|epcc|

Threaded Programming Part 1

B087928

October 29, 2015

1 Introduction

When we parallelise loops using open multi-processing we have many options on how we distribute work amungst 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 multiple loop schedules on two different loops (loop 1 and loop 2). In each case we determine the best loop schedule, and the corresponding chunksize, to solve that particular problem on 4 threads. We use this schedule, and the corresponding chunksize, to determine the speedup for that schdule on up to 16 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 5 times and took the average result as the accepted runtime for that loop schedule.

2 Loop 1

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

Here we parallelise the outer for loop such that each thread will do a number of the iterations between $i = 0$ and $i = N$. 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. Therefore, if, on 4 threads, we split this into 4 blocks and give 0-N/4 to thread 0, N/4-N/2 to thread 1, N/2-3N/4 to thread 2 and 3N/4-N to thread 3 then thread 3 will have far too little work and thread 0 will have far too much.

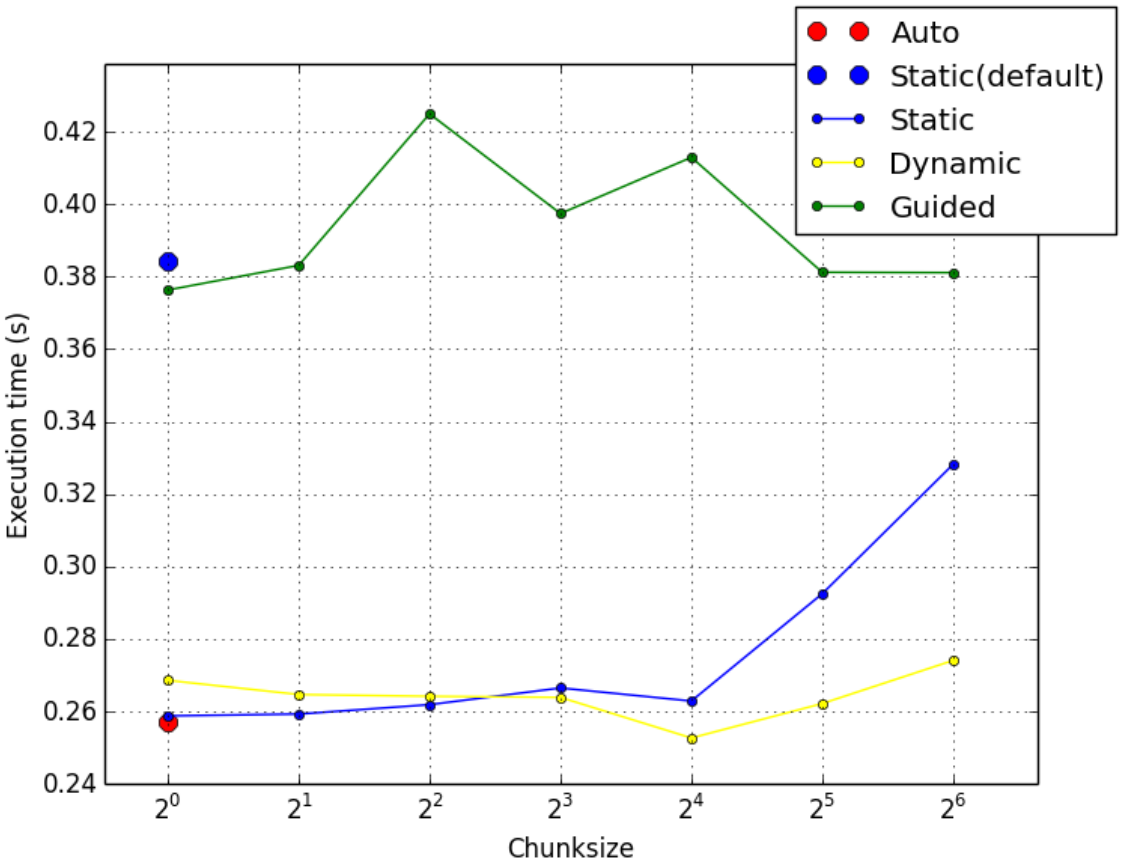


Figure 1: Execution time for loop 1 as a function of chunksize for different loop scheduling. Static(default) and Auto do not have a chunksize of 1 as seen on the graph but are included in the graph just to illustrate their execution time.

For this loop, we can clearly see that Static and Dynamic loop scheduling with specified chunk-

size both result in fast runs. We can see that Static is slightly better than dynamic for very small chunksize (1, 2, 4). Since Dynamic allocates the next chunk to the first available thread; the only reason for this is the overhead in book keeping required to implement a Dynamic loop scheduling.

By default Static uses a chunksize of number of iterations divided by number of threads. We can see that, for this problem, this is much slower than smaller chunksizes. This poor load balancing is usually because the majority of the large calculations are on a minority of the threads. This is the problem we predicted when examining the loop before attempting to parallelise. This is an example of how we may predict what loop schedule will be bad. While the difference between static and dynamic was not as predictable.

Guided starts off with a large chunksize and decreases with each new chunk allocation to a minimum of the chunksize specified. This is clearly a poor choice for this problem as the earlier iterations are the most expensive and this is reflected in figure 1.

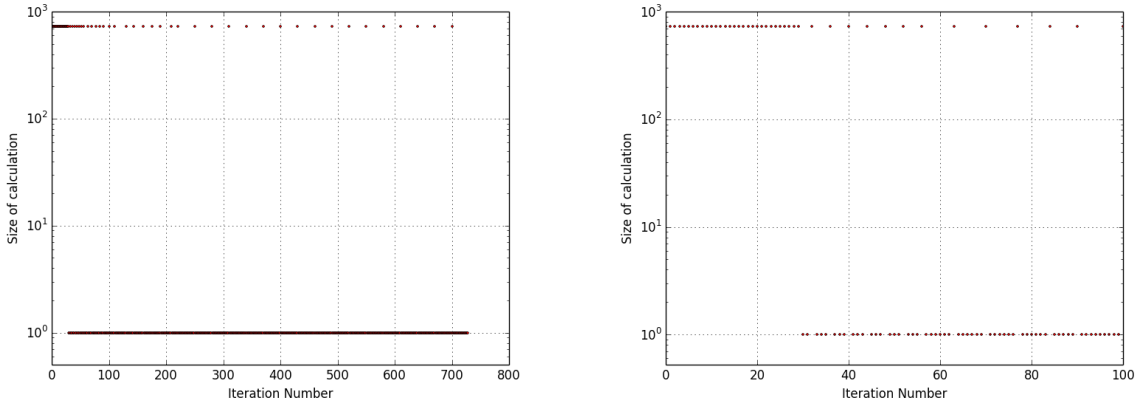
Auto allows the runtime to have full control over the assignment of iterations to threads. This is particularly useful if a loop is repeated throughout the code as this one is. The distribution of work for this loop is simple (later iterations have less work) so we can see from figure 1 that runtime develops a good load balance for this loop.

We can determine from this graph that the best loop schedule for this problem on 4 threads is Dynamic with a chunksize of $n = 16$.

3 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;
    }
  }
}
```

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 2. 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



(a) jmax[i] against i.

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

Figure 2: 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.

jmax[i] = N = 729. The computation time will be hugely dominated by these larger iterations. If we use a static, 1 schedule then we may have the following problem: threads, say, 1,2 and 3 may complete an iteration with jmax[i] = 1 then will have to wait for thread 0 which is taking much longer to complete an iteration with jmax[i] = N = 729 in order to get assigned new iterations. Therefore, we expect that static, n should give a much poorer performance than dynamic, n especially for a small chunksize (n).

It is particularly important to notice that the first 29 iterations are of size jmax = N (figure 2b). This implies that if we use a chunksize of 32 or 64, then thread 0 will have to do almost half of the total work. This is reflected in the steep disimprovement in runtime for static, 32 and dynamic, 32 in figure 3.

This 'cluster' of large iterations at the begining of the loop is also the reason that we have good performance for both static, 8 and dynamic, 8. With this chunksize we split this initial cluster almost perfectly amongst the four threads and thus the difference between static and dynamic is insignificant. When we move to chunksize 16, however, we see a much larger difference between these two scheduling methods. This can be explained in the following way: threads 0 and 1 will spend a lot of time on their first chunk while threads 2 and 3 will be finished quickly and stall with a static schedule but be assigned more chunks with a dynamic schedule.

Following the discussion above, it is obvious that default static is a poor choice and that thread one will do about half of the entire work.

As with loop 1, guided also gives a poor performance here, as the first few iterations are the

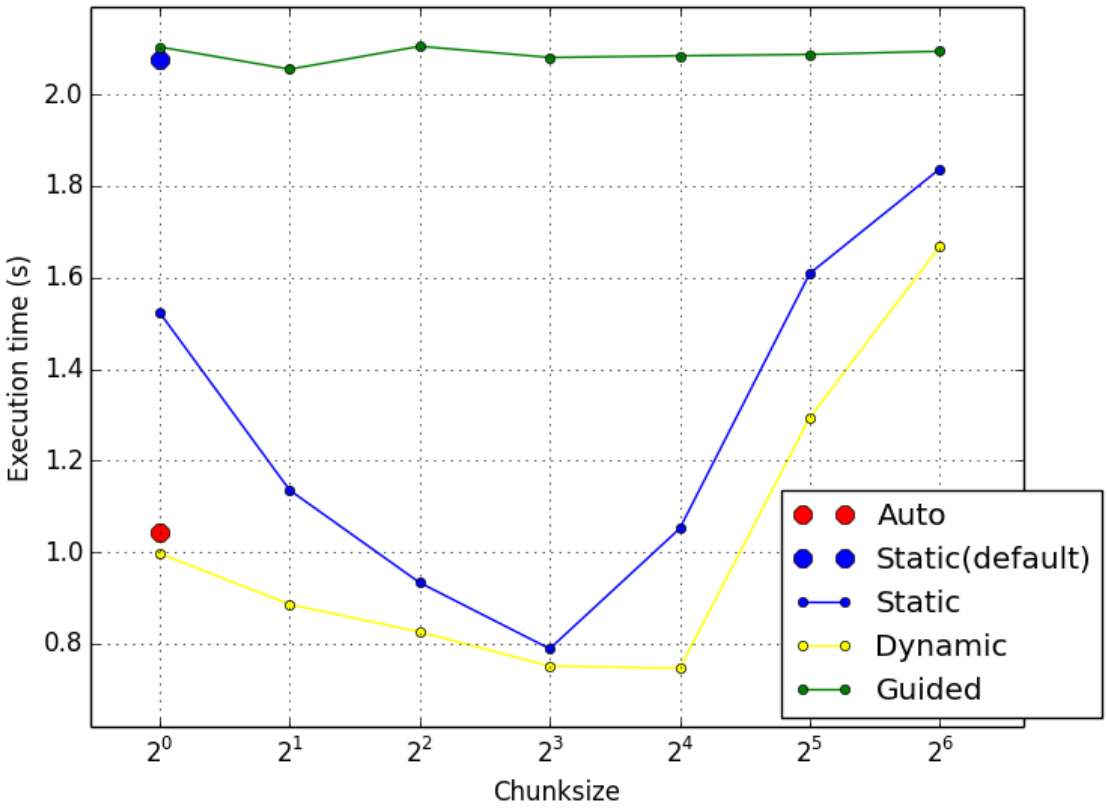


Figure 3: Execution time for loop 2 as a function of chunksize for different loop scheduling. Static(default) and Auto do not have a chunksize of 1 as seen on the graph but are included in the graph just to illustrate their execution time.

most expensive. This is obvious by looking at the problem but is also reflected in the graph.

Auto again develops a good schedule as the loop is repeated many times. However auto does not develop as good a schedule-relative to the best scheduling choice- as it does for loop 1. This is believed to be because the work balance is less straight forward in loop 2.

We can see from figure 3 that the best schedule for this loop on 4 threads is dynamic, 16.

4 Speed up

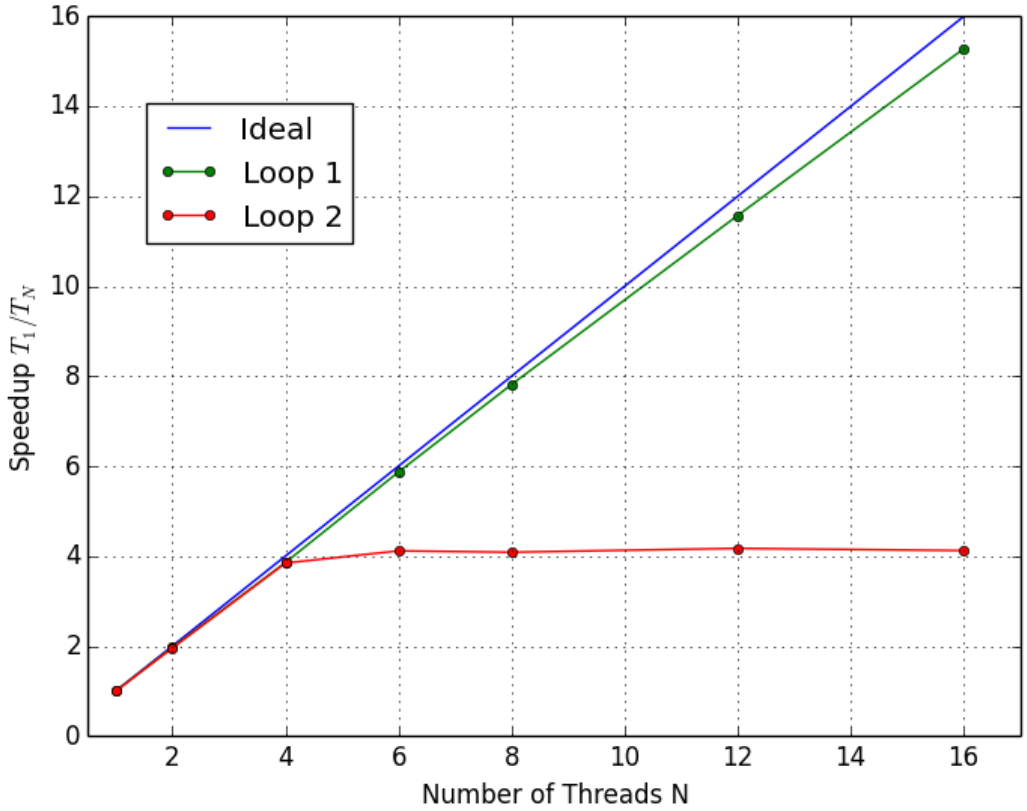


Figure 4: Speedup of loop 1 and loop 2 both with schedule dynamic, 16 compared to the ideal linear speedup.

When we double number of processes we would, ideally, expect the parallel section of our code to run twice as fast. In reality we have overheads in creating threads and scheduling as well as load balancing so we expect our real speedup to be less than the ideal, linear speedup. In sections 2 and 3 we determined the best schedule for 4 threads. This schedule is not necessarily best for a different number of threads.

In figure 4 we can see the speedups of loops 1 and 2 which, as expected, are less than the ideal linear speedup. We can see that the speedup for loop 1 scales well. However, loop 2 appears to saturate at 4 threads. To explain this we can refer to our discussion in section 3 where we discovered that for dynmaic, 16; threads 0 and 1 will spend most of their time on their first chunk while the other threads will complete many chunks. Therefore adding more threads has not improved speedup as the code takes as long as threads 0 and 1 need to finish their first (and

only) chunk.

We can solve this problem to give a better speedup by using dynamic, 8. We expect that, although this gave a slightly slower run for 4 threads, we will have a better speedup as we are no longer committing threads 0 and 1 to the cluster of large jobs. This better performance is illustrated in figure 5.

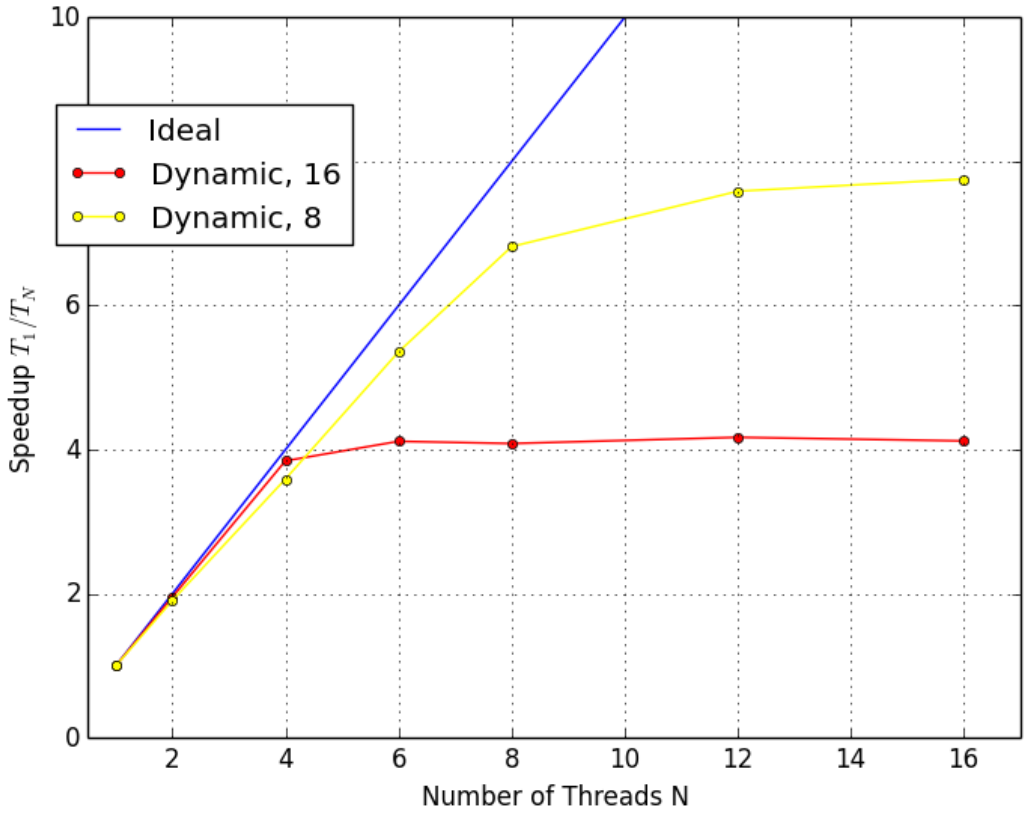


Figure 5: Speedup of loop 2 using dynamic scheduling with chunksizes 8 and 16 as compared to ideal speedup.

We have a similar plateau effect above eight 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.

5 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.

A guided schedule is a very poor choice for problems where the first few iterations are the most expensive.

Auto develops a good schedule when we repeat loops several times, especially when the work balance throughout the iterations is straightforward (follows a simple pattern).

Using static with default chunksize is a poor choice unless work is very evenly distributed throughout all iterations while we can usually get good performance with an appropriate chunksize. Static has a lower overhead than Dynamic, which is it's only advantage, but may make a difference if earlier iterations are equal in work or slightly greater than later ones since in this case we are unlikely to have threads waiting for each other.

Dynamic is a very reliable schedule for all irregular problems and generally makes up for its overhead in the time we save in threads not stalling.

The best schedule for a particular thread number is not necessarily the best schedule for a different number of threads. It is often possible to predict why this is from the problem. Therefore, for some problems, there is an advantage to know how many threads we would like to run on before we try to determine which schedule to use.

References

[1] Mark Bull, Fiona Reid, Amrey Krause, Threaded Programming Lecture Notes, Edinburgh Parallel Computing Centre (EPCC), Edinburgh University.