Scheduling in OpenMP

B083194

October 29, 2015

1 Introduction

Parallel computing has proven to be a fundamental part of modelling, both real life scenarios, and hypothetical. Without it, these complex simulations would cost too much, both financially and computationally. Parallel computers can be built relatively cheaply with commodity components, making their development more financially viable. The demand for more powerful machines is only going to rise with an increased range of applications: from big data mining to financial and economic modelling. However, often OpenMP applications fail to utilise the whole system effectively [1].

The aim of this project is to first parallelise our C code using OpenMP directives, specifically *parallel for*, before experimenting with the loop scheduling options to fully optimise our system. The code will first be run on 4 threads to determine the best scheduling option, and using this scheduling option we will then experiment with the number of threads to observe which thread count optimises the implemented schedule. Throughout this project, the *pgcc* compiler will be used along with the *–O3* flag on the command line to optimise the resultant times.

2 Parallelisation and Scheduling

2.1 The Code

The main source of parallelism in programming comes from loops, only loops with a predetermined number of iterations can be parallelised (eg for loops). In our code, lets call it Simon, there are two nested for loops, the first loop (loop 1) updates the elements of an upper trianglular matrix, a[i][j], with the cosine values of another matrtix b[i][j]. Because of its upper triangular form, the first iterations will be most expensive as the upper row is of length N, where N=729. Loop 2 updates the elements of a vector,c[i], by the log of the elements in matrix b[i][j] and a few additional constants. This loop is expected to take more time because it is a triple nested for loop. We know that there is a wide varying load because the first 30 iterations of jmax[i] are all N, whereas the later elements are mostly 1, meaning that earlier iterations in loop 2 will be most expensive. The first loop in Simon is quite easy to parallelise using the *pragma omp parallel for* directive, followed by clauses. For example the first loop (loop 1) is demonstrated below:

```
#pragma omp parallel for default(none)\
            schedule(runtime)\
            shared(a, b) \
            private(i, j)
  for (i=0; i<N; i++){
    for (j=N-1; j>i; j--){
      a[i][j] += cos(b[i][j]);
    }
  }
```

where runtime is the different shceduling options defined in a bash script, N is constant and jmax[i] is a vector of length N. The iteration variables are generally made private so that each thread has its own copy. We make a and b shared so that each thread can update their elements. Loop 2 is similarly parallelised (see appendix). When compiling, the *-O3* flag is used to optimise, if the *-O* flag is not used, optimisation levels are set to two. Specifying *-O3* means that scheduling within extended basic blocks is performed and traditional scalar optimizations such as induction recognition and loop invariant motion are performed by the global optimiser.

2.2 Scheduling options

The *schedule* clause gives a variety of options for specifying which loops iterations are executed by each thread. Within this, there are two arguments: *kind* and *Chunksize*. Different scheduling options optimise different loops. For example: *static* scheduling options work best for balanced loops, and *dynamic,n* is best used for wide varying loads. The following are the scheduling options considered in this project.

2.2.1 *Static* and *Static,n*

If a chunksize is not specified, static will break up the workload into roughly equal chunk sizes and each set of chunks gets assigned to a thread. If there the iteration count of the loop is divisible by the number of threads, then we can guarantee that each thread will receive an equal number of iterations to carry out. Hence, static is effective when working with balanced loops. However it is possible that the number of iterations is not divisible by the thread count, or the workload is not evenly distributed among the threads. *Static,n* is similar to static, except this time the chunks are of size n and the chunks are assigned cyclically to each thread in order. ”Static works well for simple calculations but not for memory bound calculations, or highly conditional applications” [2].

2.2.2 *Dynamic,n*

The *Chunksize* argument will divide up the iteration count into chunks of size n. For example, if chuck size was 10 and the total number of iterations was 200, then there would be 20 chunks of size 10. Dynamic scheduling will then allocate each chunk to a thread, and as soon as that thread has finished a chunk it will take the next chunk from the list. The threads are assigned on a first-come first-served basis. This makes *Dynamic* scheduling useful for iterations with wide varying loads. However, there is a very strong chance that remote access would occur and hence incurs suboptimal performances.

2.2.3 *Guided,n*

Guided,n is similar to *dynamic* in that it works on a first-come first-served basis, however the chunks of iterations start off large and get exponentially smaller. The size of the next chunk is proportional to the number of remaining iterations divided by the number of threads. Chunksize tells the schedule the minimum number of chunks to include. Guided can be less expensive than dynamic because it more efficiently distributes small tasks at the end of your *parallel for* to reduce the end time range among your threads. Users must be wary of the first chunk consisting of the most expensive iterations as this can lead to slow runtimes.

2.2.4 *Auto*

Auto leaves the scheduling up to the computer to allocate the work load the best it sees fit, and has no *chunksize* argument. Its increased popularity stems from its ability to work outside basic shared memory scheduling options. As the parallel loop is repeatidly executed, the runtime can develop a good schedule which has low overheads and improved load balance.

3 Experimenting With The Different Schedules

To determine the best schedule, we will run all of the possible schedules on four threads, varying the chunksize from 1 to 64. This was done on a bash script for speed and efficiency. Static with default chunksize and auto were not plotted on these graphs, however they are still analysed using tables. There are two loops being parallelised in our code, which may be optimised by different scheduling options. The different scheduling options must be plotted on two different graphs, one for each loop, so our image does not get cluttered. These are displayed in Figures 1 and 2. The results for loop 1 and loop 2 are also displayed in table 1 and 2.

Chunksize	Static	Dynamic	Guided	Auto
N/A	0.441746			0.263057
1	0.255817	0.263154	0.3433	
2	0.25407	0.22052	0.26182	
4	0.256041	0.218191	0.439508	
8	0.252506	0.217623	0.385973	
16	0.268574	0.192934	0.270828	
32	0.2636	0.257839	0.357207	
64	0.320747	0.223329	0.256789	

Table 1:Times for Loop 1

Chunksize	Static	Dynamic	Guided	Auto
N/A	2.161659			0.973949
1	1.556377	1.047986	2.137766	
2	1.005961	0.87561	2.083298	
4	0.853241	0.811357	2.086362	
8	0.811334	0.772241	2.100977	
16	1.081541	0.76678	2.089905	
32	1.546892	1.266056	2.03261	
64	1.89096	1.629445	2.021449	

Table 2: Times for Loop 2

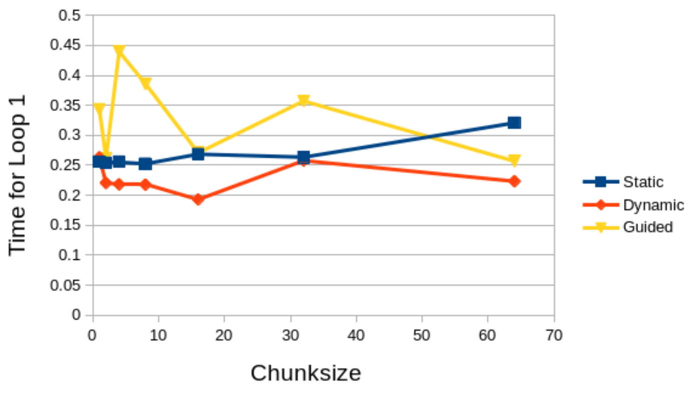

Figure 1: Time For Loop 1 Vs Chunksize For The Different Schedules

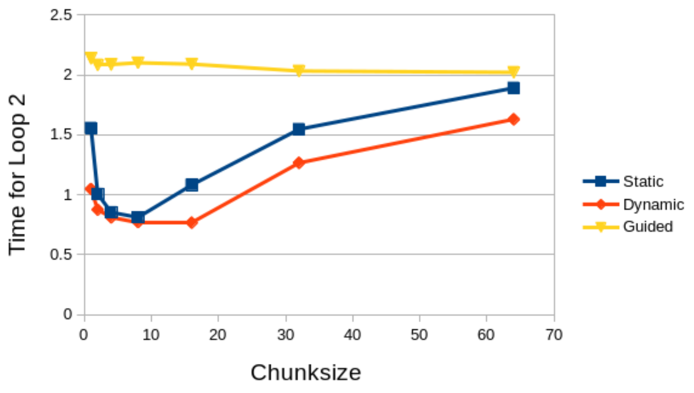

Figure 2: Time For Loop 2 Vs Chunksize For The Different Schedules

These graphs are difficult to read because they are so cluttered for low magnitude chunksizes. We notice that chunk size increases exponentially, and to declutter we simply have to standardise the x axis. The most intuitive way to do this is with a logarithmic scale of base two. The logarithmic scale will display the same data, but with even spacing for improved readability. These are displayed in Figures 3 and 4.

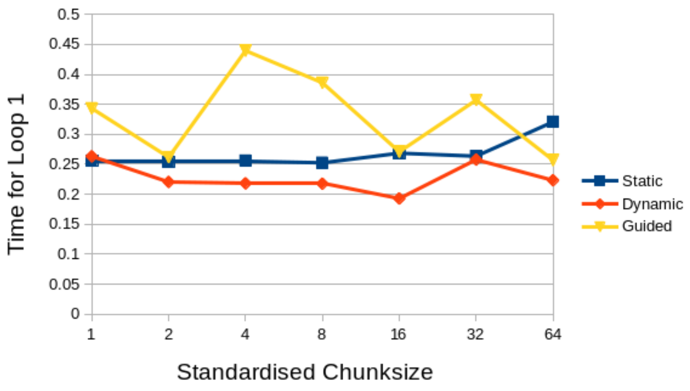

Figure 3: Time For Loop 1 Vs Standardised Chunksize For The Different Schedules

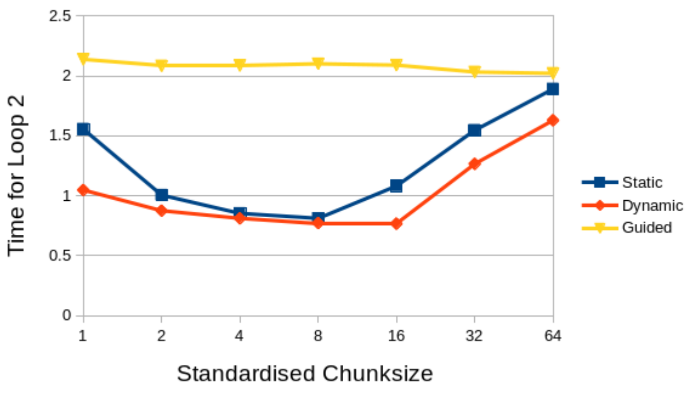

Figure 4: Time For Loop 2 Vs Standardised Chunksize For The Different Schedules

Now that the graphs are more readable, it becomes clear that the best scheduling, for both loop 1 and loop 2 on four threads, is *Dynamic, 16*. This suggests that the for loops may have a wide varying load.

For loop 1, dynamic is faster at most chunksizes compared with both static and guided, due to early iterations being so expensive. The lowest trough is dynamic 16, and so this will be used as our scheduling option in the next section.

The times for static, like dynamic, have a small range (excluding where the chunk size was not specified) which has a generally increasing trend. This is because larger chunksizes mean the first thread gets the lions share of the work.

The guided schedule approximately follows a highly variated crooked line. This is due to guided scheduling assigning the first threads the most iterations. Because these iterations are so expensive, the first thread may take a long time to finish. Auto scheduling experiences fast times, just slightly slower that the fastest dynamic times.

For loop 2, we notice that at each chunksize, dynamic is faster than both static and guided. The fastest time is achieved with chunk size 16, where smaller chunksizes incur more expensive overheads, and a larger chunksize reduces the benefit from smaller ranges of end times among the threads.

The times for static are just slower than that of dynamic, which is expected when the first iterations are most expensive. Static times trough around chucksize 8, and like dynamic, increase steeply either side of their trough, emulating an x^3 shaped plot.

Guided takes considerably longer to complete the tasks than its scheduling counterparts, except at the end points. Its shape is almost flat, with little indication of improved times. This is because loop 2 has expensive iterations early on, and guided assigns earlier chunks the most iterations. This means the rest of the threads will be waiting for the for the first threads to finish their iterations, regardless of the chunksize specified. The auto schedule completes its iterations quickly, with a time only slightly behind the quickest static and dynamic times.

4 Experimenting With The Number of Threads

Dynamic, 16 was found to be the fastest schedule, for both loops, with four threads. We will now implement that schedule with a varying number of threads (on 1, 2, 4, 6, 8, 12 and 16 threads). Naively, and ignoring Amdahl's law, we might assume that time will be inversely proportional to the number of threads. This is rarely the case.

4.1 Amdahl's Law

Amdahl's law states: "the performance improvement to be gained by parallelisation is limited by the proportion of the code which is serial". Or mathematically:

$$SpeedUp = \frac{1}{(1-P)+(P/N)}$$

Where N is the number of cores, and P is the parallel portion of the program. This equation can be separated into two parts, the sequential part $1-P$, and the parallel part P/N. If we increase the number of cores, N, for fixed P, our speedup will emulate a $sqrt(x)$ graph. This is because increasing the number of cores only affects the parallel part, a limit will be reached where increasing the number of cores no longer increases the speed up [3]. If we take the limit as N goes to infinity, we get:

$$\lim_{N\to\infty} Speedup = 1/(1-P)$$

This discredits the naive notion that more processors equates to more speedup. There is a point where speedup will not increase by a significant margin as the number of threads increase, we will refer to this as the critical point. Once you pass the critical point, the only way to increase the speed up is to increase the portion of code which is parallel. As the parallel proportion, P, increases, $SpeedUp$ increases.

4.2 Varying the number of threads

The number of threads was varied, with values ranging from 1 to 16, and then plotted against time. This is displayed in Figure 5. The minimum times for loop 1 and loop 2 are 0.053055 on 16 threads and 0.677092 on 8 threads respectively. There is a generally decreasing trend for time as the number of threads increase for both loops, however there is a slight upward curve present after thread 8 for loop 2. The time for loop two is much longer at the start, but this difference decreases significantly after the first few cores are added. Loop 2 appears to have reached its critical limit much quicker than loop 1, this can be shown better in a speed up Vs threads graph shown in Figure 6.

Figure 6 shows the speed up, $(Timeforthread1)/(TimeforthreadN)$, against the number of threads, which can be more useful for analysis. We see that the speed up for loop 2 reaches its critical point as it plateaus around 8 threads. In fact, the speed up decreases after 8 threads are used. This is due to an increase in an overhead called parallel set up time. As more threads are used, it takes longer to set them all up to execute the code. The parallel set up is part of the sequential part of the code so more processors will not speed this up. With loop 2, we can see that the overheads involved in adding more threads outweigh the potential benefits, and that 8 threads is best for this particular loop.

The speed up for loop 1 appears to increase approximately linearly as the number of threads increases, although there is some suggestion that it may

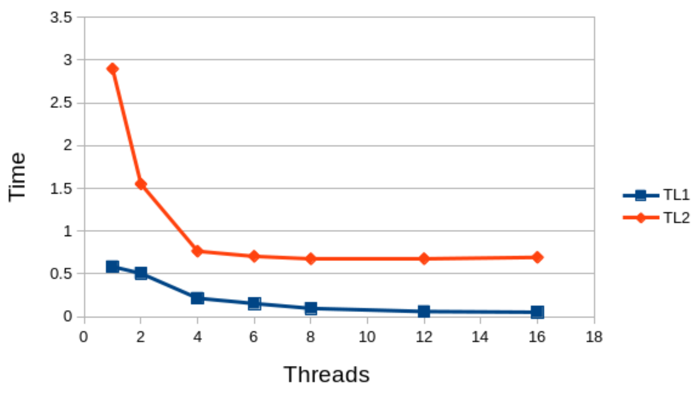

Figure 5: Time Vs Number of Threads

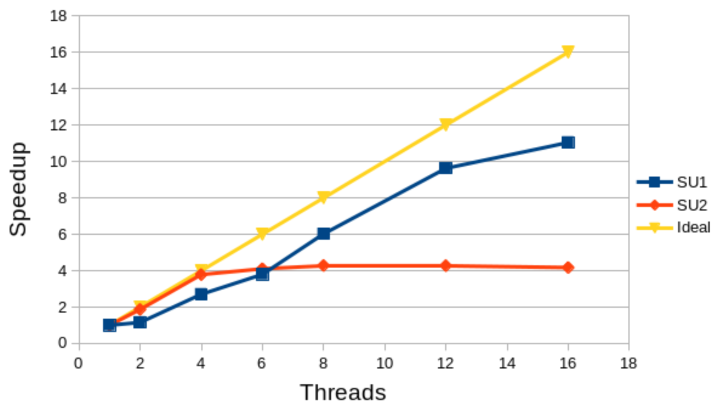

Figure 6: Speedup Vs Number of Threads

be reaching its limit as it begins to trail off towards the end point. This could be tested by running it on up to 64 cores, but that is beyond the scope of this project. The overheads, up to at least 16 threads, are less detrimental than the benefits gained by increasing the number of threads. The fastest speed up, and hence qucikest time, was 16 threads for loop 1.

5 Conclusions

Throughout this project, all 64 cores in Morar were reserved each time the code was executed, however times still may vary between successive runs. In this project the code was run 4 times, and the fastest times were taken for our results, however, an area for further development would be to run the program many times, then take the average value coupled with standard error bars either side.

In parallel computing, we must first consider the nature of our program before implementing our schedule. Different schedules work better than others on particular programs, and knowing which works best could prove worthwhile. Once the schedule which best optimises our program is found, the next step may not be to run it on the maximum number of cores available. The overheads involved in setting up the parallel region may outweigh the potential benefits, slowing our program down. Researchers should use both their intuition and knowledge when deciding an appropriate scheduling option and the number of threads they will need.

6 Appendix

```
#include <stdio.h>
#include <math.h>


#define N 729
#define reps 100
#include <omp.h>

double a[N][N], b[N][N], c[N];
int jmax[N];


void init1(void);
void init2(void);
void loop1(void);
void loop2(void);
void valid1(void);
void valid2(void);


int main(int argc, char *argv[]) {

  double start1,start2,end1,end2;
  int r;

  init1();

  start1 = omp_get_wtime();

  for (r=0; r<reps; r++){
    loop1();
  }

  end1  = omp_get_wtime();

  valid1();

   printf("TL1=%f\n", (float)(end1-start1));
  //printf("Total time for %d reps of loop 1 = %f\n",reps, (float)(end1-start


  init2();
  start2 = omp_get_wtime();

  for (r=0; r<reps; r++){
    loop2();
  }

  end2  = omp_get_wtime();

  valid2();

  printf("TL2=%f\n", (float)(end2-start2));
  //printf("Total time for %d reps of loop 2 =%f\n",reps,(float)(end2-start2)

}

void init1(void){
  int i,j;

  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      a[i][j] = 0.0;
      b[i][j] = 3.142*(i+j);
    }
  }

}

void init2(void){
  int i,j, expr;

  for (i=0; i<N; i++){
    expr =  i%( 3*(i/30) + 1);
    if ( expr == 0) {
      jmax[i] = N;
    }
    else {
      jmax[i] = 1;
    }
    c[i] = 0.0;
  }

  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      b[i][j] = (double) (i*j+1) / (double) (N*N);
    }
  }
```

```
}

void loop1(void) {
  int i,j;
#pragma omp parallel for default(none)\
                         schedule(runtime)\
                         shared(a, b) \
                         private(i, j)
  for (i=0; i<N; i++){
    for (j=N-1; j>i; j--){
      a[i][j] += cos(b[i][j]);
    }
  }

}



void loop2(void) {
  int i,j,k;
  double rN2;

  rN2 = 1.0 / (double) (N*N);
#pragma omp parallel for default(none)\
                         schedule(runtime)\
                         shared(c, b, rN2, jmax)\
                         private(i, j, k)
  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;
      }
    }
  }

}

void valid1(void) {
  int i,j;
  double suma;

  suma= 0.0;
  for (i=0; i<N; i++){
    for (j=0; j<N; j++){
      suma += a[i][j];
    }
  }
  printf("suma␣=␣%lf\n", suma);
  //printf("Loop 1 check: Sum of a is %lf\n", suma);

}


void valid2(void) {
  int i;
  double sumc;

  sumc= 0.0;
  for (i=0; i<N; i++){
    sumc += c[i];
  }
  printf("sumc␣=␣%f\n", sumc);
  //printf("Loop 2 check: Sum of c is %f\n", sumc);
}
```

References

[1] Peter Thoman, Hans Moritsch, and Thomas Fahringer. Topology-aware openmp process scheduling. In *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More*, pages 96–108. Springer, 2010.

[2] Thomas RW Scogland, Barry Rountree, Wu-chun Feng, and Bronis R De Supinski. Heterogeneous task scheduling for accelerated openmp. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 144–155. IEEE, 2012.

[3] Xian-He Sun and Yong Chen. Reevaluating amdahl's law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183–188, 2010.