Threaded Programming Part 2

B087928

December 4, 2015

1 Introduction

When we parallelise loops using open multi-processing we have many options on how we distribute work amongst the available threads. Giving threads an equal amount of itereations does not guarantee an equal load balance. A good choice of loop schedule is problem dependent and is essential to achieve optimum speedup from parallelisation. Some forms of loop scheduling are naive and will cause several threads to have to wait a long time in certain problems while others incur a higher overhead.

Knowledge of the problem to be solved can suggest the appropriate loop scheduling to be used, however it is often impossible to predict without some experimentation. In this report we examine two loops (loop 1 and loop 2). In each case we examine the speedup on up to 16 cores using a manually implemented affinity loop scheduling (see section 2). We shall compare this speedup to that obtained by the best inbuilt scheduling clauses which has been determined previously to be dynamic 16 for both loops. Note: the best was taken to be the scheduling clause that gave the lowest runtime for that loop on 4 threads.

All program executions were done on Morar, Where we reserved an entire node (64 cores) for each execution. This ensured that no other jobs would effect the shared memory environment on which our job was running. In this way we could expect that we got very similar runtimes when we ran the same job repeatedly. To ensure we had accurate runtimes, we executed each job 100 times and took the average result as the expected value of runtime and the standard deviation to be the error.

2 Code

In this section we shall examine how we implement affinity scheduling in the code.

2.1 Data Structures; shared and private variables

When we open the parallel region we manually divide the work for the outer loop amongst available threads in a static way. The number of outer loop iterations, N, is divided by the number of threads to give the number of iterations per thread (*ipt*). We refer to these sets of contiguous data as each thread's local set.

The thread id (*myid*) is multiplied by the number of iterations per thread to give the loop iteration where that thread's local set begins $lo = myid * ipt$ and the $hi = (myid + 1) * ipt$ gives the endpoint for that local set. We shall herein refer to these private variables as low and high respectively.

If N is not divisible by the number of iterations then we round up. This ensures that all threads have an equal amount of iterations except for the last one. For example, if we have 18 iterations to split amongst 4 threads then they shall be divided 5,5,5,3.

We declared two dynamic arrays before we entered the parallel region. These are shared and will be set to the size of the number of threads used. The first (*r array*) shall contain the number of remaining iterations for each thread. This shall enable any thread to see, and alter, the remaining number of iterations left for any local set. The second (*ChunkEndPoint array*) shall be set to the value of high for each thread. Which, as described above, may be different for the last local set. This can be shared or private as it will not change value. We have set it as a shared variable here.

We now assign the values in the shared arrays. Each element of *r array* is the difference between the high and low iteration value for each local set and each element of *ChunkEndPoint* is set to the high iteration value for each local set.

2.2 While Loop

Once the above variables have been declared we enter a while loop wherein the code shall spend most of its time. There are two major steps undertaken in this loop:

1. Determine the values of low and high for the next chunk of iterations that is to be executed by a given thread. Update the value of the number of remaining iterations for that local set. Set found work as true if work was found (default is false).
2. Execute loops with the high and low values determined from the previous step then return to start of while loop; or break loop if found work is false.

To execute step 1 we check the number of iterations remaining in the thread's own local set. If this is greater than zero then we choose our next chunk from this local set; update the value of remaining in the shared array and set found work to be true. If number of remaining iterations is equal to zero then the thread shall steal a chunk from the thread with the largest number of remaining iterations in its local set, update the value of remaining for that local set and set found work to be true. If all the values of remaining in the shared array is zero then found work shall be set to false and the while loop will break in step 2.

2.3 Thread Synchronisation

Thread synchronisation is the most complicated aspect of this problem. To understand how we synchronised threads we must consider what problems may arise with unsynchronised threads.

There is an implicit barrier at the end of open mp parallel regions. So if each thread works on just its local set then there is no need to consider synchronisation within the while loop. However, when threads need to steal chunks from one another we must consider the following issues:

1. Two threads must not access the shared array of remaining iterations at the same time.
2. When a thread has successfully determined the number of iterations it must do, and on which local set, it must determine at which iteration it should start and end.

Since calculating high and low requires an access and update to the shared array of remaining iterations we must always have this in a critical region. This must be the same critical region for both assigning high and low in a thread's own local set and stealing from another's since we can never know which thread will work on which chunk next.

We can determine low by subtracting the value of remaining number of iterations on that local set from the last iteration on that local set which is given by the shared array *ChunkEnd-Point*. High is then simply low plus the size of the chunk- which is determined by the number of threads and remaining iterations.

It is possible, though extremely unlikely that one thread may enter the while loop and complete all the chunks in its local set and then steal from another thread's local set before the latter thread has reached the while loop. In this case the latter thread may set the shared value of remaining iterations back to the original value even though the prior thread has stolen a chunk. To avoid this potential issue we placed a barrier just before the while loop that will ensure all threads are synchronised as they enter the while loop.

3 Speed up

Figure 1 shows the speedup for loops 1 and 2 using affinity scheduling against an ideal linear speedup. We can see a good linear speedup for loop 1 and a decent consistent speedup for loop 2. We calculate the speedup for each core 100 times and use the average as the expectation value of the speedup for that number of cores and the standard deviation as the error.

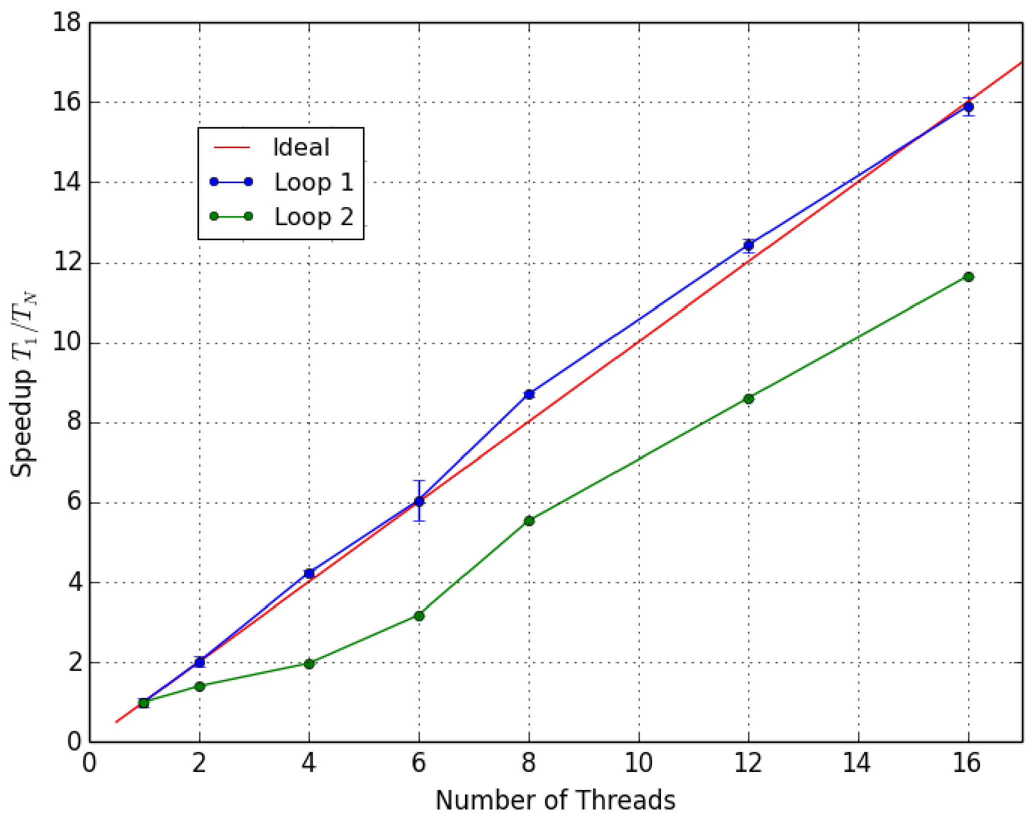


Figure 1: Speedup for loop 1 and loop 2 using affinity sharing compared to ideal linear speedup.

With affinity scheduling, no thread shall spend any time idle since once it is finished its section, or some other thread's chunk, then it shall find another chunk until there are none left. The only thread idle time shall be found when there are no chunks left to steal but other threads are still working. But as chunk sizes are a fraction of the remaining iterations the last few chunk sizes will be very small and this idle time shall be insignificant.

Since affinity scheduling causes no idle thread time we presume the poorer speedup of loop 2 is due to threads requiring more steals than for loop 1 as this shall cause threads to spend longer in the critical region.

3.1 Loop 1

First we shall discuss how the work is distributed in loop 1.

```
for (i=0; i<N; i++){
  for (j=N-1; j>i; j--){
    a[i][j] += cos(b[i][j]);
  }
}
```

Each iteration of the loop does not have an equal amount of work to do. In fact, this is a 2 dimensional array with N columns, where we are parallelising by giving each process multiple columns to iterate. However we can see from the inner loop that on each column we have a different number of calculations to be done. This corresponds to doing calculations on a lower triangular matrix, as shown, which can be a very common requirement. Thus it is very important we find the optimum way to do so.

$$\begin{pmatrix} & & & & \\ (0,1) & & & & \\ \vdots & (1,2) & & & \\ \vdots & \dots & \dots & & \\ (0,N-1) & \dots & \dots & (N-2,N-1) & \end{pmatrix}$$

It is clear that work decreases linearly between columns 0 to N-1.

Figure 2 shows the speedup using affinity and dynamic, 16 scheduling for loop 1. We can see that both of these schedules give a good linear speedup. We have a slightly better performance for dynamic, 16. This is likely because our affinity scheduling has a higher overhead. Recall that only one thread at a time may assign the values to high and low and update the shared array. When only small chunk sizes remain threads will need to get new chunks more regularly, therefore there may be threads stalling before the critical region.

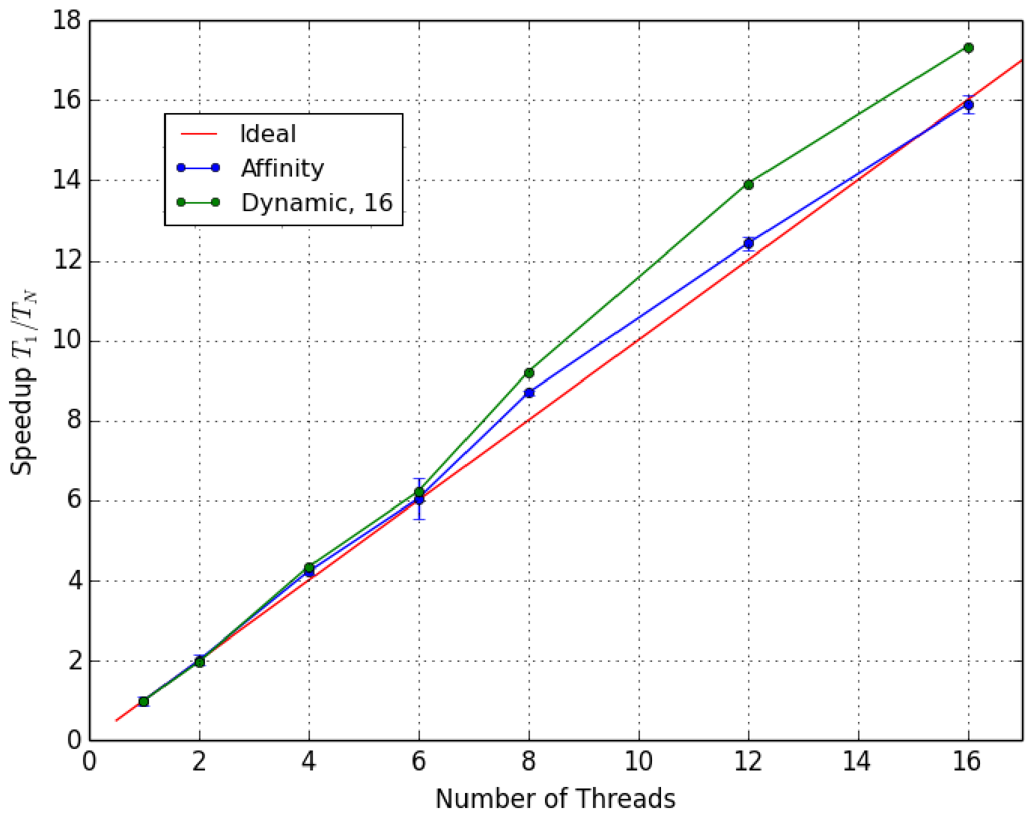
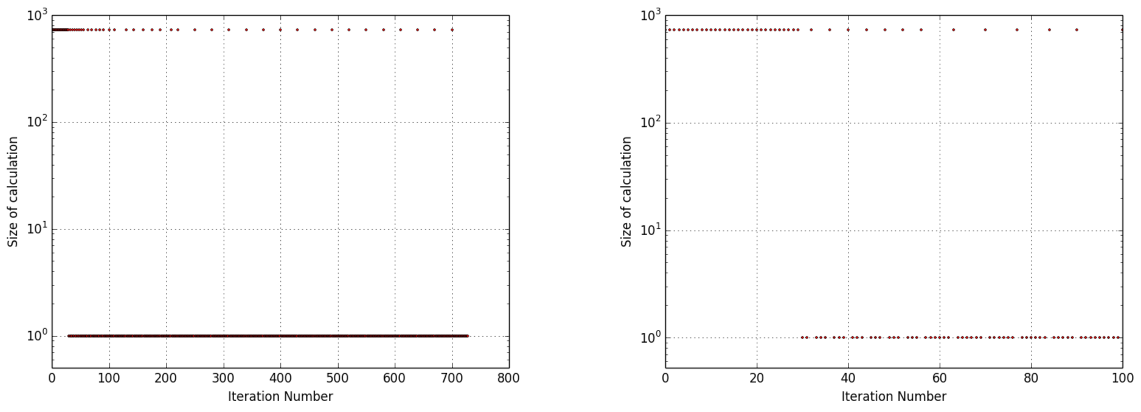


Figure 2: Speedup for loop 1 using affinity and dynamic, 16 scheduling compared to ideal linear speedup.

3.2 Loop 2

```
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;
    }
  }
}
```



(a) jmax[i] against i. (b) jmax[i] against i (zoomed).

Figure 3: In these figures we illustrate the size of the array jmax for each iteration (i) of the outer loop. The size of this array determines the number of inner loop iterations.

To understand the work balance in this problem we printed the array jmax. A plot of jmax[i] against i can be seen in figure 3. Most iterations are very short, with only one iteration of the inner loop required as jmax[i] is mostly of value 1. However, 67 of the iterations are of size jmax[i] = N = 729. The computation time will be hugely dominated by these larger iterations.

It is particularly important to notice that the first 29 iterations are of size jmax = N (figure 3b).

This 'cluster' of large iterations at the begining of the loop is the reason that we have good performance for dynamic, 16 on up to 4 threads, as seen in figure 4. With this chunksize we split this initial cluster almost perfectly amongst the first two threads (say thread 0 and 1) which will finish these chunks at a similar time as it will take threads number 2 and 3 to complete all remaining iterations. As we move to a higher number of threads, however, we get no additional speedup. This can be explained in the following way: threads 0 and 1 will spend all of their time on their first chunk while while the additional threads are only working on the inexpensive iterations and so the code still takes as long as it takes threads 0 and 1 to complete their first, and only, chunks.

We have a similar plateau effect above 8 threads using dynamic, 8. Similarly to dynamic, 16, this is because the first 4 threads spend their whole time on the cluster of long iterations at the begining. Here knowledge of the problem to be solved allowed us to predict what scehdule would give a better speedup even if it wasn't the best schedule on a particular number of threads.

We can clearly see from figure 4 that dynamic, n can give a better performance than affinity scheduling. This is believed to be for the same reason as loop 1- time spent in the critical region, especially when the chunk sizes are getting small. However it is also clear from figure 4 that it is difficult to find a chunk size that has a consistent speedup for some problems. Thus, although our implementation of affinity scheduling has more overhead in assigning chunks to threads, it shall have a very consistent speedup as no threads are ever stalling- thus this overhead is the only expense.

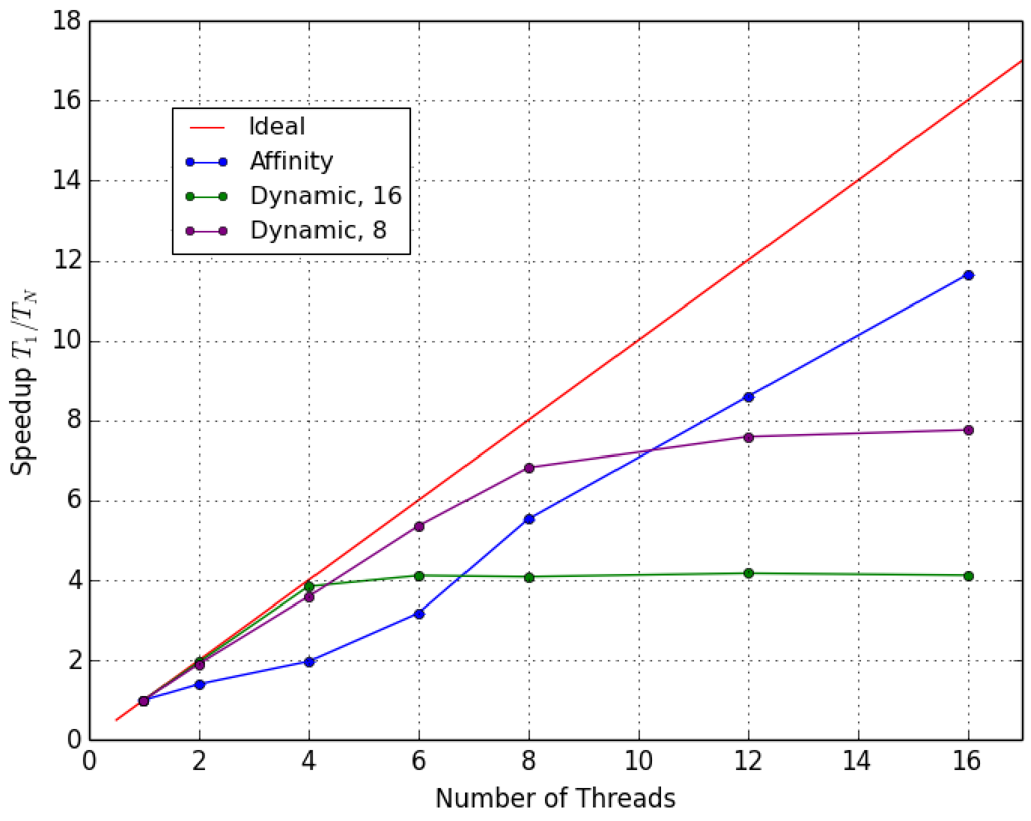


Figure 4: Speedup for loop 2 using affinity sharing, dynamic, 16 and dynamic, 8 compared to ideal linear speedup.

Given the discussion above we'd expect, within a margin of error, for affinity scheduling to have a speedup which is a straight line which follows the one between 8 and 16 cores. So we wish to determine the cause of the of the curve between 1 and 8 cores.

Consider that each local set is of size N/p, where p is the number of threads. We have that the first chunk for each is of size r/p where r is the number of remaining iterations in the local set.

Thus each thread's first chunk is N/p^2. Thus for number of threads 2, 4 and 6 we have initial chunk sizes of 183, 46 and 21 respectively. Therefore we have the problem here that thread 0 in each case is given most, or all, of the 'cluster' of expensive iterations at the beginning and hence is completing a considerable time after all other threads have completed their own, and the rest of thread 0's local set.

From this discussion it is clear that for a sufficiently high number of cores we will get a very reliable and predictable speedup for all problems.

4 Conclusions

Choice of the best schedule is problem dependent. Knowledge of the problem to be solved may give a good idea of what scheduling methods will be most effective. However, experimentation is often required to find the optimum schedule.

Following our discussion at the end of section 3.2 it is clear that affinity scheduling is very good for getting a good speedup for a large number of threads. Due to the fact that threads spend little or no time idle we have a very consistent, reliable speedup. For some problems we may get a poorer speedup on a small number of threads depending on the distribution of work but this should improve as we increase the number of cores as initial chunk sizes get smaller. This is especially important for loops with very irregular work loads such as loop 2. Which is instantly obvious when we consider that when experimenting on 4 threads we determined that dynamic, 16 was the best schedule but this gave no further speedup for a higher number of threads for reasons explained in section 3.2.

The main difference in performance between affinity scheduling and dynamic, n is that affinity scheduling has a high overhead but shall always give a consistent and reliable speedup while using the dynamic clause is very problem dependent. Furthermore, affinity scheduling may become more reliable (less problem dependent) for a high number of threads as initial chunk sizes are smaller. Affinity is the only scheduling scheme that we have examined that guarantees that no thread shall be idle while there are still iterations to be completed.