Threaded Programming

TP Assessment 1

Exam number B137122

1. Introduction

Multithreaded programming is an important part in reaching faster executions of programs since semiconductor advancement in making higher frequency processors has slowed. This report uses two distinct loops to analyse the performance of the various schedule clauses and the speedup gained by using an ever-increasing number of threads. First, we will give a short introduction into the loops structure. The report continuous with explaining the testing methodology this includes implementation, test scenarios, test settings. Thereafter we show and analyse the results of the tests and finally, we summarise the findings in the conclusion.

1.1.Loops

The program most important element are the two loops shown here:

Loop 1

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

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;
                }
        }
}
```

We can see that both loops are very different in their structure. This can be best seen by looking at the workload of both loops as shown in figure 1. While loop 1 has a steadily falling workload, loop 2 is very different showing spikes in the workload.

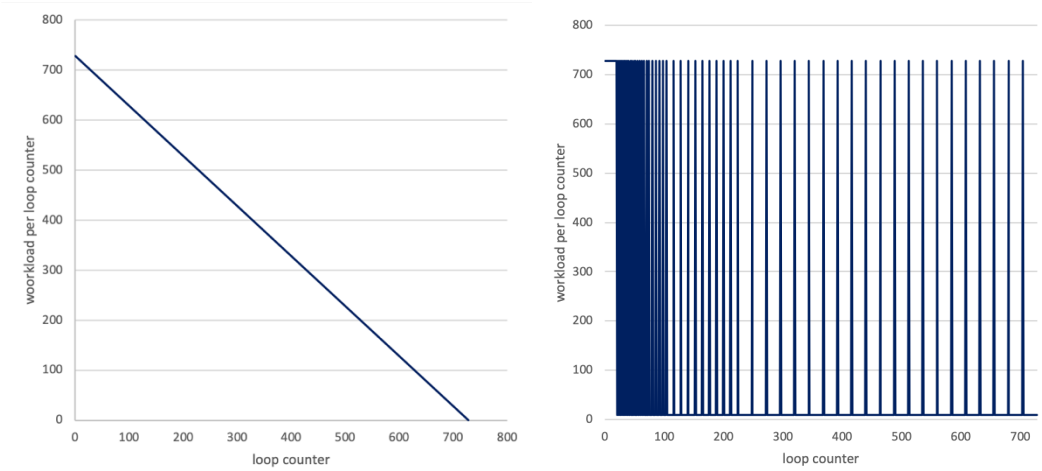

Figure 1: Workload per loop counter. Left figure shows workload of loop 1, right plot shows workload of loop 2.

2. Methodology

2.1.Implementation

The serial code for the coursework was provided by EPCC and is written in C. However, to apply it for testing the performance of multithreaded programming, the program needed to be parallelised by adding the OpenMP parallelization directives to loop 1 and loop 2. The parallelization directive added was

#pragma omp parallel for schedule(*schedule*, *n*) private(*private_variables*) shared(*shared _variables*) default(none)

where *schedule* is the name of the SCHEDULE clause option, *n* is the chunksize, *private_variables* are the names of the variables that are not to be shared (in our case the iterators) and *shared _variables* the variables that can be accessed by every thread.

To investigate the performance changes of the parallelised loops by OpenMP, test scenarios were conducted in the following manner.

In the test scenario 1 the number of threads processing the program was set to 4 and held constant during the testing, while the SCHEDULE clause option and the chunksize n were variable. The options are shown in table 1.

In contrast to scenario 1, test scenario 2 was used to investigate the performance of the loops by increasing the number of threads. To keep things manageable, the tests were not run on every possible configuration of scenario 1. Instead we build on the results of that scenario and used the best performing options for schedule type and chunk size for loop 1 and loop 2 respectively. Since the loops behave very differently the best options for loop 1 were different to the ones for loop 2.
For loop 1 the SCHEDULE clause option was set to “guided,n” and the chunk size to 8. For loop 2 the SCHEDULE clause option was set to “dynamic,n” and the chunk size to 16.

2.2.Test settings

All testing of the program was performed on the Cirrus UK National Tier-2 HPC with the Intel compiler and the full optimization option “-O3”. To account for possible inconsistencies on the HPC computation nodes, 3 runs were conducted for every test case.

Table 1: Options of different scenarios.

Scenario 1		Scenario 2
Schedule	Chunksize (n)	Threads
• "auto"	• 1	• 1
• "static"	• 2	• 2
• "static,n"	• 4	• 4
• "dynamic",n	• 8	• 6
• "guided,n"	• 16	• 8
	• 32	• 12
	• 64	• 16

3. Results

In the following sections we will discuss the results of the program run with the different scenarios described in table 1. First, we go through the outcomes of scenario 1 and determine the optimal schedule type and the best chunk size for each of the loops. In the following subsection 3.2 we examine the results of scenario 2 and determine the speedup gain by increasing the number of threads.

3.1. Scenario 1, loop 1

As in table 2 displayed he results show that the difference between the different schedules is rather small. The worst performance has the schedule type "static" without chunktype (figure 3). The difference between the schedule type "dynamic,n" and "guided,n" is marginal (figure 4). And in the case where the chunk size is set to 32 "dynamic" performs even better than "guided,n". Further, we can see an optimal chunk size between 8 and 16. The difference between those two lies in the area of 10^{-4} seconds (table 2) and can be depended on other factors such as busyness of the network. The best performing scenario for loop 1 is "guided,n" with a chunk size of 8. "Guided,n" is the logical winner, since the dynamically assigned blocks are decreasing after every work block is finished, and the longer the loop runs the smaller the workloads gets.

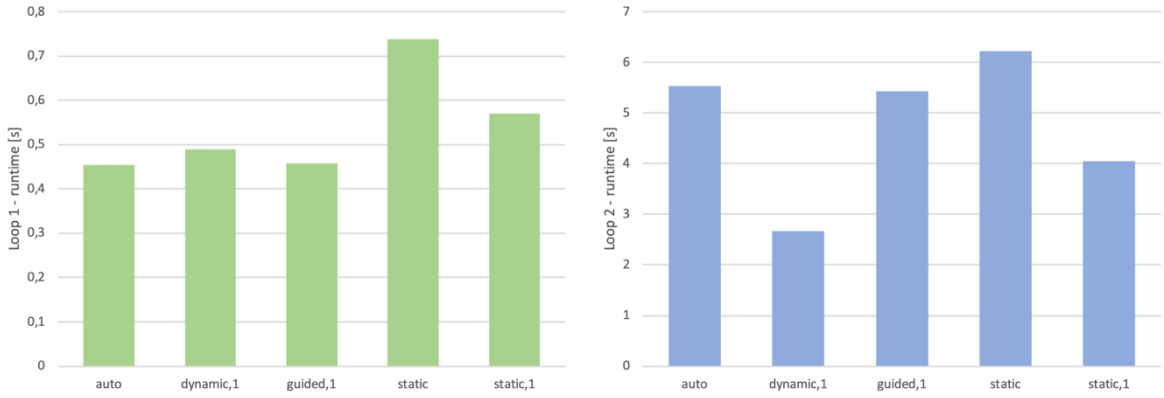

Figure 3: Comparison of different schedule options.

Table 2. Results of the computation. * is the best performing time for loop 1, ** is the best performing time for loop 2

Schedule Type	Number of Schedules	Loop 1				Loop 2			
		Run 1	Run 2	Run 3	Mean	Run 1	Run 2	Run 3	Mean
static,n	1	0.586	0.586	0.541	0.571	3.870	4.159	4.135	4.055
static,n	2	0.438	0.438	0.438	0.438	3.001	3.018	2.999	3.006
static,n	4	0.430	0.430	0.430	0.430	2.777	2.755	2.735	2.756
static,n	8	0.432	0.432	0.432	0.432	2.333	2.339	2.337	2.336
static,n	16	0.442	0.442	0.442	0.442	3.110	3.112	3.111	3.111
static,n	32	0.468	0.468	0.468	0.468	4.792	4.795	4.795	4.794
static,n	64	0.524	0.524	0.525	0.524	5.438	5.442	5.442	5.440
dynamic	1	0.493	0.493	0.485	0.490	2.653	2.707	2.663	2.674
dynamic	2	0.438	0.438	0.437	0.438	2.630	2.659	2.625	2.638
dynamic	4	0.430	0.430	0.429	0.430	2.556	2.517	2.593	2.555
dynamic	8	0.424	0.424	0.423	0.424	2.207	2.207	2.206	2.207
dynamic	16	0.422	0.422	0.421	0.422	2.203	2.203	2.202	2.203**
dynamic	32	0.422	0.422	0.419	0.421	3.879	3.880	3.878	3.879
dynamic	64	0.452	0.452	0.450	0.452	4.913	4.914	4.913	4.913
guided,n	1	0.457	0.457	0.461	0.458	5.428	5.430	5.429	5.429
guided,n	2	0.416	0.416	0.421	0.418	5.429	5.429	5.429	5.429
guided,n	4	0.421	0.421	0.422	0.421	5.428	5.430	5.429	5.429
guided,n	8	0.415	0.415	0.420	0.417*	5.429	5.429	5.429	5.429
guided,n	16	0.415	0.415	0.421	0.417	5.428	5.429	5.429	5.429
guided,n	32	0.420	0.420	0.425	0.422	5.428	5.429	5.429	5.429
guided,n	64	0.422	0.422	0.429	0.425	5.429	5.429	5.429	5.429
auto	-	0.480	0.456	0.425	0.454	5.502	5.669	5.431	5.534
static	-	0.764	0.726	0.725	0.738	6.226	6.229	6.234	6.230

3.2.Scenario 1, loop 2

The results for loop 2 are more distinguishable compared to loop 1. As with the previous loop the worst performer is “static” (figure 3) without any dedicated chunksize, followed by “auto” that also takes no chunksize as an option and schedule type “guided,n”. The steadily decreasing blocks in the “guided,n” schedule type cannot take advantage of their algorithm where loop 2 is having too much workload in the beginning, that the later threads finish their work and wait until the first ones finish their jobs (figure 1). The “dynamic,n” scheduler preforms by far best (figure 4). With the dynamic allocation of the work blocks the algorithm has the ability to harness the unique structure of loop 2 by dynamically allocating the work among the threads. Furthermore, the performance can be increased by optimizing the chunk size. While the chunk size is not affecting the “guided,n” scheduler at all, the other schedules performance can be optimized by adjusting the chunk size. Table 2 shows the best performing option, with the optimal chunksize of 16 and the “dynamic,n” scheduler. that is also one with an execution time of 2.203 seconds. This is just a tiny bit faster than “static, 8” but whole 3.226 seconds that is 1.46 times faster than the “guided,n” schedulers best time.

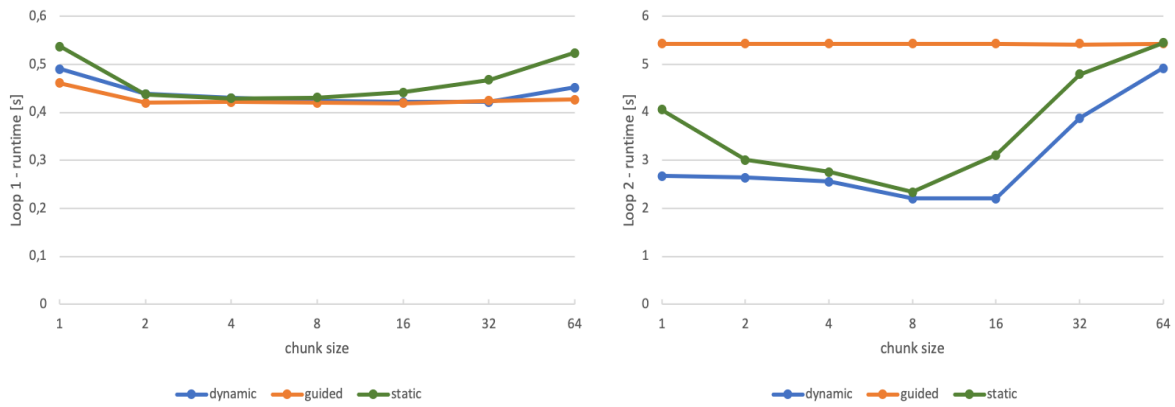

Figure 4: Comparison of the actual and ideal speedup for loop 1 and loop 2.

Table 3: Runtime for loop 1 and loop 2 with the best configurations on different amounts of threads. Loop 1 with schedule "guided, 8". Loop2 with schedule "dynamic, 16".

	Loop 1				Loop 2			
Threads	run 1	run 2	run 3	mean	run 1	run 2	run 3	mean
1	1.649	1.643	1.683	1.659	8.657	8.659	8.657	8.657
2	0.829	0.826	0.846	0.833	4.402	4.401	4.406	4.403
4	0.425	0.423	0.492	0.447	2.204	2.204	2.204	2.204
6	0.364	0.328	0.333	0.342	2.071	2.072	2.072	2.072
8	0.237	0.262	0.254	0.251	2.072	2.072	2.072	2.072
12	0.173	0.197	0.178	0.183	2.071	2.073	2.072	2.072
16	0.184	0.137	0.162	0.161	2.077	2.072	2.072	2.074

3.3. Scenario 2, loop 1

As shown in figure 5, loop 1 can make use of the additional threads. Using 2 threads gives nearly ideal speed up, while adding more threads slightly decreases the speed up ratio as can be seen in table 4. This loop is scalable and with 16 threads there is still a speedup of 10.

3.4.Scenario 2, loop 2

In contrast to loop 1, loop 2 is just scalable up to a certain point, thereafter the runtime remains static even when additional threads are added (table 3). This can be explained by the loop's shape of a workload. The workload spikes there is a minimum time that cannot be surpassed because the thread that gets assigned the last heavy workload (spike) needs a certain amount of time to accomplish the task. Hence assigning more threads to the loop cannot undercut the time needed to finish this workload. This can be seen in figure 5 where the utilization of up to 4 threads brings almost perfect speed up and flattens sharply afterwards to a maximal speed up of 4.18 (table 4).

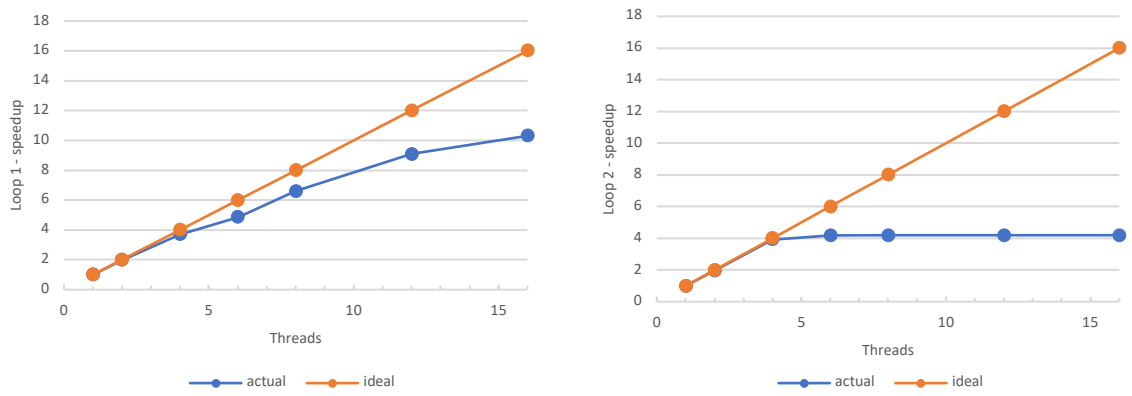

Figure 5: Comparison of the actual and ideal speedup for loop 1 and loop 2.

Table 4: Comparison of the actual and ideal speedup for loop 1 and loop 2.

	Loop 1			Loop 2		
Threads	actual	ideal	ratio	actual	ideal	ratio
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.99	2.00	0.99	1.97	2.00	0.98
4	3.71	4.00	0.93	3.93	4.00	0.98
6	4.85	6.00	0.81	4.18	6.00	0.70
8	6.61	8.00	0.83	4.18	8.00	0.52
12	9.09	12.00	0.76	4.18	12.00	0.35
16	10.30	16.00	0.64	4.17	16.00	0.26

Conclusion

In this report we showed the loop 1 and loop 2 are very different in their structure as can be seen in figure 1. Loop 1 has the structure of gradually decreasing workload per iteration. The schedule type for loop 1 is rather unimportant since the runtimes for “static,n”, “dynamic,n” and “guided,n” were very similar. Nevertheless, it makes sense that the best performing schedule option was “guided,8” where the work is scheduled dynamically while the block size decreases with every work package given to a thread. Furthermore, loop 1 scales good and can take advantage of additional computing power, due to the mentioned workload structure.
Loop 2 with its heavy workload spikes is every case different from loop 1. While “guided,n” performs best on loop 1, it performs very bad on loop 2. This is due to its workload distribution it takes advantage of steadily decreasing workloads and since the loop has high workloads in the beginning of the loop and the guided,n scheduler cannot take advantage and many cores are idle waiting for one core to finish the task. Hence the best performing schedule option for loop 2 is “dynamic,16”, were Loop 2 is just scalable to 4 threads, where it has almost ideal speedup. However, adding more than 4 threads for computation of loop 2 is a waste of recourses since the speedup flattens at 4.18 and never surpasses that value.