Using OpenMP to Implement the *Affinity* Work Sharing Schedule to Parallelise *For* Loops

Threaded Programming: Coursework Part 2

Exam No. B138101

November 29, 2018

1 Introduction

The aim of this project is to implement an iteration scheduling algorithm for work sharing between threads in OpenMP. Instead of using tranditional OpenMP directives to implement schedules such as *Static* or *Dynamic* to parallelise *for* loops, an alternative algorithm is manually programmed via parallel regions and other OpenMP mechanisms to implement its scheduling policies. The name of the algorithm in question is the *Affinity* schedule, as originally described by Subramaniam and Eager (1994). A brief description of *Affinity* scheduling and the details of its implementation are discussed in sections 3 and 4 respectively.

To test the implementation, the running times for 1000 repetitions of two parallelised loops are recorded in the programme. Also, the programme is run using 1, 2, 4, 6, 8, 12 and 16 threads, and is executed repeatedly for 5 times for each configuration in order to gain a reliable measure of the underlying running times. The averages of these times for each thread count configuration were used for data collection. In order to evaluate the algorithm's performance, the variations in execution time with thread count are compared with that of the best work sharing schedules found in Part 1 for corresponding loops. The results of these experiments are presented and discussed in sections 5 and 6 respectively.

In order to verify the correctness of the *Affinity* implementation, the final numerical outputs of the computations from the parallelised loops for all configurations tested are compared with that of a serial version of the programme. Therefore, it can be confidently said that the implementation is correct.

Each execution of the programme was submitted to a compute node of the Cirrus High Performance Computing Cluster, and run exclusively of any other submitted jobs. Each compute node contains two 2.1 GHz, 18-core Intel Xeon processors, 256 GB of memory and access to a shared file system.

The C code was compiled on the Cirrus head node using the Intel compiler and the flags specified were such that a high level of sequential optimisation was ensured.

2 Brief remarks on loops to be parallelised

The provided C code in which the schedule is implemented contains two *for* loops to be parallelised. As previously mentioned, each is executed 1000 times over the course of the programme. The nature of the loops was described in detail in Part 1, and so such material will not be repeated. The most salient points about the loops, however, are thus; each contained a fixed number of iterations (729), the required compute power over the iterations was steadily decreasing in the case of the first loop and irregular (albeit heavily clustered towards the beginning) for the second, the inner most statements of the loops accessed elements from shared arrays and computed them through basic mathematical functions (namely the cosine and logarithm), adding the resulting outputs to elements of another shared array. Finally, such statements within the loops were homogeneous for all iterations, in that no there was no logic which changed the statements executed on a per-iteration basis.

3 An outline of *Affinity* scheduling

The main distinction underpinning the *Affinity* schedule in comparison with other similar mainstream schedules, such as *Guided*, is that has a propensity for threads to "steal" chunks of iterations by means of reassigning units of work from the most loaded thread to others. Based on the work by Subramaniam and Eager (1994), the scheduling algorithm can be described as follows:

- For each thread, a work sharing *for* loop consisting of n iterations, in a programme with a capacity to execute across p threads, is assigned a local set of contiguous iterations of size n/p (or rounded upwards to the nearest integer).
- While a thread has not finished its own local set, it serially executes iterations in chunks of size R/p (or rounded upwards to the nearest integer), where R is the number of remaining iterations in the local set.
- Once a thread has terminated the execution of its local set, it determines the thread with the largest remaining workload, and executes chunks, whose size is a fraction $1/p$, of this thread's remaining iterations. This "iteration stealing" is repeated until there are no remaining iterations of the loop left to complete.

4 Implementation details

Given the algorithm's reassigning of some iterations between threads, it is necessary to develop an implementation with particular attention to three aspects. Firstly, data structures such that the progress through a thread's local set is not only recorded, but shared. This is in order to maintain that iterations are not accidentally executed multiple times when any thread begins work on iterations outside of its local set. Secondly, the parallelised execution of chunks of iterations, be they either part of a thread's local set or part of the next chunk of the most overloaded thread, all the while on a per-thread basis. Thirdly, there must be no race conditions created as threads need to be synchronised in deciding which iterations are going to be/have been executed, such that no single iteration is accidentally executed more than once. It is also worth noting the importance of declaring private and shared variables in order to maintain variable integrity over the work sharing process.

After the discussion on the implementation in subsection 4.1, some evaluating remarks discussing drawbacks and possible improvements to the approach are included in subsection 4.2.

4.1 Main scope of implementation

Start with the sharing of data on a thread's progress through its local set. Before any computation takes place, there must exist a structure to hold such information. This implementation uses a *struct* data type, called *Thread* to do so. The collective variables within the *Thread struct* contain both the upper and lower boundaries of one thread's local set of iterations, and the number thereof which have been completed (plus those which have been assigned to a thread and are pending completion). For the running of each parallelised loop, an array, of length equal to the number of threads, in which each element is a *Thread struct* corresponding to one of the threads, is shared within the parallel region. Since the array is shared, and from the information encapsulated within each *Thread struct*, any other thread may calculate, straightforwardly, the next chunks upper and lower bounds of iterations based on the mathematical expressions laid out in section 3.

The main advantage of using a *struct* data type is that it allows a collection of variables under one name, thereby making it easy to access its underlying variables. An alternative approach would be to hold each variable in separate arrays, whereby each array contains only on type of information (say the number of iterations completed), and the index of the information would correspond to its thread number. However the accessing the information by indexing in this way would make for slightly more fiddly programming than the simple, all-encompassing *struct* implementation.

Secondly, take the parallelised execution of selected iterations. In order to select manually which ones are to be executed, the main scope of the programme which controls this implements parallel regions. This allows control on a per-thread level since the region spawns multiple threads to run simultaneously. Upon entering the region (of which the aforementioned array of *Thread structs* is passed as a shared variable) each thread determines the upper and lowers bounds of the iteration numbers of its local set based on the expressions in section 3. These variables are then used to initialise those of each thread's respective *Thread struct* element in the shared array. Next, the thread calculates the iterations of the first chunk to execute based on the information in the shared array, and marks these as completed/pending completion (the synchronisation aspects of this are later discussed). The thread then enters a *do-while* loop, in which the iterations in this chunk are executed, and then the next chunk of its iterations is calculated and, as before, marked as complete.

The motivation for using a *do-while* loop is that it checks whether or not to go back to the beginning based on a check at the end of the loop. This means that, since the next chunk can be computed at the end of the loop, and if it is the case that there are no remaining iterations in the local set, the *do-while* loop terminates here based on this condition (else it loops back to execute the next chunk in the local set). Alternatively, the use of a *for* loop would be inappropriate here since it requires knowing how many cycles will take place before starting (which is not possible due to the "iteration stealing" of the *Affinity* schedule). A *while* loop, even though does not have such a limitation, only checks the termination condition at the beginning of the loop, and thus extra logic would be necessary to break out upon calculating whether there are more iterations in its local set.

Once a thread has finished its local set, a very similar flow of execution brings the thread to execute iterations from outside of it. The main difference here, however, is that the bounds of the chunks of these iterations are calculated based on the thread with the largest number

of remaining iterations, and are not part of the this thread's own local set. Such information is computed via iterating through the *Thread structs* in the shared array, finding the thread with the largest number of remaining iterations, calculating the chunk boundaries and marking these iterations as complete. Once there are no remaining iterations left to complete, the final *do-while* is broken, and the parallel region is ended.

Thirdly, take the synchronisation of shared information between the threads. This is the most important aspect of implementation due to the schedule's nature of dynamically reassigning iterations between threads during execution. As a result of this, a poor implementation, which perhaps includes race conditions between threads, would render the final results of such a programme redundant, since the output cannot be reliably replicated.

In order to combat the issue, a concurrency mechanism is required for the shared array of *Thread structs* to keep it in a consistent state throughout execution. This requires restricting access by the underlying threads to it in order to avoid concurrency issues such as the Lost Update Problem. This is implemented via making use of OpenMP's *critical* directive, which restricts threads from entering the section if another thread is already inside, effectively serialising threads' execution of statements within. Every read or write request to this shared array is encompassed within a *critical*. It is therefore ensured that, for every read or write of the array, each thread sees the information within in a consistent and current state, and no operation on the array by a thread will cause another thread's data to become invalidated. There are, however, drawbacks to this approach, as later explored, in that threads may incur frequent (albeit brief) overheads of idle time when waiting to access the array.

An alternative approach to implement such concurrency control would be to use OpenMP's *locks* to encompass every read/write access to the shared array. While this would not solve the issue of frequent idle times, the effect of using such mechanisms would be more or less identical to that of using the *critical*, since both directives control threads in similar ways. While using *locks* could have easily been implemented, *critical* directives are generally more syntactically pleasant and convenient to use, since they neither have to be initialised, nor manually set/unset.

It is also important to briefly mention another mechanism used to ensure consistency of the shared array. Upon calculation of the per-thread variables (local set bounds and number of iterations completed), these are initialised into the shared array. (Parallelising this is acceptable since each thread always writes to a different element in the array). However after this point, a *barrier* directive is used to ensure that all threads have done this before the execution of any iterations begins. While is unlikely that, if the *barrier* were not implemented here, a thread finishes its local set and tries to access the uninitialised variables corresponding to another thread, this mechanism is included for the sake of completeness.

4.2 Remarks on the limitations and possible improvements of this implementation

The implementation described here is, by no means, perfect. There are various limitations which contribute to slow downs in execution time, thus reducing overall performance. For example, consider the *critical* directives used whenever the shared array of *Thread structs* is accessed during a threads execution of its local set. In many cases, it will be such that threads are not intending to access the same index in the array, and therefore such assesses could execute concurrently with no race conditions ensuing. This would reduce execution time. However, given that only a single thread may enter a *critical* at a time, there are overheads induced and threads are left unnecessarily idling.

To improve on this, the programme could include read/write locks which would be implemented on a specific variables within the shared array, rather than excluding access to the entire

array at once. These would prohibit access to an individual variable when a lock is place, and only when unlocked would access be granted. Such an approach would provide greater flexibility for the programme, since the variable detailing the number of iterations completed for different threads could be accessed simultaneously. However, the locking mechanisms supported by OpenMP do not work in this manner, and read/write locks for variables are not supported in this API.

It is important to note however, given that the *critical* sections in the current implementation are not computationally expensive, they are completed quickly and thus the idle time of other threads is minimal. Finally, given the stochasticity of the execution of the programme within the parallel region, it is improbable (yet still possible) that any two threads attempt to enter a *critical* section simultaneously, since much of their running time is spent executing iterations instead.

5 Presentation of results

5.1 Loop 1

As determined from Part 1, the best work sharing schedule to parallelise Loop 1 was *Dynamic 32*. The experimental results of this configuration, along the *Affinity* schedule, are shown for this loop in the visualisation below in Figure 1.

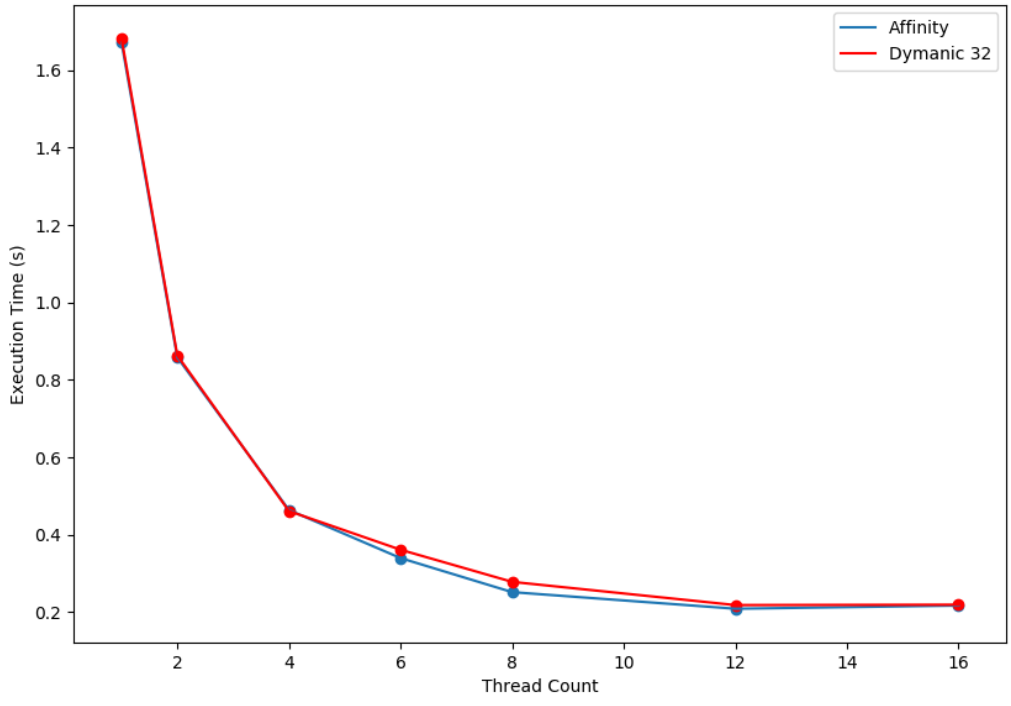

Figure 1: Variation in execution time (s) with thread count for the schedules *Affinity* and *Dynamic 32* in parallelising Loop 1

As shown in Figure 1, both work sharing schedules exhibit very similar performance. For both schedules, the execution time decreases with increasing thread count in a manner, similar to

an inverse correlation. The fastest execution time for both schedules (with 12 threads deployed) brings approximately an 7.7x speed up from serial execution. However, there is no significant decrease in execution time on a per-thread count basis between the two schedules tested, and thus there is no major benefit to using either schedule to parallelise this loop.

5.2 Loop 2

The best schedule to parallelise the second loop, as determined from Part 1, was *Dynamic 16*. The results of from the experiments for Loop 2 of this schedule and that of the *Affinity* are shown in Figure 2.

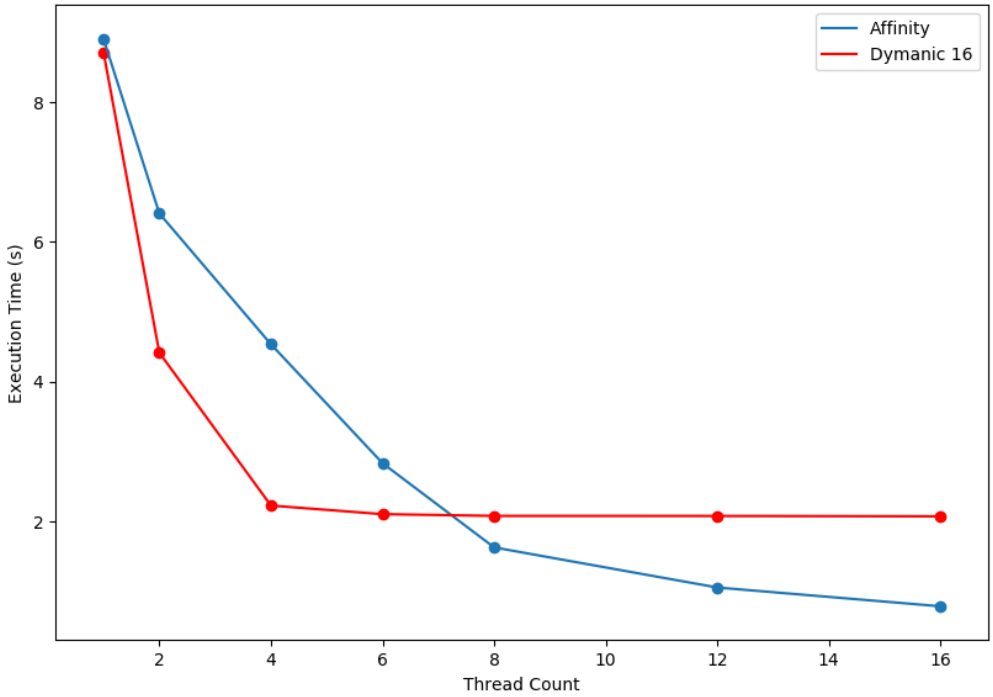

Figure 2: Variation in execution time (s) with thread count for the schedules *Affinity* and *Dynamic 16* in parallelising Loop 2

As can be seen from Figure 2, there is more distinction in the behaviours between the two schedules than was the case with Loop 1. The *Dynamic 16* configuration initially outperforms that of *Affinity* up to 6 threads deployed, after which point *Affinity* starts to outperform. The overall trends show that *Dynamic 16* decreases in execution time up to a deployment with 6 threads, after which the performance plateaus. The fastest speed up for this schedule is approximately 4.2x. However, no such plateau is shown for the *Affinity* schedule, as its decreases continuously with increasing thread count, resembling an inverse correlation. The fastest execution time of the *Affinity* schedule brings an 11.3x speed up.

6 Discussion of results

Starting with the second loop, in order to explain the results shown, consider the effect on the size of a thread's local set of iterations with threads deployed in execution in the *Affinity* schedule. As the number of threads increases, according the algorithm as described in section 3, the size of the local set decreases, meaning that any particular thread will, on average, spend less time executing it. Furthermore, the average size of a chunk of iterations executed within a local set decreases still, giving yet more granularity to the programme. As a result of this, each thread has more capacity to aid others by reducing their workloads, and thus groups of smaller iterations will be dynamically reassigned between threads more often during execution. The overall effect of this is that it makes the algorithm increasingly similar to a *Dynamic* schedule with shortening chunksizes.

We can therefore explain the improvement in performance of the *Affinity* schedule in comparison with the *Dynamic 16* by looking around the interception point between 6-8 threads. The average chunksize of iterations executed in the *Affinity* configuration with 6 threads is approximately 5.8 (there is, however, a large variation due to the decay in chunksizes with progression through the local set). This number is smaller than the chunksize of 16 used in the *Dynamic* schedule, meaning that it could be expected to see an outperformance by *Affinity* since the iterations are being dynamically reassigned between threads more often. A possible explanation for the absence thereof, may be due to the overheads in synchronisation in the *Affinity* algorithm. As average chunksizes decrease and iterations are reassigned more frequently, the threads thus enter the *critical* regions more often, increasing overheads and idle times. This would, to some extent, have subdued the performance increase expected from the smaller chunksizes.

However, looking at 8 threads, where it can be observed that the *Affinity* schedule start to outperform *Dynamic 16*, the average chunksize drops to approximately 3.8. Beyond this point, it this follows that the gains from smaller chunksizes are strong enough to offset the overheads when encountering the *criticals*.

The main rational behind the improvement with smaller chunksizes, is that such an effect tends to make the *Affinity* schedule more dynamical with increasing granularity. And it is generally true that *Dynamic* schedules, particularly with short chunksizes, tend to perform well when tasked with managing iterations in loops with a widely varying workload, which is certainly the case for the second loop. This explains the eventual outperformance compared with *Dynamic 16*, since there is a fixed chunksize throughout all thread counts in this configuration.

Secondly, consider the results shown for Loop 1. It can be seen that there is no increase in speedup for the *Affinity* configuration in comparison with *Dynamic 16*. While all the same effects from increasing thread count in the *Affinity* schedule (decreasing local set size, decreasing average chunksize, more reassignments) will still be present, the lack of improvement is likely a result of the nature of the workload distribution of iterations in Loop 1. As mentioned in section 2, there is a far more regular distribution of compute power required across the loop than is the case for Loop 2. Since *Dynamic* schedules tend to perform best on loops without such regularity, the regularity here is a plausible an explanation for lack of relative performance increase. It can also be noted that the overheads brought on from synchronisation outweigh any such expected relative performance increases. In order to confirm this statement however, a more efficient implementation with fewer overheads, such as that described in subsection 4.2 involving the use of read/write locks on individual variables, must be tested.

7 Concluding remarks

The project sought to implement manually the *Affinity* work sharing schedule in OpenMP. The programme was the tested on two different *for* loops with a varying number of threads, and the execution times compared with the best schedules determined from Part 1. From the results seen, it can be concluded that the *Affinity* schedule performs strongly, generally matching (if not besting) the results shown from the other scheduling configurations. Based on the experiments, it can thus be concluded that it performs best with many threads deployed during execution and on loops with a widely varying workload across their iterations, as a result of the decreased underlying chunksizes, and thus more reassignments of chunks between threads.

It is, however, important to recall that there are various limitations to this implementation, which likely subdued relative performance increases. However such limitations could not easily be overcome with the capacity provided in OpenMP. Further work could take into account some suggestions made in subsection 4.2 to mitigate such limitations, and perhaps use a different multiprocessing API, such as Java's Concurrency library to do so.

References

Subramaniam, Srikant and Derek L. Eager (1994). *Affinity Scheduling of Unbalanced Workloads*. Supercomputing 1994. Washington, D.C.: IEEE Computer Society Press, pp. 214–226. ISBN: 0-8186-6605-6. URL: `http://dl.acm.org/citation.cfm?id=602770.602810`.