EDINBURGH NAPIER UNIVERSITY

SET10108 – Concurrent and Parallel Systems

Coursework 1

40056161

06/11/2015

Contents

Introduction

The aim of this coursework is to parallelise the LINPACK Benchmark to improve performance. Using different concurrency and parallel CPU techniques in an attempt to optimise the algorithm.

LINKPACK Benchmark

LINPACK Benchmarks are a measure of a system's computing power (Wikipedia, 2015). It measures how quickly a computer solves a matrix of linear equations using floating point arithmetic. Since 1993 it has been used to rank the TOP500 supercomputers as it can be run by any machine, regardless of the architecture (TOP500, 2015). For this coursework a matrix size of 1000 will be used.

Methodology

Approach

The goal of the coursework is to improve performance. The serial algorithm will first be measured to serve as a benchmark for all future tests. Analysing the gathered data should hopefully identify any performance bottlenecks. Once a bottleneck is identified it will be worked on using various parallelisation techniques. This process will be iterated upon with the goal to achieve a significantly improved algorithm.

The first approach to improve the algorithm will utilise the OpenMP construct `parallel for`. OpenMP is a library that supports shared memory concurrency. It allows the developer to parallelise their code without significantly having to rewrite it (Yliluoma, 2008). OpenMP uses pre-processor compiler arguments to allow existing code to run on multiple threads.

The next approach will involve tweaking the parallel for to use different OpenMP scheduling methods. Four main scheduling types will be explored: static, dynamic, guided and runtime. Scheduling is the division of work over threads and the different approaches decide in what manner the processes are assigned to their threads.

Gathering Results

Results will be gathered by using the Concurrency Visualizer and measuring times. To ensure consistency of results, each approach will be run 100 times and an average will be taken of those iterations. The data will be outputted to a csv file and Microsoft Excel will be used for data analysis and data visualisation. The purpose of the initial investigation is to determine which part of the algorithm is the most computationally intensive. The timings used start and end before any file output. Meaning than any I/O processes will not be included in the measurements provided.

In regards to the Concurrency Visualizer, when recording CPU usage each variation of the algorithm will be run 5 times. This ensures that there will be no anomalies in the figures collected. All raw data will be accessible in the appendices.

Hardware

The machine used for the experiments will be a 2011 Dell XPS15.

- CPU – Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz, 1995 Mhz, 4 Core(s), 8 Logical Processor(s)
- RAM – 6GB
- GPU – NVIDIA GeForce GT 540M
- OS – Windows 8.1 64-bit

Hypothesis

Amdahl's law states that there will always be a serial part of the algorithm that hinders performance, unless the algorithm is 100% parallelizable (Michalove, 1992). Due to the nature of the supplied code, it would be difficult to make it completely parallel as the algorithm would need to be rigorously analysed to determine what percentage is parallelisable.

$$T_{parallel} = \frac{(B \times T_{serial})}{N} + 0.1 \times T_{serial}$$

$B = \%\ of\ algorithm\ that\ is\ parallelisable$

$N = number\ of\ processors$

As the machine used for testing has 8 logical processors the maximum speedup should theoretically be 8 times as fast as the serial algorithm (under the assumption that the method is 100% parallelisable). It is expected that the serial computation will be slightly lower than an 800% performance increase due to physical external factors such a CPU context switching and I/O operations.

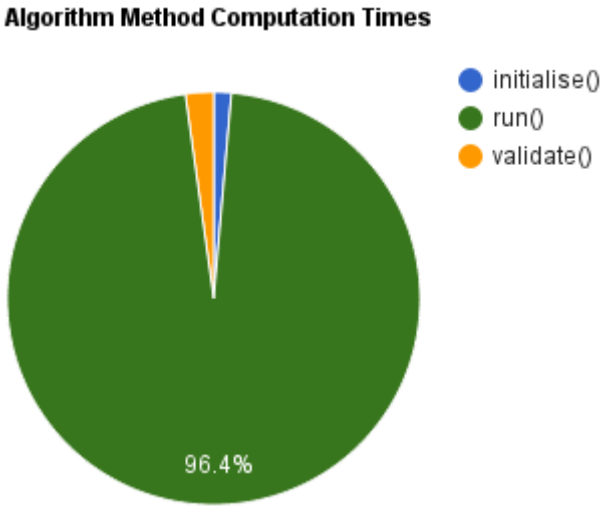
As for scheduling, it is suspected that scheduling a parallel for in OpenMP should have some general improvement opposed to no scheduling at all. It will be interesting to examine the difference in the 4 methods of scheduling and how they affect performance and CPU utilisation.

Results

Serial Solution

Using the provided code from Moodle, the serial solution will be used as a comparison for the implemented parallel solution. The initial task carried out was identifying the immediate bottlenecks. This was done by timing the algorithms methods called in main and then recording them appropriately.

Initial measuring of methods called in Main			
Name of Method	`initialise()`	`run()`	`validate()`
Average Time milliseconds (100 iterations)	58.7	4110.12	93.58

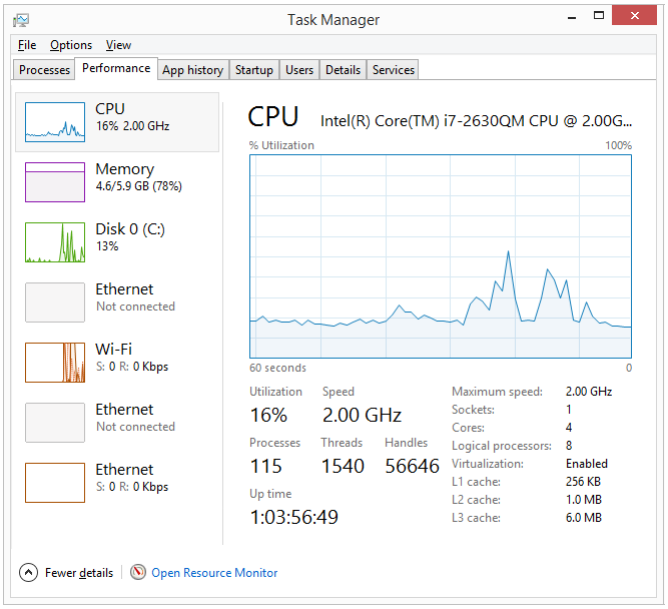


It is apparent from the above table and graph that the `run()` method takes the longest to compute (96.4% of processing time). On further inspection it was discovered that this method called two other methods and therefore was not parallelisable. The next step was to identify what is going on inside of the two methods method and pinpoint the bottleneck. The method `dgefa()` 'performs Gaussian elimination with partial pivoting' whilst `dgesl()` 'Solves the system a * x = b'.

Name of Method	`dgefa()`	`dgesl()`
Average Time milliseconds (100 iterations)	4310.6	12.69

From these preliminary results it shows that the method that performs the Gaussian elimination is the most performance intensive method. Since `dgefa()` is the main bottleneck it will be focussed on in the attempt to improve overall algorithm performance.

When analysing the serial algorithm using Window's built in task manager the algorithm (and all background tasks) appear to only utilise 15% to 17% of the CPU. Running Visual Studios Concurrency Visualizer on the serial implementation shows that the existing algorithm only utilises one thread.



The overall time for the entire serial algorithm averaged out at **4461.13** milliseconds. Taking this into consideration, it is an obvious decision to focus on the `dgefa()` method as it accounts for **96.63%** of the total computation time.

OpenMP Parallel For

Parallel for is a directive that allows loops to be run on multiple processors. This is implemented by adding a declaration before the existing loop. The programmer must be careful when implementing parallel for as any variables that are modified within the loop must be declared as private (The Supercomputing Blog, 2009).

```
176                 // Row elimination with column indexing
177 #pragma omp parallel for num_threads(num_threads) private(t, col_j)
178                 for (int j = kp1; j < n; ++j)
179                 {
180                     // Set pointer for col_j to relevant column in a
181                     col_j = &a[j][0];
182 
183                     t = col_j[l];
184                     if (l != k)
185                     {
186                         col_j[l] = col_j[k];
187                         col_j[k] = t;
188                     }
189                     daxpy(n - kp1, t, col_k, kp1, 1, col_j, kp1, 1);
190                 }
```

In the above code example, parallel for is added in line 177. The variables `t and col_j` have been declared as private as they are being modified inside the loop. This means that each thread will have their own independent copies of these variables. The parallel for uses `num_threads` to determine how many threads to use. For this algorithm `parallel for` uses all possible threads by setting the `num_threads` variable to the result of `thread::hardware_concurrency()`.

Implementing parallel for made a significant improvement in regards to the speed of the application with an average overall time of **580.18** milliseconds.

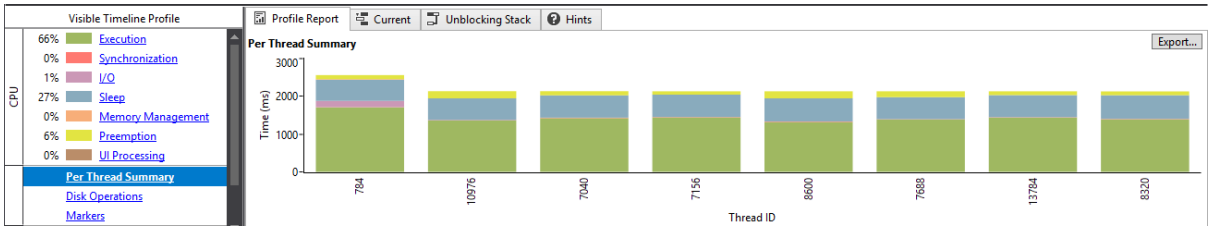
$$\left(\frac{Serial\ Time}{Parallel\ Time}\right) * 100 = \ speedup\ \%$$

$$\left(\frac{4461.13}{580.18}\right) * 100 = \ 768.92\%$$

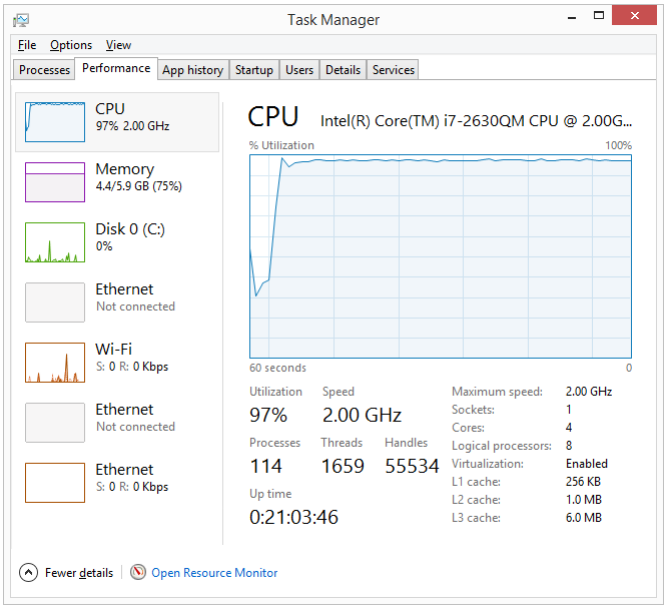
Theoretically the algorithm should be improved eightfold as it is utilising 8 threads instead of 1. It is interesting to find that the speedup behaves as expected (nearly 800%). It is worth noting that these calculation are based on the overall time, not just the improved method. This is a vast improvement in terms of performance, after this initial test the previous timings were run to compare the improvement of the methods within `run()`.

Name of Method	`dgefa()`	`dgesl()`
Average Time Serial	4310.6	12.69
Average Time Parallel For	482.24	7.03
Speedup	925.39%	55.08%

It is interesting to note that even though the code change was made in the Gaussian elimination method, the solver method is improved by 55%. When analysed with the Concurrency Visualizer it shows 8 threads being utilised by the algorithm which explains the significant performance increase.



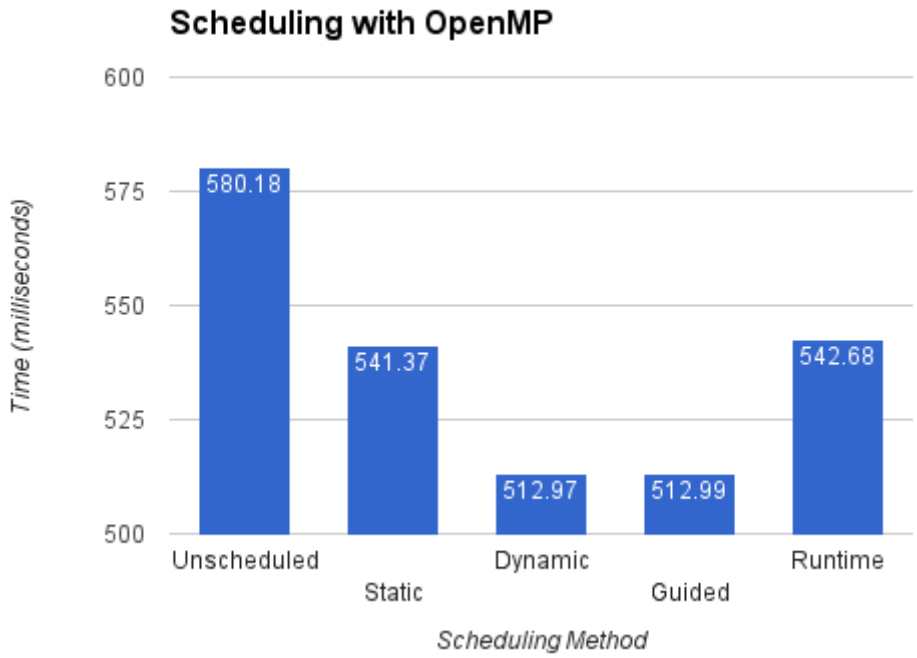
Since multiple threads are being utilised the CPU is working a lot more than the serial implementation. The task manager shows up to 98% of the CPU being utilised by the algorithm when `parallel` for is used. This reinforces the fact that OpenMP is an effective method in regards to parallelising existing code.



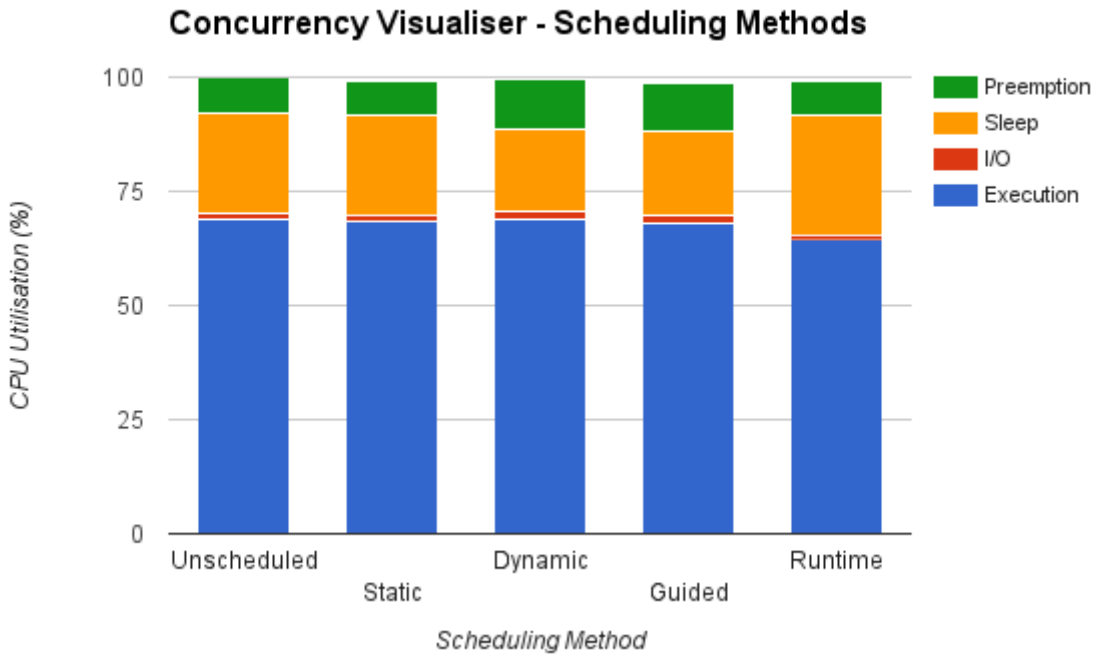
OpenMP Scheduling

The parallel for loop will be modified to test different methods of scheduling. Each iteration will be recorded using the concurrency visualizer and the timings for the overall algorithm will be recorded. The purpose of these test is to determine the optimal scheduling method. Implementing these different method is simple as it only involved adding an extra declaration to the `parallel for` statement.

- Static scheduling divides the for loop into ‘chunks’. By default the ‘chunksize’ is the loop count divided by the number of threads. Due to time constraints the chunksize was not altered during the experiments.
- Dynamic scheduling assigns threads to the loop whilst it is executing. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue (Green, 2014).
- Guided scheduling is similar to dynamic scheduling except the chunk size starts off large and decreases to better handle load imbalance between iterations.
- Runtime Scheduling determined when the algorithm begins to run.



The gathered results show that all methods have at least a 7% increase over the unscheduled `parallel for` implementation. It is interesting to see that Static and Runtime have similar performance, and Dynamic and Guarded have almost exactly the same computation time.



The two highest performing scheduling methods (Dynamic and Guided) have an increased amount of CPU space assigned to Pre-emption. The two lower-performing methods (Static and Runtime)

appear to take more time in Sleep than in Pre-emption which could explain why these methods are not as effective in terms of reducing computation time.

Conclusion

The aim of the coursework was to improve the performance of an existing serial implementation of the LINPACK 1000 algorithm. This has been achieved by implementing features of the OpenMP library.

After extensive analysis of the existing serial algorithm a major bottleneck was removed by implementing `parallel for`, resulting in an initial 768.92% speedup. On further analysis it showed that OpenMP fully utilised the hardware it was running on by creating multiple threads and using 97% of the CPU (as opposed to 16% for the serial implementation).

Further investigation was taken to explore the different scheduling methods that OpenMP included. Four scheduling methods were tested resulting in a minimum improvement of 7% (Runtime scheduling) and a maximum improvement of 11% (Dynamic scheduling). Each method was analysed with the Concurrency Visualizer and showed that the higher performing methods allocated more time to Pre-emption.

There is the opportunity to test many other different methods to improve the algorithm. Techniques such as C++11 threads and MPI could be explored to investigate how efficient they are in parallelising the code. Further work could also be taken that focuses on other parts of the algorithm such as the initialisation and validation methods.

Bibliography

Stroustrup, B., 2015. *C++11 - the new ISO C++ standard.* [Online]
Available at: http://www.stroustrup.com/C++11FAQ.html
[Accessed 3 Novemeber 2015].

The Supercomputing Blog, 2009. *Tutorial – Parallel For Loops with OpenMP.* [Online]
Available at: http://supercomputingblog.com/openmp/tutorial-parallel-for-loops-with-openmp/
[Accessed 04 November 2015].

TOP500, 2015. *The TOP500 Project | About.* [Online]
Available at: http://www.top500.org/project/
[Accessed 19 October 2015].

Wikipedia, 2015. *LINPACK benchmarks.* [Online]
Available at: https://en.wikipedia.org/wiki/LINPACK_benchmarks#LINPACK_1000
[Accessed 19 October 2015].

Yliluoma, J., 2008. *Guide into OpenMP: Easy multithreading programming for C++.* [Online]
Available at: http://bisqwit.iki.fi/story/howto/openmp/
[Accessed 03 November 2015].

Appendices

Appendices are included within the accompanying folder.

- Serial.cpp – Original Serial implementation of the algorithm with timings.
- Parallel.cpp – Parallelised version with OpenMP. Options for different scheduling methods and timings.
- Output File – Collection of all data gathered in each experiment

All progress can be viewed Github at: https://github.com/cgathergood/Concurrent-and-Parallel-Coursework.git