Multi-threaded Programming: Scheduling and Affinity Coursework

Jack Tyler: 27513556

April 15, 2019

In the early days of computing, computational power progressed at an alarming rate; constant field scaling meant that transistors could be fabricated smaller and smaller, allowing them to use less power, and for the clock speed of a processor to be increased. However, transistor manufacture has hit a physical limit for scaling (Mcfarland and Flynn, 1995), and as such, increasing the clock speed any further would require highly capable cooling solutions impractical for use. As a result, processor companies have had to find other methods to speed up computer processors, to keep computational power, and Moore's Law, advancing. Perhaps most prominent is the introduction of parallelism, where multiple processor units may exist on a processor 'chip' and execute asynchronously and independently.

This type of parallelism is one of the most common implementations currently used in high-performance computing, to decrease program execution time. However, care must be taken as to how each processor unit – each thread – is synchronised to complete a computational task in unison, and in particular, some methods of synchronisation, or scheduling, may provide better performance for a particular application.

This report investigates the most effective scheduling algorithm for a particular computational problem on the ARCHER supercomputing cluster. In particular, it studies the effectiveness of scheduling algorithms provided by the OpenMP standard, compared to a static partitioned affinity schedule work-stealing algorithm, specifically designed for the computation problem at hand, and identifies many of the limitations and shortcomings in both the paralellisation of the algorithm, and in the analysis method. Knowledge of the OpenMP standard and parallel computing nomenclature is assumed in this report.

Scheduling algorithms

Scheduling algorithms determine the distribution and synchronisation of work to the threads on the processor. It's possible to quantify a scheduling algorithm's performance based on its effectiveness in minimising four separate overheads: synchronisation, where threads must idle to wait for another thread to finish some task; communication, where interactions between processors do not proceed optimally, such as cache misses and non-local memory accesses; worker initialisation, where time is requried to initialise, schedule and destroy pools of parallel workers; and load imbalances, where some threads are idle while others are still performing work. The latter is perhaps the most important for this runtime optimisation, since the loops studied here are relatively poorly balanced. Indeed, a major metric used in this thread is the amount of time threads would spend idling.

We may define two distinct types of self-scheduling algorithms: fixed, and variable.

```
do i = 1,N
    do j = N,i,-1
        a(j,i) = a(j,i) + cos(b(j,i))
    end do
end do
```

Figure 1: Loop 1; note that this loop is well-balanced, and we should expect each thread to perform approximately the same amount of work for this loop.

In fixed systems, each processor will be assigned a fixed amount of work at loop initialisation, and continue computing until all their assigned work is completed. While this can reduce scheduling overhead, in potentially only distributing and synchronising once, this method can be inappropriate for imbalanced loops, since some loops may idle unnecessarily while other threads are still working, providing a bottleneck on execution times. In the OpenMP standard, we may also specify the chunk size – that is, the number of iterations assigned to each thread – which can adjust scheduling performance depending on our load balancing. Small chunk sizes will incur large schediling overheads, but will be able to smooth out load imbalances by constantly reassigning work, regardless of other threads being busy. Similarly, large chunk sizes will reduce scheduling overheads, but amplify and load imbalances.

Variable algorithms will adjust their chunk sizes as the loop progresses. In the OpenMP standard, two such algorithms exist: `DYNAMIC` and `GUIDED`. `DYNAMIC` functions by assigning variable chunk sizes to threads, and working on a 'first-come, first-served' basis. This works well for imbalanced loops, since work can be reassigned while other threads are still idling, at the cost of scheduling overheads.

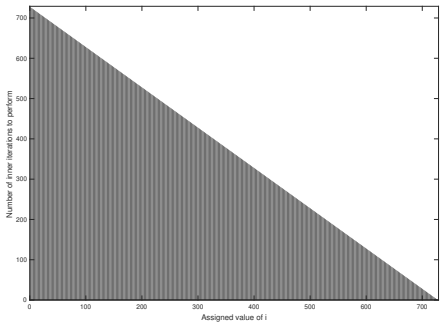
`GUIDED` functions similarly, but by assigning initially large numbers of iterations and then getting exponentially smaller. By doing so, the scheduling overheads are kept relatively small – with the large chunk assignments at the beginning – but the possibility for good load balancing is still present at the end of the iterations, when small chunks are used.

Most compilers will also support an `AUTO` option, which lets the compiler determine the scheduling to implement (be it OpenMP-standard or otherwise). However, there is no transparency in the scheduling implemented here, although there is the possibility of the scheduling changing to reflect observed behaviour with some compilers.

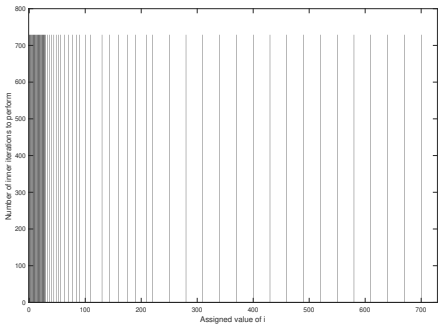
Introduction to the Problem

The problem statement for this work is, at first glance, straightforward: to parallelise two examplar computation loops, and investigate the effect of OpenMP-standard scheduling algorithms, or indeed a third-party algorithm, on program runtime. However, it's worthwhile to first understand and consider potential approaches and pitfalls that may be experienced from an examination of the loop structures.

Loop 1 (Figure 1) is a simple, yet relatively imbalanced nested do-loop; each repetition of the inner do-loop is variable based on the value of `i`. As a result, threads that are assigned a different value of `i` will perform a different amount of work (Figure 2a). We might therefore expect scheduling algorithms that work well with a slight work imbalance to perform well on this problem, such as `GUIDED,`n, or `DYNAMIC,`n. However, since the distribution of work is readily predictable, we may see good performance from the `AUTO` option, which gives the compiler the authority to change the scheduling algorithm as it sees fit (including during runtime).



(a) The number of inner-loop iterations in Loop 1 a thread must perform based on the value of `i` it is assigned; while this is a slight work imbalance, the number of iterations to perform is readily predictable.



(b) The number of inner-loop iterations in Loop 2 a thread must perform based on the value of `i` it is assigned; note that threads may be required to perform either 1 or 729 inner iterations. This is a severe work imbalance that would need to be addressed effectively to maximise performance.

Loop 2 (Figure 3) is a poorly-balanced loop. The `i`-th element of `jmax` is initialised to either 1 or N, depending on the value of `i`. As a result, a thread which is assigned the work of a value of `i` equal to N will perform far more work than the threads with only 1 inner iteration (Figure 2b). Loop scheduling algorithms that are suitable for loops with widely fluctuating work levels, such as `DYNAMIC` or `GUIDED`, would be expected to work well here. More specifically, since the majority of work is scheduled in the first 30 iterations, and approximately every 30 iterations thereafter, performance may be best for chunk sizes less than 30. The work repeated every 30 iterations will also prevent threads from becoming idle and accepting more work.

OpenMP-standard Scheduling: Varying chunk sizes

Implementation

Loop 1 and Loop 2 represent 'embarassingly parallel' problems, and as such, parallelising the routines was relatively trivial: both loops were enclosed in a `!$OMP PARALLEL DO` construct, and set such that their arrays were shared (as well as `rN2` in the case of Loop 2), with all other variables set private. The scheduling algorithm was then set to be allocated at runtime – `SCHEDULE(RUNTIME)` – and the associated `OMP_RUNTIME` environment variable was exported to the compute node at job submission, which was controlled by a lightweight governing bash script that would parametrically generate the PBS submission file for every possible combination of scheduling algorithm and chunk size. The program was designed to take in the number of threads to use as a command line argument to the program execution. The results of an `OMP_GET_SCHEDULE` call as part of the main Fortran program were printed to the screen to ensure the correct scheduling algorithms; the call returns an integer of type `omp_schedule_kind` that can be cross-checked against the scheduling kinds contained in the `omp_lib_kinds` module as part of `libgomp`.

Submitting all of the jobs as a batch also meant that the timing results were more accurate, as jobs were almost-guaranteed to be computed on the same node. Each job was repeated five times, and an average taken, to remove the effects of variable computational performance on each run.

```
do i = 1,N
    do j = 1, jmax(i)
        do k = 1,j
            c(i) = c(i) + k * log(b(j,i)) * rN2
        end do
    end do
end do
```

Figure 3: Loop 2; note that this loop is poorly-balanced as a result of the inner loop being iterated to either 1 or N, depending on the value of `i`.

Results

The results for Loop 1 are given in Figure 4. Clearly, the `AUTO` scheduling performs worse than the serial case (note that, while the AUTO execution time has been extended across all chunk sizes, the chunk size was not specified and this has been done for clarity only.) It is not immediately clear why this scheduling option performs this poorly, as there is no transparency in the scheduling implemented, but it is clear that this option should be avoided.

The performance of the `STATIC, n` scheduling routine decreases with increases in chunk size. This is to be expected, and highlights the differences in balance in the loop; when contiguous chunks of n iterations are distributed, the first threads will always perform more work than the final threads as a result of the work distribution for Loop 1. For low chunk sizes, the high-work loops are relatively well-distributed amongst all threads, and thus there is relatively little idling and generally decreased performance time. However, when high chunk sizes are used, the first thread takes the majority of the work, and thus acts as a bottleneck for the program execution: the higher the chunk size, the longer the final threads are idling waiting for the first thread(s) to finish.

As expected, `DYNAMIC` and `GUIDED` perform well for this loop. Since `DYNAMIC` assigns chunks of loop iterations to threads on a first-come, first-served basis, at small chunk sizes threads which are initially assigned less work will be assigned more work whilst other threads are still computing. `GUIDED` functions similarly, but the loop iterations start off large and get expontentially smaller: thus, for loops with initially high amounts of work, such as this, we would expect good performance.

`DYNAMIC,2` produces the best performance for this problem, but not by any considerable margin. It provides the best overall execution time of 0.03449s, but this is only .23% above the average performance of the `GUIDED` method. In fact, `GUIDED` has generally superior performance at most chunk sizes when compared to `DYNAMIC`, and helps to smooth the slight imbalance in the loops to maximise performance. Therefore, while `DYNAMIC,2` is the fastest scheduling choice here, `GUIDED` may prove to be more versatile across a wider range of chunk sizes.

Also note that the performance of most of the routines with chunk size 1 is also worse than the serial version, highlighting the scheduling overheads of OpenMP.

The results are relatively similar for Loop 2, and are provided in Figure 5.

Again, the `AUTO` scheduling performs poorly, and routines with chunk size 1 perform worse than the serial case.

Because of the unusual loading in this loop, the first 30 iterations are required to do the maximum amount of work. Therefore, scheduling algorithms that spread this initial work out across threads

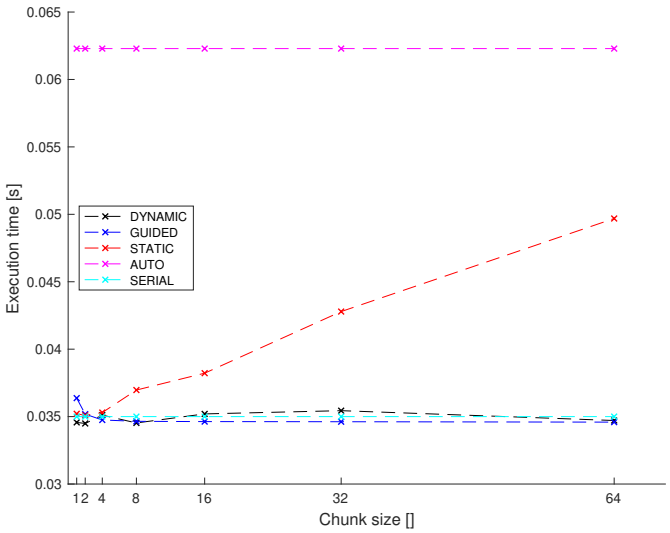


Figure 4: Loop 1 execution performance for different scheduling algorithms: `DYNAMIC2` provides the best performance here overall, but `GUIDED` performs better across a wider range of chunk sizes.

would reduce idling times for the unloading threads and thus increase performance. This explains the behaviour of many of the scheduling algorithms above: when the chunk size begins to increase – in the case of `DYNAMIC` and `GUIDED`, the minimum chunk size – then fewer threads get assigned the initial amounts of work, and thus execution times will rise.

Note again that performance generally suffers with chunks over 32, which is as predicted upon first examination of the loop, as a result of high-work iterations being spaced 30 iterations apart.

Interestingly, `DYNAMIC` has constant performance for all chunk sizes. This may suggest that, since the chunk size passed to the scheduler specifies only the minimum chunk size used, the algorithm is using only large chunk sizes.

`GUIDED`, and specifically `GUIDED,8`, has the best execution time for this problem. This is again expected, with `GUIDED` allowing threads with low amounts of work to accept more, and also forcing the latter portions of the loop iterations to be assigned in exponentially smaller chunks, spreading the workload better.

OpenMP-standard Scheduling: Thread numbers

Now we have determined the 'best' scheduling options for each loop, we may investigate their thread-scaling. Ideally, we would experience a true linear scaling, where the execution time decreases as a linear function of the number of threads used. However, consider Amdahl's Law, which is often-cited in parallel computing:

$$\text{Speedup}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

which gives the factor of speedup, where s represents the local speedup – say, by parallelising – and p represents the amount of the program that is available to be parallelised. Amdahl's Law provides a good metric for determining the maximum speedup we can expect from a parallel program, and provides the deduction that speeding up programs by using parallelism will be limited by our serial

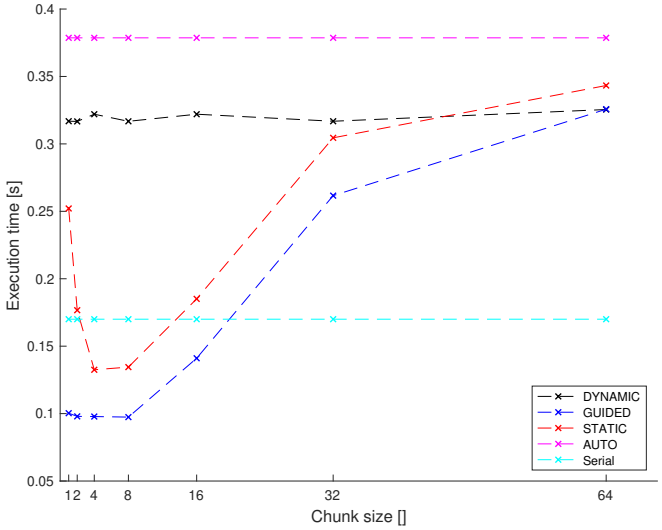


Figure 5: Loop 2 execution performance for different scheduling algorithm. `GUIDED,8` provides the best performance for Loop 2, but performance generally reduces with increasing chunk size. This is a quirk of the unusual loading for Loop 2.

regions. In general, our processor efficiency – how well we are using all our threads – is a major limitation on program runtime.

As a result, it is rare (or near impossible) to see true linear scaling; even for a program that has 95% of its code base eligible to be paralellised, speed-up will never exceed a factor of 20, no matter how many threads are used.

Figure 6 provides the speedup for both loops as the number of threads is varied. It's immediately obvious that program execution times decrease with increasing thread numbers, as expected.

However, the scaling is generally poor, even despite the relatively large amount of parallelisable source code (recall that we are timing only the loop execution). Loop 1's scaling factor is 0.7332 per additional thread, and Loop 2 0.1233.

There are several reasons why this performance is lower than the theory predicts. For Loop 1 (Figure 7), the optimal scheduling configuration has a low chunk-size, which is likely to incur significant scheduling overheads. Similarly, while the loop is only mildly imbalanced, we are also expecting some communication overhead when threads are reassigned work and the data must be moved in memory.

DO SOME PROFILING HERE?!?!

For Loop 2 (Figure 8), the scaling is generally much worse. Much of this, as discussed previously, will be due to synchronisation overheads when threads are kept working, and the extra few high-work iterations spaced evenly through the runtime will prevent threads becoming available. Other overheads, such as communication overheads, will also be present, but dominated by the synchronisation overheads.

Affinity Scheduling

In view of the performance of the OpenMP-standard loops, attention now turns to an implementation of static partitioned affinity scheduling, an algorithm developed by Markatos and LeBlanc (1994). The algorithm is designed for parallel loops nested inside serial loops, and aims to ensure that cache misses and processor idling times are minimised.

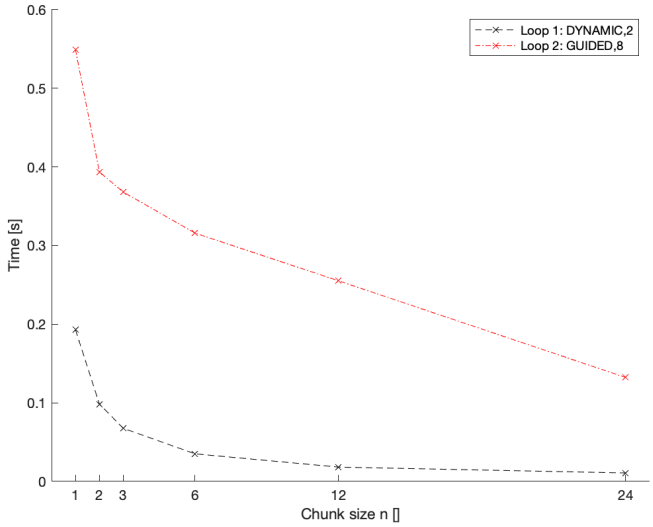


Figure 6: Loop execution times with respect to the number of threads used; we do not obtain linear scaling here, as a result of OpenMP overheads, but we do see a general increase in performance with thread numbers, as expected.

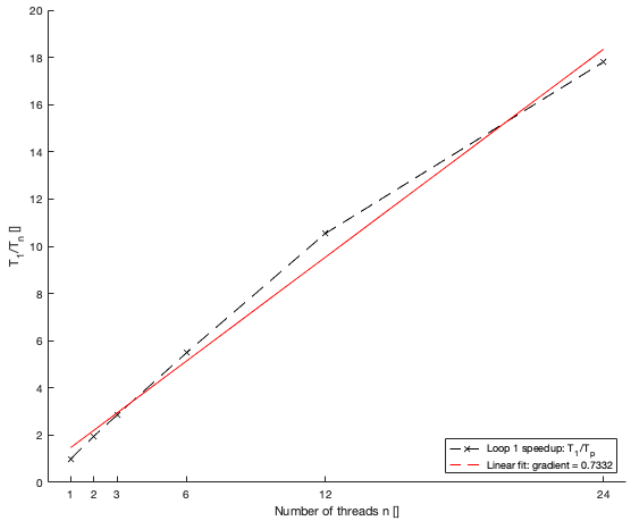


Figure 7: The speedup for Loop 1 compared to the serial execution times. For low thread numbers, the scheduling overhead is not overcome, and performance suffers.

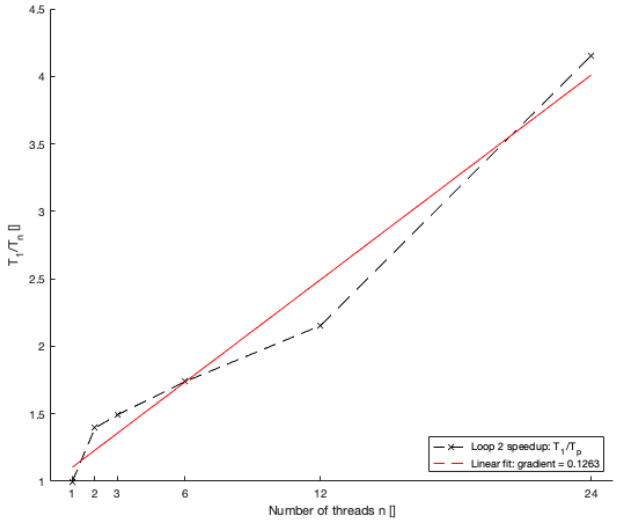


Figure 8: The speedup for Loop 2 compared to the serial execution times. The scaling here is poor as a result of the loading keeping threads working when they would otherwise be idle.

The algorithm functions by allocating an equal proportion of the total work to each thread. When a thread finishes each of its initial allocation, it will 'steal' work from the thread with the most remaining of its initial allocation.

Much of the algorithm's speed comes from the determinstic loop allocation at the initial execution of the loop, which ensures that the ith processor holds the ith chunk of iterations. By doing so, the data required for repeated executions of the loop will already be stored in cache, removing the need for relatively expensive memory movements. Also, the algorithm assumes, at first, that the loop is perfectly balanced, and therefore takes initially large chunks of work. Thus, there is only a scheduling overhead at the beginning of the loop.

When each thread does finish its local thread, work is reassigned to smooth the load imbalance and prevent threads from idling. By only reassigning at the point of imbalance, unlike `DYNAMIC` or `GUIDED` which begin to perform large re-scheduling in the latter portions of the loop, the scheduling and memory re-allocation overhead is only incurred when required, and memory is moved at most twice when work is reallocated.

Implementation

A description the algorithms' implementation follows:

Each thread is initially assigned P/N iterations, where P is the total number of iterations and N the number of threads. Should N not divide evenly into P, then the remainder r is scattered across the first r threads. The number of iterations each thread is to perform is stored in an array indexed by the thread identifier retrieved from `OMP_GET_THREAD_NUM`. Since this array must be shared in the parallel region, the synchronisation of, and access to, the array must be closely controlled. `CRITICAL` sections or `LOCK`s must be implemented to prevent race conditions and unintended overwrites. A discussion on this is given later. If `LOCK`s are used, another array is created that holds the lock from every thread, indexed by the thread identification number.

Other shared variables, such as the number of threads, and the value of P/N, are also shared. By default, all other variables are scoped as `PRIVATE` in all parallel regions.

At this point, each thread is begins work on $1/P$ iterations of its initial assignment, and updates

the (shared) number of iterations array to announce to the other threads that it has taken those iterations to be executed. This contiguous assignment ensures that repeated iterations will access data already stored in local caches.

This assignment is again enclosed in either a `CRITICAL` region, or `LOCK`s are used to prevent race conditions on each element of the array. From here forwards, the number of iterations array tracks the number of iterations remaining for each thread.

The upper and lower bounds of iteration counter are then determined parametrically from the thread identification number and the number of iterations remaining; the algorithm starts at the 'bottom' of each thread's chunk and moves upward.

In order to aid the execution of this type of scheduling algorithm, the `loop1` and `loop2` subroutines now take in two `integer`-type arguments, that set the upper and lower bounds for the outer loop. Owing to the variable scoping in the internal procedures in Fortran, the shared arrays `a`, `b` and `c` may be accessed without specification of their OpenMP scope.

This process is repeated for all threads until one or more completes its initial assignment of threads. At this point, it is likely, especially in imbalanced loops, that other threads will still be working. So as to minimise idling times, each 'finished' processor – that is, one that has completed its initial assignment – will use the array of remaining iterations to determine which other thread is currently the most loaded. The finished thread will then 'steal' work from the most loaded thread, appropriately emunlating the thread identifiers of the most loaded thread to properly update the shared arrays to announce to other threads that it has taken some of the loaded thread's iterations to be executed. So that no two finished threads are assigned the same work, `CRITICAL`s or `LOCK`s are used again.

When all threads have finished their local assignment, the loop has completed.

Results: Affinity Scheduling

The results for Loop 1 using affinity scheduling are presented in Figure 9. For the equivalent number of threads as the scheduling methods visited earlier (6), performance is superior.

Again, this is to be expected. The initial, deterministic assignment of iterations to

`CRITICAL` vs. `LOCK`s

There are two methods of general synchronisation defined in the OpenMP standard: `CRITICAL` and `LOCK`s. `CRITICAL` may be thought of as protecting code sections: regions may only be entered by one thread at a time, and thus there is a significant synchronisation overhead whilst threads are waiting to enter the `CRITICAL` region. `LOCK`s act on data, and restrict access to certain parts of data when a thread 'locks' or 'unlocks' a portion of the data.

Two versions of the affinity scheduling algorithms were implemented: one using critical regions, and another using locks. By doing so, the best-performing implementation could be selected for timing, and the impact on speed investigated.

The results of this investigation are presented in Figure 13. It is found that, for Loop 1, the differences are negligible, and for Loop 2 the differences are up to 10% in run time. This may suggest that the original `CRITICAL` regions in Loop 1 were unlikely to hinder run time, implying that the amount of time threads spend doing work compared to assigning work, is low.

Alternatively, for Loop 2, preventing threads from iterating at the start of a `CRITICAL` region appears to be a large bottleneck in program execution.

Such timings or profilings would need to be performed to ascertain the impact of either synchronisation structure on the program. In general, where `CRITICAL` regions reduce performance, they are far easier to implement. For large code bases, it may be a trade-off between programming time and execution time.

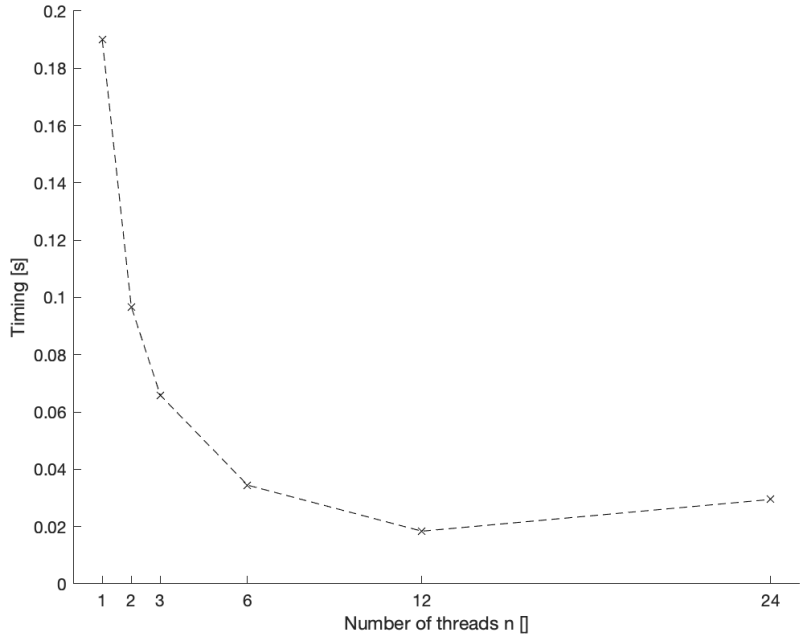


Figure 9: Execution times for Loop 1, scheduled using the static partitioned affinity scheduling algorithm. We obtain better performance as we increase the number of threads, until the array chunks no longer fit in the processor cache.

References

E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994. ISSN 1045-9219. doi: 10.1109/71.273046.

Grant Mcfarland and Michael Flynn. Limits of Scaling MOSFETs. (January), 1995.

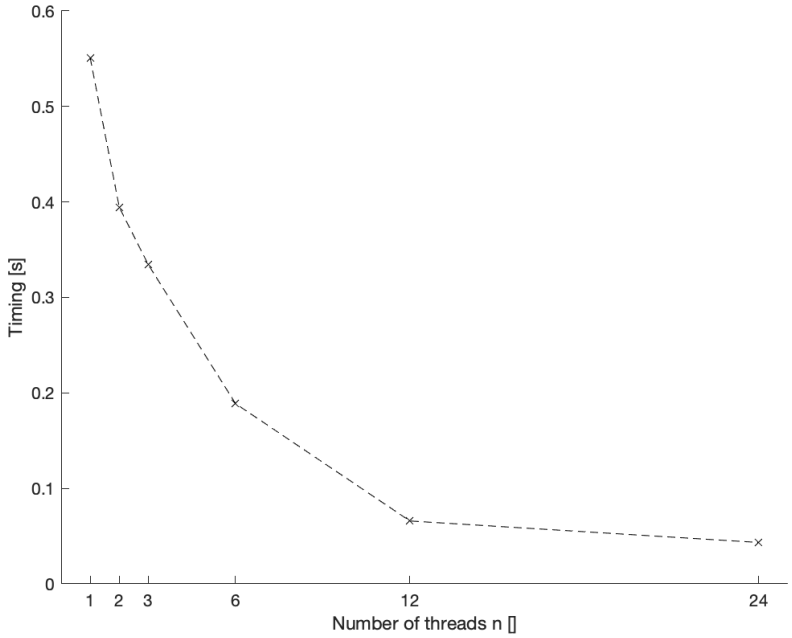


Figure 10: Execution tiems for Loop 2, scheduled using the static partitioned affinity scheduling algorithm. Again, we obtain better performance as we increase the number of threads, although the performance does not scale one-to-one with the number of threads.

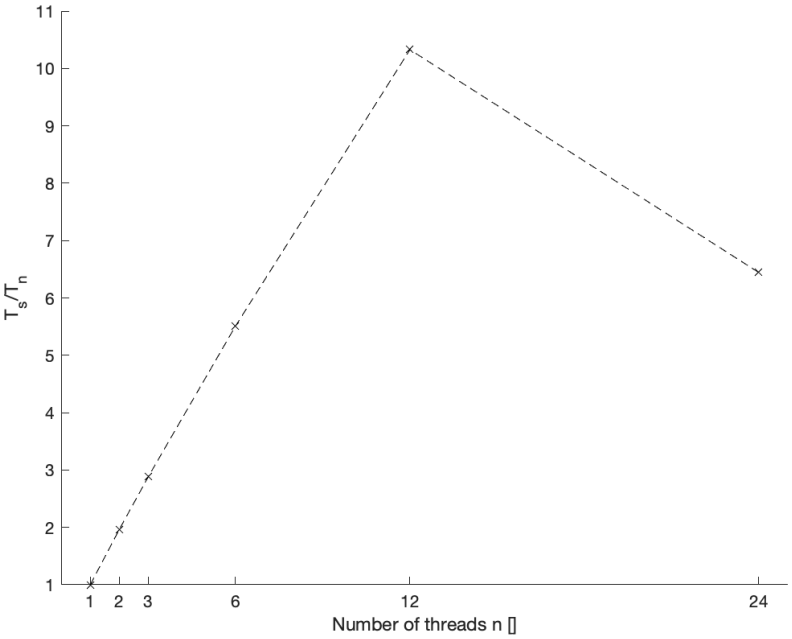


Figure 11: The ratio of execution time using n threads to serial execution time for Loop 1; ...

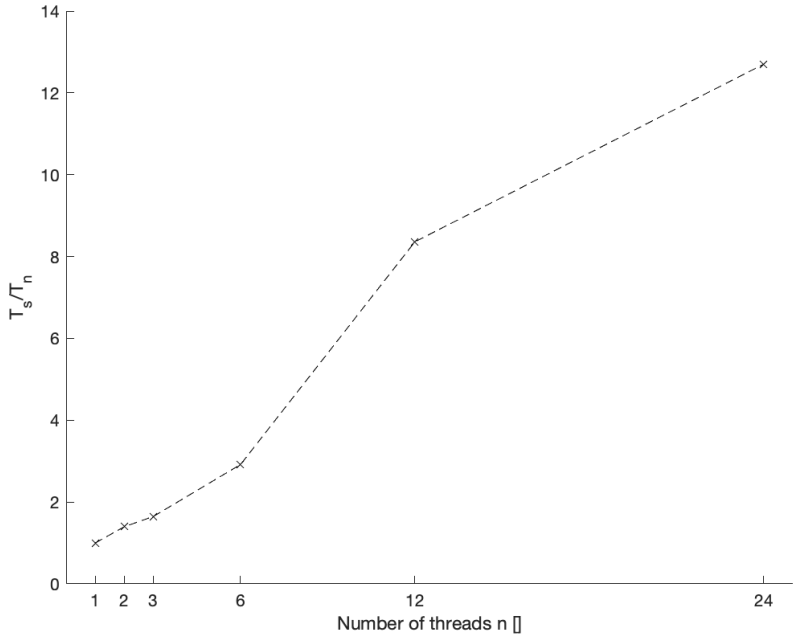


Figure 12: The ratio of execution time using n threads to serial execution time for Loop 2; ...

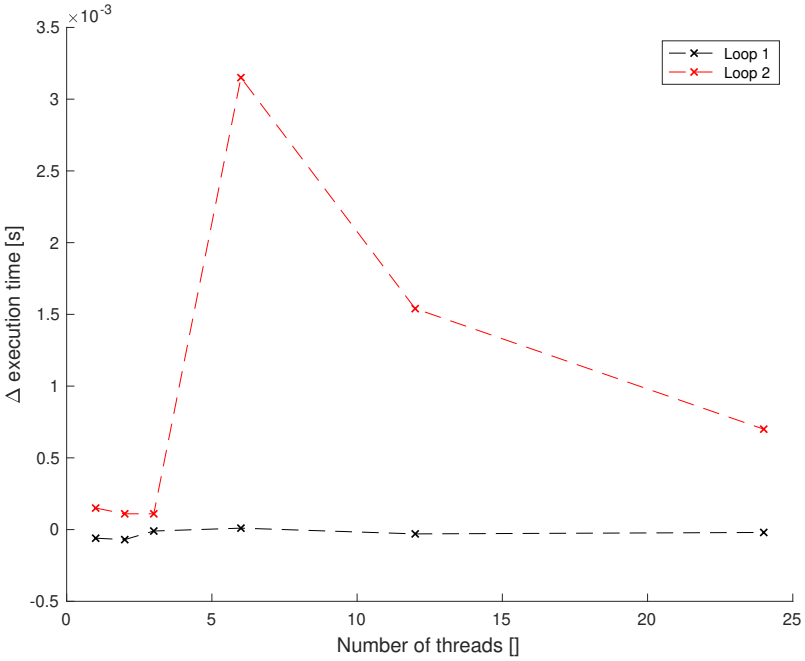


Figure 13: The change in execution time when using critical regions, as compared to locks; locks are found to be faster by approximately 10%.