C1424548

CM3103: High Performance Computing

Professor David Walker

OpenMP Coursework

November 28, 2017

Contents

1 My Environment

I compiled and ran all code on my own PC. I compiled the code using **MinGW** running on **64-bit Windows 10**, and ran the resulting executables in the same environment.

I used a **4-core intel i5-6600K CPU** with each core clocked at **3.5GHz**, and no hyperthreading or similar technologies. This processor can perform **4.67 GFLOPS per core**, meaning 18.61 GFLOPS total with full utilisation.

Info taken from the following addresses (27th Nov 2017):
https://asteroidsathome.net/boinc/cpu_list.php
https://ark.intel.com/products/88191/Intel-Core-i5-6600K-Processor-6M-Cache-up-to-3_90-GHz

2 Methodology

2.1 Process

I compiled two separate executables from my parallelised code, one using static scheduling, and one using dynamic scheduling. The programs take thread count and chunk size as command-line arguments, and output the processing time in seconds, using the supplied timing mechanism.

I parallelised both blur value calculation, and overwriting the original values, but left the output loop sequential as it relies on writing data in a specific order.

Next, I wrote a simple python script to run the executables with different parameters and output the results into .csv files. I ran each configuration 10 times and took the mean values to reduce noise for external factors. All of my timings were measured in seconds, and have been displayed as such on the graphs.

Finally, I loaded the output data into MATLAB to generate bar graphs. I created two graphs from each dataset, to illustrate the impacts of both variables (thread count, and chunk size).

2.2 Test Data

I tested 16 different thread counts (1 to 16). I chose this range because 16 is 4-times my number of cores, as prescribed in the coursework brief.

For chunk size, I used two different sets of values, one linearly-spaced, and one logarithmically-spaced. Each set contained 27 values from 1 to 222988. I picked 27 values because it yielded a nice logarithmic series with no repeated numbers in the low-range. I chose the 1 to 222988 range because each loop over the whole image requires a total of $521 \times 428 = 222988$ iterations, meaning chunk-size bigger than 222988 is redundant, as it would be reduced to 222988 anyway. I generated the logarithmic data to get more resolution in the lower chunk sizes, as the first 2-3 values in the linear data always cover a huge time range, while the rest of the data is mostly flatter in comparison.

3 Results

3.1 Sequential Run-Time

The first thing I did was time the supplied sequential code, measuring $0.047s \pm 0.003s$ over 10 runs. This value is shown on all the graphs I have produced, as a benchmark to compare the parallel timings with.

Having a task with such a low run-time makes it easy to see the comparative overheads of the two scheduling methods, which could be drowned out by longer-running tasks.

3.2 Static Scheduling

3.2.1 Grouped by Thread Count

The following bar graphs show, for every tested thread count, how execution time (in seconds) varies with chunk size when using static scheduling. For each thread count, there were 27 chunk size values tested, shown as 27 bars at each thread count value on the x-axis. Each group of bars is displayed in ascending order from 1 to 222988. The red line on each graph represents the time taken for the original sequential code to run.

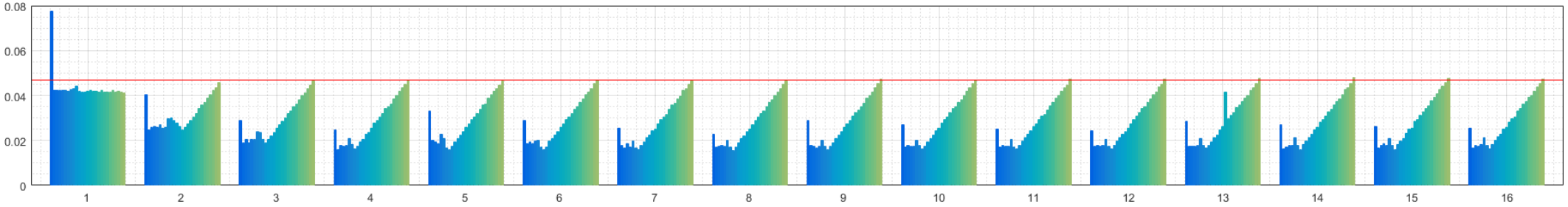

Figure 1: *Chunk sizes linearly spaced*

Figure 2: *Chunk sizes logarithmically spaced*

3.2.2 Grouped by Chunk Size

The following bar graphs show, for every tested chunk size, how execution time (in seconds) varies with thread count when using static scheduling. For each chunk size, there were 16 thread count values tested, shown as 16 bars at each chunk size value on the x-axis. Each group of bars is displayed in ascending order from 1 to 16. The red line on each graph represents the time taken for the original sequential code to run.

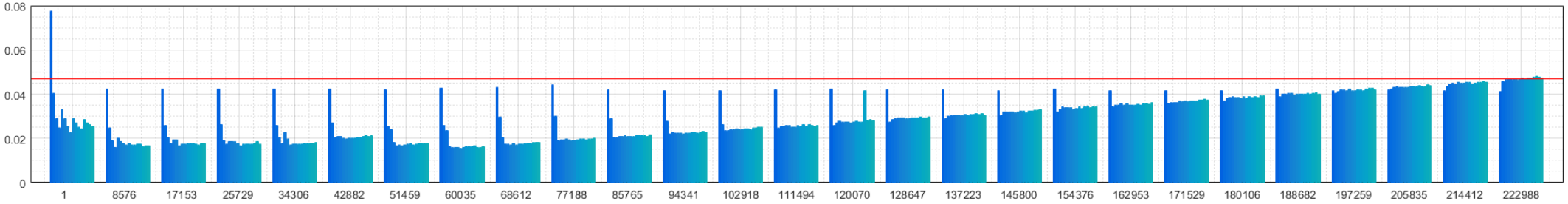

Figure 3: *Chunk sizes linearly spaced*

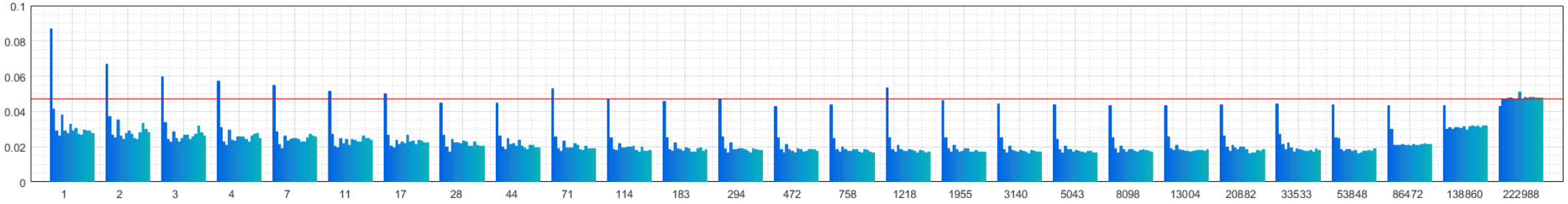

Figure 4: *Chunk sizes logarithmically spaced*

3.2.3 Analysis

The graphs above show that when using static scheduling, there is a clear relationship between chunk size and execution time. This can be seen clearly in Figures 3 and 4. Thread count on the other hand seems to have little baring on the achievable execution times, but instead leads to seemingly random variation in the optimal chunk size, as illustrated in Figure 2. This is unintuitive at first, as an increasing thread count should in general demand smaller chunks to evenly spread the workload. In reality, it seems like the relationship is more dependant on the specific numbers. More samples would be needed to conclusively determine a pattern.

Another interesting feature is that at all thread counts except 1, a chunk size of 222988 (the highest value) performs almost identically to the sequential code. When the chunk size 222988, every single iteration is contained in a single chunk meaning the work is all done by the first thread. The fact that it keeps pace with sequential execution shows that almost all the overhead in static scheduling comes from the number of chunks. At smaller chunk sizes, there is a big slowdown for all thread counts.

The slowdown at larger chunk sizes can be explained by the general rule that the less chunks you have, the less likely it is that you can split them evenly between a given number of threads. The optimal chunk size is determined by a balance of this effect with the overheads from static scheduling at small chunk sizes.

The biggest anomaly in the data the behaviour when the thread count is 1. At this thread count, the program still ran significantly faster than the sequential code for a lot of chunk sizes, even including 222988, where performance matched the sequential at every other thread count. This is shown clearly in Figure 1 In fact, after the chunks get big enough to alleviate the overhead, the run-time remained at a fairly constant level slightly below the sequential time. I do not have any explanation for this other than behind-the-scenes optimisations in OpenMP, or some difference in interaction with the thread management system in Windows.

3.3 Dynamic Scheduling

3.3.1 Grouped by Thread Count

The following bar graphs show, for every tested thread count, how execution time (in seconds) varies with chunk size when using dynamic scheduling. For each thread count, there were 27 chunk size values tested, shown as 27 bars at each thread count value on the x-axis. Each group of bars is displayed in ascending order from 1 to 222988. The red line on each graph represents the time taken for the original sequential code to run.

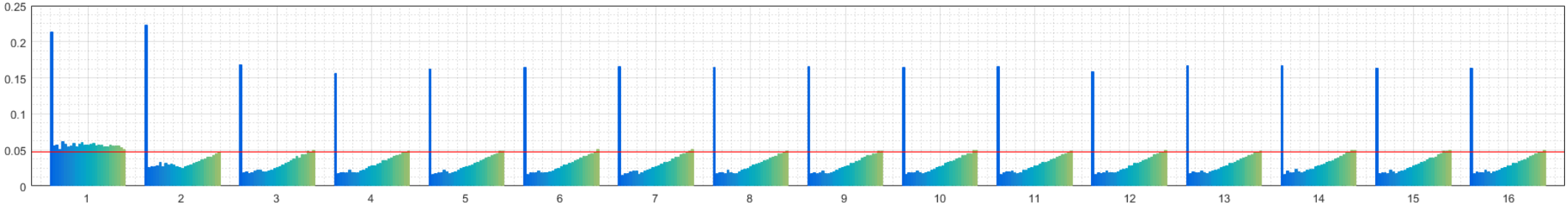

Figure 5: *Chunk sizes linearly spaced*

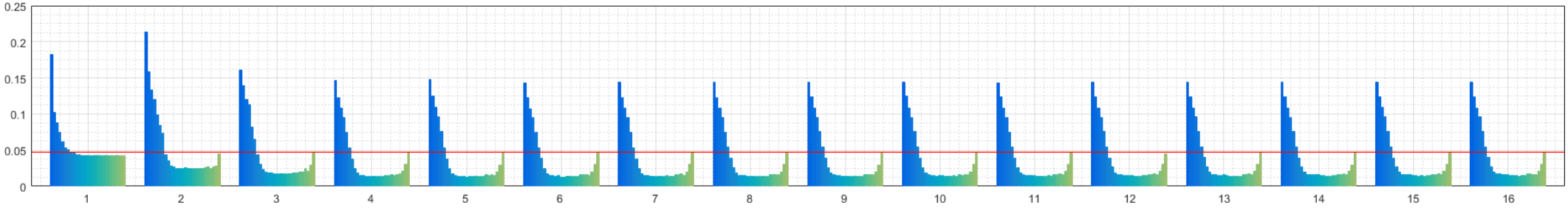

Figure 6: *Chunk sizes logarithmically spaced*

3.3.2 Grouped by Chunk Size

The following bar graphs show, for every tested chunk size, how execution time (in seconds) varies with thread count when using dynamic scheduling. For each chunk size, there were 16 thread count values tested, shown as 16 bars at each chunk size value on the x-axis. Each group of bars is displayed in ascending order from 1 to 16. The red line on each graph represents the time taken for the original sequential code to run.

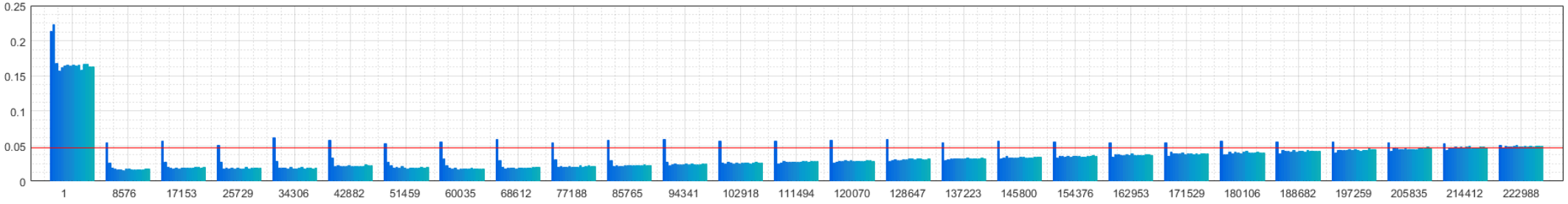

Figure 7: *Chunk sizes linearly spaced*

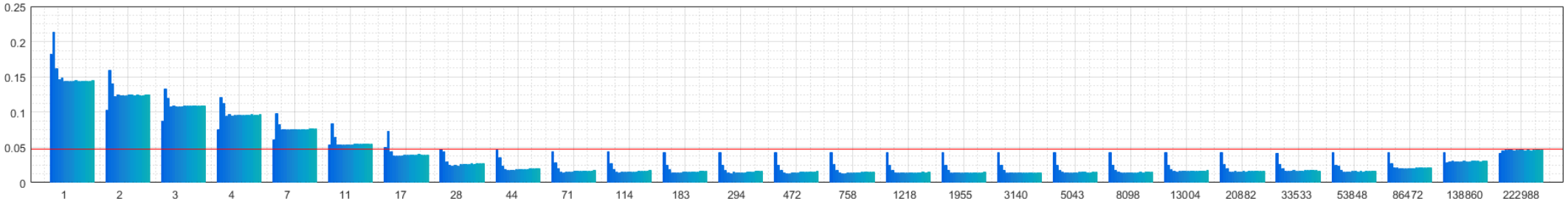

Figure 8: *Chunk sizes logarithmically spaced*

3.3.3 Analysis

For dynamic scheduling, the relationship between chunk size and execution time is very similar to that of static scheduling. At the lowest chunk sizes, the execution time is high, probably due to overhead in assigning new chunks to threads. Then, at higher chunk sizes, the execution time increases again because of the innate inefficiency in trying to balance work that is split into fewer units.

Like with static scheduling, the optimal chunk size lies at a balance between the overheads at either end of the spectrum. In dynamic scheduling, the optimum is at a much lower value. The maxima at the low-extremes of chunk size are over twice the run time as those in static scheduling, however they are also drop off much sooner. This is illustrated best by comparing the graphs showing the data grouped by thread count (Figures 1 and 5, and 2 and 6).

Another difference from static scheduling is that the position of the optimal chunk size seems much more consistent, always appearing around the middle of the logarithmic range ($\approx \sqrt{total\ iterations}$), and immediately after the first bar in the linear range. It is worth noting that when comparing Figures 2 and 6, while the dynamic readings appear more linear in the middle section of the dynamic data, it is only because the y-axis scale is much smaller.

Just like in static scheduling, at all thread counts except 1, chunk size of 222988 performed almost identically to the sequential code. Once again, in the logarithmic dataset, at a thread count 1, dynamic scheduling somehow outperforms the sequential code for higher chunk sizes despite only utilising a single thread. This however is not reflected in the linearly spaced data. This difference between the two datasets suggests inconsistency in performance at this thread count. As the logarithmic data matches the static scheduling results, I think it is more likely to represent the typical pattern.

In both the linear and logarithmic data, the worst performance came at a chunk size of 1, and a thread count of 2. This is different to static scheduling where the worst performance was at chunk size = 1, thread count = 1. This is probably because of overheads from assigning work between multiple threads on-the-fly. Something that is totally avoided in static scheduling.

4 Conclusions

First of all, as expected, there were large performance gains achieved by parallelising the task, for both scheduling mode. If you only consider performance, static and dynamic scheduling will yield very similar results at their respective optimal configurations.

The optimal chunk size for Dynamic scheduling is more consistent between different thread counts, staying at around $\sqrt{total\ iterations}$. The optimal chunk size for static scheduling is less predictable, though is usually still in roughly the same area.

Dynamic scheduling is can yield very bad performance, with maximum execution-times over twice as high as those from static scheduling. Such excessive overheads should be avoidable by choosing reasonable parameters, however with a good configuration you can still only match the performance of static scheduling. Because of this, I believe that static scheduling is the best choice for most situations.

Dynamic scheduling has the innate benefit of being equally efficient when the work required for each chunk is inconsistent. When this is the case dynamic scheduling is the clear choice. Dynamic scheduling could also be a good choice if your processing cores have uneven expected performance, either because of hardware differences, or because of other workloads.

The biggest surprise to me is that there seems to be no correlation between thread-count and achievable performance past 2 threads. I expect that this is because the task demanded a small fraction of my total processing power, and the application was run in an environment with lots of other processes. A task that demanded more CPU resources for each iteration might have seen a bigger performance increase from spreading the load over multiple cores.

I also did not expect chunk size to have such a large impact on performance. Again, I expect that a task that took more processing time per-iteration would suffer proportionally smaller overheads from task scheduling. Regardless of this, it is clear that chunk size should always be taken into consideration, and very low chunk sizes should be avoided entirely, along with obviously excessive chunk sizes.