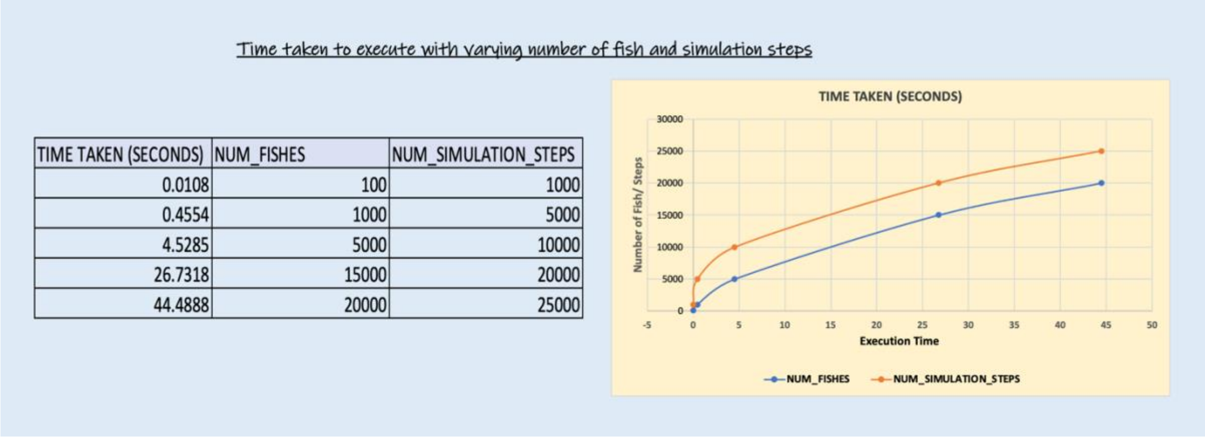
-Vary `NUM_FISHES`: 1000

-Vary `NUM_SIMULATION_STEPS`: 5000
Time Taken: 0.4554 seconds.

-Vary `NUM_FISHES`: 5000
-Vary `NUM_SIMULATION_STEPS`: 10000
Time Taken: 4.5285 seconds.

-Vary `NUM_FISHES`: 15000.
-Vary `NUM_SIMULATION_STEPS`: 20000
Time Taken: 26.7318 seconds.

-Vary `NUM_FISHES`: 20000.
-Vary `NUM_SIMULATION_STEPS`: 25000
Time Taken: 44.4888 seconds.



Time taken to execute with varying number of fish and simulation steps

TIME TAKEN (SECONDS)	NUM_FISHES	NUM_SIMULATION_STEPS
0.0108	100	1000
0.4554	1000	5000
4.5285	5000	10000
26.7318	15000	20000
44.4888	20000	25000

Observations:

- With smaller values, the execution times are relatively fast. However, as we increase the number of fish or simulation steps, the execution time increases accordingly.
- Further, it is noticed that the increase in time is more with change in simulation steps compared to change in number of fish.

Experiments with Parallel (Multi-threaded) Program

Experimentation Scenarios

Fix a (large) number of fishes, and experiment with speedup for at least five different numbers of threads.

Experiments with difference thread scheduling methods and their analyses.

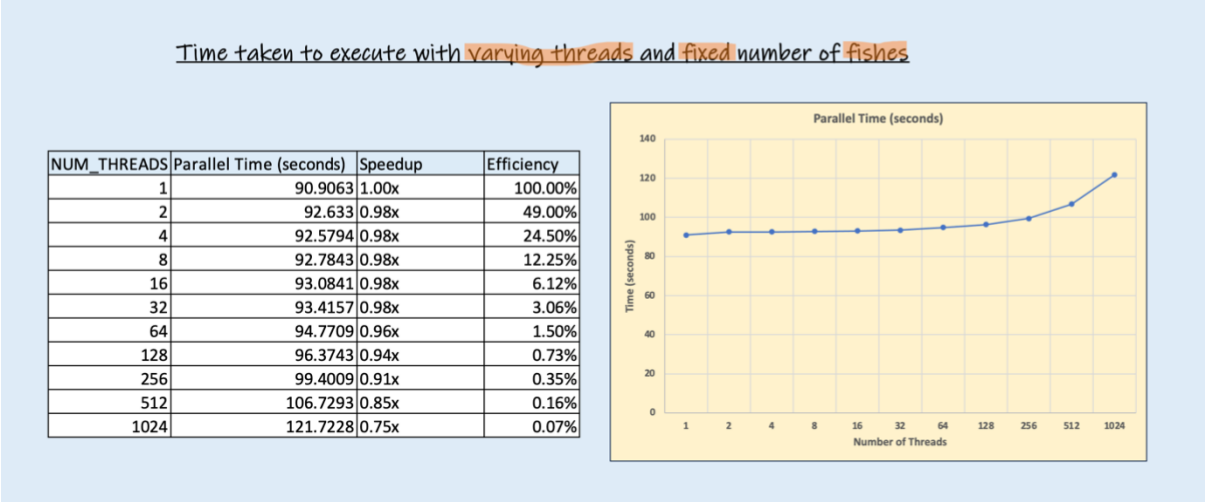
1. Variable threads, fixed fishes, and auto scheduling

Description:

- In this scenario, the OpenMP runtime autonomously determines the scheduling based on system and program characteristics.
- The number of fishes is fixed at 1,000,000, while number of threads varies from 1 to 1024.

\`NUM_FISHES\`: 1000,000
\`NUM_SIMULATION_STEPS\`: 1000
‘OMP_SCHEDULE’ = "auto"

Sequential Time: 90.9917 seconds



Time taken to execute with varying threads and fixed number of fishes

NUM_THREADS	Parallel Time (seconds)	Speedup	Efficiency
1	90.9063	1.00x	100.00%
2	92.633	0.98x	49.00%
4	92.5794	0.98x	24.50%
8	92.7843	0.98x	12.25%
16	93.0841	0.98x	6.12%
32	93.4157	0.98x	3.06%
64	94.7709	0.96x	1.50%
128	96.3743	0.94x	0.73%
256	99.4009	0.91x	0.35%
512	106.7293	0.85x	0.16%
1024	121.7228	0.75x	0.07%

Observations:

- As we increase the number of threads, the parallel program does not show significant speedup. In fact, the speedup is close to 1x (or sometimes less), indicating that the parallel version is not significantly faster than the sequential one.
- The efficiency of the parallel program decreases as we add more threads. This suggests that as we increase the number of threads, the program becomes less efficient in utilizing the available computational resources.
- The overhead of managing threads seems to outweigh the benefits of parallelism as the program achieves only 24.5% of ideal efficiency.

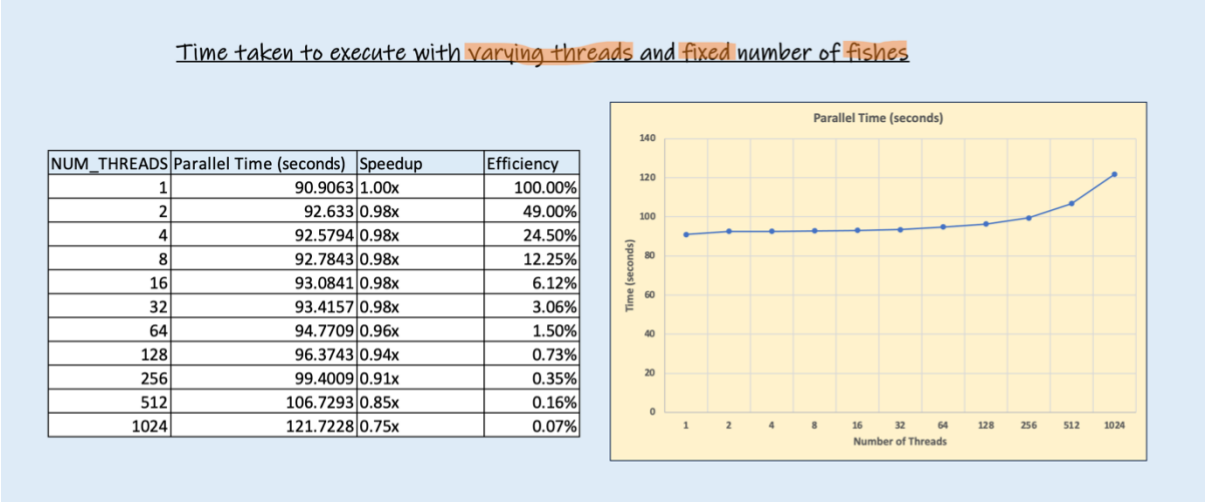
2. Variable threads, fixed fishes, and static scheduling

Description:

- The scenario employs static scheduling with a fixed chunk size of 10. It divides loop iterations into chunks of equal size and assigns these chunks to threads in a round-robin manner.
- The number of fishes is fixed at 1,000,000, while number of threads varies from 1 to 64.

`NUM_FISHES`: 1000,000
`NUM_SIMULATION_STEPS`: 1000
'OMP_SCHEDULE' = "static, 10"

Sequential Time: 93.7687



Time taken to execute with varying threads and fixed number of fishes

NUM_THREADS	Parallel Time (seconds)	Speedup	Efficiency
1	90.9063	1.00x	100.00%
2	92.633	0.98x	49.00%
4	92.5794	0.98x	24.50%
8	92.7843	0.98x	12.25%
16	93.0841	0.98x	6.12%
32	93.4157	0.98x	3.06%
64	94.7709	0.96x	1.50%
128	96.3743	0.94x	0.73%
256	99.4009	0.91x	0.35%
512	106.7293	0.85x	0.16%
1024	121.7228	0.75x	0.07%

Parallel Time (seconds)
Time (seconds)
0
20
40
60
80
100
120
140
1
2
4
8
16
32
64
128
256
512
1024
Number of Threads

Observations:

- Speedup remain consistently below 1 for all cases, indicating suboptimal parallel execution.
- Efficiency values are below 1 and relatively stable, suggesting that parallelization overhead might be affecting efficiency.
- Static scheduling with a chunk size of 10 does not seem to effectively balance the workload, contributing to performance issues.
- The program does not seem to scale efficiently with increasing threads.

3. Variable threads, fixed fishes, and dynamic scheduling

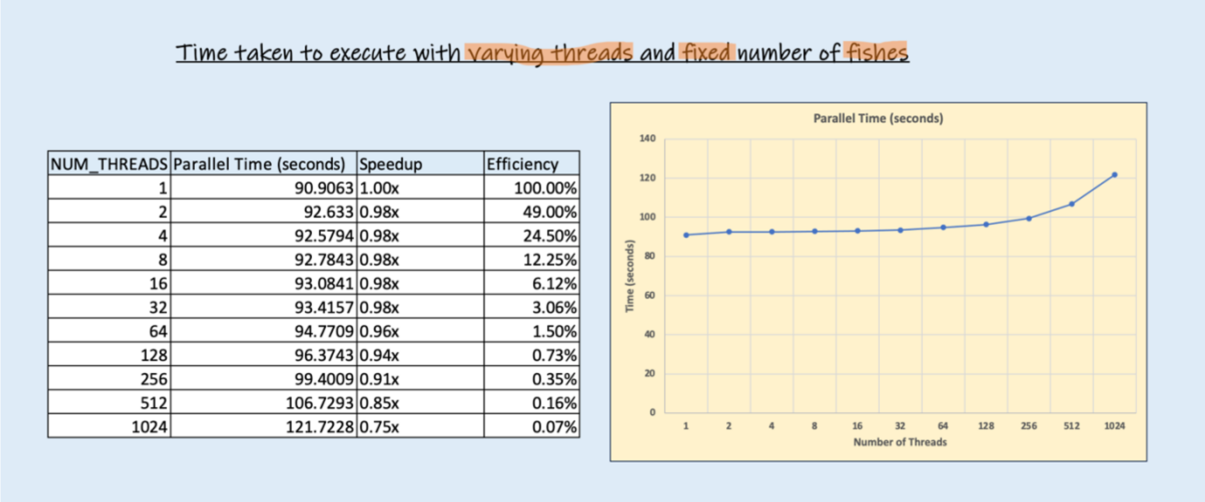
Description:

- The scenario employs dynamic scheduling with a chunk size of 100. It assigns chunks of loop iterations to threads dynamically to balance the workload.

- The number of fishes is fixed at 1,000,000, while number of threads varies from 1 to 64.

`NUM_FISHES`: 1000,000
`NUM_SIMULATION_STEPS`: 1000
‘OMP_SCHEDULE’ = " dynamic,100"

Sequential Time: 90.9846



Time taken to execute with varying threads and fixed number of fishes

NUM_THREADS	Parallel Time (seconds)	Speedup	Efficiency
1	90.9063	1.00x	100.00%
2	92.633	0.98x	49.00%
4	92.5794	0.98x	24.50%
8	92.7843	0.98x	12.25%
16	93.0841	0.98x	6.12%
32	93.4157	0.98x	3.06%
64	94.7709	0.96x	1.50%
128	96.3743	0.94x	0.73%
256	99.4009	0.91x	0.35%
512	106.7293	0.85x	0.16%
1024	121.7228	0.75x	0.07%

Observations:

- Speedup values consistently hover around 1, indicating that parallel execution provides little benefit over sequential execution.
- Efficiency values remain below 1, suggesting that parallelization overhead might be impacting efficiency.
- Dynamic scheduling does not appear to be effective in optimizing performance for this workload.
- The program does not scale efficiently with increasing threads.

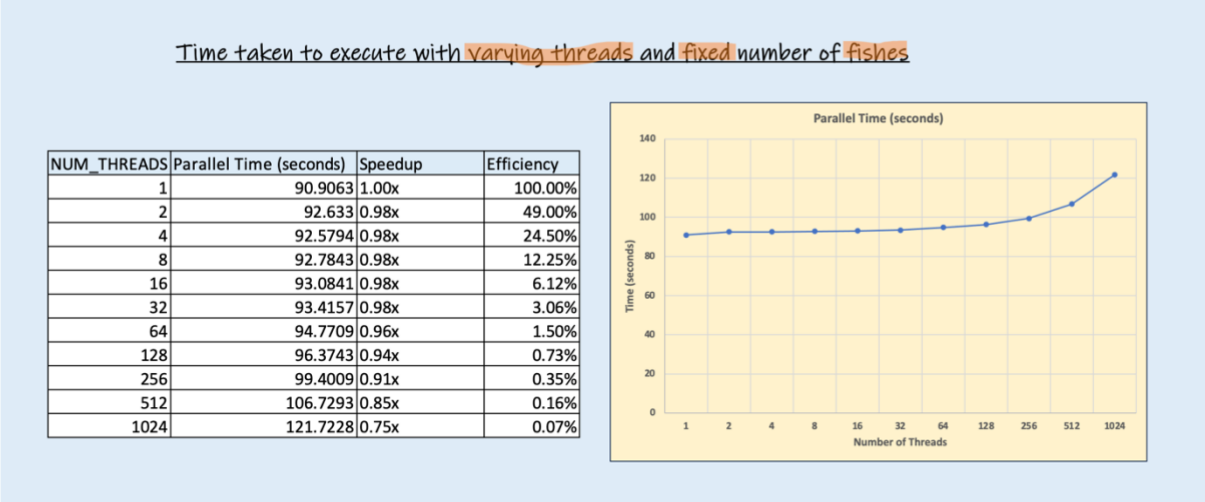
4. Variable threads, fixed fishes, and guided scheduling

Description:

- The scenario employs guided scheduling with a chunk size of 100. It initially assigns larger chunks of work to threads and gradually reduce the chuck size to balance the workload.
- The number of fishes is fixed at 1,000,000, while number of threads varies from 1 to 64.

`NUM_FISHES`: 1000,000
`NUM_SIMULATION_STEPS`: 1000
'OMP_SCHEDULE' = " guided,100"

Sequential Time: 91.0319



Time taken to execute with varying threads and fixed number of fishes

NUM_THREADS	Parallel Time (seconds)	Speedup	Efficiency
1	90.9063	1.00x	100.00%
2	92.633	0.98x	49.00%
4	92.5794	0.98x	24.50%
8	92.7843	0.98x	12.25%
16	93.0841	0.98x	6.12%
32	93.4157	0.98x	3.06%
64	94.7709	0.96x	1.50%
128	96.3743	0.94x	0.73%
256	99.4009	0.91x	0.35%
512	106.7293	0.85x	0.16%
1024	121.7228	0.75x	0.07%

Observations:

- Speedup values, like in previous scenarios, remain consistently below 1, indicating that parallel execution is not significantly faster than sequential execution.
- Efficiency values are below 1, suggesting the presence of parallelization overhead and suboptimal resource utilization.

- Guided scheduling does not appear to be effective in this context, as the program's efficiency remains low.
- The program does not scale efficiently with increasing threads.

Overall Analysis:

- Across all scenarios and scheduling methods, the program faces significant performance challenges, with speedup values consistently below 1.
- Efficiency values consistently indicate the presence of parallelization overhead, potentially affecting the efficient use of available resources.
- It appears that factors other than scheduling, such as the nature of the computation or potential bottlenecks, are influencing the program's efficiency and speedup.
- The choice of scheduling method (auto, static, dynamic, or guided) does not significantly impact the program's overall performance.

Fix number of threads, and experiment with speedup for at least five different numbers of fishes

Experiments with difference thread scheduling methods and their analyses.

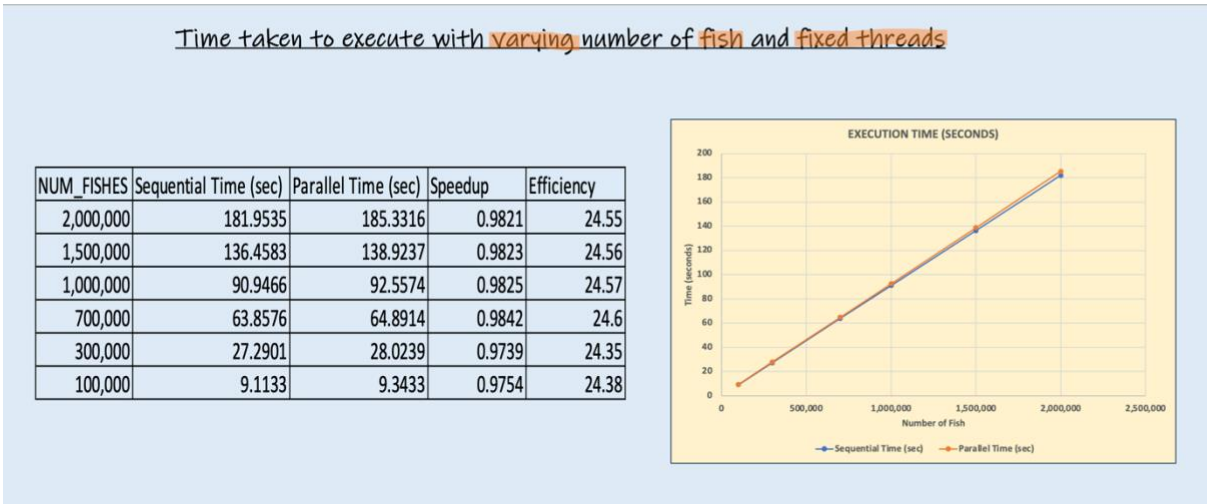
Note: In scenarios 1-4, we observed optimal performance when NUM_THREADS were set to 4. Based on this finding, we have maintained this configuration as a constant value while conducting scenarios 5-8.

5. Variable fishes, fixed threads, and auto scheduling

Description:

- In this scenario, the OpenMP runtime autonomously determines the scheduling based on system and program characteristics.
- The number of threads is fixed at 4, while number of fish varies from 2,000,000 to 100,000.

`NUM_THREADS`: 4
`NUM_SIMULATION_STEPS`: 1000
‘OMP_SCHEDULE’ = "auto"



Time taken to execute with varying number of fish and fixed threads

NUM_FISHES	Sequential Time (sec)	Parallel Time (sec)	Speedup	Efficiency
2,000,000	181.9535	185.3316	0.9821	24.55
1,500,000	136.4583	138.9237	0.9823	24.56
1,000,000	90.9466	92.5574	0.9825	24.57
700,000	63.8576	64.8914	0.9842	24.6
300,000	27.2901	28.0239	0.9739	24.35
100,000	9.1133	9.3433	0.9754	24.38

Observations:

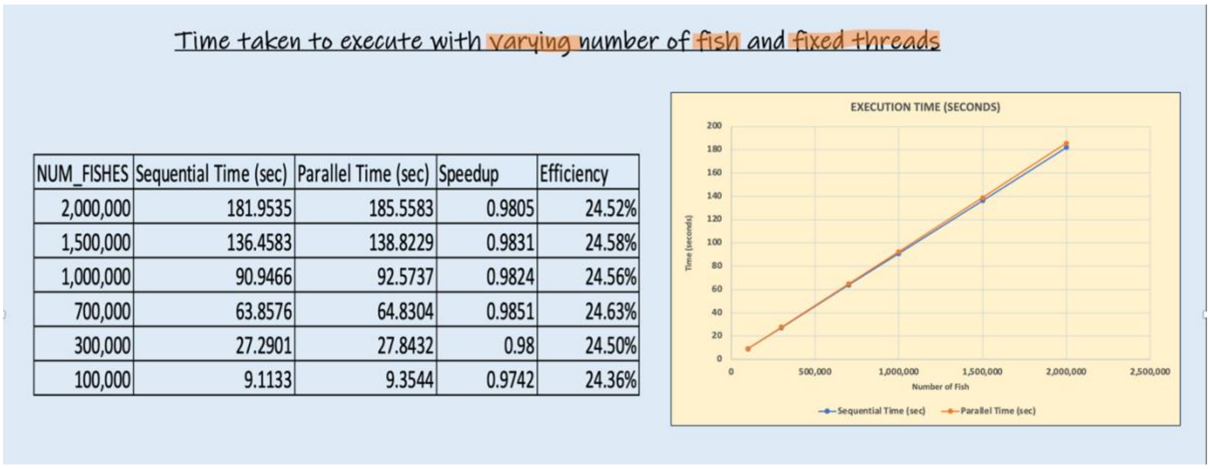
- For 2,000,000 fishes, speedup is 0.9821, indicating a minor improvement in execution time compared to the sequential version.
- As the number of fishes decreases, the speedup remains in the range of 0.9754 to 0.9842, suggesting consistent performance improvements with parallel execution.
- Efficiency values range from 24.35 to 24.60, suggesting that there is parallelization overhead, and the program's efficiency is not optimal.

6. Variable fishes, fixed threads, and static scheduling

Description:

- The scenario employs static scheduling with a fixed chunk size of 10. It divides loop iterations into chunks of equal size and assigns these chunks to threads in a round-robin manner.
- The number of threads is fixed at 4, while number of fish varies from 2,000,000 to 100,000.

`NUM_THREADS`: 4
`NUM_SIMULATION_STEPS`: 1000
‘OMP_SCHEDULE’ = "static, 10"



Time taken to execute with varying number of fish and fixed threads

NUM_FISHES	Sequential Time (sec)	Parallel Time (sec)	Speedup	Efficiency
2,000,000	181.9535	185.5583	0.9805	24.52%
1,500,000	136.4583	138.8229	0.9831	24.58%
1,000,000	90.9466	92.5737	0.9824	24.56%
700,000	63.8576	64.8304	0.9851	24.63%
300,000	27.2901	27.8432	0.98	24.50%
100,000	9.1133	9.3544	0.9742	24.36%

Observations:

- Speedup values range from approximately 0.9742 to 0.9851. The speedup is relatively modest but slightly better than in Scenario 1.
- Efficiency values range from 24.36 to 24.63, suggesting that there is still parallelization overhead, but it's relatively consistent across different numbers of fishes.

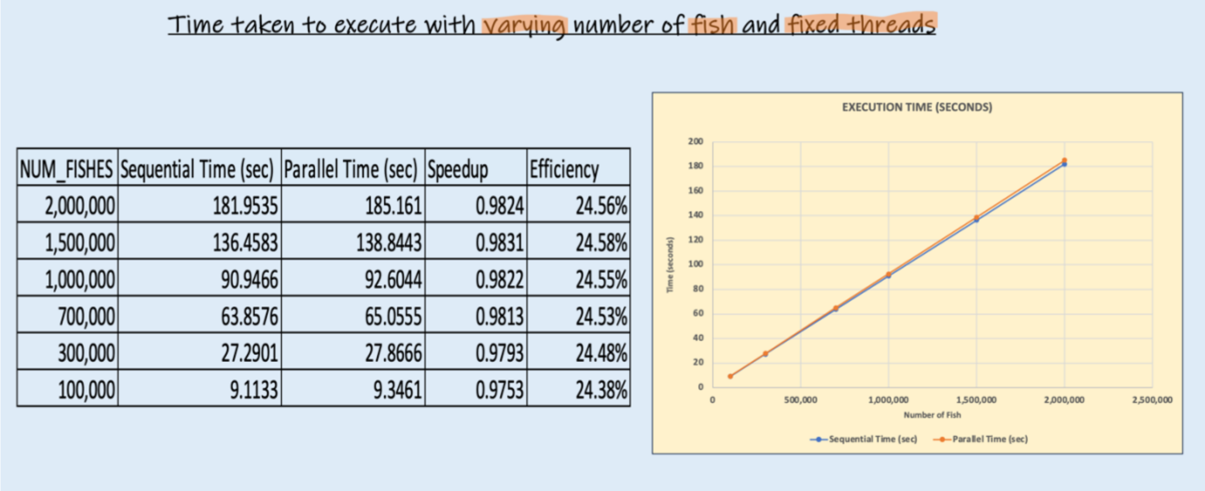
7. Variable fishes, fixed threads, and dynamic scheduling

Description:

- The scenario employs dynamic scheduling with a chunk size of 100. It assigns chunks of loop iterations to threads dynamically to balance the workload.

- The number of threads is fixed at 4, while number of fish varies from 2,000,000 to 100,000.

`NUM_THREADS`: 4
`NUM_SIMULATION_STEPS`: 1000
‘OMP_SCHEDULE’ = " dynamic,100"



Time taken to execute with varying number of fish and fixed threads

NUM_FISHES	Sequential Time (sec)	Parallel Time (sec)	Speedup	Efficiency
2,000,000	181.9535	185.161	0.9824	24.56%
1,500,000	136.4583	138.8443	0.9831	24.58%
1,000,000	90.9466	92.6044	0.9822	24.55%
700,000	63.8576	65.0555	0.9813	24.53%
300,000	27.2901	27.8666	0.9793	24.48%
100,000	9.1133	9.3461	0.9753	24.38%

Observations:

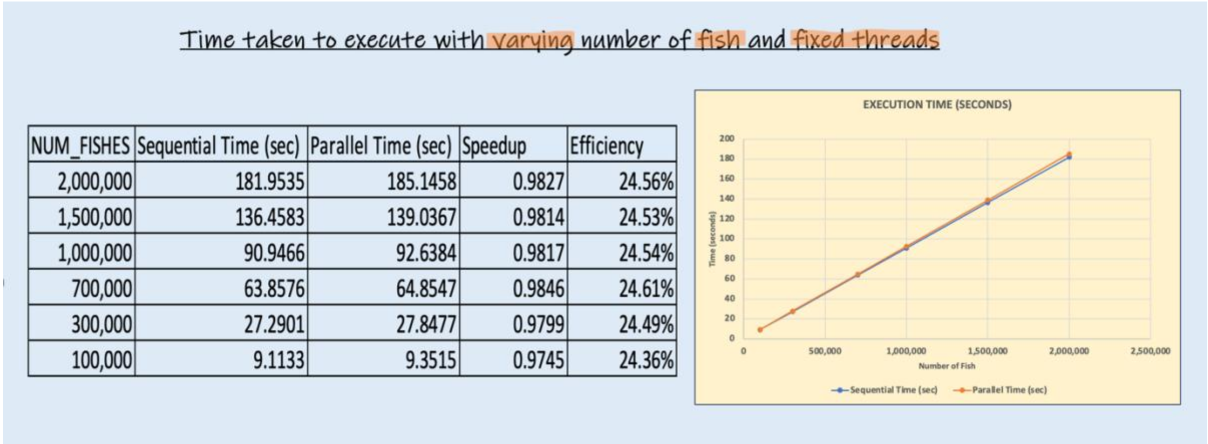
- Speedup values range from approximately 0.9753 to 0.9831, indicating that parallel execution is slightly faster than the other scenarios.
- Efficiency values are between 24.38 and 24.58, suggesting that there is still parallelization overhead, but it's relatively consistent across different numbers of fishes.

8. Variable fishes, fixed threads, and guided scheduling

Description:

- The scenario employs guided scheduling with a chunk size of 100. It initially assigns larger chunks of work to threads and gradually reduce the chuck size to balance the workload.
- The number of threads is fixed at 4, while number of fish varies from 2,000,000 to 100,000.

`NUM_THREADS`: 4
`NUM_SIMULATION_STEPS`: 1000
‘OMP_SCHEDULE’ = " guided,100"



Time taken to execute with varying number of fish and fixed threads

NUM_FISHES	Sequential Time (sec)	Parallel Time (sec)	Speedup	Efficiency
2,000,000	181.9535	185.1458	0.9827	24.56%
1,500,000	136.4583	139.0367	0.9814	24.53%
1,000,000	90.9466	92.6384	0.9817	24.54%
700,000	63.8576	64.8547	0.9846	24.61%
300,000	27.2901	27.8477	0.9799	24.49%
100,000	9.1133	9.3515	0.9745	24.36%

Observations:

- Speedup values range from approximately 0.9745 to 0.9846, indicating that parallel execution is like the other scenarios.
- Efficiency values range from 24.36 to 24.61, suggesting that there is still parallelization overhead, but it's relatively consistent across different numbers of fishes.

Overall Analysis:

- Across all scenarios and scheduling methods, the program faces significant performance challenges, with speedup values consistently below 1.
- Efficiency values consistently indicate the presence of parallelization overhead, potentially affecting the efficient use of available resources.
- It appears that factors other than scheduling, such as the nature of the computation or potential bottlenecks, are influencing the program's efficiency and speedup.

- The choice of scheduling method (auto, static, dynamic, or guided) does not significantly impact the program's overall performance.

Factors Affecting Parallel Execution Performance

We think that the following points could be potential reasons for the limited improvement in parallel execution:

1. **Parallelization Overhead:** Creating and managing threads, as well as synchronizing them can introduce overhead. In our scenarios, this overhead may be a significant portion of the execution time, leading to slower parallel execution.

2. **Bottlenecks:** Parallel programs often have bottlenecks, which are parts of the code that cannot be parallelized effectively. If there are sections of the code where computations depend on previous results or sequential processing is required, these bottlenecks can limit the overall speedup.

3. **Amdahl's Law:** Amdahl's Law states that the speedup of a program is limited by the portion of the code that cannot be parallelized. Even if most of the code can be parallelized, the presence of a sequential component can severely limit the potential speedup.
   In our case, below part of the code hasn’t been parallelized as it is inherently sequential and has dependencies between each step. This could also be one of the reasons for the lack of improvement performance.

```
int main() {

  // CODE HERE

 for (int step = 0; step < NUM_SIMULATION_STEPS; step++) {

   // TODO: COMMENT OUT THE PRINTF STATEMENT FOR EXPERIMENT
   printf("Simulation Step: %d\n", step);
   printf("#########################################################\n");
   simulationStep(fishes, NUM_FISHES);
   calculateObjectiveFunction(fishes, NUM_FISHES);
   calculateBarycenter(fishes, NUM_FISHES);
   printf("--------------------------------------------------------\n");
 }

  // CODE HERE
}
```

For the above code snippet, the inherent dependencies pose significant challenges for effective parallelization. Here are the key reasons why parallelization is not feasible in this context:

1. **Data Dependency:** In the given code, each step of the simulation (`simulationStep`), the calculation of the objective function (`calculateObjectiveFunction`), and the calculation of the barycenter (`calculateBarycenter`) depend on the results of the previous step. These calculations are inherently sequential and cannot be parallelized within the same simulation step.

2. **Sequential Dependence:** The simulation steps are sequential; each step depends on the state of the fishes after the previous step. Parallelizing the entire loop would require each step to be independent of the others, which is not the case here.

3. **Race Conditions:** If we were to parallelize the entire loop, we would introduce race conditions, as multiple threads would be trying to update and access the same `fishes` data concurrently. This would lead to unpredictable and incorrect results.