Threaded Programming

TP Assessment 2

Exam number B137122

1 Introduction

The parallel execution of programs critical, since most modern processors have multiple processors. Independent loops present themselves as a relevant source to parallelise programs. To parallelise a loop efficiently is primarily dependent on the workload distribution of a loop. Loop schedulers are responsible for distributing loop iterations are among the threads. Thus, the right choice of the scheduling type is a critical aspect of loops to achieve excellent performance.

OpenMP has several already build in scheduling types. However, prior knowledge about the problem like load imbalance or the system architecture is essential to optimise scheduling and granularity of work (chunk_size) to achieve the best performance. It is often not possible to know how the distribution of the workload, hence we present an approach affinity scheduling. That accounts for some shortcomings.

In this report, we introduce and explain manual implementation affinity loop scheduling in OpenMP. We continue to examine the structure and workload distribution of loop 1 and loop2. We then test the manually implemented affinity loop schedule on 1 to 16 cores. Afterwards, we compare the results of the affinity scheduler with the best performing in-build schedules before concluding.

2 Methodology

In this section, we give first an introduction to the two loops provided for this assignment. After that, we describe the implementation of the affinity scheduler and the difficulties of implementing parallel code. Finally, we outline the test settings.

2.1 Loops

For evaluating the loop schedulers, we use two very different loops. In the following sections, we introduce both loops and give insight to the structure and the workloads of the loops.

Loop 1

Loop 1's definition is as follows:

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

Where N is the number of iterations, i and j are auxiliary variables, a and b are i x j matrices.

The loop is performing an update on a 2-dimensional array with N columns and N rows. However, by looking at the loop, we can see that the inner loop decreases the number of iterations after every iteration of the outer loop. Hence the loop is not actualising every field in the array just half as can be seen in the matrix

0, N-2 0, N-3 … … 0, 1
1, N-2 1, N-3 … 1, 2
2, N-2 … 2,3
… …
N-1, N-2

Figure 1 shows the linearly decreasing workload with increasing loop counter. Since we are parallelising the outer loop, every row is assigned to a thread. Since every row has a decreasing workload threads are assigned an ever decreasing workload, and threads with a higher row number finish faster than threads with a lower row number.

Loop 2

Loop 2's definition is as follows:

```
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;
            }
      }
}
```

Where N is the number of iterations, i, j and k are auxiliary variables, b is a i x j matrix, c and jmax are I dimensional vectors, rN2 is a scalar.

This loop is more complicated than loop 1, due to the vector jmax they are dependent on and two inner loops. The workload distribution is shown in Figure 1, where we can see the workload per loop counter. Key to understanding this workload pattern is the vector jmax.

Jmax is initialised with the following pattern:

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

Where N is the number of iterations, i, is an auxiliary variable and jmax is an N-dimensional vector.
Examining jmax, we can observe the pattern in Figure 1 where with increasing index the frequency of setting an index to N is decreasing. It is notable to mention that in the case of N=729 from the total 67 elements that are set to N, 30 elements are the first 30 set to N. That concludes that the newly assigned indexes contain a huge compute load and the later ones just a fraction of it.

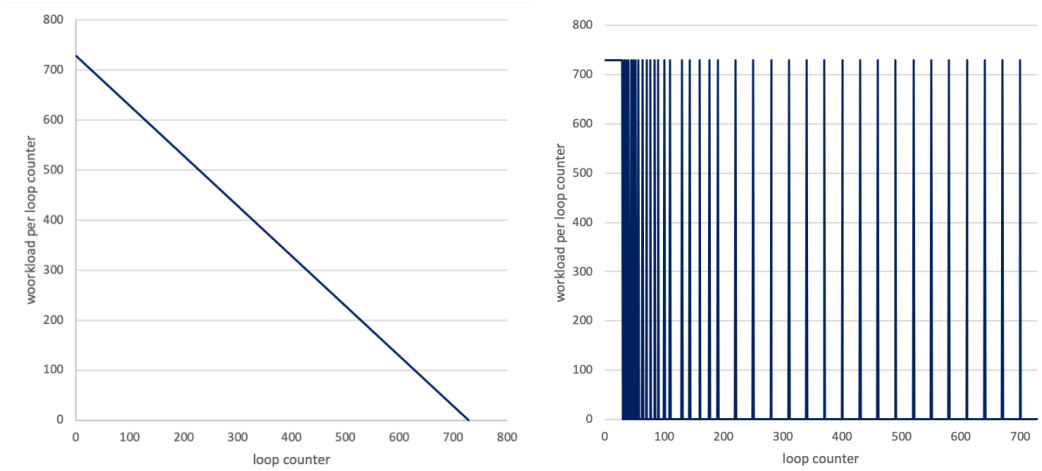

Figure 1: Workload per loop counter. The left Figure shows the workload of loop 1. The right Figure shows the workload of loop 2

2.2 Affinity scheduling

We examine the implementation of the affinity loop scheduling algorithm in OpenMP. We mainly describe in three parts how the shared data structures are used and how the threads are synchronised.

In the first part the affinity scheduling code we are

- declaring and initialising all the shared variables
- starting the parallel region
- distributing the number of iterations equally among the threads in a
- synchronising all threads
- starting the while loop
- distributing the number of iterations equally among the threads in a critical region

Shared variables

number available threads (`$num_thrds`)

Before we enter the parallel region in the code, we declare two one dimensional arrays with the size of the number available threads. These arrays are shared and help us later to keep track of each threads allocated iterations.

Furthermore, we compute the iterations per thread (`$ipt`) by dividing the number of outer loop iterations (`$N`) by the number of threads. In case `N` divided by `$num_thrds` has a remainder bigger than 0, the number is rounded up. This procedure ensures that each thread gets the same number of iterations assigned.

Parallel region

We start the parallel region with the OpenMP directive and assign the variables as mentioned earlier as shared variables to allow every thread to have access to them. When starting the parallel region, all are forked to the threads, and every thread is executing the same code.

Hence without dividing the data in chunks for every thread, the code would run in parallel, but every thread would compute the same, and there would be no speedup in the execution of code. Thus, we must distribute the work for every thread. We accomplish this by using the in-build runtime library function `$omp_get_thread_num()` to store thread id of every thread in the private variable `$myid`.

We then use `$myid` to divide the iterations (workload) into equal chunks for the threads. This division is done by multiplying `$myid` by `$ipt` to get the starting point (`$lower_limit`) and multiplying (`$myid` +1) to get the endpoint (`$higher_limit`) for every thread. If the last thread gets assigned a `$higher_limit` that is higher than `N` it is set to `N`.

Every thread shall know the `$lower_limit` and `$higher_limit` of the other threads. Thus the limits are stored in the shared arrays `$limits_lo` and `$limits_hi`. In this case, we do not need to worry about writing simultaneously to a shared variable since the threads access just their element in the array with `$myid`. We also declare and initialise the variables $lo and $hi as auxiliary variables, so a given thread calls loop1 or loop2 to compute the right iterations.

Barrier

After declaration and initialisation of variables and before we enter the while loop, we need to synchronise the threads, so all of them start at the same time entering the while loop. We implement this by setting a barrier before the while loop. This procedure is necessary as one thread may access the while loop completes its local set and steals another threads chunk before the other thread even enters the while loop. This particular case can happen when we assign a high number of threads, and some chunks are executed very fast.

While loop

After we assured the synchronisation of the threads, all threads enter the while loop where it iterates until there are no more iterations for computation anymore. While being in this loop the

We check if a given thread still has iterations left to compute
`IF` there are still iterations left continue executing
`ELSE` we determine from which thread the idle thread can steal form and synchronise the threads
We check if there is still work to be done
`IF` yes we call the loops (loop1 and loop2) to be executed
`ELSE` we break out of the loop

Critical region

Inside the while loop, we check if a given thread still has iterations left to compute and update `$lo` and `$hi` that are later used for executing loop1 and loop2. The update of `$hi` and `$lo` requires access and update to the shared arrays `$limits_lo` and `$limits_hi`. To ensure that multiple threads cannot write to the shared array, we need to put this in a critical region.

Set thread ID

In this region, we check if a given thread has still iterations from its local set available
`IF` iterations in the local set are available -> set `$thrdid` to `$myid` and continue calculation on local set
`ELSE` find the most loaded thread -> set `$thrdid` to the id of the most loaded thread

Update workload

After determining on which `$thrdid` a thread should work on we calculate the remaining iterations and update the variables `hi` and `lo` for the thread as well as the shared array `$limits_lo`, so other threads get access to the updated distribution of the iterations. Afterwards, we exit the critical region since all necessary variables are updated.

Call loop1 or loop2

Now we are able to call the main computing function loop1 or loop2 to be executed on the iterations determined by `$lo` and `$hi`.

2.3 Test settings

All testing of the loops was performed on the Cirrus UK National Tier-2 HPC with the specifications shown in Table 1. The programs were compiled with the Intel compiler and the full optimisation option "-O3" using the following command:

```
icc -O3 -qopenmp -std=c99 -o loops loops.o -lm
```

To account for possible inconsistencies on the HPC computation nodes, we executed each of the loops for 10 runs. To conduct the testing, we used 1, 2, 4, 6, 8, 12 and 16 threads.

Table 1. Cirrus node specifications

	Cirrus node
Sockets	2
Processor	Intel Xeon E5-2695
Clock speed	2.1GHz
Architecture	x86_64
L1D cache	32KB
L1I cache	32KB
L2 cache	256KB
L3 cache	45MB
RAM	256GB
Physical CPU cores	18
HT CPU cores	36

3 Results

In the following sections, we discuss the performance of the implemented affinity scheduler on the two loops introduced in section 2.1. Furthermore, we compare the results with the best performing built-in OpenMP schedule types for these loops.

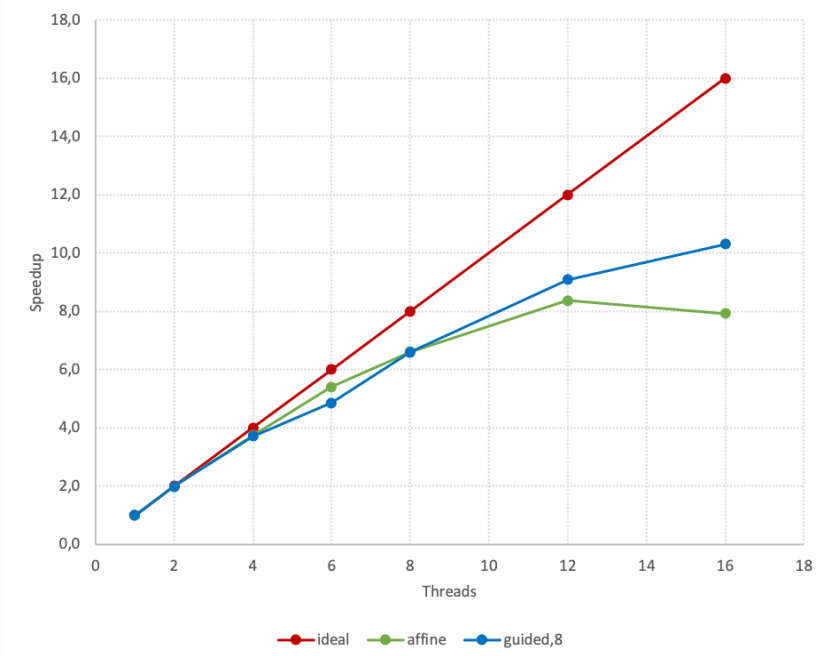

Figure 2. Comparison of the actual and ideal speedup for loop 1 using the affine schedule and the guided schedule.

Table 2: Runtime for loop 1 and loop 2 with the best configurations on different numbers of threads. Loop 1 with schedule "affinity" and "guided, 8". Loop2 with schedule "affinity " and "dynamic, 16".

Threads	Loop 1 affinity	guided,8	Threads	Loop 2 affinity	dynamic,16
1	1,658	1,659	1	8,789	8,657
2	0,843	0,833	2	6,300	4,403
4	0,444	0,447	4	4,467	2,204
6	0,308	0,342	6	2,768	2,072
8	0,250	0,251	8	1,596	2,072
12	0,189	0,183	12	0,958	2,072
16	0,293	0,161	16	0,796	2,074

3.1 Loop 1

Loop 1, as described in section 3.1 is a loop where with increasing loop counter, the workload is linearly decreasing. Hence a loop scheduler that can redistribute the workload to idle threads efficiently with linearly decreasing workload has an excellent performance. Figure 2 shows the comparison between the best performing scheduler from coursework 1 was "guided,8" and the affinity scheduler. As we can see both the "guided,8" as well as the affinity schedule are performing very well on loop 1 up to 4 threads. With increasing numbers of threads the difference between the ideal and the actual speedups of both "guided,8" and affinity scheduler start to diverge. The increasing overhead can explain this divergence since hat only one thread at a time can update its workload. With small chunk sizes remaining towards the end, threads must get new chunks more often and if to many threads need to access the critical region it can start to be a bottleneck. Furthermore we can see in Table 2 that the affinity scheduler minimum execution time is while using 12 threads and adding 4 more threads results in a worse time. Utilising more than 12 threads seem to add more overhead than computing power and result in a worse performance while the build-in "guided,8" scheduler still reducing its execution time.

3.2 Loop 2

Loop 2 is the loop with a high load imbalance and is described in detail in section 3.1. Running the affinity schedule on loop 2 with a variety of thread numbers shows that it performed best with the maximum tested 16 threads with an execution time of 2 as shown in Table 2. That means it can use additional threads to perform even better.

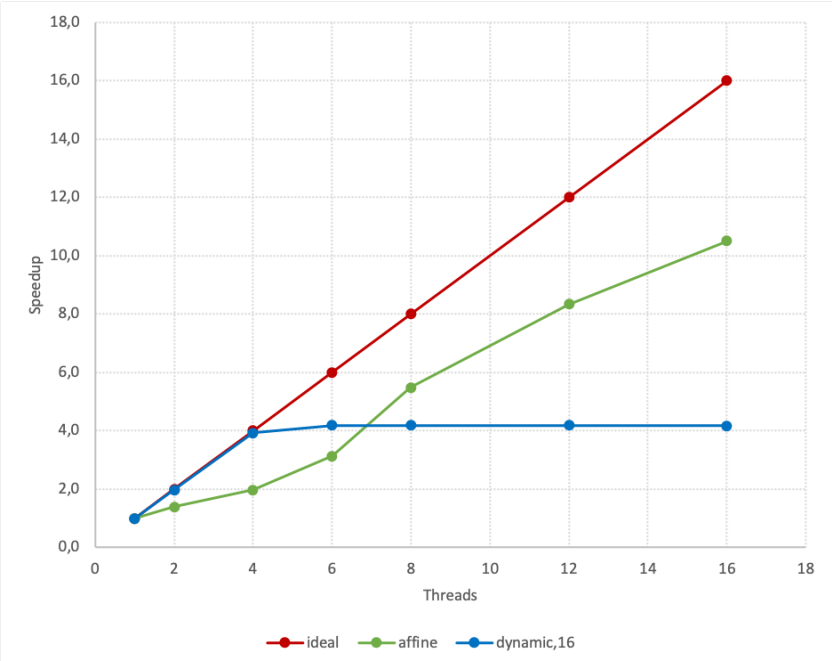

Figure 3. Comparison of the actual and ideal speedup for loop 2 using the affine schedule and the dynamic,16 schedule.

Table 3: Comparison of the actual and ideal speedup for loop 1 and loop 2. Loop 1 with schedule “affinity” and “guided, 8”. Loop2 with schedule “affinity “ and “dynamic, 16”.

Loop 1				Loop 2			
Threads	ideal	affine	guided,8	Threads	ideal	affine	dyn,16
1	1	1,0	1,0	1	1	1,0	1,0
2	2	2,0	2,0	2	2	1,4	2,0
4	4	3,7	3,7	4	4	2,0	3,9
6	6	5,4	4,9	6	6	3,1	4,2
8	8	6,6	6,6	8	8	5,5	4,2
12	12	8,4	9,1	12	12	8,3	4,2
16	16	7,9	10,3	16	16	10,5	4,2

Figure 3 shows the speedup of the affinity schedule compared with the best performing in-build scheduler from coursework 1 “dynamic,16” and the ideal speedup. We can see that the affinity scheduler scales pretty well especially with a high number of threads.

Looking closer we can see that affinity schedule is performing worse than dynamic,8 up to 6 threads as shown in Table 3. The excellent performance of dynamic until thread 4 is due to the preferable distribution of the workload among the threads. The chunk size splits the workload almost entirely for the number of threads. While the thread that gets the first chunk is busy working on the high workload, the other threads work on the rest of the workload and all of them finish almost concurrently. This occurrence happens since the first workload is a multiple of the following workloads. However, increasing the number of threads above 4 shows no improvement in speedup since the thread with the first workload is the lowest common denominator. Adding more threads is not helping to get any additional speedup since the first chunk is computational very expensive that it does not matter how fast the rest of the chunks are finished, they have to wait for the first one to finish. This behaviour we can see in Figure 3 where the speedup flattens at 4 threads and stays on that rate no matter how many new threads are available.

In contrast to the “dynamic” schedule, the affinity schedule speedup shows a very different trend. By comparing the results of 1 to 6 threads, the affinity schedule performs worse. This bad performance is most probably due to the time spent in the critical region to redistribute the remaining workload. Especially when the chunk sizes are getting smaller, the threads are accessing the critical region more frequently blocking other threads in updating their workload and additionally redistributing the work without taking into account any data locality. Although all threads are executed on the same CPU, due to the NUMA architecture of modern CPUs the access time to the cache memory, is dependent on how close the cache is to the executing core. However, looking at 8 threads and grater, we can see that affinity scheduling is surpassing dynamic,16 and shows a constant speedup increase most likely to continue beyond 16 threads.

4 Conclusion

As CPUs are equipped with more and more cores, parallelisation of applications is becoming increasingly important. Loops are essential code sections for parallelisation, since they often contain the most substantial workload. Loop schedulers do distribution of workload among the threads.

The affine scheduler is a scheduler that is not in-build in OpenMP. Hence we had to implement the scheduler by ourselves. The section Methodology describes the implementation of the scheduler and allude to the difficulty of synchronising threads at the right time to prevent race conditions. The description of the implementation shows that writing clean parallel programs is not a trivial undertaking.

As we have seen in the section results, the choice of the best schedule is dependent on the problem. Often it is unknown how the workload of a problem is distributed and to find the optimal schedule for the problem often requires testing of various schedulers.
We identified this problem by executing the affinity scheduler on two loops with very different workload distributions and comparing them to the best performing in-build schedules. Considering the results of loop 1, we can identify the overhead the affinity schedule struggles with when using high numbers of threads. In the

case of using more than 12 threads, the affinity schedule performance even decreases. The explanation for this behaviour is that loop1's linearly decreasing workload, which produces much overhead due to a large number of threads that are stealing small chunks from each other.
Investigating the results on loop2, we can conclude that the affinity schedule shows impressive speedup for a large number of threads on an unbalanced workload. The results show that although the built-in "dynamic,16" has a better speedup while using up to 6 threads, since using more than 4 threads with "dynamic,16" results in the other threads being idle and waiting for the threads with high workload to finish. Affinity scheduling, in contrast, assigns workload continuously to treads that fished their workload, which gives this schedule type a very consistent, reliable speedup.