OpenMP Coursework

Ben Kitching-Morley

April 23, 2019

1 Introduction

In this project the performance of various OpenMP scheduling methods has been analysed through execution of test code (written in C) on the Archer supercomputer. OpenMP is an API used for shared memory based multiprocessing. In contrast to MPI which can be used to run code across many nodes or computers, the shared memory nature of OpenMP limits its application to a single compute node. An program utilizing OpenMP can be run on any number of threads, as a thread is a computational concept, however, there is usually little payoff from using more threads than the number of cores in the CPU. Each login compute node on Archer has 24 cores, meaning that code was tested across a range of thread numbers from 1 to 24.

The test code used in this project consists of two nested loops, loop1 and loop2, which pose different challenges for optimization. The first loop is uniform in that each iteration of the outermost loop is associated with the same amount of computational work. On the other hand in the second loop the iteration over the second iterable changes as an irregular function of the outermost iterable.

The first part of this project is focused on the inbuilt scheduling options in OpenMP, which are as follows:

- Static - Here the iterations are broken into chunks of size chunksize. The threads execute the chunks cyclically. For example if there are 8 iterations, 3 threads and the chunksize is 2 then thread 1 executes iterations 1 and 2, and then 7 and 8, thread 2 executes iterations 3 and 4 and thread 3 executes iterations 5 and 6. If no chunksize is provided the iterations will be split as evenly as possible into a number of chunks equal to the number of threads.
- Dynamic - This schedule works in a similar way to the Static schedule, execept that each chunk waiting to be executed is given to the first thread that is free when that chunk is reached. In this way which threads execute which chunk is uncertain.
- Guided - This schedule is similar to dynamic, except that instead of the size of chunks being fixed, chunks instead decrease in size exponentially as the loop is executed. More specifically the next chunk is a given by $\frac{No.\ iterations\ left}{No.\ threads}$. If the chunksize parameter is given then this number gives the minimum size of chunks.
- Auto - This scheduling method leaves the exact details of the parallelism up to runtime. In order for this method to optimize a loop well its necessary for it to be called several times so that a good schedule can be evolved.

In the second part of this project the Affinity scheduling algorithm (As described in the coursework sheet) was implemented in c. This method was then tested in a similar way to the inbuilt OpenMP scheduling methods.

1.1 Analysis of loop structure

loop 1 is a simple nested loop over i and j from 1 to 729, with the computational content of the loop being the following function call to cos() from the math.h library. We therefore expect the workload of loop1 to be roughly uniform. Moreover, changes in workload would likely affect broad ranges of i as opposed to individual values. For example, if cos(x) were harder to evaluate if its gradient were steeper, or if the lookup for b[i][j] were more time consuming for i larger than a certain value.

```
a[i][j] += cos(b[i][j]);
```

In contrast loop2 has very dramatic CPU workload dependence on i. The form of loop2 is detailed below, with its key feature being the jmax parameter.

```
for (i=0; i<N; i++){
  for (j=0; j < jmax[i]; j++){
    for (k=0; k<j; k++){
      c[i] += (k+1) * log (b[i][j]) * rN2;
```

The value of jmax[i] is either initialized to 1 or to N, with the latter case representing a significantly larger amount of work. More explicit enumeration of the initialisation of jmax shows that it takes the value of N only when $i\%(3 * (i//30) + 1)$ evaluates to zero. As i increases this occurs less and less frequently, meaning that loop2 is significantly front loaded.

2 Inbuilt scheduling options

All of the inbuilt scheduling methods described in the introduction were executed on Archer, using a range of chunk sizes and run on various thread numbers. All runs were repeated multiple times so that a more accurate mean result could be obtained and standard deviations could be found.

2.1 Implementation

The c test code was made parallel by a pragma omp parallel for directive before the outermost loop of each of the 2 nested for loops. The code snippet taken from ***parallel.c*** shown below details the implementation for loop1. Note that schedule(runtime) was used so that the code could be applicable to any OpenMP inbuilt scheduling method.

```
void loop1(void) {
  int i,j;
  #pragma omp parallel for default(none) shared(a, b) private(i, j) schedule(runtime)
  for (i=0; i<N; i++){
    for (j=N-1; j>i; j--){
      a[i][j] += cos(b[i][j]);
    }
  }
}
```

Submission to Archer was done via. .pbs scripts, of which four were used in producing the data seen in the plots in this section:

- ***single.pbs*** This was used to run a sequential version of the test c code. The result of this was later used to calculate the speedup as a result of the OpenMP parallelism.

- ***auto.pbs*** and ***static.pbs*** Were used in running the auto and static schedules with unspecified chunksizes accross a range of thread numbers.
- ***parallel.pbs*** This script represents the majority of CPU time in generating data for this part of the project. The BASH code shown below was used in this file to cycle through different scheduling algorithms, chunksizes, thread numbers and repeat runs.

```
for SCHEDULE in static dynamic guided; do
  for CHUNK_SIZE in 1 2 4 8 16 32 64; do
    export OMP_SCHEDULE="${SCHEDULE}, $CHUNK_SIZE"
    for NUM_THREADS in 1 2 3 6 12 24; do
      export OMP_NUM_THREADS=$NUM_THREADS
      (time aprun -n 1 -d $OMP_NUM_THREADS ./$OMPPROG) 2>&1 > my_output.txt
```

Each run the time taken for loop execution was appended to a .csv file. The output of the loop totals was collected into a c_logs directory so that each run could be validated. The analysis of the results.csv data was done using the pandas package in Python.

2.2 Results

2.2.1 Performance on six threads

The parallel code was executed on six threads in Archer using the four inbuilt scheduling methods of OpenMP. Each schedule was run with chunksizes running through $[1, 2, 4, 8, 16, 32, 64]$. In the case of auto and static runs were also executed where no value for chunksize was specified. In all cases at least 15 repeat runs were executed. Table1 shows the reference times for the single threaded runs

loop	time/s
loop1	0.041178
loop2	0.538096

Table 1: Times taken for code execution of sequential code.

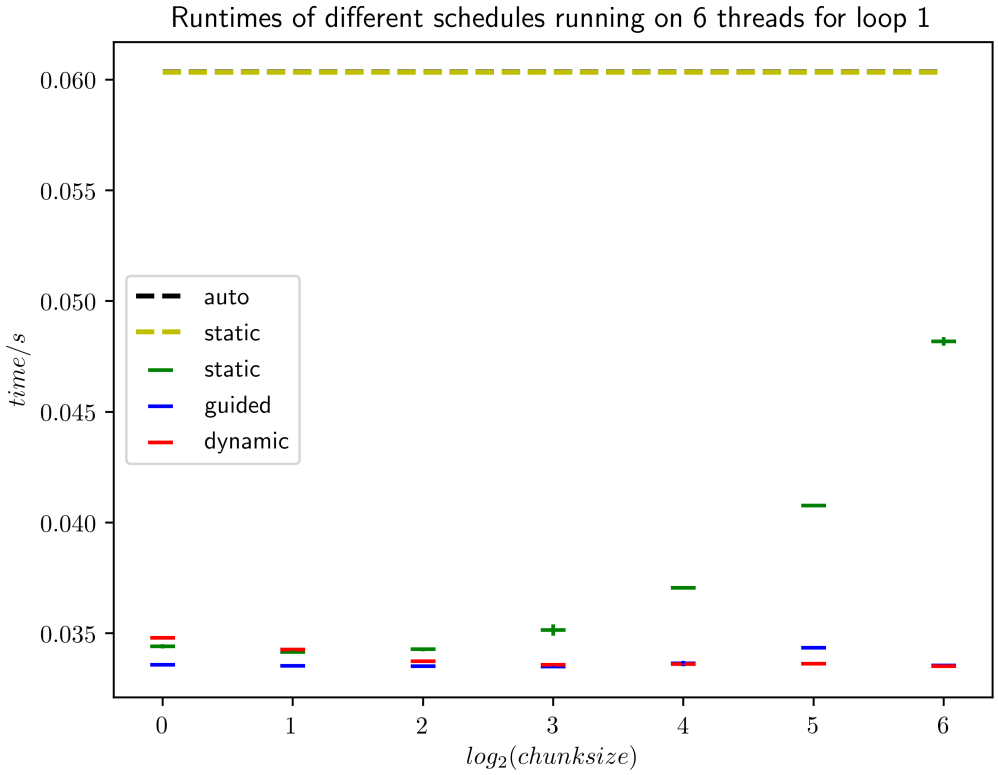

Figure 1: Average execution time for loop1 of different scheduling methods running on 6 threads. Vertical bars show the standard deviation of execution time.

One thing that is clear from fig.1 is that while all three scheduling methods have similar performance at small chunksize, as chunksize increases static becomes dramatically less efficient. One explanation for this is noting that 729 iterations doesn't divide particularly evenly by the chunksize. For example, with a chunksize of 64 729 divides to around 11.4. This means that thread 6 will be idle for around 30% of the time. This has the side effect that each of the other 5 threads have an extra 7 iterations as opposed to chunksize=1. Since the order is fixed then whichever thread is limiting of these 5 will determine the runtime. The second

effect that could explain this pattern is the following. If the workload slowly varies as i changes for whatever reason, then larger chunksizes might leave one thread with more work than another. Since thread positions are fixed there is a good chance this thread will be the limiting thread apon execution.

Because of the ability of the execution to be given to the first available thread, dynamic and guided can't suffer from this "doomed from the start" issue where 1 thread will always have a greater workload than the others. Instead that thread will take the longest at the start, and so end up with the final smaller chunk to finish off the execution.

Lastly, it should be noted that for small chunksizes dynamic performs worse than guided. This can be understood by considering that dynamic involves a great deal of overhead as a result of the first come first serve issuing of tasks to threads. While guided also has this to some extent it is greatly reduced by the fact that guided issues large chunks at the start of the scheme.

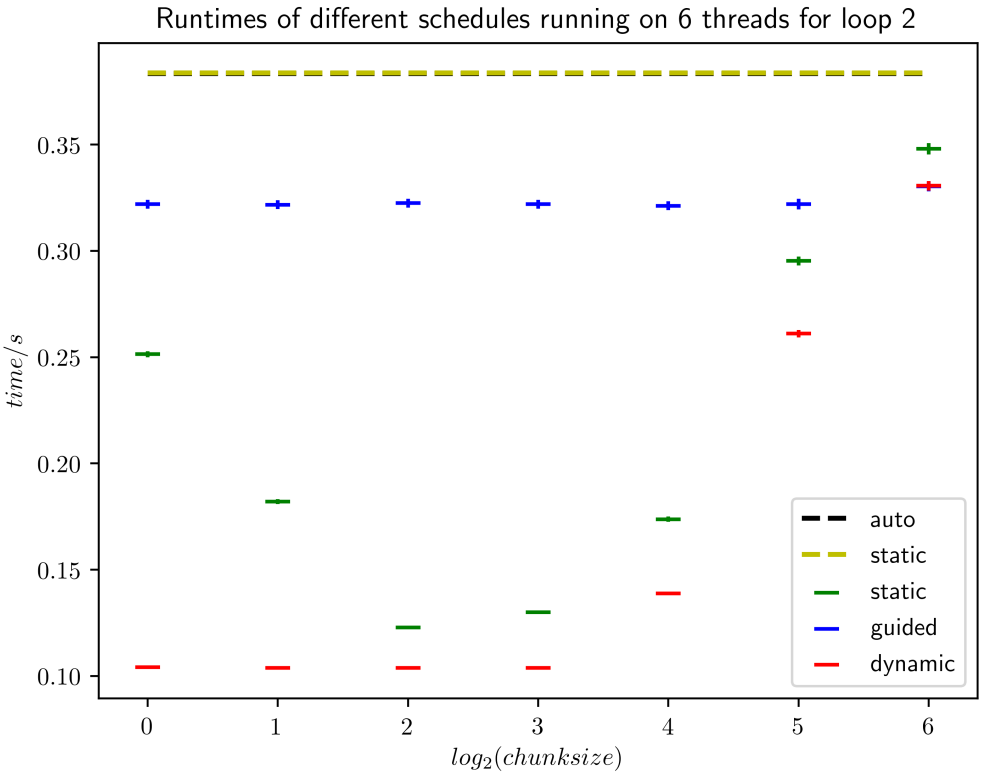

Figure 2: Average execution time for loop2 of different scheduling methods running on 6 threads. Vertical bars show the standard deviation of execution time.

The poor performance of all scheduling methods at large chunksizes is expected given the highly front-loaded nature of loop2(). This results in the first thread carrying a large fraction of the workload for the entire task. Given this it is possible for this single beginning chunk to be the final chunk executing when the program finishes. This same reasoning explains why guided performs so poorly for any chunksize, as in this schedule the first few chunks are large regardless.

In this loop where the workload is so dependent on i, it is inevitable that static will suffer from one thread being more loaded than others at the start. Moreover, because the total execution time of this loop is much greater, the speed up static offers with less overhead becomes less of a significant factor. It is therefore

unsuprising that dynamic outperforms static at all chunksizes.

Prehaps the most surprising feature of these results is the degradation of performance of static as the chunksize gets small. Explaining this requires further analysis of the expression for determining jmax. As well as noting that this expression front loads the loop, it is also apparent that when i is a multiple of 30, then the expression evaluates to zero and there is larger computational work for that iteration. Considering the case of chunksize=1, since static orders the threads cyclically, whichever thread starts on i=0 will also execute all multiples of 6. As 6 factors into 30 then this thread will execute all multiples of 30, giving it a considerably larger CPU workload and likely making it the most loaded thread. This effect is also present for chunksize=2 to a lesser extent, as in this case 2 threads will share the load of executing the multiples of 30.

2.3 Varying thread number

The best performing schedules from the previous analysis on six threads were tested over a range of thread counts for each of the loops. Through reference to the time taken in the sequential code, the speed up was calculated and plotted against the thread number in each case.

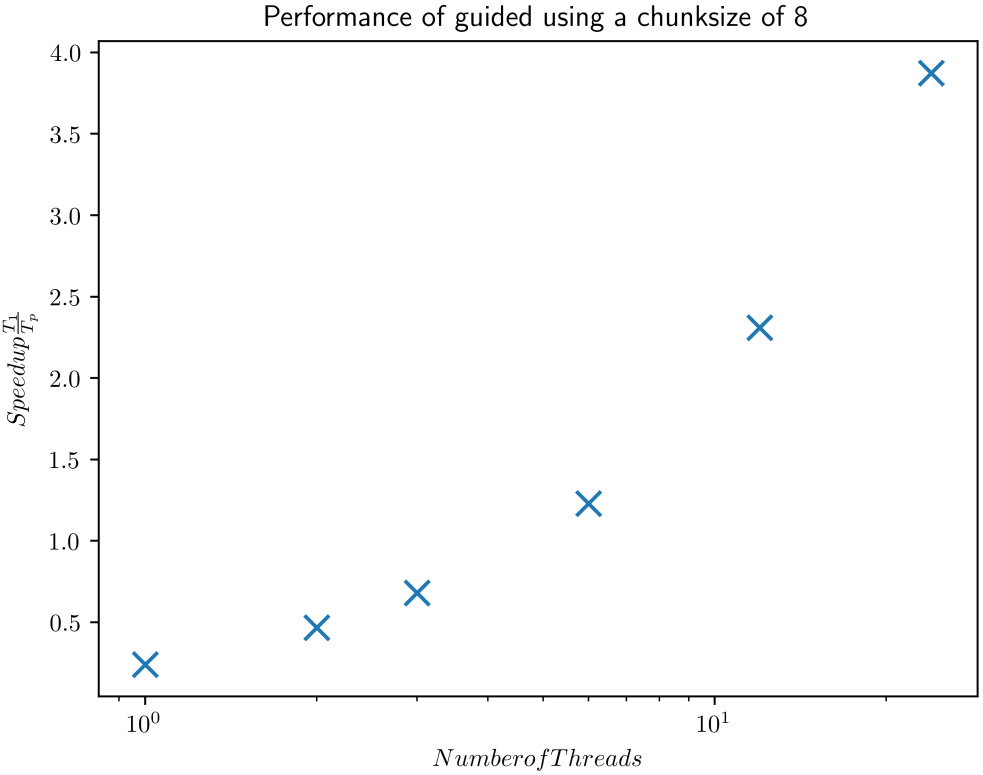

Figure 3: Average execution time for loop1 across a range of thread numbers using the best performing schedule for 6 threads.

Falling in line with expectations the speed up resulting from this scheme correlates positively with the number of threads. Performance at smaller thread numbers is significantly affected by the additional overhead

involved with implementing OpenMP routines. As a result, speed up of higher than 1 is only observed for 6 threads and higher. Users who don't have access to a supercomputer will therefore be unlikely to see any improvement on their code speed from implementing OpenMP directives on their local machine.

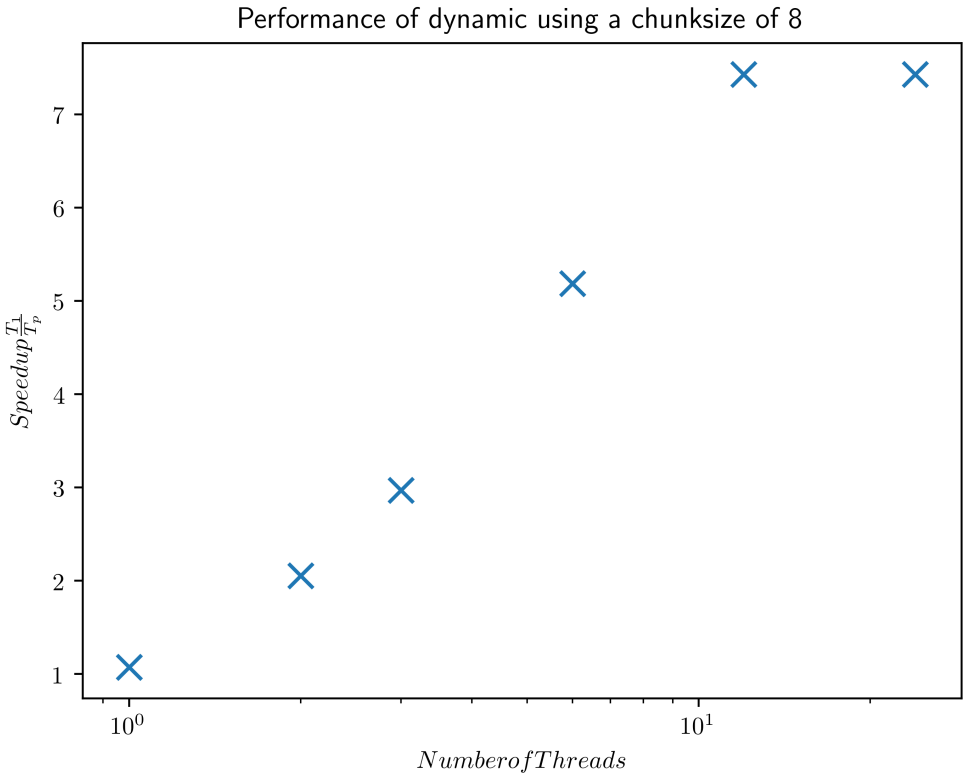

Figure 4: Average execution time for loop2 across a range of thread numbers using the best performing schedule for 6 threads.

fig.4 shows a much more impressive speed up for the parallel code as compared to the sequential than the loop1 schedule. This can be explained by noting that loop2 involves more computationally intensive iterations, so there is a better pay off for the increased overheads involved in parallel code. No speed up is observed in going from 12 threads to 24 threads. This can again be explained by analysis of the jmax formula. for i¡30 the expression evaluates to zero for all i. Therefore, with a chunksize of 8 some threads will have at least 8 computationally intensive iterations. Calculation of the total number of these intensive operations yields 67. For 12 threads this already means that these 67 are not split optimally over the threads as 67 / 12 = 5.6. By further increasing the thread number no performance gain is seen as the threads which initialize with 8 intensive iterations are the limiting factor in the execution time.

3 Affinity scheduling

3.1 Implementation

Implementation of the affinity scheduling method was done using c and executed on archer in the same manner as previous methods. The following pseudo-code outlines the methodology used in the implementation of the algorithm, for reference the c code is in the script. ***affinity.c***.

```
1 num_threads = argv()
2 num_remaining = N
3 local_sizes = initializer_sizes(N, num_threads)
4 local_lower = initializer_lower(N, num_threads)
5
6 #pragma omp parallel shared(num_remaining, local_sizes, local_lower, num_threads)
7     thread_num = omp_get_thread_num()
8
9     while(num_remaining >= 0)
10         #pragma omp critical
11             if(local_sizes[thread_num] > 0)
12                 temp = local_sizes[thread_num]/num_threads
13
14                 if(temp == 0) : chunksize = 1
15                 else : chunksize = temp
16
17                 lower = local_lower[thread_num]
18                 upper = lower + chunksize
19
20                 local_lower[thread_num] = upper
21                 local_sizes[thread_num] -= chunksize
22                 num_remaining -= chunksize
23
24           else
25                 index = max_index(local_sizes, num_threads)
26
27                 if(num_remaining == 0) : num_remaining = -1
28
29                 else
30                     Do the same as for updating a threads own local set except index
31                     with [index] instead of [thread_num]
32
33         if(num_remaining >= 0)
34             loop(lower, upper)
```

- **line 1 - num_threads** is the number of threads to be used is obtained from a commandline argument passed into the compiled script
- **line 2 - num_remaining** keeps track of the number of iterations yet to be assigned to a thread
- **line 3 - local_sizes** is an array which will keep track of the number of iterations left in each threads local set. so **local_sizes[i]** is the number of iterations left in thread i's set
- **line 4 - local_lower** is an array that keeps track of the lower bound of the next chunk in each threads local set. For example, if thread 3's next chunk is from i=60 to i=65, then **local_lower[3]** equals 60.

- **line 6** - Initiate the parallel region with sharing of the above variables, which will allow all threads to see the current state of the schedule.
- **line 7** - **thread_num** stores each threads ID
- **line 9** - Threads should continue to execute iterations as long as there are iterations left to be completed
- **line 10** - It is very important to avoid race conditions that any updates to the shared parameters which represent the state of the schedule occur within a critical region
- **line 11** - Check if the thread in questions local set is empty. If it isn't then execute iterations from its local set
- **line 12** - new chunksize set at $\frac{1}{p}$ times the remaining size of the local set
- **lines 14-15** - The chunksize should be at least 1
- **lines 17-18** - Set the lower and upper bounds on the region of i that this thread will iterate over.
- **lines 20-22** - Update the schedule tracking parameters
- **line 24** - In the case this threads local set is empty
- **line 25** - Obtain the identity of the "most loaded" thread
- **line 27** - This line has been included because of the possibility of 2 threads passing the while loop when only one iteration is left to go. One thread will enter the critical region first and set **num_remaining** to zero. When the second thread enters the critical region, this check will cause it to set **num_remaining** to -1 which acts as a flag for later use.
- **lines 29-31** - If there are threads left to be executed in other threads local sets then the scheduling parameters will be set as they are from lines 12 to 22, except using the index of the most loaded thread as opposed to the threads own ID
- **line 33** - If there is no flag (see line 27)
- **line 34** - Run the loop between lower (inclusive) and upper (exclusive). This section of the algorithm is outside of the critical region and so can be executed by multiple threads at once

3.2 Results Overview

The performance of the affinity scheduling method is compared to the schedules which gave the best performance on 6 threads.

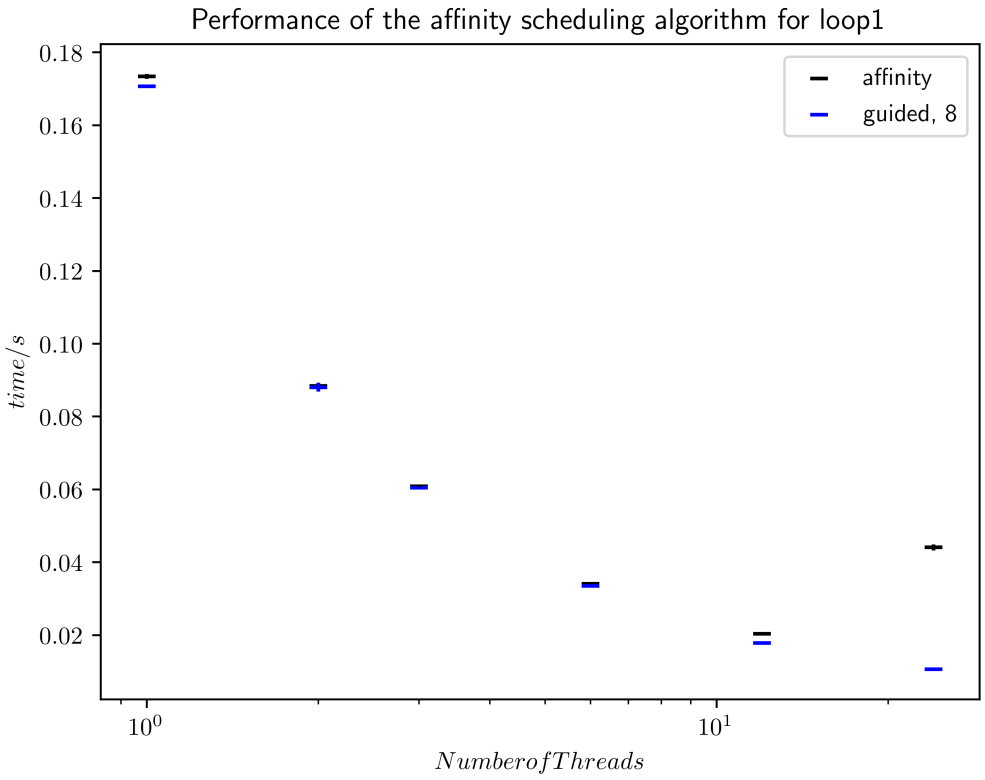

Figure 5: Average execution time for loop1 across a range of thread numbers using the affinity scheduling method. The best performing schedule for 6 threads is included for comparison. Vertical lines represent the standard deviation of execution times.

In the case of fig.5 the affinity scheduling algorithm closely follows the best performing inbuilt schedule on six threads (guided, 8). This follows expectations as similarly to guided, affinity executes chunksizes which decay exponentially in size as the algorithm proceeds. However, the initial size of the guided chunksize is independent of chunksize (unless the stated chunksize is larger). This contrasts to the affinity schedule where the size of the initial chunks scale as $\frac{1}{num\_threads}$. The effect of this is a large increase in overhead as the number of threads becomes large. The critical region may well become the limiting factor in performance for the 24 thread run, where the affinity method performs very poorly.

Given the extra programmer resources required in implementing the affinity scheduling algorithm it is not suitable for simple loops such as loop1, where the inbuilt OpenMP algorithms are highly efficient.

For small chunksizes fig.6 demonstrates the poor performance of the affinity scheduling method as compared to dynamic, 8. This can be explained though recalling previous analysis of the jmax function. The initial chunk of thread 1 is the first $\frac{N}{num\_threads^2}$ iterations of the outer loop. Because the first 30 iterations all have jmax=N, then a large amount of computational work sits at the beginning of this loop. Therefore, at small thread numbers the performance of the affinity scheduling algorithm is degraded by a single thread being

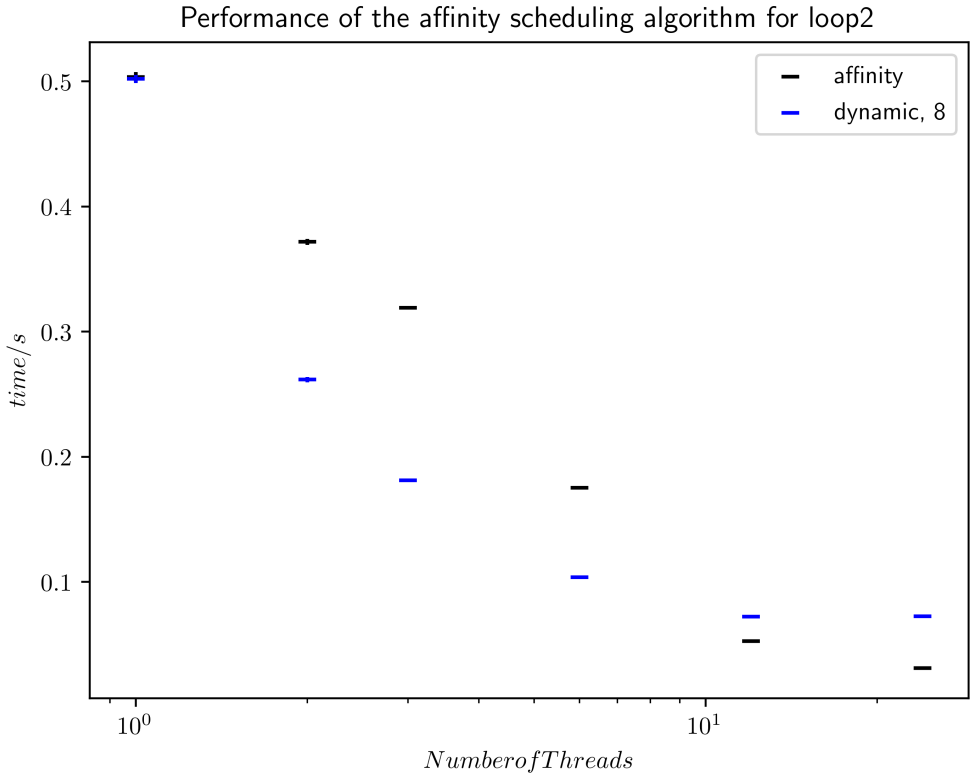

Figure 6: Average execution time for loop2 across a range of thread numbers using the affinity scheduling method. The best performing schedule for 6 threads is included for comparison. Vertical lines represent the standard deviation of execution times.

overloaded at initialization.

However, for similar reasons the converse becomes true at high thread numbers. Here (dynamic, 8) is limited by the first 8 iterations of the first thread, putting a floor under its runtime. This effect can be seen by the fact no speed up is achieved in going from 12 to 24 threads. In comparison, the affinity scheduling algorithm initialises very small chunks at higher thread numbers and so is able to loadshare more effectively between all of the threads, giving it better performance compared to dynamic, 8.

The affinity scheduling method as implemented in this report does suffer from increased overhead, as compared to the inbuilt methods. For reference, while (dynamic, 8) performs poorly on 24 threads, (dynamic, 2) has very impressive performance with a runtime of 0.0278441s, as compared to affinity's 0.030947s.

3.3 Comparision to each of the scheduling methods

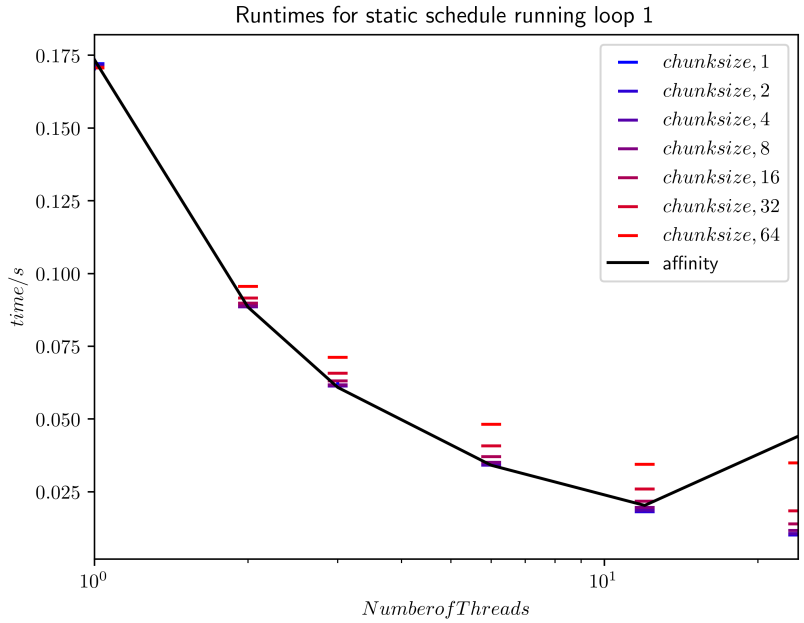

Figure 7: Performance of the affinity scheduling method against static for loop 1

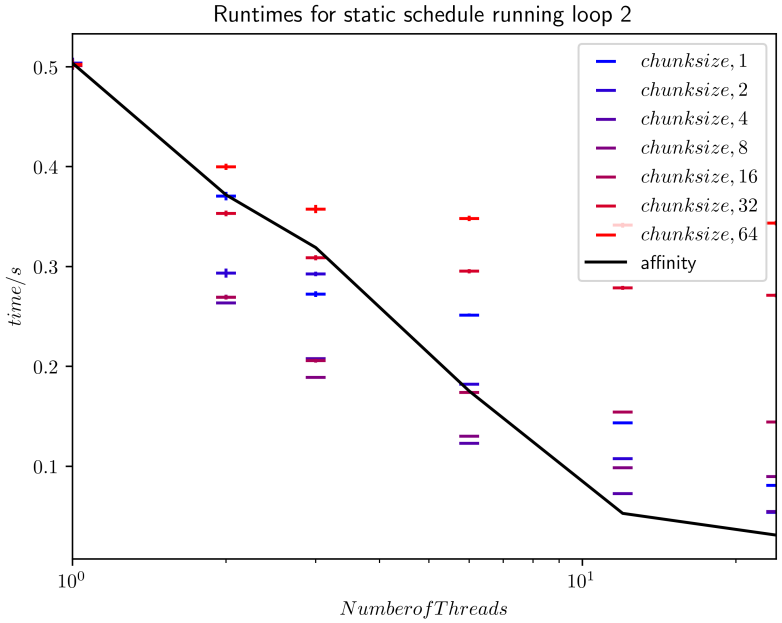

Figure 8: Performance of the affinity scheduling method against static for loop 2

Figure 7 demonstrates strong performance of the affinity scheduling algorithm against the static scheduling method across all chunksizes, and over all thread numbers except 24 threads. At 24 threads, the low overhead of static as compared to affinity gives it a decent speed up in comparison.

Consequently, static is a reasonably well performing method for loops where the workload doesn't change much as the loop progresses. However, for much more variable loops such as loop2 the superior ability of affinity to alter the schedule as execution progresses allows it to much more effectively share the work load amongst all threads. This is especially true when the number of threads is large.

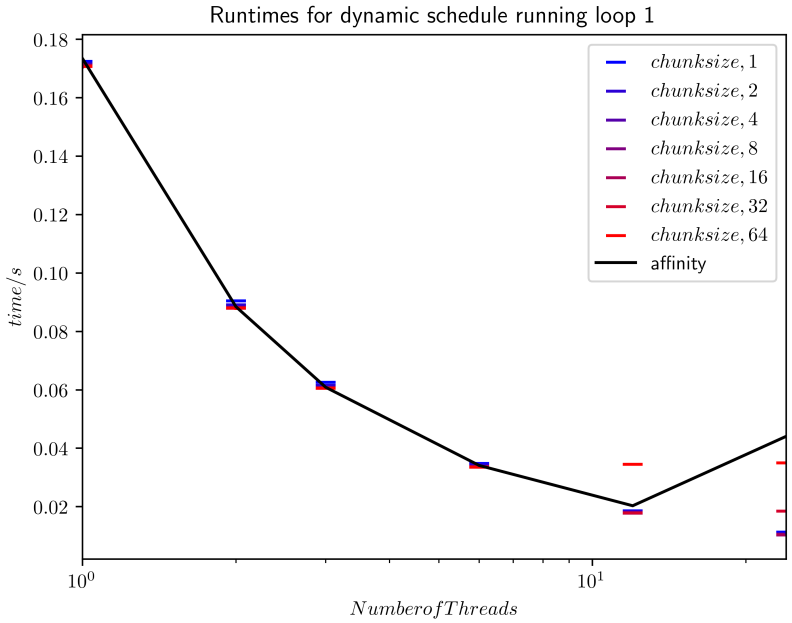

Figure 9: Performance of the affinity scheduling method against dynamic for loop 1

Figure. 9 displays the highly aligned performances of the affinity scheduling method against dynamic. The only difference between the two being at high thread numbers, where the affinity method is held back by its increased overhead. This is possibly an indication that the algorithm hasn't been implemented as efficiently as possible. Figure. 11 elucidates a similar picture for the performance of affinity in comparison to guided when acting on simple loops such as loop1.

In comparison, Figure. 10 shows a great disparity in the speed-up offered by multi-threading between the affinity scheduling method and dynamic for loop2. At low thread numbers the large starting chunk sizes hold back affinity. In comparison at higher thread numbers it scales comparably to the most optimal chunksizes for dynamic.

As previously discussed in this report, the execution of smaller initial chunksizes, at high thread numbers, gives affinity a performance advantage as compared to guided for loop2. This is clearly displayed in Figure. 12.

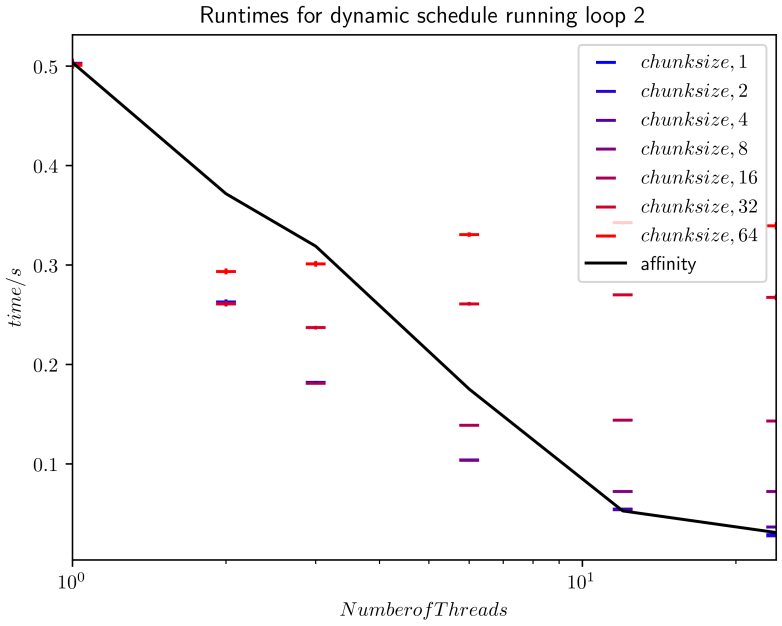

Figure 10: Performance of the affinity scheduling method against dynamic for loop 2

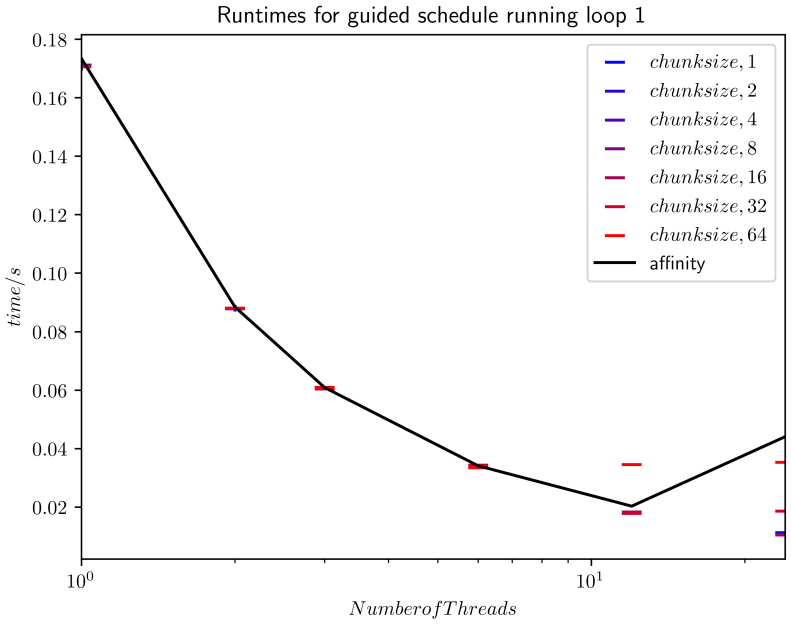

Figure 11: Performance of the affinity scheduling method against dynamic for loop 1

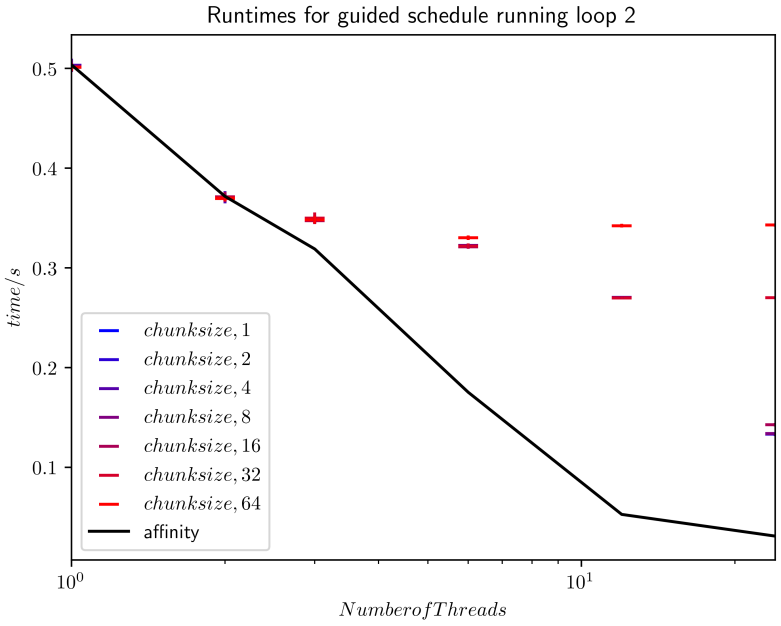

Figure 12: Performance of the affinity scheduling method against dynamic for loop 1

4 Conclusion

One thing that is clear from the results of this analysis is that there is no scheduling method that is best in any situation. For the optimal performance it is in the best interests of the programmer to study the form of the for loop they are attempting to make parallel. If the load is roughly evenly balanced through the loop, and the run-time of each iteration is low then static is an appropriate choice. For more demanding but still evenly distributed loops then guided offers the best performance.

The more challenging situation that can arise is when the loop is unevenly loaded. In this case dynamic is usually the best option, but the programmer should experiment with different chunksizes to get the optimum performance. If the code is going to be implemented over a large range of thread numbers, then affinity scheduling may over a good compromise method. While this report found that this scheduling algorithm was never the fastest, it did find that it performed respectably on both loop and over a range of chunksizes. It therefore can be expected to offer more reliable speed up than a specific inbuilt OpenMP method.

5 Code listing

Submitted as part of this coursework submission are the following files:

C scripts

- **single.c** - Script containing the sequential code, for use in calculating the speed-up as a result of OpenMP directives

- **parallel.c** - Script used to generate timings for the inbuilt OpenMP schedule methods
- **affinty.c** - Script implementing the affinity scheduling algorithm

.pbs scripts

- **single.pbs** - Script used for executing the sequential code on Archer. When run this script also outputs **a_answer.txt** and **c_answer.txt** which store the correct answers for the loop1 sum and for the loop2 sum. These answers are used to validate the parallel code later on
- **parallel.pbs** - This script is used in running all of the chunksize specified runs, e.g. (static, n), (dynamic, n) and (guided, n)
- **auto.pbs** - Used in running the auto schedule
- **static.pbs** - Used in running the static schedule without specifying a chunksize

.py scripts

- analysis.py - Script used in data analysis through the pandas library. It is also used in producing all the plots used in this report.

.csv files

- results.csv - Contains all results used in producing plots in this report in the canonical csv format

.png files

- **six_threads_loop_1.png** - Figure 1
- **six_threads_loop_2.png** - Figure 2
- **best_6_loop1.png** - Figure 3
- **best_6_loop2.png** - Figure 4
- **affinity_loop1.png** - Figure 5
- **affinity_loop2.png** - Figure 6
- **affinity_loop1.png** - Figure 5
- **static_loop_1.png** - Figure 7
- **static_loop_2.png** - Figure 8
- **dynamic_loop_1.png** - Figure 9
- **dynamic_loop_2.png** - Figure 10
- **guided_loop_1.png** - Figure 11
- **guided_loop_2.png** - Figure 12