Allen Antony 22706998

CITS5507 Project 1 Report

INTRODUCTION

Fish School Search (FSS) is a swarm intelligence algorithm that optimises a specified objective function through the behaviour of many individual agents traversing the solution space. This algorithm initialises a 'school' of fish that individually move in random steps, gaining weight as they reach more favourable regions of the solution space. Like other swarm intelligence algorithms, FSS displaces each fish based on the weighted average of all individual movements in the school, simulating the fish school collectively moving to regions with more food. The algorithm balances exploration of the search space and exploitation through the concept of volitive movement, where fish will either deviate away from the 'centre-of-mass' of the school (known as the barycentre), or cluster towards this barycentre depending on whether the total weight of the school has increased from the previous iteration.

In this project, the multi-threaded parallelisation of a simplified version of the FSS algorithm has been attempted through OpenMP. The purpose of this project is to measure the speed-up of parallelisation under different conditions, such as number of fish, number of threads, worksharing scheduling types, worksharing chunk sizes, etcetera, and so we have opted to forgo the complexities of the entire FSS algorithm. This simplified version of FSS omits the collective component of fish movement, limiting the algorithm to 3 steps: individual fish movement, feeding, and calculating of the barycentre coordinates. The fitness function used is (maximising) Euclidean distance from the origin. Pseudocode for the sequential implementation of this algorithm is shown in the next section.

ALGORITHM ANALYSIS

Sequential Implementation pseudocode

```
**for** each fish **do**:
Initialise fish with random position
Initialise $wt = wt_{INITIAL}$
Evaluate $f$
**for** NROUND iterations **do**:
    **for** each fish $i$ perform individual swimming:
            $x_i(t+1) = x_i(t) + rand(-1, 1) \cdot step_{ind}$
        Evaluate $f_i(t+1)$
        Reset position if $\Delta f_i < 0$
        Update max fitness variation $\Delta f_{max}$
    **for** each fish $i$ perform feeding:
```

```
                    $wt_i(t+1) = wt_i(t) + \frac{\Delta f}{\Delta f_{max}}$
        **if** $wt_i(t+1) \; > \; 2 \cdot wt_{INITIAL}$ **then:**
                    $wt = 2 \cdot wt_{INITIAL}$
    **for** each fish $i$ **do**:
        Update sum with $x_i \cdot wt_i$
        Update sum with $y_i \cdot wt_i$
        Update sum with $wt_i$
    Compute $b_x = \frac{\sum_{i=1}^{N} x_i \cdot wt_i}{\sum_{i=1}^{N} wt_i}$
    Compute $b_y = \frac{\sum_{i=1}^{N} y_i \cdot wt_i}{\sum_{i=1}^{N} wt_i}$

                    $b = \sqrt{b_x^2 + b_y^2}$
```

Parallel implementation:

The most obvious method of parallelisation is simply dividing the initialised fish among the required number of threads. This method reduces the number of vectors of communication to aggregate variables such as max fitness variation across the school and provides a balanced load to each thread. In contrast, dividing the lake among threads would require per-iteration re-computation of fish allocations for each thread (i.e., as fish move across regions), would result in unbalanced loads (e.g., empty regions), and a greater overhead in aggregating shared variables.

The parallelisation construct used is the *#pragma omp for* worksharing directive. For multiple stages in the parallel implementation of FSS, local variables must be reduced across threads (e.g., Δf_{max}). The *omp for* worksharing directive provides the *reduction* clause to perform efficient binary reduction, which is not supported via other worksharing methods such as *omp task* (although reduction across tasks will be added in OpenMP 5.0) and must be manually implemented through synchronisation methods such as locks and atomic statements*.* Furthermore, the *omp ATOMIC* directive does not support *max* operations, requiring the much less efficient *omp CRITICAL* to find maximum fitness variance. These factors resulted in an execution time that was 5-10x slower than the counterpart due to the communication overhead.

Since the implemented FSS algorithm is a simplified version of the original, there is little opportunity to introduce asynchronous execution via the *nowait* clause to remove barriers. The individual swimming region uses a *reduction* clause that requires a thread barrier, and the manual implementation of such is not worth the overhead cost. Since the barycentre calculation region is at the end of the *omp parallel* block, the thread barrier is not removable. We can, however, introduce a *nowait* clause in the fish feeding region since (assuming static scheduling and identical chunk sizes), the thread will access the same chunk of ‘fed’ fishes in the following region. The following section illustrates the pseudocode in simple terms.

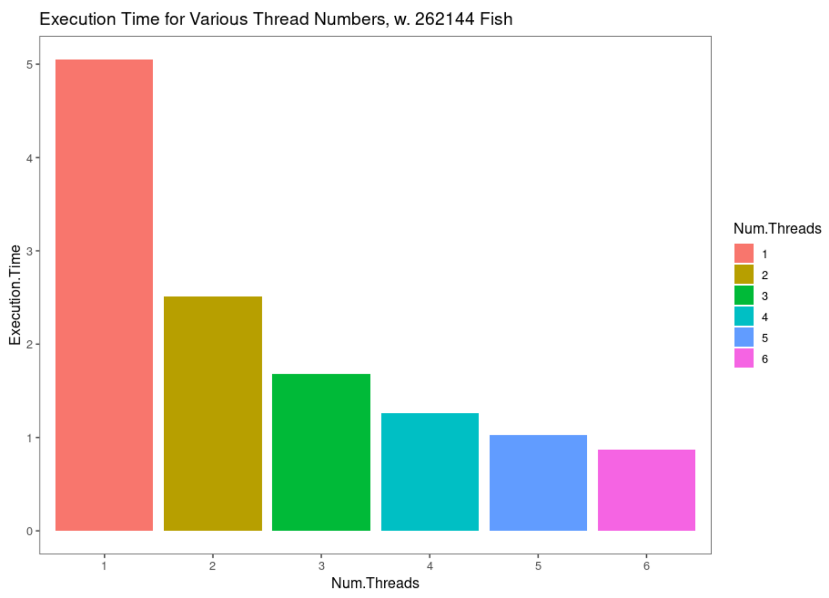
Parallel implementation simplified pseudocode

```
for each fish initialise fish with random position
for NROUND iterations do:
        #pragma omp
#pragma omp for reduction(max:$\Delta f_{max}$)
for each fish perform individual swimming
#pragma omp for nowait
        for each fish perform feeding
        #pragma omp for reduction(+: $wt, x \cdot wt, y \cdot wt$ )
        for each fish update barycentre numerator and denominator
        Calculate barycentre
```

THREADS AND FISHES EXPERIMENTS

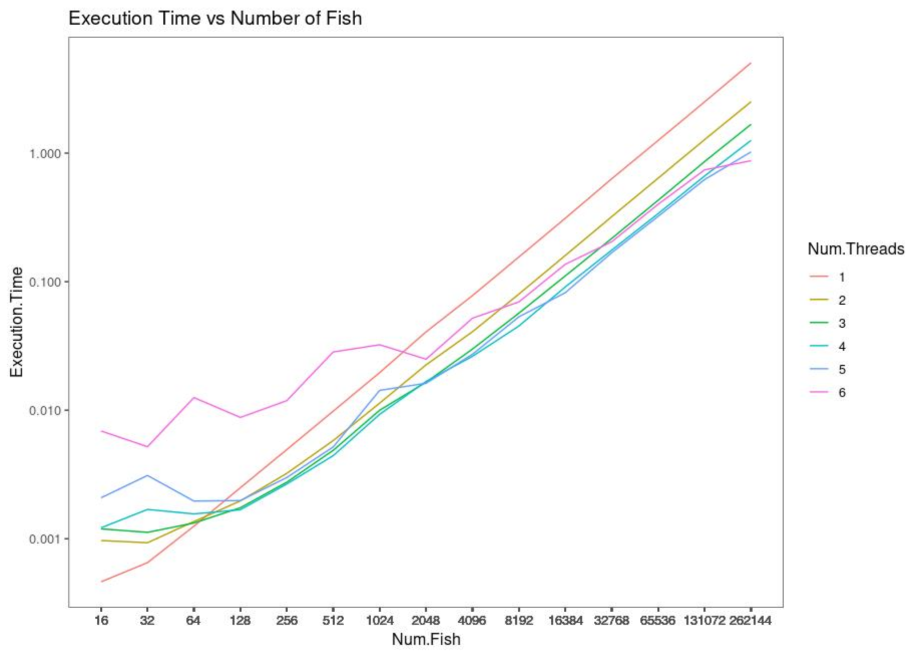
- For all experiments, the number of rounds was fixed to 500 as this variable (predictably) linearly increases execution time with no other interesting behaviours.
- All experiments were performed on an Intel i5 10600K processor (6 cores with hyperthreading off).

Execution time vs Number of Threads



We can immediately observe the efficacy of the parallel implementation from the halving of the execution time between 1 thread (execution time of the sequential program) and 2 threads. With a low number of threads, for a sufficiently large number of fishes, the overhead of multi-threading (caused by generating and merging threads, and waiting for thread synchronicity at barriers, etcetera), is negligible compared to the execution time of the program, resulting in the large immediate speed-up. The pattern continues, with 3 threads having approximately a 3x speedup, 4 threads a 4x speedup and 5 threads a 5x speedup. It's only until the 6th thread that there is a noticeable decline in the speedup rate per thread, as the overhead cost, rising exponentially with number of threads, is a greater proportion of the total execution time.

Execution Time vs Number of Fishes (log-log scale)

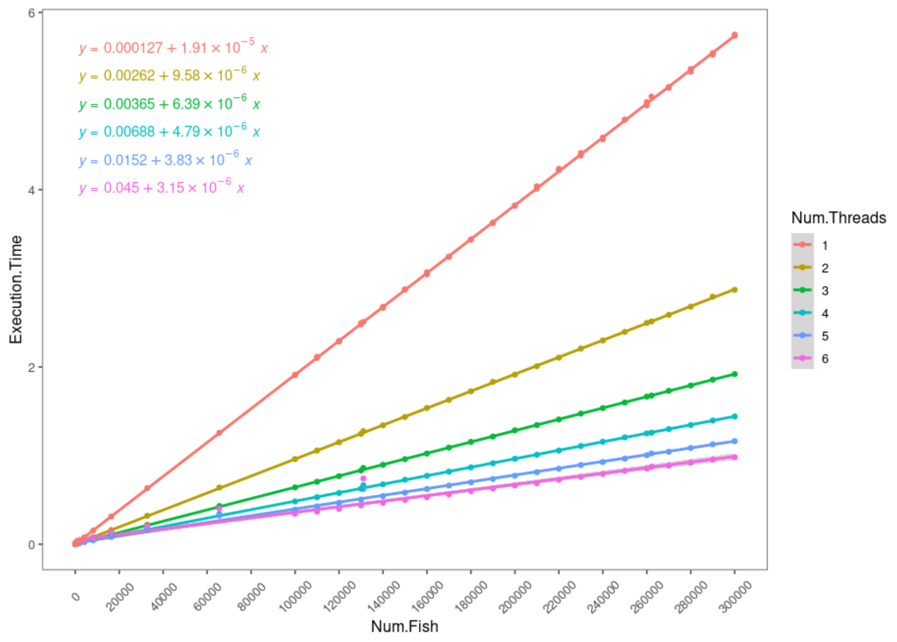


As described in the previous section, the parallel implementation has an overhead cost that rises (non-linearly) with number of threads, verified from the 6-threaded program's execution time intersecting (surpassing) the sequential execution time at 2048 fishes, while the other parallel programs intersected the line at 64 and 128 fishes. Interestingly, 262144 fishes are required before the 6-threaded program becomes the most efficient parallel implementation (surpassing the 5-threaded program).

As number of fishes increase, the parallel program allocates a greater amount of work to each thread to execute synchronously. Assuming that the rate of speedup does not change with number of fishes, the *amount* of speedup increases, and the balance between the overhead cost and the execution speedup changes. The intersection with the line for sequential execution time marks the point where the execution speedup becomes more significant than the overhead cost.

All parallel programs have a ratio of synchronous execution time to overhead cost. As we increase the significance of parallelism by increasing the amount of synchronous work that can be performed, it is predicted that increasing number of threads will more quickly result in speedup. One of the ways this can be performed is by utilising a more compute intensive objective function, which was performed in section 3.

Execution Time vs Number of Fishes (linear scale)



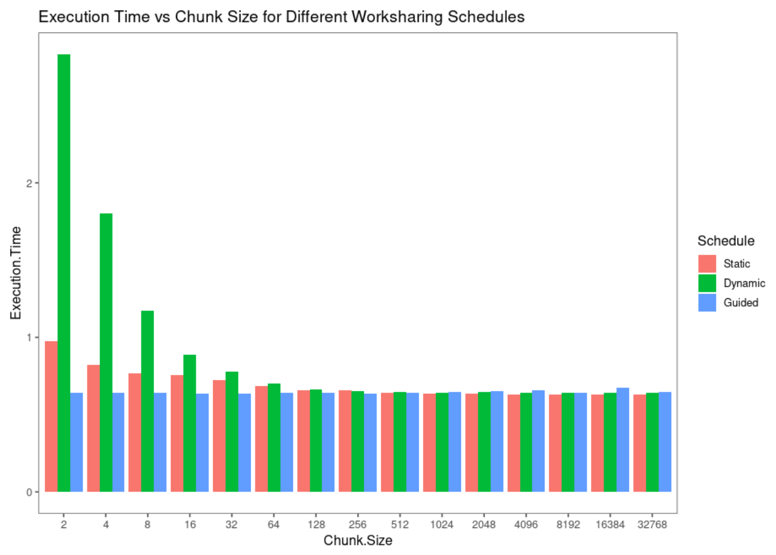
As expected, 6 approximately linear graphs are produced when viewing execution time vs number of fish on a linear scale. We observe 3 key features from the graph:

- As the number of threads increase, the relationship deviated further from linearity. However, this may be a systematic testing error as using all cores of the system may have resulted in the FSS process competing with other running processes for CPU allocation.
- When applying linear regression to the data points, equations in the form $y = ax + b$ are formed, where b is equal to the overhead cost, and $\frac{y_{seq}}{y_{parallel}}$is equal to the speedup from parallelism.
- Since execution time forms a linear relationship with number of fish, the overhead cost is constant for all problem sizes. This means that there is no time delay due to false sharing, cache contention, and idle threads, all of which would scale with problem size. The method of parallelism is hence efficient.
- We can verify the assertion made in the previous section as b likely has an exponential relationship with number of threads, approximately doubling for every additional thread from 2 threads. More specifically, it approximates: $b \sim 0.00262 \cdot 2^{t-2}$. Although negligible for low numbers of threads, it remains to be seen how the overhead cost will scale for 32 and 64 core systems.

SCHEDULING EXPERIMENTS

Execution Time vs Chunk Size

Using 4 threads, 131072 fish, and chunk sizes that are factors of 131072



False sharing is when multiple threads synchronously access the same L1 cache line when attempting to read/modify logically independent data values, causing the cache line to become invalidated and greatly increasing interconnect bandwidth as the cache line is constantly updated. In the case of the FSS algorithm, a single fish, stored contiguously in memory as a C struct, has 5 float values, resulting in a total size of 20 bytes. The Intel core i5 has a 64-byte cache line size. This means that at each thread must be allocated at least 4 *fish* structs to ensure that adjacent threads do not cross the cache line boundary. We observe this from the above graph, where the execution time for static and dynamic scheduling drops drastically at a chunk size of 4.

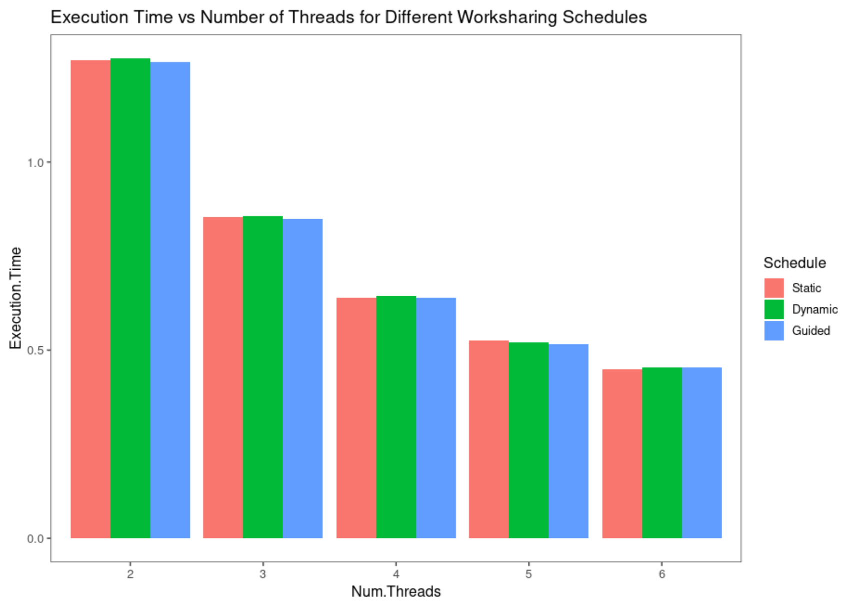
The three scheduling types are the following:

- Static scheduling allocates 'chunks' of the iterations of an *omp for* loop in a round-robin fashion until all iterations have been exhausted. This runs the risk of load imbalance between threads, especially for high chunk sizes, but with a lower overhead cost.
- Dynamic scheduling stores iterations in an internal work queue, where a 'chunk' of iterations is dynamically allocated to a thread that is idle. This reduces the occurrence of idle threads but has a high overhead cost for managing the queue.
- Guided scheduling is like static scheduling, except the allocated chunks reduce in size (to the specified chunk size) over the course of the loop.

Since fish have randomly allocated positions across the lake, and they move in random individual steps, for a large school size such as the one used for this experiment, the load distribution between threads will be approximately even. Hence, there is no advantage to performing dynamic or guided scheduling. Rather, it is predicted that for larger thread sizes, static scheduling will surpass dynamic scheduling in performance as the number of dequeue operations increase. This is demonstrated from the drastic decrease of the execution time of dynamic scheduling with the increase of chunk size, as the runtime environment is required to allocate chunks less often to threads.

Execution Time vs Number of Threads

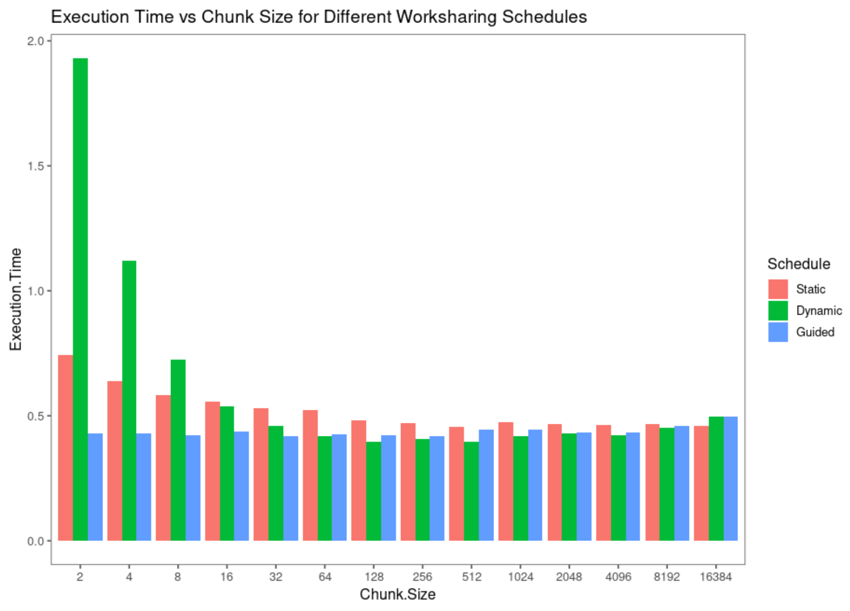
Using 131072 fish and chunk size of 512



As we vary the number of cores used, utilising the most effective chunk size from the previous figure, the same trend is observed, where the execution time between different scheduling types is almost identical.

Execution Time vs Chunk Size with Hyperthreading

Using 8 threads, 2 logical cores, 6 physical cores, 131072 fish, and chunk sizes that are factors of 131072



Interestingly, when we use hyperthreading, the performance of dynamic and guided scheduling surpasses static scheduling. Hyperthreading is a process by which each processor has an extra set of CPU registers and control units, allowing it to logically parallelise a problem whilst utilising the same ALU (arithmetic logic unit) for computation. This allows two threads to be scheduled concurrently in a single processor, reducing downtime in the case of the processor stalling (e.g., cache miss, thread idling, etc.). Without hyperthreading, the computational overhead of dynamic scheduling must be borne by one of the physical cores, and thread execution may need to be stalled to (infrequently) load the work queue into cache and allocate chunks. In this case, the work queue may be managed by one or more logical cores, allowing other threads to execute without interruption.