|epcc|

Threaded Programming: Affinity Scheduling

B083194

December 3, 2015

Contents

1 Introduction

Open MP scheduling is a powerful tool for speeding up a programs execution time through parallelisation. Dividing up the work between processors is usually done through built-in scheduling options. There are many scheduling options built into Open MP directives, however, often OpenMP applications fail to utilise the whole system effectively [1]. It is possible to design scheduling options from scratch, and one such schedule option is called *Affinity scheduling*. This is constructed "by hand" using a parallel region.

The aim of this project is to construct affinity scheduling 'by hand' using a parallel region, then measure it's performance. Affinity scheduling will be applied to an already parallelised program with two loops, which have verification steps after each loop. The code will be run on 1, 2, 4, 6, 8, 12 and 16 threads. The results from the performance tests will be compared against the best built-in Open MP schedule from the previous project. The `pgcc` compiler will be used throughout, along with the `O3` flag on the command line, to optimise the performance.

2 Affinity scheduling

Affinity scheduling usually achieves the highest speedup of all the commonly used scheduling options [3], however it is not trivial to construct. Affinity scheduling is a non-standard feature of Open MP, and this section will take a step by step approach to explain its design, and how it was constructed within `loops2.c`. Affinity scheduling can be described as follows:

- Each thread is initially assigned a (contiguous) local set of iterations.
- For a loop with n iterations, and p threads, each thread's local set is initialised with n/p iterations. (If p does not divide n exactly, choose a suitable distribution of the extra iterations.)
- Every thread executes chunks of iterations whose size is a fraction $1/p$ of the remaining iterations its local set, until there are no more iterations left in its local set.
- If a thread has finished the iterations in its local set, it determines the thread which has most remaining iterations (the most loaded thread) and executes a chunk of iterations whose size is a fraction $1/p$ of the remaining iterations in the most loaded thread's local set.
- Threads which have finished the iterations in their own local set repeat the previous step, until there are no more iterations remaining in any thread's local set. [2]

2.1 The Code

The code has two loops, each run for 100 reps, to be parallelised without using work-sharing loop directives. However, because the iterations of both loops are passed to the same function, only this function requires parallelisation. The first loop (Loop 1) contains a double nested for loop which updates the elements of an upper triangular matrix, so earlier iterations will be most expensive. The second loop (Loop 2) contains a triple nested for loop which updates the elements of a vector, $c[i]$. This too has expensive early iterations, because the first 30 elements of an updating variable,jmax[i], are all 729, whereas later elements are mostly one. This is displayed in Figure 1.

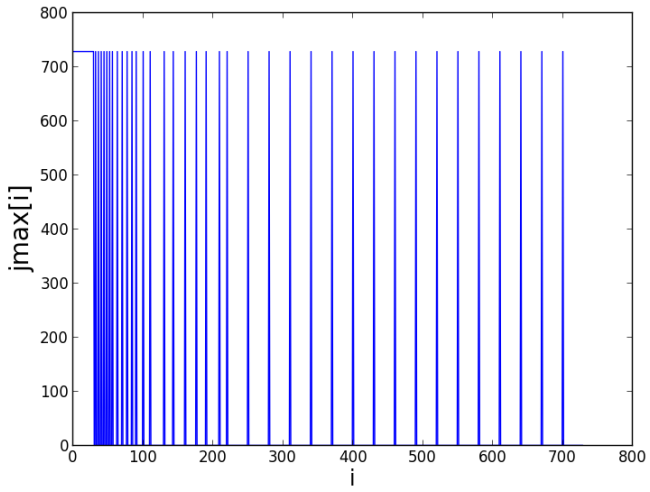


Figure 1: Spikes represent where $jmax[i] = 729$, earlier iterations have more spikes, and hence take longer.

Affinity scheduling will be applied to the function `runloop`, which has already been parallelised "by hand" to follow a static schedule without a specified chunk size. This is a good basis on which to start implementation of affinity scheduling, because affinity, like static, requires each thread to get an approximately equal workload.

2.2 Constructing the Algorithm (i)

Constructing an affinity schedule has two parts. Firstly, each thread does the work it has been assigned, with earlier chunksizes more loaded. Secondly, idle threads must help out the "most loaded" thread by taking some of its workload (work steal). The first step is to define some new variables: the number of remaining iterations for each thread and the number of iterations to be assigned to the next chunk. Then a formula for working out chuck size must be designed before creating a while loop which continues to iterate until there is no more work for each thread. Inside this while loop, the parameters are redefined for the next iteration.

The code thus far divides up the work, assigning the largest chunk size to be completed first, with the chunk sizes getting progressively smaller. In this respect, it is similar to guided scheduling. The next stage of the process is more complicated, with additional data structures required.

2.3 Constructing the Algorithm (ii)

To incorporate "stealing" work from other threads, a structure (struct) is introduced. The struct has two members: `high` and `remaining`. The struct is initiated within the function before it is assigned a memory location, using the `malloc` function. This is done using a single directive, so that only one thread allocates this memory. The struct is contained within the shared clause of the parallel region, so that all threads can can read from it and write to it. Structs are used because they can hold multiple pieces of information in one memory location. This makes programs cleaner and increases readability and reproducibility. The struct and its members are created as follows:

```
struct block
{
  int high;
  int remaining;
};
```

The individual members of the struct are then assigned values within a critical region, this ensures synchronisation. For this reason, members will be updated throughout the rest of the parallel region using critical directives. Without these critical regions, a thread may steal the same iterations that have already been completed, and the results produced would be incorrect.

Similarly to Section 2.2, a while loop is used for each thread to carry out the work initially assigned to it. However, once this work has been completed, each thread does not lie idle. A `do while` loop is introduced, containing a critical region, which finds the location of the thread with the highest number of remaining iterations, and the number of iterations at this location. As long as work is remaining, the threads which have finished their own work steal the work from the most loaded thread. The remaining work is then updated. The critical region ends and the work, and stolen work, is completed. The memory used for the struct is then freed at the end, to ensure no memory leakages.

The single and critical directives ensure synchronisation, which in turn ensures protection from the race condition. If a thread finishes it's allocated work, it can only search for and find work through critical regions. The updates of the remaining work are also updated using a critical region. These directives create an environment completely free from the perils associated with the race condition, whilst also maintaining high performance.

3 Synchronisation tests

Even if we are certain that the race condition has not affected our program, we must conduct tests to prove this. The first test is simple, testing that we get the correct sum for any number of cores. If a race condition is present, it can force a program to do the same iterations twice, and produce incorrect results. Table 1 shows the results obtained from the validation tests already in the code.

Threads (n)	Loop 1 check	Loop 2 check
Serial result	-34302.147477	-2524264.460320
1	-34302.147477	-2524264.460320
2	-34302.147477	-2524264.460320
4	-34302.147477	-2524264.460320
6	-34302.147477	-2524264.460320
8	-34302.147477	-2524264.460320
12	-34302.147477	-2524264.460320
16	-34302.147477	-2524264.460320

Table 1: Expected results and achieved results test results

All the results match that found from running the code in serial, as expected. However, this test does not guarantee synchronisation. Another test is to print the iterations a thread performs, when a thread is stealing, which iterations it carries out, and how many are remaining. Then you can manually count up to make sure iterations are synchronised. The results from 2 threads, with one 1 rep, are shown below.

```
Thread 1 iterating from 365 to 547 with 182 remaining
Thread 0 iterating from 0 to 183 with 182 remaining
Thread 1 iterating from 547 to 638 with 91 remaining
Thread 1 iterating from 638 to 684 with 45 remaining
Thread 1 iterating from 684 to 707 with 22 remaining
Thread 1 iterating from 707 to 718 with 11 remaining
Thread 1 iterating from 718 to 724 with 5 remaining
Thread 1 iterating from 724 to 727 with 2 remaining
Thread 1 iterating from 727 to 728 with 1 remaining
Thread 1 iterating from 728 to 729 with 0 remaining
Thread 1 stealing from thread 0 iterating 183 to 274 with 91
remaining
Thread 1 stealing from thread 0 iterating 274 to 320 with 45
remaining
Thread 1 stealing from thread 0 iterating 320 to 343 with 22
remaining
Thread 1 stealing from thread 0 iterating 343 to 354 with 11
remaining
```

```
Thread 1 stealing from thread 0 iterating 354 to 360 with 5
remaining
Thread 1 stealing from thread 0 iterating 360 to 363 with 2
remaining
Thread 1 stealing from thread 0 iterating 363 to 364 with 1
remaining
Thread 1 stealing from thread 0 iterating 364 to 365 with 0
remaining
```

Similar results were obtained for all numbers of threads, and hence synchronisation is intact. This output depicts how expensive the early iterations are. Thread 0 only does half of its own iterations, and thread 1 steals the rest.

4 Results

All the results were produced on Morar, with 64 cores reserved. The results for all graphs have been run five times, with the average value taken as the final result. The following graphs were plotted using Python.

4.1 Execution Time and Speedup

Performance, in terms of execution time and speedup, are shown below in Figure 2 and 3.

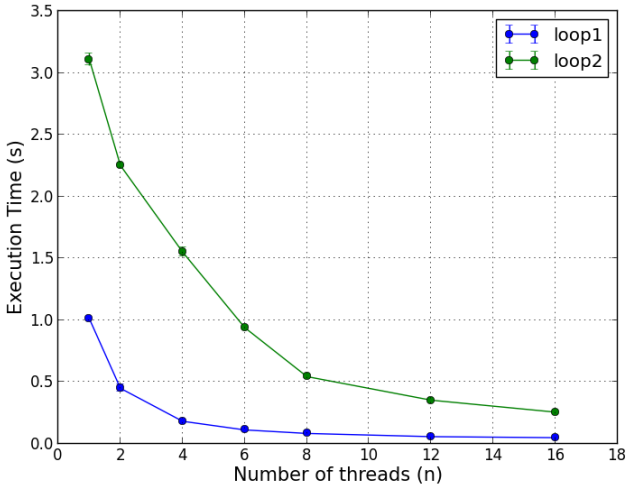


Figure 2: Execution time in seconds for both Loop 1 and Loop 2 on affinity scheduling. This was plotted for n = 1, 3, 4, 6, 8, 12 and 16 threads. Error bars have been plotted around each point.

Loop 2 is a triple nested `for` loop, and so we expect it to take more time than Loop 1. Both loops follow the same trend, where more increasing threads reduces execution time. Earlier iterations of both loops are the most expensive, however because affinity scheduling steals work, this effect is reduced. Next we will look at the speedup, shown in Figure 3.

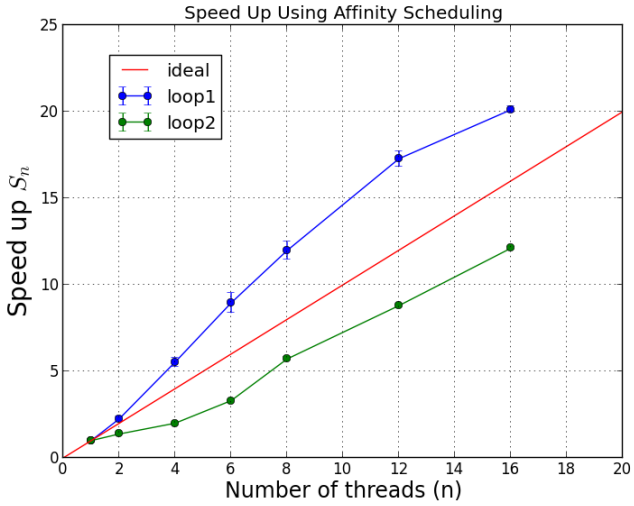


Figure 3: Speedup for Loop 1 and Loop 2 with affinity scheduling. Error bars have been plotted around each point. This was measured for n = 1, 2, 4, 6, 8, 12, 16 threads. The ideal speedup is plotted in red. Note super-linear speedup for Loop 1.

In threaded programming, speed up is the time of thread 1 (T_1) divided by the time of the nth thread (T_n), where n = 1, 2, 4, 6, 8, 12, 16.

$$Speedup_n = T_1/T_n \tag{1}$$

Ideal speedup is when speedup=X for X processors.

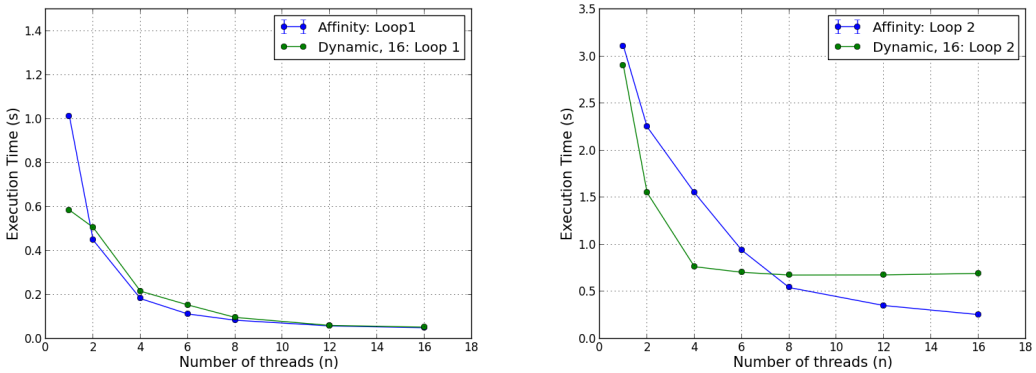
Loop 1 achieves super-linear speedup, which means the speed up was better than ideal for all numbers of threads. This disagrees with Amdahls law and intuition. How can doubling the number of processors more than double the speedup, when significant portions of the code are written in serial? An explanation can be found by looking at computer hierarchies and the cache effect [4]. As the numbers of processors increase, so does the accumulated capacity of cache memory. More of the working set can now fit into cache, and for small working sets, they can completely fit in. The work set for Loop 1 must be small enough to fit into cache memory, and hence super-linear speedup is achieved.

Loop 2 achieves good speedup, which doesn't show any signs of plateauing. It falls just below ideal speed up. This is the desired result for most computations. Often speedup plateaus when early iterations are the most expensive, because extra processors can saturate the performance. However, even though affinity scheduling has its most expensive iterations first, idle threads steal work from the most loaded thread. With

smaller chunk sizes in later iterations, the parallel finish time is likely to be very close for each thread.

4.2 Comparison With Dynamic, 16

In the previous project, we found Dynamic, 16 to be the fastest built-in scheduling option for both Loop 1, and Loop 2. In this subsection we are going to compare the speed up and the execution time of both loops for both scheduling options. The execution times for Loop 1 and Loop 2 are shown in Figure 4.



(a) Loop 1 Execution Times

(b) Loop 2 Execution Times

Figure 4: The execution time of Loop 1 and Loop 2 for Dynamic, 16, and affinity scheduling. This was measured for n = 1, 2, 4, 6, 8, 12, 16 threads.

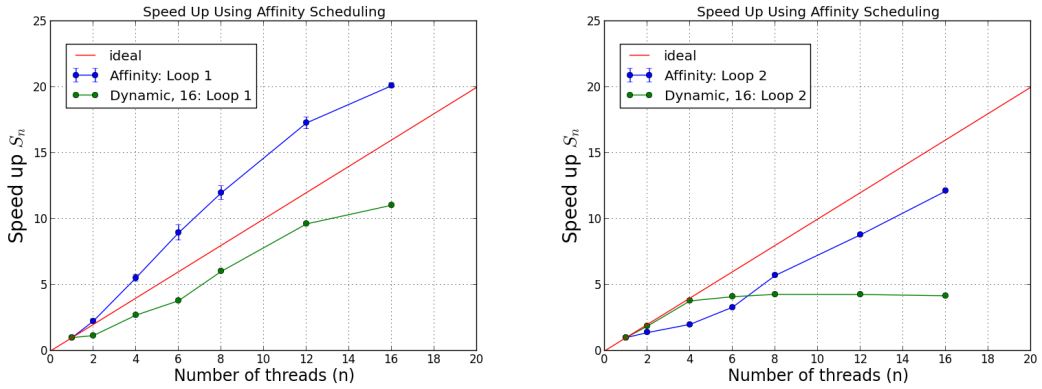
Both scheduling options have very similar execution times for Loop 1. Although dynamic, 16 is fastest to begin with, affinity scheduling times sharply decreases, and have lower execution times for all numbers of processors greater than two, however the difference is marginal.

For Loop 2 both execution times are around the same value for one processor. This time, it is dynamic that sharply decreases, before leveling out for greater than 6 processors. Execution times for affinity scheduling steadily decreases as the number of processors increase. Although initially slower than dynamic, for more than 8 processors, affinity scheduling is faster. In fact, the execution times for dynamic increase for more than 8 processors.

Affinity's lower execution times for higher numbers of processors can be explained by parallel end times. Dynamic scheduling is most effective for wide varying loads, and as early iterations are most expensive, we have exactly that. However, because dynamic has a consistent chunk size of 16, there are sizeable ranges of end times among threads. With affinity, earlier chunk sizes of are largest, this means later chunk sizes are small. As long as the first iterations are not too expensive, affinity scheduling will have a small range of end times among threads, hence faster times. Affinity also benifits

from varying chunk sizes. As the numbers of processors increase, the chance of earlier iterations being assigned to one or two cores is minimised. This is due to the chunk size being inversely proportional to the number of processors, as defined in the equation for chunk size.

The speedup for Loop 1 and Loop 2 is displayed in Figure 5. For loop 1, we already saw that the speed up was super-linear with affinity scheduling. The speed up for dynamic is also very good, it falls just under the ideal line.



(a) Loop 1 Speedup

(b) Loop 2 Speedup

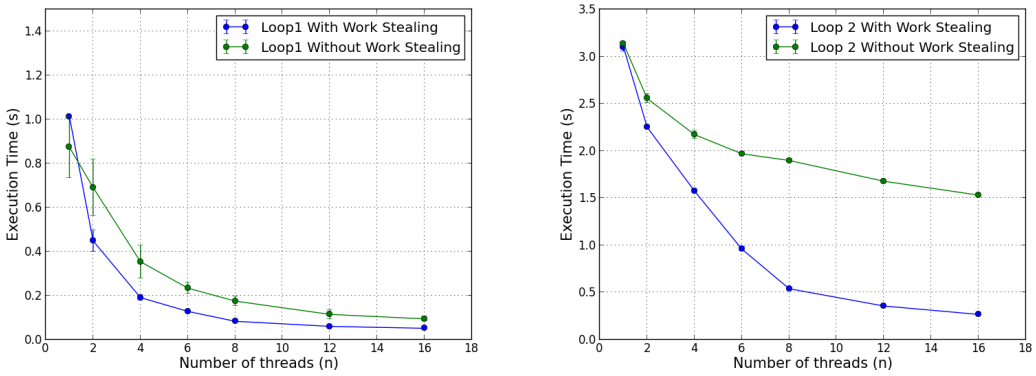
Figure 5: The speedup of Loop 1 and Loop 2 for Dynamic, 16, and affinity scheduling. This was measured for n = 1, 2, 4, 6, 8, 12, 16 threads.

The speed up for dynamic on Loop 2 is close to ideal. However, after 4 cores the speedup plateaus. This result is expected for most programs. A significant portion of the code is in serial, and so Amdahls comes into play. Dynamic, 16 gives the bulk of the work to the first two threads (the first 30 iterations take the longest). Adding more threads just saturates the performance and speedup plateaus. Affinity scheduling continues to speed up because the chunk size changes as more threads are added, so the bulk of the workload is never assigned to a small number of processors. This combined with work stealing, and the low range of end times make affinity scheduling better for higher numbers of threads.

5 Work Stealing

Work stealing is perhaps the most powerful tool in affinity scheduling, but exactly how powerful? The execution times for loop 1 and loop 2 are shown in Figure 6. The first thing we notice is the higher variability rate for affinity without work stealing. This is explained by the larger range of end times for the threads. Instead of idle threads stealing the work of the most loaded thread, the loaded thread must do all the work it has been assigned. This is particularly perilous when you have a wide varying load, like ours. This results in the loops with work stealing being faster for multiple threads

(where stealing can actually take place). The ratio of execution time for affinity without work stealing:affinity with work stealing (Stealing ratio) is shown in Table 2.



(a) Loop 1 Execution Time

(b) Loop 2 Execution Time

Figure 6: The execution time for Loop 1 and Loop 2 for Dynamic, 16, and affinity scheduling. This was measured for n = 1, 2, 4, 6, 8, 12, 16 threads, and includes error bars.

The ratio is close to 1 to start with, but as the number of threads increase, the effect of work stealing becomes more pronounced. For 16 threads this increases to 5.716(3d.p.). This is a huge difference, and really displays the difference work stealing can make on work sharing.

Threads (n)	Stealing Ratio
1	1.004407899
2	1.135161466
4	1.379212293
6	2.05150027
8	3.516115996
12	4.70403364
16	5.715712959

Table 2: Threads and stealing ratio. All values were run 5 times, with average values taken as final.

6 Conclusions

Built in work sharing directives are useful for optimising performance. However, one must first consider the nature of their code before applying a scheduling option. Iterations with wide varying loads can be handled well by dynamic schedules, but dynamic schedules cannot vary their specified chunk size during execution. Affinity scheduling

is not trivial to construct, but it's ability to change chunk size according to how many processors are used and how many iterations remain, serves it well. Furthermore, work stealing speeds up execution times, producing phenomenal speedup results.

Affinity scheduling yields quicker times than dynamic scheduling for high numbers of processors. For lower numbers of threads dynamic, 16 was as good for Loop 1, and better for Loop 2. This is because the first 30 iterations are the most expensive, leaving the first two threads to be assigned the highest workload, regardless of the number of processors.

Super-linear speedup was achieved for Loop 1 with affinity scheduling, which can be attributed to accumulated cache memory. This only works for Loop 1 because it is small enough to fit in, whereas Loop 2 is too big to completely fit inside the accumulated cache. The speedup for Loop 2 was more consistent with affinity scheduling than dynamic, which plateaued after 8 threads. Affinity scheduling performs better than dynamic in this project for high numbers of threads, and in general is better for programs with expensive early iterations.

References

[1] Thoman, Peter, Hans Moritsch, and Thomas Fahringer. "Topology-Aware OpenMP Process Scheduling." Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More. Springer Berlin Heidelberg, 2010. 96-108.

[2] MSc in High Performance Computing Coursework for Threaded Programming Part 2

[3] AyguadÃl', Eduard, et al. "Is the schedule clause really necessary in OpenMP?." OpenMP Shared Memory Parallel Programming. Springer Berlin Heidelberg, 2003. 147-159.

[4] Speedup https://en.wikipedia.org/wiki/Speedup