CITS3402 High Performance Computing Project 1

Parallel implementation of search based on Fish School Behaviour

Sijing Aloysius Loh 24028283

Thai Hoang Long Nguyen 23147438

Introduction:

Multi-dimensional optimization poses a significant challenge in computer science, with heuristic algorithms offering valuable approaches to finding optimal solutions within complex problem spaces. One such heuristic method, Fish School Behaviour (FSB), takes inspiration from the collective behaviour of fish to tackle optimization problems efficiently.

The project scenario envisions a vast lake inhabited by a school of fish, with food resources distributed unevenly. The objective is to guide the fish towards regions rich in food. Each fish can eat, move, and collectively orient itself with the school. Through this project, we simulate the behaviour of the fish school using parallel computing techniques.

The simulation occurs in discrete rounds, where each fish performs actions such as eating and swimming. Furthermore, the entire school collectively orients itself toward a barycenter, symbolising the school's centre of mass. This project explores the parallelization of this simulation, aiming to optimise its performance and efficiency.

In the following sections, we will delve into the algorithm, present both sequential and parallel implementations, conduct experiments with varying parameters like changing the number of fish, threads, threads scheduling and threads scheduling using Setonix platform also we will be conducting other experiments using different Operating System, and analyse the results.

Algorithm

In the sequential program, we first create a struct for a fish which holds the x and y coordinates, its current weight, and the last change in objective function. We then create an array for a school of fish and dynamically allocate memory in the heap for

that array. In the array, each fish is given a random x and y coordinate and a predetermined initial starting weight of 10.
Next, we start the timer and begin the simulation. The main ‘for’ loop runs each step of the simulation. In each iteration of the simulation loop, there is a nested ‘for’ loop where we calculate the current objective of each fish, then change its x and y coordinates by a random amount between 0 and 0.1. With the new x and y coordinates, we can calculate the new objective function and subsequently, the difference between the old and new objective functions, ‘delta_fi_max’.

In the second nested ‘for’ loop, we iteratively compare the change in objective function of each fish to determine the largest change in objective function.

In the final nested ‘for’ loop, we calculate the new weight of each fish with a restriction between 0 and 2*10. Finally we calculate the barycentre of the pond.

This is repeated for every step of the simulation and once the last step finishes, we stop the timer, free the memory and end the program.

In the parallel program, the first nested ‘for’ loop is parallelised using ‘#pragma omp for’ to distribute the number of fish amongst each thread. Each thread is allocated a certain number of fish to calculate the objective function, generate new coordinates, and calculate the change in objective functions for.

Reduction with the ‘max’ operator on ‘delta_fi_max’ is used for the second nested ‘for’ loop to parallelise finding the maximum change in objective function. Reduction is required for data sharing among the threads for correct determination of the maximum change in objective function.

Reduction with the ‘+’ operator on the barycentre’s numerator and denominator is also used for the last nested ‘for’ loop to parallelise the calculation of fish weight and the barycentre.

Experiment and Analysis

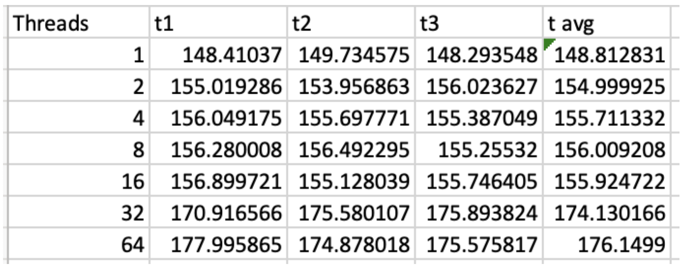
Varying number of threads

Table 1 and Graph 1 presents data comparing the execution times of FSB on the Setonix supercomputer. The experiments were conducted using a fixed number of 1,000,000 fish, with the only varying parameter being the number of threads, ranging from 1 to 64, while maintaining a constant simulation length of 3000 steps.

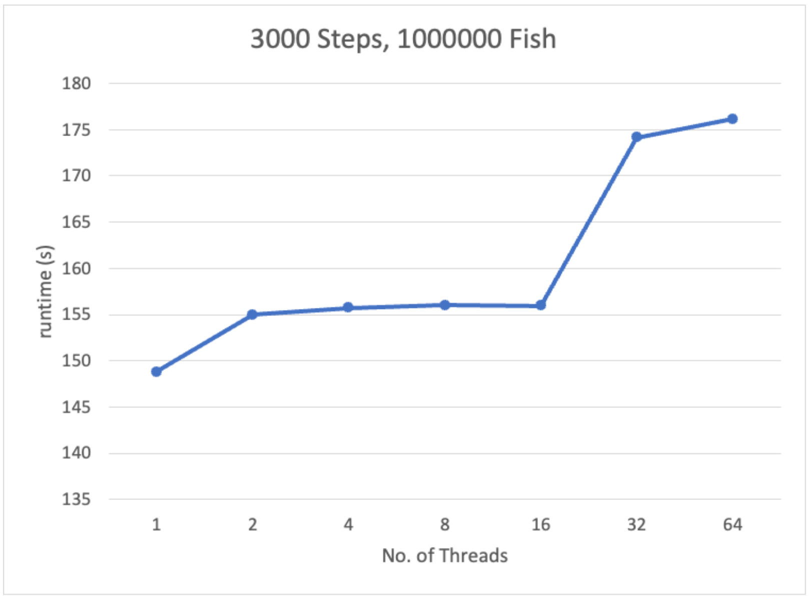
It is observed that in general, the execution time increases as the number of threads increases. This may seem counterintuitive to what a parallel program is supposed to do, since more threads should mean that each thread does less work leading to a lower execution time. However, our observation is likely due to the overheads of parallelisation.

In parallel programs, thread creation, synchronisation and management may have overheads. This is a likely explanation of why the sequential implementation is faster than the parallel implementation regardless of the number of threads allocated.

Table 1 and graph 1



Threads	t1	t2	t3	t avg
1	148.41037	149.734575	148.293548	148.812831
2	155.019286	153.956863	156.023627	154.999925
4	156.049175	155.697771	155.387049	155.711332
8	156.280008	156.492295	155.25532	156.009208
16	156.899721	155.128039	155.746405	155.924722
32	170.916566	175.580107	175.893824	174.130166
64	177.995865	174.878018	175.575817	176.1499



Varying number of fish

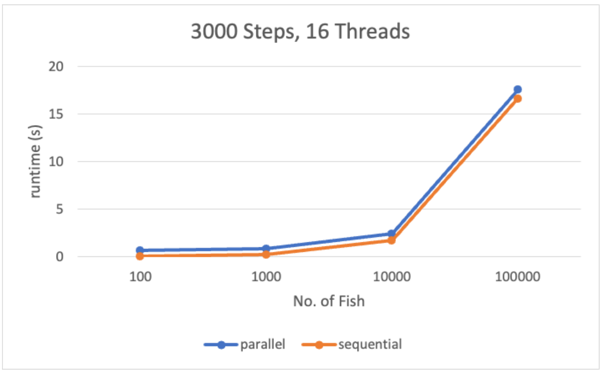
Table 2 and Graph 2 presents data comparing the execution times of parallel and sequential implementations of the FSB on the Setonix supercomputer. The experiments were conducted using a fixed number of 16 threads for both implementations, with the only varying parameter being the number of fish in the simulation, ranging from 100 to 1,500,000, while maintaining a constant simulation length of 3000 steps.

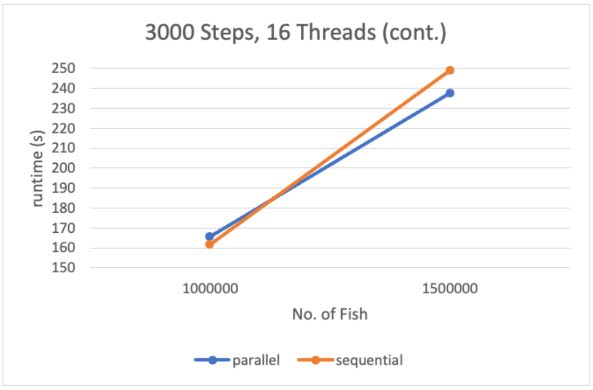
First, it's evident that as the number of fish increases, there is a natural increase in execution time for both parallel and sequential implementations. This behaviour is consistent with the idea that simulating more entities (in this case, fish) requires more computational resources and time.
It is also observed that the execution time of the sequential implementation is lower than that of the parallel one between 100 and 1,000,000 fish. This is likely due to the same reasons mentioned above. However, the difference in execution times gets proportionally smaller as the number of fish increases. The parallel program eventually becomes faster in execution when the number of fish is set to 1,500,000. This is consistent with our theory that the

parallel program was experiencing some overheads which was causing it to execute slower than the sequential one.

Table 2 and graph 2

Fish	t1	t2	t3	parallel	sequential
100	0.622065	0.622087	0.622619	0.622257	0.017915
1000	0.785081	0.786097	0.78248	0.7845527	0.167837
10000	2.359709	2.36375	2.399141	2.3742	1.673823
100000	18.048874	18.050807	16.511226	17.536969	16.61842
1000000	170.900622	161.246245	164.981733	165.70953	161.6427
1500000	245.136412	233.863834	233.757665	237.58597	248.9636





Different scheduling methods

In the third experiment, we embarked on a parallel implementation with a fixed number of 1,000,000 fish and a constant set of 16 threads. However, what set this experiment apart was the exploration of different scheduling methods: Static, Guided, and Dynamic. These diverse strategies were tested for their execution times in the context of the Fish Simulation Benchmark (FSB).

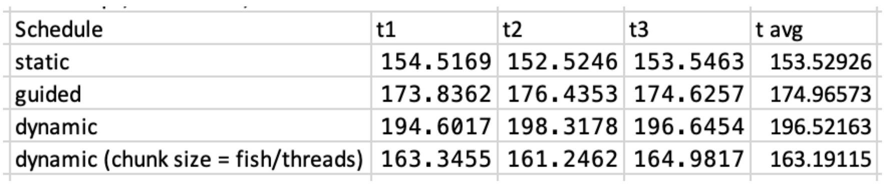
The results, as revealed in Table 3 and Graph 3, showcased that Static Scheduling emerged as the fastest with an average execution time of 153.5 seconds. The rationale behind this efficiency lies in its even distribution of workload among threads before execution, resulting in remarkably uniform execution times across multiple runs. This uniformity and reliability are paramount when predictability in execution times is a priority.

On the other hand, Guided Scheduling introduced adaptability to the workload by starting with larger chunk sizes and dynamically adjusting them as the simulation progressed. However, this adaptability came at a slight cost, with an average execution time of 175.0 seconds compared to Static Scheduling.

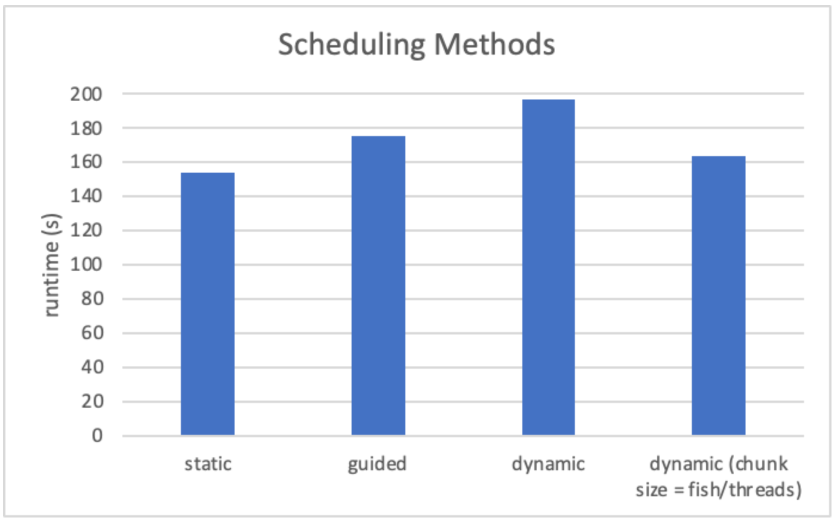
Dynamic Scheduling, while highly flexible and beneficial in scenarios with unpredictable or skewed workloads, incurred overhead due to frequent chunk allocation and reallocation. Consequently, it posted the longest execution times among the four scheduling methods, averaging 196.5 seconds. After further experimentation, we decided on a chunk size of 62,500 which seems to have given promising results. It achieved an average execution time of 163.2 seconds.

In summary, the choice of thread scheduling strategy should align with specific workload characteristics and performance objectives. Static Scheduling offers uniformity and reliability, Guided Scheduling introduces adaptability at a slightly increased execution time and Dynamic Scheduling is highly flexible but comes with overhead. Each method has its strengths and should be chosen based on the specific needs of the parallel application at hand.

Table 3 and graph 3



Schedule	t1	t2	t3	t avg
static	154.5169	152.5246	153.5463	153.52926
guided	173.8362	176.4353	174.6257	174.96573
dynamic	194.6017	198.3178	196.6454	196.52163
dynamic (chunk size = fish/threads)	163.3455	161.2462	164.9817	163.19115



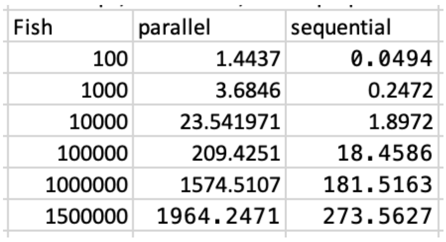
Additional experiment: Dell personal computer vs Pawsey Supercomputer

In the final experiment, a replication of the initial scenario was conducted, utilising a fixed number of threads but varying the number of fish. This time, the experiment was executed on a Dell laptop running Ubuntu as the operating system, in contrast to the previous Setonix supercomputer environment. The primary objective was to observe whether different operating systems would yield distinct results. A comparison between the data from Table 2 (Setonix) and Table 4 (Dell's Computer) reveals notable differences in execution times, both for parallel and sequential simulations.

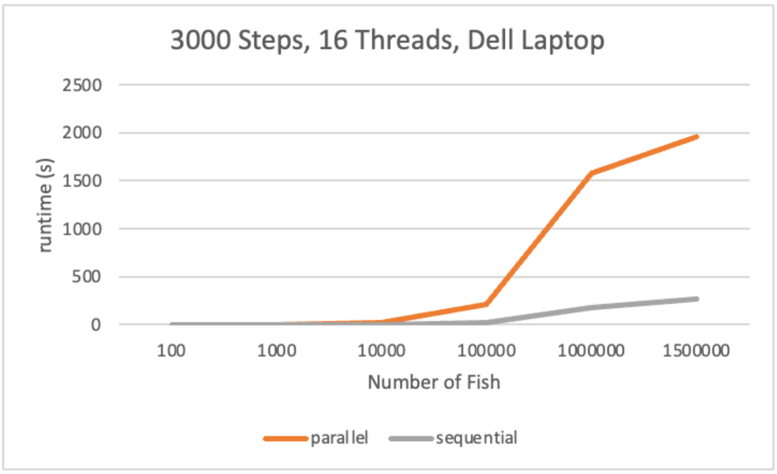
In the Setonix environment, parallel execution times were similar to their sequential counterparts. This was expected, given the computational power of a supercomputer. For instance, with 1,000,000 fish, the parallel execution time averaged 165.71 seconds, while the sequential execution time was 161.64 seconds.

On the Dell laptop, the disparity between parallel and sequential execution times was significantly more pronounced. With 1,000,000 fish, the parallel execution time averaged 1574.51 seconds, while the sequential execution time was 181.52 seconds. Additionally, both parallel and sequential execution times were notably longer compared to Setonix, which was expected due to the differences in computing resources.

Table 4 and graph 4



Fish	parallel	sequential
100	1.4437	0.0494
1000	3.6846	0.2472
10000	23.541971	1.8972
100000	209.4251	18.4586
1000000	1574.5107	181.5163
1500000	1964.2471	273.5627



Conclusion:

In conclusion, this comprehensive exploration of the FSB simulation using parallel computing has provided valuable insights into its performance and efficiency under varying conditions and environments.

In the initial experiments, we observed the impact of changing the number of threads and fish on execution times. It became evident that while, in theory, parallel processing offers significant advantages, the overhead associated with it can offset these gains, especially on smaller scale computations.

The analysis of different thread scheduling methods further highlighted the importance of selecting an appropriate strategy based on workload characteristics. Static scheduling excelled in providing consistent and reliable execution times, while guided scheduling offered adaptability at a slightly increased cost. Dynamic scheduling, while flexible, introduced overheads.

Finally, the comparison between the Setonix environment and a Dell computer revealed the substantial influence of computational resources. Setonix demonstrated faster execution times across the board while the difference in execution times between parallel and sequential programs were more significant on the Dell computer.

In summary, this project underscores the significance of thoughtful design choices, including thread management and scheduling strategies, to harness the full potential of parallel processing in FSB simulations. The results offer valuable guidance for optimising simulation performance and scalability across various computational environments.