CITS5507 Project 1 Report

Joo Kai Tay (22489437)

2023-09-18

# Project Overview

This project aims to test various sequential and parallel implementations of search based on the Fish School Behavior (FSB) algorithm. The performance for each experiment will be measured by the time taken to execute the program given a number of fish and steps.

# Fish simulation

## Fish Structure

The file fish.h contains the definition of a fish in this simulation. The struct Fish contains the following parameters:

* **double euclDist**: This is the euclidean distance of the fish from the origin in the previous step
* **double x\_c**: This holds the x coordinate of the fish in the current step
* **double y\_c**: This holds the y coordinate of the fish in the current step
* **double weight\_c**: This holds the weight of the fish in the current step
* **double weight\_p**: This holds the weight of the fish in the previous step

typedef struct fish {  
 double euclDist;  
 double x\_c;  
 double y\_c;  
 double weight\_c;  
 double weight\_p;  
} Fish;

## Fish methods

fish.h also contains declarations for functions that are implemented in fish.c

This function dynamically allocates memory in the heap for an array of Fish structures and initializes each fish with random coordinates within a 200x200 square which represents the pond. The parameter numfish is used to specify the number of fish instead of a #define NUMFISH C preprocessor constant to allow for experiments with different numbers of fish.

Fish\* initializeFish(int numfish)

This function updates the weight of a fish during a simulation step. It takes 3 parameters:

* **Fish\* fish1**: A pointer to the fish that will be updated
* **double maxObj**: This is the maximum difference in distance between the previous step and the current step
* **int step**: The current step of the function

void eat(Fish\* fish1, double maxObj, int step)

This function updates the position of a fish instance if its current weight is less than two times the STARTWEIGHT. The new position is calculated randomly within -0.1 and 0.1 and is then clamped to ensure it remains within a 200 x 200 square.

void swim(Fish\* fish1)

# Experiment Methodology

The experiments make use of functions declared in sequential.h and parallel\_functions.h. Each experiment has its corresponding experimentX.c and experimentX.sh files. To run an experiment, SSH into Setonix and use the command: sbatch experimentX.sh to run the desired experiment. The reason for using this method of breaking up the experiments is due to the limits of computation time imposed by Setonix.

Each experiment was ran 10 times and the results shown are an average of those 10 simulations.

NUMSTEPS and NUMFISH are not defined using C preprocessor definitions for the ease of testing using different values of each.

## Base Case - Sequential

The base case for this simulation was implemented using the void sequential(Fish\* fishArray, int numfish, int numsteps) function implemented in sequential.c. The FSB problem was tackled as such:

1. The outer loop runs for a number of iterations equal to NUMSTEPS.
2. The first of the inner loops iterates over all the fish present and finds the maximum difference in the position of the fish in the current and the fish in the previous round.
3. The second of the inner loops uses the maximum difference found in step 2 and performs the eat and swim operations on each fish in the array.
4. The final of the inner loops calculates the numerator and denominator variables for the barycentre.
5. The barycentre is calculated using the values from step 4.

## Base Case - Parallel

The base case for parallel functions in this simulation is the void parallelReduction(Fish\* fishArray, int numfish, int numsteps) function implemented in parallel\_functions.c.

1. The first inner loop is parallelized using #pragma omp for reduction(max:maxDiff) inside the parallel region. The reduction clause ensures that the maxDiff variable is updated in a thread-safe manner, to avoid any race conditions when updating the variable.
2. The second inner loop is parallelized using #pragma omp for as the eat and swim operations are thread-safe, therefore, no special clauses need to be attached to this loop.
3. The last inner loop is also parallelized using #pragma omp for reduction(+:sumOfProduct,sumOfDistance) to avoid race conditions when updating the two variables.

# Experiment 0 - Memory Caching

* Source file: experiment0.c
* Run with: sbatch experiment0.sh
* Number of threads: 4
* Number of steps: 10,000
* Number of fish: 10,000

## Sequential Memory Cache

The first experiment conducted was related to memory caching. Section 3.1 describes the base case sequential function that has 3 inner loops. We attempt to improve performance by condensing the three inner loops into two. The following 2 functions were used in this experiment and their implementations can be found in sequential.c.

1. void sequentialCondensed(Fish\* fishArray, int numfish, int numsteps): The original version of the sequential code with 3 inner loops as described in section 3.1.
2. void sequential(Fish\* fishArray, int numfish, int numsteps): The sequential code that condenses the eat, swim and accumulation of barycentre variables into one loop.

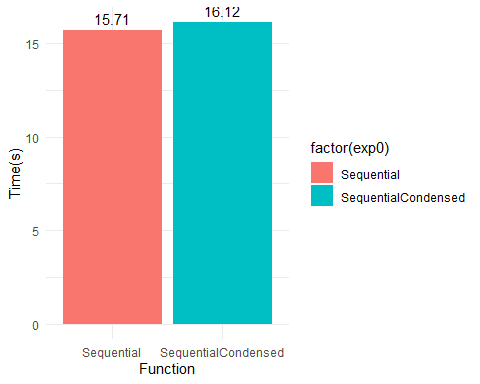
The plot below shows the result of this experiment. The sequential function with 3 inner loops outperforms the function with 2 inner loops by a clear margin. This might be due to the spatial and temporal locality of accessing memory.

Spatial locality refers to the tendency of the program to access data that is located close to the data it has accessed recently.

In the sequential function, each loop accesses and modifies a different set of properties of the Fish objects in the array. This could potentially lead to better spatial locality, as each loop would be working within a smaller memory footprint, allowing more efficient use of the cache and better prefetching.

In the sequentialCondensed function, different properties of the Fish objects are accessed and modified within the same loop. This can potentially result in lower spatial locality and more cache misses, as the memory accessed in the loop is more scattered, and the CPU needs to load different parts of the objects into the cache more frequently.

The results of this experiment show that reducing the number of loops might not be the best option. In cases where the dataset is large enough that it doesn’t fit well into the cache, having better spaital locality improves performance.



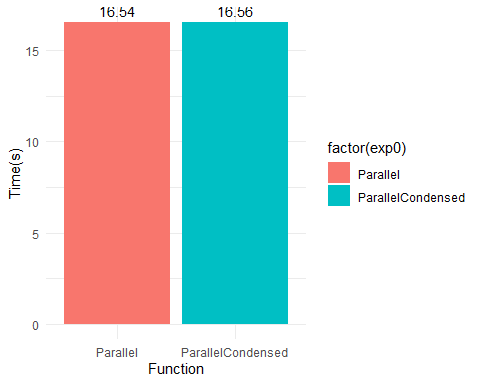
## Removing Parallel Regions

Section 4.1 demonstrates that removing and inner loop may degrade performance due to worse spatial locality of memory. However, this is more complicated when it is applied to the parallel code as described in section 3.2. By removing one of the inner loops, we also remove the associated parallel region and the overheads that come with it, such as thread creation and destruction as well as the implicit barrier that comes at the end of a parallel region. Therefore, the hypothesis is that the difference in the parallel code would be smaller than the difference in the sequential code. The following 2 functions were used in this experiment and their implementations can be found in parallel\_functions.c

1. void parallelReduction(Fish\* fishArray, int numfish, int numsteps): The original version of the parallel code with 3 inner loops and parallel regions as described in section 3.2.
2. void parallelReductionCondensed(Fish\* fishArray, int numfish, int numsteps): The parallel code that condenses the eat, swim and accumulation of barycentre variables into one loop and removes the associated parallel region.

The plot below shows the results of this experiment. It can be observed that the difference between the two functions is negligible. This confirms the hypothesis that the removal of parallel regions would give a performance benefit to the function. In this case, it was enough to close the gap between the two different functions when compared to the sequential versions.

For clarity, any comparison against the base functions in future experiments will be done against sequential and parallelReduction.



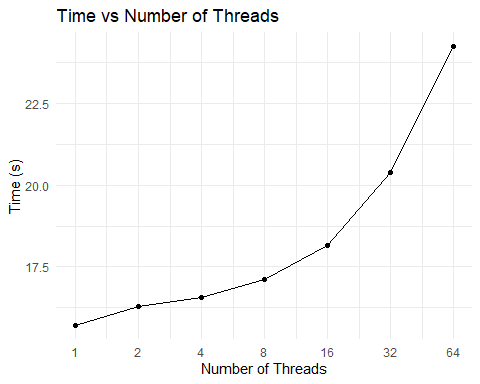
# Experiment 1 - Time taken as number of threads increase

Experiment 1 investigates the effect of changing the number of threads on the time take for computation. This experiment makes use of the base case functions for both sequential and parallel as described in sections 3.1 and 3.2. The number of steps and fish are held constant and the only variable changed is the number of threads as defined by setting the environment variable OMP\_NUM\_THREADS.

* Source file: experiment1.c
* Run with: sbatch experiment1.sh
* Number of threads: 1, 2, 4, 8, 16, 32, 64
* Number of steps: 10,000
* Number of fish: 10,000

The graph below shows the results of the experiment. It can be observed that the increase in time as the number of threads increase is exponential. One possible conclusion that can be drawn from this is that the overhead introduced by parallelization which includes thread creation and synchronization outweighs the benefits that parallel computing provides in this case. As the computation done in each of the inner loops is very small, as the number of threads increase, the overhead from handling all those threads vastly outstrips the performance gains from parallelization. This is why we see the exponential growth in time taken as the number of threads increase.

Future experiments will be run with 4 threads as with higher number of threads, it would not be easy to determine if any performance penalties come from having too many threads or from the additional constructs being tested.

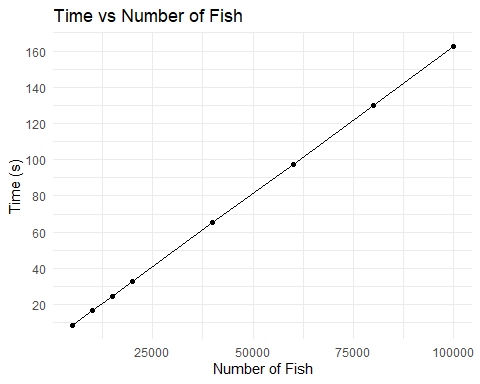


# Experiment 2 - Investigating time vs number of fish

Experiment 2 investigates the increase in time taken as the number of fish increases. This experiment makes use of the base case parallel function. The number of steps and threads are held constant and the only variable changed is the number of fish.

* Source file: experiment2.c
* Run with: sbatch experiment2.sh
* Number of threads: 4
* Number of steps: 10,000
* Number of fish: 5000, 10000, 15000, 20000, 25000

The plot below shows the increase in time as the number of fish increases is linear.

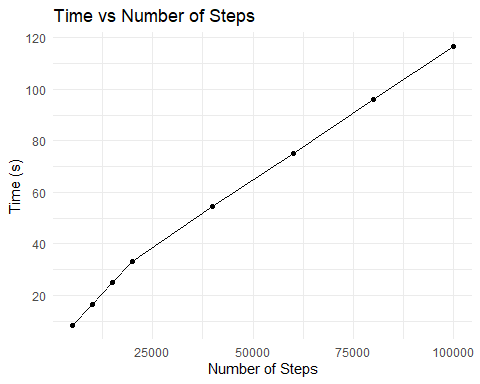


# Experiment 3 - Comparing Steps and Fishes

Experiment 3 investigates the increase in time taken as the number of steps increases. This experiment makes use of the base case functions parallel function. The number of fish and threads are held constant and the only variable changed is the number of steps.

* Source file: experiment3.c
* Run with: sbatch experiment3.sh
* Number of threads: 4
* Number of steps: 5000, 10000, 15000, 20000, 25000
* Number of fish: 10,000

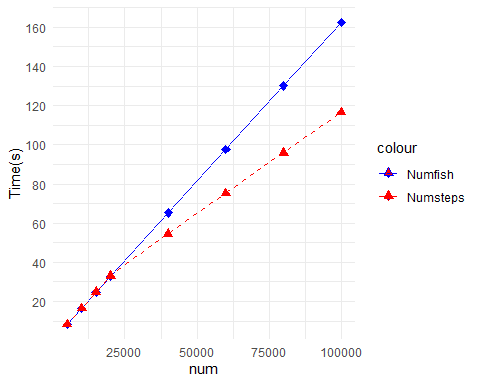
The plot below shows that the initial gradient is steep, the rate of increase in time taken slows down as the number of steps increase.



## Changing steps vs changing fishes

The following graph shows the results for experiments 2 and 3 overlayed on top of each other. It can be observed that from 5000->20000 steps and fishes, the increase in time taken is identical. However, the difference in time starts to be seen from 20000->100000 where the growth in time from increasing the number of fish starts to rapidly outpace the growth in time from increasing the number of fish.

This behavior is also something unexpected as the time complexity of the program is **O(numsteps \* numfish)**. By this complexity reasoning, we expect the function’s time taken to scale linearly as observed in the relationship between time taken and numfish. However, this relationship does not hold true when varying numsteps, which mean that there might be other factors such as hardware or memory limitations.



# Experiment 4 - Thread Scheduling

## Thread Scheduling Methods

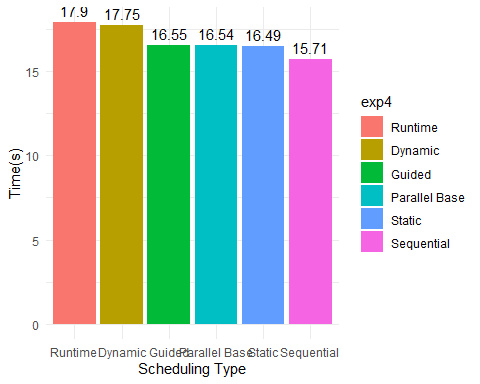
Experiment 4 investigates the difference in time taken for different scheduling methods and compares them to the base cases for both sequential and parallel. The function void parallelSchedule(Fish\* fishArray, char\* scheduleType, int numfish, int numsteps) which is implemented in parallel\_functions.c is an extension of the base case parallel function that implements four different scheduling types.

1. **Static**: Iterations are divided into equal-sized chunks and each thread gets a chunk of iterations to execute. In this experiment, the chunk size is not specified in the program, therefore, OpenMP divides the iterations into equal parts, and each thread gets num\_iterations / num\_threads iterations to execute.
2. **Dynamic**: Dynamic scheduling assigns a chunk of iterations to a thread, and once the thread is finished, it retrieves the next chunk to process. This can help balance the workload if the time to process each iteration can vary. Given that a chunksize was not specified, OpenMP assigns one iteration at a time to the threads.
3. **Guided**: Guided scheduling is a variation of dynamic scheduling where the chunk sizes decrease exponentially. Early on, threads get larger chunks, and as they finish, they get smaller chunks. This can help in situations where there’s a lot of variation in workload, as threads that finish early can take on more work.
4. **Runtime**: The scheduling type and chunk size are determined by the environment variables or runtime library routines.

* Source file: experiment4.c
* Run with: sbatch experiment4.sh
* Number of threads: 4
* Number of steps: 10,000
* Number of fish: 10,000

The chart below shows the results of the different thread scheduling methods sorted in descending order. These are some observations about the data collected:

1. **Static**: Given that the workload is fairly consistent across iterations of the loop, static scheduling is the fastest among the various thread scheduling methods tested as it does not incur any of the overhead of dynamically managing threads. For this number of threads, using the static scheudling method outperforms the base parallel case.
2. **Guided**: Guided scheduling is very close in performance to static scheduling which indicates that it adapts fairly well to the workload in the FSB simulation.
3. **Dynamic**: Given that the performance of dynamic scheduling is significantly worse than static and guided, it suggests that the overhead of managing the dynamic assignment of chunks outweighs the benefits of potentially more balanced workloads especially in this FSB simulation where the workload across the various iterations of the loop does not vary much.
4. **Runtime**: Runtime scheduling defers the decision of which schedule type to use until runtime. The scheduling type is set by an environment variable. Since it’s the slowest, it could be because the environment’s default choice isn’t optimal for this specific workload, or there might be additional overheads involved in determining the schedule at runtime.



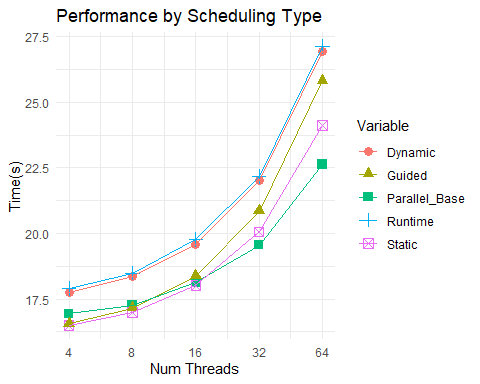
## Different scheduling methods across multiple threads

A sub-experiment was conducted after reviewing the results of experiment 4. In experiment 4b, the same scheduling methods were tested with different number of threads to determine if the conclusions drawn remain the same as the number of threads increase. This was achieved by modifying the export OMP\_NUM\_THREADS in experiment4.sh.

* Source file: experiment4.c
* Run with: sbatch experiment4.sh
* Number of threads: 4, 8, 16, 32, 64
* Number of steps: 10,000
* Number of fish: 10,000

The graph below plots the time taken for each scheduling method as the number of threads increase. The following observations can be drawn from the data:

1. The values for **Dynamic** and **Runtime** are consistently similar as the number of threads increase and do not diverge like the other 3 scheduling methods tested. given this pattern, the scheduling method selected at runtime in Setonix is very possibly dynamic which is what results in the very similar behaviors in both graphs.
2. **Static**: While this was the best performer at 4 threads, performance slightly degrades as the number of threads increases. This could be due to overheads from managing more threads, or the fact that the iterations might not be divided evenly among threads as their number increases, leading to some threads sitting idle while others are still working.
3. **Guided**: This method performed very similarly to static and parallel\_base from 4 to 16 threads, however, past that point it is observed that the performance of the guided scheduling type degrades rapidly as the number of threads increase. The overheads associated with adapting the chunk size dynamically increase with more threads which could explain this rapid increase in time taken.
4. The parallel base outperforms all the other methods at higher thread counts as it does not need to deal with the overhead of managing the different schedules for a large number of threads.

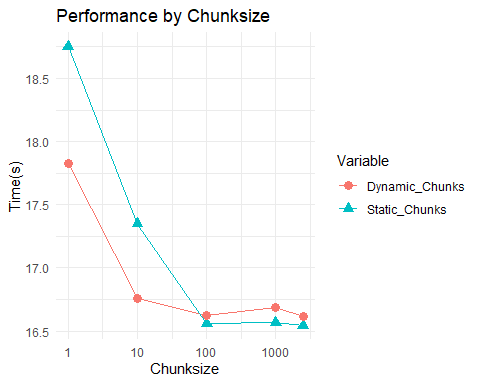


## Different Chunksize

For the **static** and **dynamic** scheduling methods, there is an option to set the chunksize that each thread executes. In sections 7.1 and 7.2, this option was not exercised. Resulting in the default division methods used. For static, OpenMP divides the iterations into chunks that are approximately equal in size and assigns one chunk to each thread. For dynamic, OpenMP assigns one iteration at a time to the threads.

The plot below shows the results of the experiment.

1. **Static**: In static scheduling, when the chunk size is small (e.g., 1), each thread will be assigned a very small amount of work in each cycle, leading to frequent synchronization and hence, higher time consumption. As the chunk size increases, the overhead due to synchronization decreases, leading to better performance. This improvement plateaus after chunksize exceeds 100.
2. **Dynamic**: In dynamic scheduling, a smaller chunk size means more frequent synchronization and higher overhead, similar to static scheduling. However, dynamic scheduling can handle load imbalance better than static scheduling as it assigns iterations one by one to idle threads. This is reflected in dynamic scheduling performing better at lower chunksizes before eventually falling behind static scheduling.



# Experiment 5 - Tasks and Sections

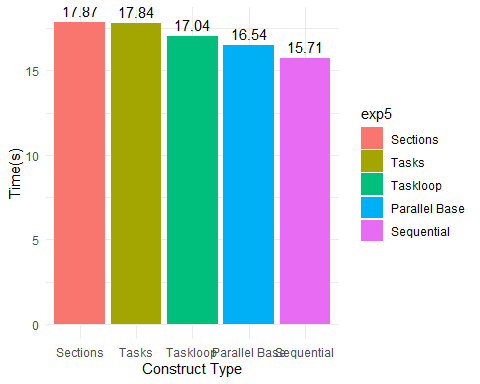
Experiment 5 investigates the time savings that can be garnered from using different worksharing constructs such as sections and tasks. Three different methods will be compared to the base sequential and parallel cases:

1. void parallelSections(Fish\* fishArray, int numfish, int numsteps): This function makes use of sections from OpenMP. The sections construct is used to define multiple independent blocks of code that can be executed in parallel.
2. void parallelTasks(Fish\* fishArray, int numfish, int numsteps): This function makes use of tasks from OpenMP. Inside each loop, tasks will be dynamically generated at runtime to handle a small portion of the calculation. As this does not make use of the reduction clause, a #pragma omp critical directive needs to be used before modifying any of the shared variables to prevent race conditions.
3. void parallelTaskloop(Fish\* fishArray, int numfish, int numsteps): This function makes use of #pragma omp taskloop directive. The taskloop directive combines the concept of tasks and loops by creating tasks for each iteration of a loop. This means that each iteration can potentially run as a separate task, which allows for better load balancing especially when the iterations have varying execution times.

* Source file: experiment5.c
* Run with: sbatch experiment5.sh
* Number of threads: 4
* Number of steps: 10,000
* Number of fish: 10,000

The chart below shows the results of the different worksharing constructs sorted in descending order. These are some observations about the data collected:

1. At 4 threads, none of the worksharing constructs outperform the base parallel case.
2. **Tasks** and **Sections** performed similarly and were noticably slower than taskloop. This suggests that the overhead from using either of these constructs is not worthwhile for such a small amount of threads.

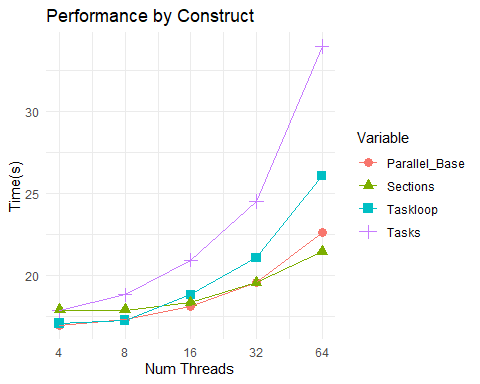


## Tasks and Sections across multiple threads

Similar to experiment 4, a sub-experiment was conducted to compare the performance of the different worksharing constructs as the number of threads increased. In experiment 5b, the same scheduling methods were tested with different number of threads to determine if the conclusions drawn remain the same as the number of threads increase. This was achieved by modifying the export OMP\_NUM\_THREADS in experiment5.sh.

* Source file: experiment5.c
* Run with: sbatch experiment5.sh
* Number of threads: 4
* Number of steps: 10,000
* Number of fish: 10,000

1. **Taskloop**: Taskloop execution time gradually increases with the number of threads. While the difference between 4 and 8 threads is minimal, the time continues to increase more significantly as the thread count grows.
2. **Tasks**: The Tasks construct has a more pronounced increase in execution time as the thread count rises. The overhead of managing individual tasks seems to become quite significant, especially as the number of threads increases, as evident from the steeper rise in times from 32 to 64 threads.
3. **Sections**: While this construct performed the worse at 4 threads, it shows a more consistent and gradual increase in time with a rising thread count. Interestingly, the time taken for 8 threads is a tiny bit less than that for 4 threads, which could be within the margin of error or might indicate that the sections are more effectively utilized with 8 threads.



# Conclusion

When considering the results of Experiments 0 to 5, it can be concluded that the sequential code is the best solution to this version of the FSB algorithm. However, a consistent theme throughout these experiments is that the work done in the loops is not sufficient to justify the benefits of parallelization. It should be noted that the algorithm implemented is a heavily simplified version of the actual FSB algorithm which lacks components such as running the collective-instinctive movement operator and collective-volitive movement operator. The inclusion of these and other missing components would greatly increase the work done in each step. If the work in each iteration is sufficient, the algorithm would benefit from the implementation of parallelization..

If it was determined that the code would benefit from parallelization, the following could be implemented to further boost performance:

1. Use the **static** scheduling type to increase performance
2. Use **taskloop** if the number of threads is small and **sections** if the number of threads is large.