Project 1

Max Chatfield, 22176321
Nicholas Clarke, 10417777

September 2023

1 Introduction

This project has implemented and experimented upon various parallel strategies in the *Fish School Behavior*[1] algorithm. Specifically, the eat and swim actions have been modelled, along with a simple calculation of the barycentre of fish.

We present some implementation details and experimental results. In particular, we have focused on correctly timing with a different number of fishes, different number of threads, different chunk and scheduling strategies as well as testing various openMP constructs.

2 Implementation

We present a brief overview of the steps in the sequential and parallel algorithm.

1. Initialize all fishes with a random position in the grid, and set their weights to a fixed value. Calculate their euclidean distance to the origin.
2. Begin the main simulation loop.
3. Iterating over each fish, perform a movement in a random direction. Calculate their updated euclidean distances and store a δ_f value, which is the difference between the updated euclidean distance and the old distance.
4. Iterating over each fish, find the maximum of the δ_f.
5. Iterating over each fish and using the maximum value found, update each fishes weight based on the formula in the specification.
6. Iterating over each fish, calculate the total barycentre.
7. repeat from step 2 until all steps of the algorithm have been exhausted.

We briefly go over some implementation details used for experiments and other notes we found during the course of our analysis. Timing is conducted from the start of the main simulation loop and ends when all steps (defined

in code as the preprocessor variables STEPS) have been exhausted. During the initial exploration of results, it was found that timing needed to be treated carefully. The details of how timing was conducted are explained below in section 3.1.

Each of the 'iterating over each fish' steps are loop items that are subject to parallelisation. Their behavior were varied by changing the number of threads, changing the schedule strategy and chunk size. For finding the maximum δ_f, it was further identified that this is possible in at least three ways. The first was to use the 'max' reduction clause in openMP to calculate the max values. The second involves using the 'critical' clause to define a critical region on the max variable, so that threads can update and use that value in a thread-safe manner. Finally, a parallelisable quick sorting algorithm was also used to find the max value.

For the calculation of the barycentre, there were multiple openMP strategies employed. The first is to use a reduction clause on the barycentre calculation, which was done by reducing on two variables, the barycentre numerator and denominator. The second is to use a critical section when updating the barycentre. The last was to use the 'atomic' openMP clause, which allows the operation to be done by a single thread in such as a way that any other thread reading or writing the variable will only operate after any other thread had finished writing to it.

For generating random numbers, it was discovered through experimentation that the rand() system call[2] is not appropriate in multi threaded code. This is because the times increased dramatically with more threads and naive usage of rand(), which is due to the thread unsafe nature of the function. Instead, the rand_r() system call has been used. This has the side affect of allowing each thread to manage its own random number generation, and therefore allowing for deterministic results in some cases.

The X-Y coordinates, weight, euclidian distance, and δ_f of each fish were stored in a struct. Doubles were used for these values, rather than floats, so as to increase the memory size of each, and thus potentially show a greater effect of CPU cache on the process during our exploration of chunk sizes and scheduling.

3 Experimental results

3.1 Timing analysis

Four timing functions were chosen available that are available in POSIX environments. Setonix runs CrayOS[3], based on the SUSE Linux environments and indeed, the timing functions are available on Setonix. The functions chosen are the clock system call, omp_get_wtime, as well as the clock_gettime[4] call with two clocks CLOCK_MONOTONIC and CLOCK_PROCESS_CPUTIME.

There is no difference between clock and POSIX Process CPU time, and between omp_get_wtime and POSIX monotonic measurements from examining table 1. Importantly, the dataset includes many thread values as seen in Figure

Method	Mean	SD
clock	130.84	11.09
omp_get_wtime	34.17	38.71
POSIX montonic	34.17	38.71
POSIX Process CPU time	130.84	11.09

Table 1: Timing statistics for the four measures used in the analysis. The data used are taken from sequential and parallel experiments with differing number of threads, constant number of fishes and reduction clauses for barycentre and δ_f calculations.

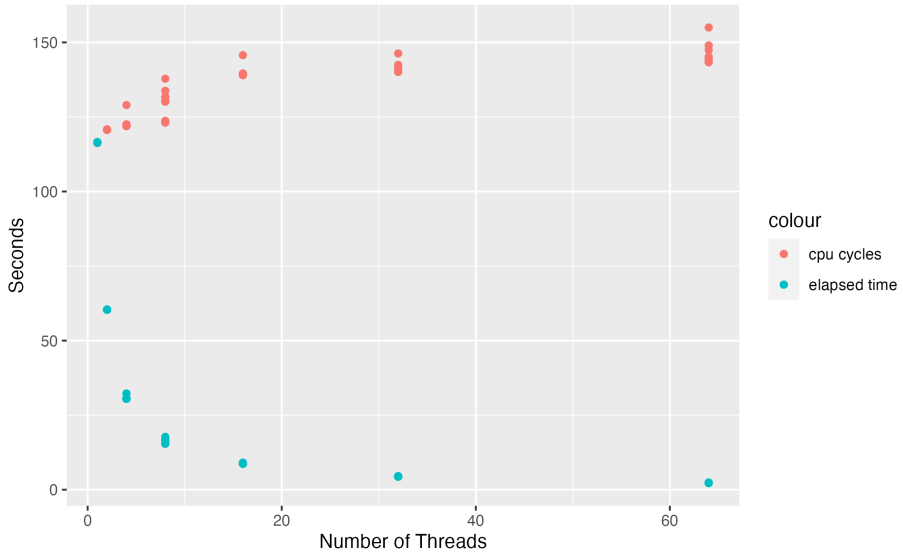

Figure 1: Elapsed time and cpu cycles when varying thread number. 1000000 fishes where used along with 1000 steps. Reduction clauses where used for barycentre and δ_f calculations and a static schedule with default chunk size was used in all loops.

1, which means the standard deviations include any variation resulting from changes in running behavior over using different threads. Therefore, the analysis can reasonably use clock and omp_get_wtime methods for timing.

What are the differences between the two and why should both be used? Referring to Figure 1, the near-constant trend of the cpu cycles shows it is measures the total cpu cycles over all cpus used. For instance, with one thread, the cpu cycles is around 125, and with two threads, it has remained very similar. Thus, it seems with each cpu parallelising, it is dividing the amount of work equally, therefore running for less cpu cycles individual but still summing to the same total that one cpu would use.

Finally, elapsed time from omp_get_wtime measures the actual 'wall clock' time that has passed in the program. This is useful for evaluating the actual speed of executions. Elapsed time will be used as a primary measure for execution speed and cpu cycles will be used as a secondary measure to measure a form of effectiveness in program executions.

3.2 Thread number analysis

The analysis of the threads indicates they are parallelising effectively and resulting in speed up of around halving every time the number of threads doubles as seen in figure 1. To see how well the cpus are working together, we define a *clock factor* CF, equal to

$$\text{CF} = \frac{\text{CPU Cycles}}{\text{Elapsed time} * \text{Threads}} \tag{1}$$

The idea behind this factor is to measure the ratio between the cpu cycles and the elapsed time. Since cycles and elapsed share the same units, their ratio correspond to how many cpus were able to work in parallel with one another. So, if there are 2 threads, total cpu cycles are 4s and the elapsed time is 2s, we achieve CF of $\frac{4}{2*2} = 1$. The thread term allows comparison when experimenting over different threads. Correspondingly, a CF of 1 indicates effective parallelisation, CF < 1 indicates threads could not parallelise well together, and CF > 1 seems to indicate threads are able to work harder than their theoretical limit.

Figure 2 demonstrates that the threads are indeed working effectively together, with some variation in 8 threads. However, does this mean necessarily that there is a speed up compared to the sequential program?

We can examine the speed up while using more threads by examining the *Speed-Up Factor* SUF, defined as

$$\text{SUF} = \frac{\text{Mutlithreaded Program Elapsed}}{\text{Sequential Elapsed}} \tag{2}$$

Indeed, we expect that if 64 threads were able to perfectly speed up the program, which is taken to be mean the elapsed time equals the sequential

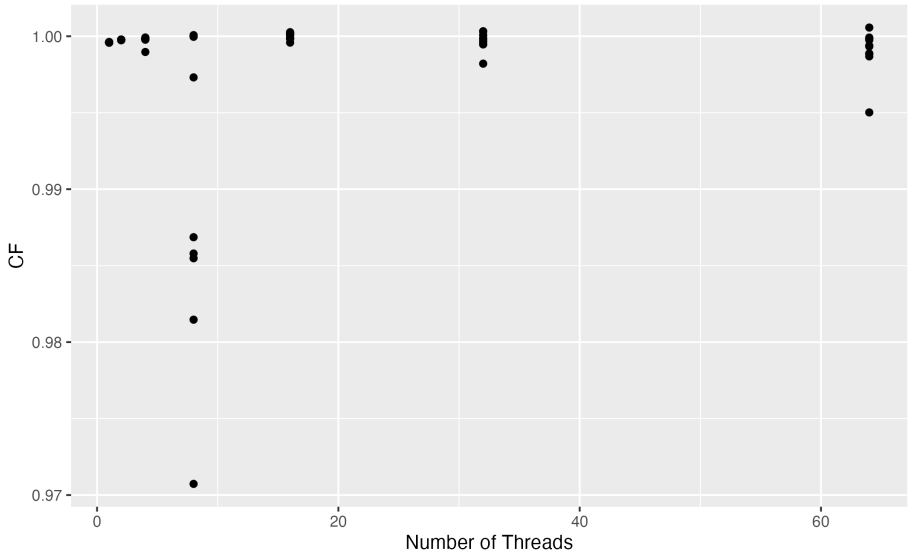

Figure 2: CF factors of experimental results when varying threads. CF is defined in equation 1.

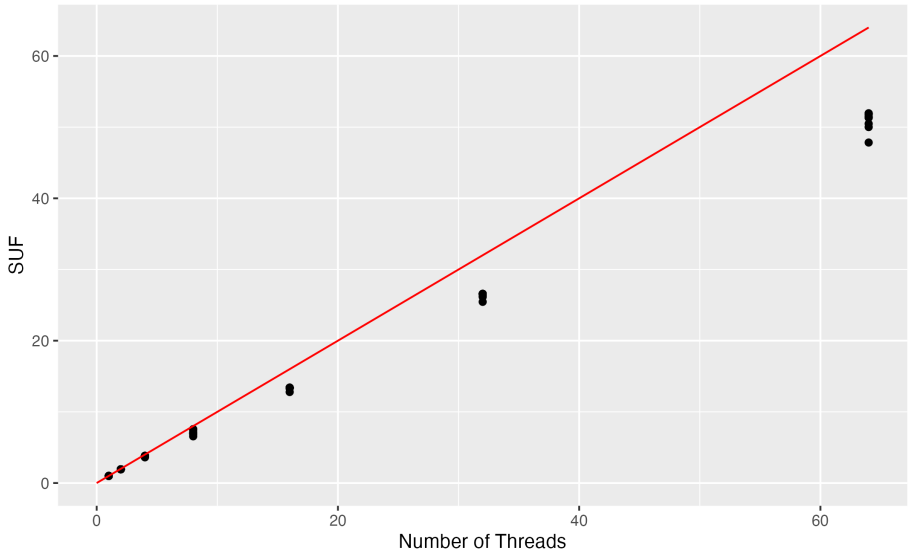

Figure 3: Comparing experimental results elapsed time to the elapsed time of the sequential program. The median sequential elapsed time was taken. The red line shows the theoretical speedup if all threads were able to perfectly speed up the program.

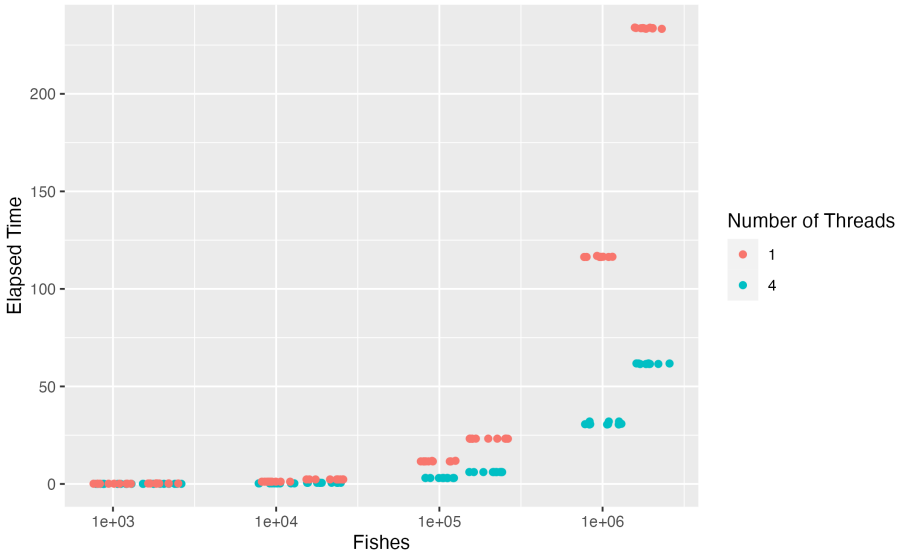

Figure 4: Elapsed program time with different number of fishes. Fishes are plotted transformed onto a log10 scale.

elapsed multiplied by the number of threads, we would expect to see a SUF of 64. However, this is not the case as reflected in figure 3.

Therefore, we have identified two classes of parallelisation performance. The first corresponds to the within program parallelisation referring to how well the program can utilize the threads within itself, measured with CF. The second class is how well the parallelisation has resulted in a speed up, which we can measure with SUF.

3.3 Fish Number analysis

Fish analysis was conducted over the fish numbers 1000, 2000, 10000, 20000, 100000, 200000, 1000000 and 2000000. Steps were kept constant at 1000 and the analysis was performed over 1 thread and 4 threads.

Elapsed time seems to follow a trend where doubling the number of fishes results in roughly double the elapsed time spent as seen in figure 4. To see this trend clearly, examining the log10 transform of both axes is illustrative. Refer to figure 5 to see the trend. The behavior suggests a linear relationship between the log transforms. Then,

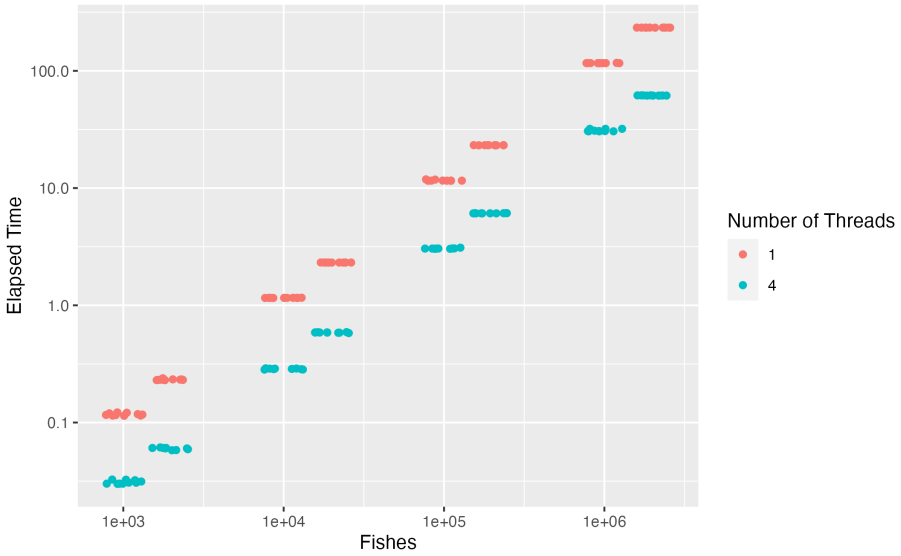

Figure 5: Elapsed program time with fishes, transforming both axes onto log10 scales.

$$\begin{aligned}
\log10(\text{Elapsed Time}) &= m\log10(\text{Fishes}) + c \\
\text{Elapsed Time} &= 10^{m\log10(\text{Fishes})+c} \\
\text{Elapsed Time} &= 10^{\log10(\text{Fishes}^m)+c} \\
\text{Elapsed Time} &= 10^c\text{Fishes}^m
\end{aligned}$$

The fit in figure 5 suggests $m \approx 1$ for both numbers of threads. Thus, we conclude that elapsed time is linearly dependant on the number of fishes.

We now examine how well the programs where able to achieve within parallelisation using the CF measure. Figure 6 demonstrates experimental results. The trend established with large number of fish is that the program with four threads was able to achieve very close values to the theoretical within parallelisation. We note that the variation present for low number of fishes may indicate the overhead from openMP calls and other sequential calls became apparent.

Finally, the speed-up factor was calculated by taking the median of the elapsed time for each number of fishes, comparing between the sequential program and the four threaded program.

Figure 7 demonstrates the SUFs on the data set. We notice all achieve good SUFs above or around 3.8. This indicates the multithreaded program was effectively parallelising and solving the fishes task faster. We note the anomalous result with 10000 fishes. This seemed to occur because of large variation in results after running on Setonix, perhaps due to the conditions at that time.

Figure 6: Clock factor measurements for different number of fishes.

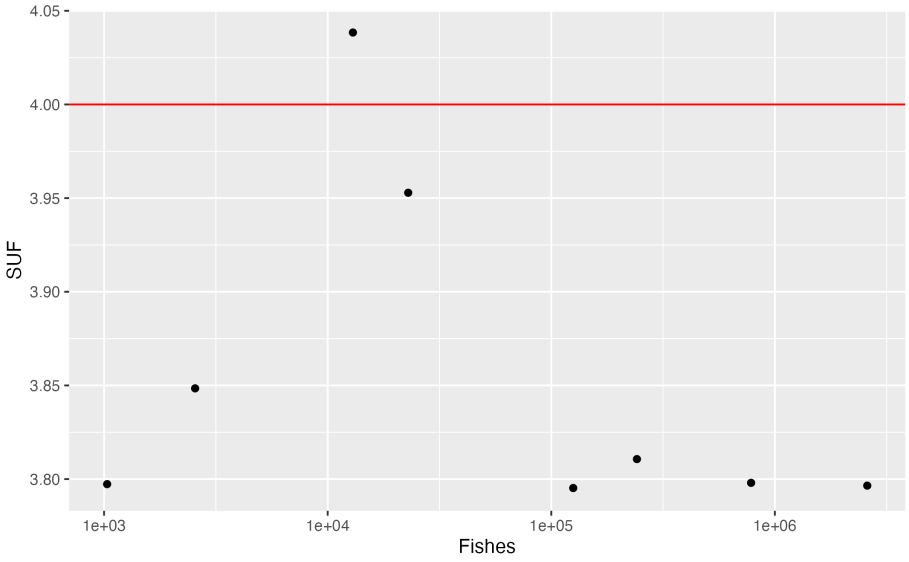

Figure 7: SUF measurements for different number of fishes when using four parallel threads compared to a single sequential process. The red line indicates the theoretical SUF which is four.

3.4 Scheduling analysis

Extensive testing of different chunk sizes and scheduling types was performed. It seemed important to test how memory caches interacted with parallelisation process. For this reason, the process was made as memory intensive as possible, by running the process for a very large number of fish.

A 32-core process was used with 20 million fish swimming for 1,000 steps each execution. In each step, the 20 million iterations were split into parallel threads of the following chunk sizes:

- Unspecified chunk size, which should by default be an even split into 32 chunks of 650k iterations per thread
- Explicitly declared chunk size of 650k, to check this was identical to the default
- Three chunk sizes aligning with Setonix's memory architecture[5][6] and the 40 byte size of each fish struct:
  - 1.6k iterations per chunk $\approx$ 64KB, the L1 cache size
  - 12.8k iterations $\approx$ 512KB, the L2 cache size
  - 100k iterations $\approx$ 4MB, 1/8 of the L3 cache which is shared between 8 cores
- Three further chunk sizes which split the 20 million fish completely evenly between threads:
  - 156,250 iterations = 1/4 of 625k
  - 78,125 iterations = 1/8 of 625k
  - 625 iterations = 1/1000 of 625k
- The deliberately inefficient chunk size of 500k iterations

All chunk sizes were tested using static, dynamic, and guided scheduling, executing 10 times each.

OpenMP's runtime scheduling was also tested. Runtime scheduling performed so poorly that the limitations on the time length of a job submitted to Setonix prevented this experiment being run with 20 million fish. Instead, runtime scheduling was performed with 1 million fish and compared to a similarly-sized static-scheduled process.

3.4.1 Chunk size

Initial testing focused on static scheduling using chunk sizes based on the CPU memory caches. This showed that the 100k chunk size was a notable outlier when compared to the 1.6k and 12.8k and default chunk sizes, as shown in Figure 8.

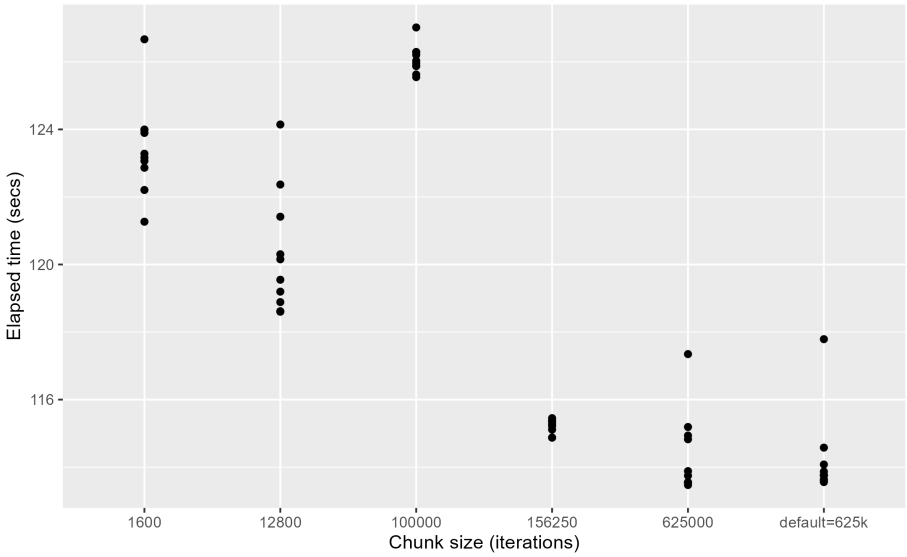

Figure 8: Initial exploration of the effect of chunk size on the speed of parallelised code using static scheduling.

This unexpected result led to exploration of potential reasons for such a performance decrease. It seemed most likely that this particular result was due to the inefficiency of splitting 20 million iterations into 200 100k chunks, and then allocating those 200 tasks to 32 threads. This split meant that all threads would do 600k iterations and then 8 of those threads would do an additional 100k iterations while the other threads waited.

To attempt to model this, we created a waste factor which calculates the hypothetical extra capacity wasted by these waiting threads, assuming that all threads worked at the same speed.

$$\text{Waste} = \frac{ceiling\left(\dfrac{\text{Total Iterations}}{\text{Num Cores} \times \text{Chunk Size}}\right)}{\dfrac{\text{Total Iterations}}{\text{Num Cores} \times \text{Chunk Size}}} - 1$$

The 100k chunk size showed better performance than its waste factor might predict. This may be evidence of some improvement from better optimisation for the cache environment. Additionally, the 8 threads performing the extra work will have been the first to complete their first 600k iterations, and thus the drop in speed would be expected to be lower than this waste factor. In later runs of the program the 500k chunk size was specifically chosen to maximise this wastage, as this chunk size divides the work into 40 chunks to be allocated among the 32 threads.

Chunk size	Median (secs)	Ratio to default	Waste	Chunks/thread
1,600	123.2	1.083	0.00096	390.63
12,800	119.9	1.054	0.00352	48.83
100,000	126.0	1.108	0.12	6.25
625	126.5	1.112	0	1000
78,125	121.3	1.067	0	8
156,250	115.3	1.013	0	4
625,000	113.8	1.000	0	1
Default	113.8	1	0	1
500,000	161.3	1.418	0.6	1.25

Table 2: Comparison of various chunk sizes using static scheduling.

There also seemed to be some inefficiency from smaller chunk sizes, presumably due to greater overhead in allocating chunks to threads. The graph of all experiments (Figure 9) showed a general trend to shorter execution times as chunk size increased.

3.4.2 Dynamic and guided scheduling

As can be seen in Figure 9, dynamic scheduling was found be to less efficient than the default static scheduling in all experiments. It would seem that this project did not suit that type of scheduling. The work involved in the calculations of this algorithm was very evenly spread across the iterations, thus it was to be expected that static scheduling would outperform the other types. Had there been more variable calculation time for different iterations, then dynamic may have helped to even out the work between threads.

An interesting result seen in the data (Table 3) is that guided scheduling performs better than static does at lower chunk sizes. This is understandable, as guided scheduling will begin with larger chunk sizes and decrease the size of chunks towards the target chunk size as the loop proceeds. Thus the guided schedule clause should result in a process with lower average chunks per thread than a static schedule with that same specified chunk size.

3.4.3 Runtime

Runtime scheduling was also used, but it performed roughly 30 times slower than static scheduling using the same number of cores. We thus used a 1 million fish version of the program and compared it to a static scheduled version using the same number of fish, the results of which are shown in table 4.

3.5 Ordered clause

To measure the impact of making parallel for loops ordered, two additional versions of the code were created. In one version all parallel for loops contained an ordered clause, and in the other only the weight calculation had an ordered

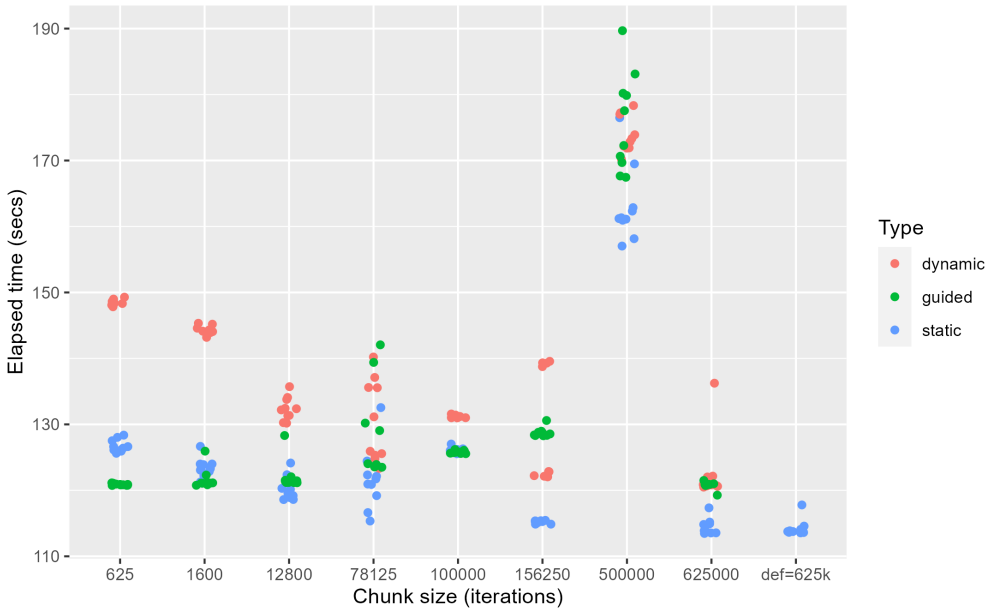

Figure 9: Comparison of speeds of static, dynamic, and guided scheduling at all chunk sizes.

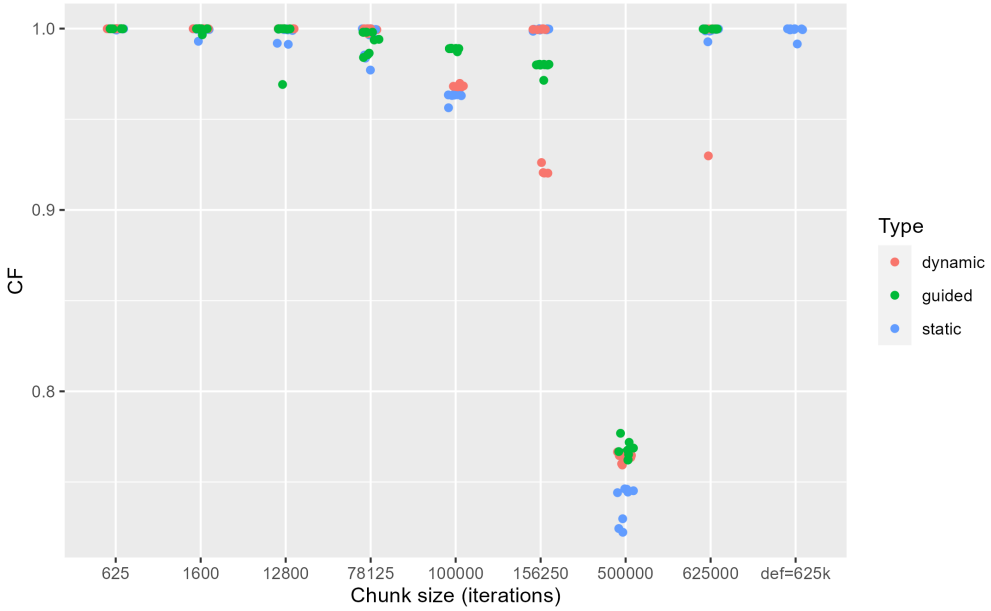

Figure 10: Clock factor measurements for different scheduling and chunk sizes.

Scheduling	Chunk size	Median (secs)	Ratio to default
dynamic	625	148.5	1.305
guided	625	120.9	1.063
static	625	126.5	1.112
dynamic	78,125	128.5	1.130
guided	78,125	124.0	1.090
static	78,125	121.3	1.067
dynamic	625,000	120.8	1.062
guided	625,000	120.9	1.063
static	625,000	113.8	1.000
static	Default	113.8	1

Table 3: Comparison of static, dynamic, and guided scheduling.

Scheduling type	Median (secs)	SUF
Static	4.39	26.53
Runtime	131.22	0.887

Table 4: Performance of runtime scheduling with 32 threads and 1 million fish, as compared to static.

clause. The fish school algorithm had no need for the use of an ordered clause, but it seemed useful to assess how an ordered clause would affect the performance of parallel code in general. These versions of the code were run using 1 million fish. The results of these were compared to the performance of parallelised code without any ordered clauses. Speed-up factors were calculated comparing to the performance of the sequential code, as seen in table 5.

3.6 Atomic, Critical and Sorting Analysis

Further analysis was carried out with a fixed number fishes and steps but varying threads, while using different openMP constructs and methods for the maximum δ_f calculation and the barycentre calculation. We briefly describe the methods used.

- **Atomic:** This test was conducted by using the atomic clause in the barycentre calculation. The maximum δ_f calculation was done with the

Code	Median (secs)	SUF
Parallel	4.39	26.55
Weight-change loop ordered	10.97	10.61
All loops ordered	153.85	0.757

Table 5: Measurements of the performance drop observed when an ordered clause was added to parallelised loops with 32 threads.

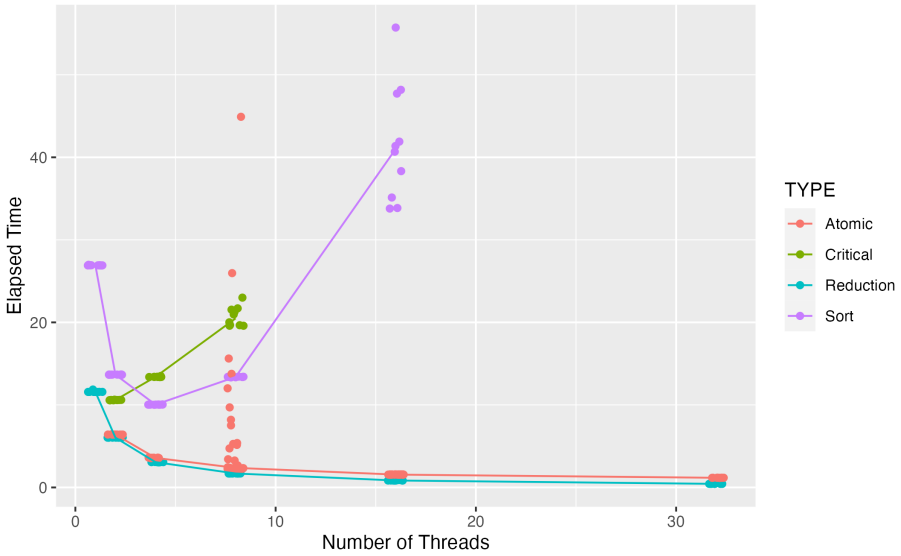

Figure 11: Experimental results with different openMP clauses and the sorting algorithm used. Fishes were kept constant at 100000 with 1000 steps. Trend lines are fitted for each group to the medians found for experiments over each number of threads.

reduction openMP construct. This was not applicable to the sequential program and so starts at two threads.

- **Critical:** This test was conducted using critical regions in both the barycentre calculation and maximum δ_f calculation. This is not applicable to the sequential program and so starts at two threads.

- **Reduction:** This test was conducted using the reduction construct. The barycentre calculation used an additive reduction and the maximum $delta_f$ calculation used a max reduction. We argue that although technically the sequential program does not use a reduction clause, it is comparable in this case as theoretically the reduction clause using one thread should be very similar as the behavior in the sequential program. This was the default behavior used in all other experiments besides this section.

- **Sort:** This test used a parallelisable quick sort algorithm to find the maximum δ_f value. The reduction clause was used for the barycentre calculation. The sequential program was also able to use the sorting algorithm and thus is comparable.

The elapsed times of the programs are shown in figure 11. Both atomic and reduction are the best strategies in terms of elapsed time. Critical became so poor that experimental results where difficult to collect after eight threads.

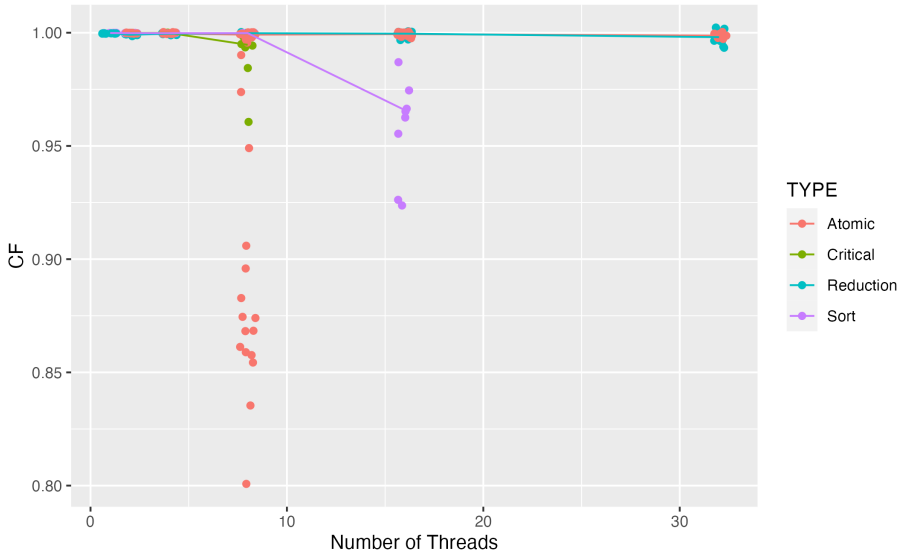

Figure 12: Clock factor measurements over the atomic, critical, reduction and sort variants. Trend lines are fitted to medians of each group.

Sorting initially resulted in improvements in performance when increasing the thread numbers, but then the performance deteriorated.

We hypothesize that critical became poor because of the large number of threads attempting to access the same critical regions repeatedly. Since this was preventing any other thread from doing calculations and they had to wait, the performance became worse and worse. We theorize that the behavior for two threads was only just better than the sequential algorithm because with only two threads, the overhead from the critical region was not too high and was able to parallelise reasonably well. But after that point, the overhead was very bad.

Sorting had interesting behavior in that we saw a reversal in the performance. It is unclear what causes the performance to degrade, but a hypothesis concerning cache behavior is put forward. As more and more threads are introduced, it may the be case that there is lots of thrashing occurring as threads partition contiguous regions of the array for work, resulting in very inefficient cache usage and bad performance.

We see that figure 12 supports the notion that the threads are working less effectively. The CF drops for sorting as threads increase, demonstrating the increasing idling occurring. We also notice the odd behavior of atomic with threads equal 8. This was likely resulting to experimental conditions on Setonix at that time, sometimes resulting in variations in thread behavior. Many observations were taken for atomic with threads equal to 8 to stnadardize the behavior.

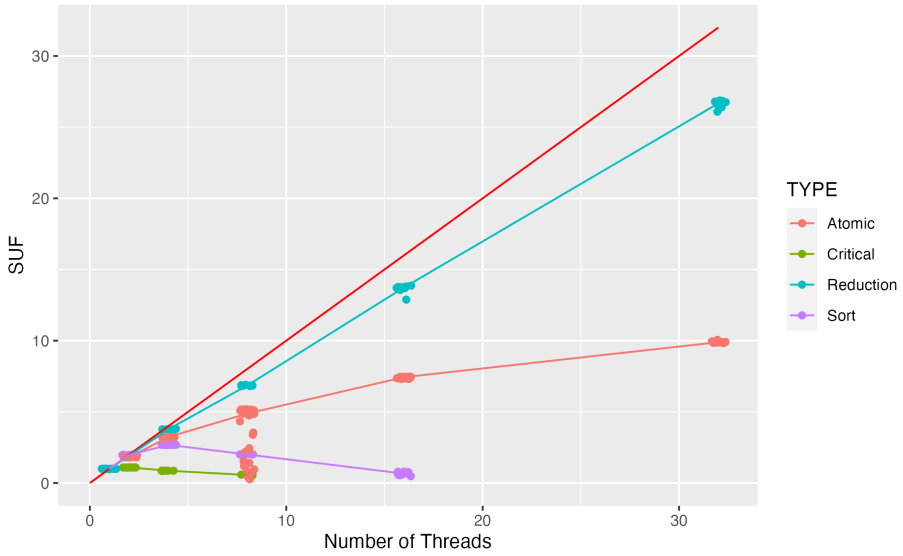

Figure 13: SUF measurements over atomic, critical, reduction and sort variants. SUFS for the sort variant were computed from the median of the sort sequential run times and each sort parallel run time. SUFs for everything else were computed from the median of the reduction sequential run times and each parallel run times.

Finally, we examine the SUFs, which demonstrate that although atomic may have been able to get good elapsed times along with reduction, it actually was not parallelising well at all, at about a third of the theoretical SUF.

4 Conclusion

Performing experiments on Setonix has demonstrated large variation in performance under different conditions and using different parallel strategies. We saw predictable and unpredictable behavior. Predictable behavior included good parallelisation achieved when adding threads using reduction clauses, and speed up remaining constant when adding more fishes. Unpredictable behavior included cpu conditions and usages during different times resulting in large variations in some experiments.

For this algorithm, modifying the schedule type and chunk size from their defaults did not show any performance improvements. The default chunk size — an even split of iterations between each core — completed faster than any other setting. Static scheduling performed better than dynamic at all chunk sizes. Guided scheduling showed some benefits over static when both are using smaller chunk sizes. Runtime scheduling performed especially poorly.

The effects of using an ordered clause in parallel threads were measured.

Sorting was illustrative to see hypothesized cache thrashing take place. There was also demonstrated weaknesses in using critical sections in loops shared by many threads and a predictable performance drop occurred.

Finally, the results show that the best results according to speed-up and parallelisation efficiency were achieved when using the reduction clauses, along with static and default chunk sizes.

References

[1] C. Bastos-Filho, F. Lima Neto, A. Lins, A. Nascimento, and M. Lima, "A novel search algorithm based on fish school behavior," Nov. 2008, pp. 2646–2651. DOI: 10.1109/ICSMC.2008.4811695.

[2] M. Kerrisk, *Rand(3)*, Jun. 24, 2023. [Online]. Available: https://man7.org/linux/man-pages/man3/rand.3.html.

[3] A. Espinosa, *Setonix software environment*, Jun. 9, 2023. [Online]. Available: https://support.pawsey.org.au/documentation/display/US/Setonix+Software+Environment.

[4] M. Kerrisk, *Clock_gettime*(3), Jul. 14, 2020. [Online]. Available: https://www.man7.org/linux/man-pages/man3/clock_gettime.3.html.

[5] A. Espinosa, *Setonix general information*, Sep. 15, 2023. [Online]. Available: https://support.pawsey.org.au/documentation/display/US/Setonix+General+Information.

[6] Wikipedia, *Zen 3*, Sep. 24, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Zen_3.