University of Edinburgh

MSc High Performance Computing with Data Science

Threaded Programming

Coursework Part 1

Exam No:

B139964

October 26, 2018

Contents

List of Figures

1 Introduction

In 1965, Gordon Moore had predicted that the amount of transistors could be fitted in a chip would be double every two years. This would allow for faster processors for the same chip area. Consequently, programs could run faster reducing execution times. This prediction was proved to be accurate for several decades leading to a vast amount of innovations. However, in the recent years, processor speeds hit a barrier due to challenges involving in further scaling down the dimensions of the transistors inside a chip.[4] At the same time applications require faster execution times. The idea of creating a single-core processor that can perform faster in order to speed up applications is not very realistic anymore and it does not serve practitioners very well.

Consequently, it is becoming increasingly popular for these people to shift from sequential programs into parallel ones. The use of multiple core machines and programming models offers an alternative approach into accelerating applications by essentially parallelise them. One of the most popular models is called threaded programming as it is most often used on shared memory parallel computers.[1] OpenMP API is one of the most popular and widely used APIs of threaded programming that provides programmers with an interface for developing parallel applications using the notion of threads and tasks.

In order to construct parallelism, OpenMP uses parallel regions as its basic parallel construct where once the master thread encounters one, a team of threads is created using the fork/join model.[2] Essentially, parallelism exists only inside the parallel region. Work inside the parallel region can be divided between threads using directives. Moreover, the programmer can choose the scope of the loops inside a parallel region or how each loop iteration will be executed by which thread, by selecting the appropriate clauses.[3]

1.1 Aims and Objectives

In this piece of work we are given a sequential piece of code. The part of the code we are interested in contains two loops. Each loop is timed for how long it takes to run 1000 repetitions. Additionally, after each loop has run, a verification test is included. This work aims to achieve the followings:

- Parallelise the two loops of the given sequential code using OpenMP directives, without necessarily understand what the code tries to solve. Use the OpenMP scheduling options to specify how threads will execute the work.
- Carry out experiments using the different OpenMP scheduling kinds and observe the impact of each kind for a constant number of threads available. Moreover, investigate how the chunk-size of each kind affects the performance of the code and select the best option for each loop.
- Observe how the performance of each loop changes, after the best scheduling option has been set, by varying the number of threads available and recording the achieved speed-up.
- Describe and explain the results obtained from each experiment.

1.2 Structure of the Report

The following sections aim to give a description of the process followed to achieve the aims mentioned in Section 1.1. A high-level explanation of how the provided code works and how to parallelise it is given in Section 2. In Section 3 the experiments performed are described and the results are presented along with an explanation. Finally, in Section 4 the conclusions drawn from the experiments along with possible future work are presented.

2 Approach

2.1 Sequential Code

The main computational part of the sequential code provided we are interested in, is consisted out of two routines called `loop1` and `loop2`. Inside each routine, nested loops exist performing operations on data.

2.1.1 Routine loop1

Before running the first routine, `loop1`, the arrays required to perform the calculations are initialised. The routine uses a 2-dimensional input array `b[][]` and writes the results of the operations performed on input data to a 2-dimensional output array `a[][]`.

Every time `loop1` is run, a nested loop of 2 for-loops is executed iterating through the contents of `b[][]`. The function evaluated on those contents is accumulated in `a[][]`. It is worth noticing that the outer loop iterates from $i = [0...N)$ while the inner loop from $j = (i...N)$ therefore the workload for every iteration of the outer loop decreases as $i \rightarrow N$ where $N = 729$ for the purposes of this work.

A total of 1000 runs of `loop1` are performed and the total execution time is recorded. After the execution of all runs is finished, a validation step exists where the contents of `a[][]` are accumulated. The total sum is printed in the form of a message. Followed the validation step, a message is printed containing information about the total runs `loop1` was executed, the total time taken for all the runs.

2.1.2 Routine loop2

The procedure that involves the execution of the second routine, `loop2`, is quite similar to the one followed in executing `loop1`. In other words, it is consisted of an initialisation stage, a timed execution stage of 1000 runs and a validation stage.

The initialisation stage involves initialising the 2-dimensional input array `b[][]` and the 1-dimensional output array `c[]`. Also, another 1-dimensional array `jmax[]` is initialised to take values of 1 or N based on the evaluation of an expression. The expression evaluated is of the form

$$expr = i \texttt{ mod } (3 * (\frac{i}{30}) + 1) \tag{1}$$

where $i = [0, N)$ and basically checks if i is a multiple of the result on the right of `mod`. If `expr` in Equation 1 is 0 then `jmax[i]` $= N$, otherwise is set to 1. Consequently, as $i \rightarrow N$, the number of `jmax[i]` set to N decreases. In other words, the elements of the `jmax[]` array set to N are not evenly distributed throughout the entire space of the array, with most of them lie at the beginning of the array, while after the number diminishes significantly.

Every time `loop2` runs, a nested loop of 3 for-loops is executed iterating through the contents of `b[][]`. The function evaluated on those contents is accumulated in `c[]`. The outer loop iterates from $i = [0...N)$. Observing the two inner loops, one can observe that the first iterates from $j = [0...$`jmax[i]`$)$, where `jmax[i]` can be either 1 or N depending on i. Moreover, the second inner loop, iterates from $k = [0, j)$. Consequently, one can deduce that whenever i is such that `jmax[i]` $= 1$, then the two inner loops do not perform any work. On the other hand, the work performed by the inner loops is increasing as $j \rightarrow N$ whenever i is such that `jmax[i]` $= N$.

2.2 Parallelisation Approach

In order to parallelise the provided code, OpenMP directives are used. Only the outer loops in the routines `loop1` and `loop2` are parallelised using the directive `#pragma omp parallel for`. Additional information inside the parallel region directive can be specified through clauses.

It is important to specify the scope of all the variables inside the parallel region to maintain correctness of the program and avoid any race conditions. As neither `loop1` or `loop2` contain any dependencies, all the variables are set into shared using the `shared()` clause. Only the ones that are used as indexes in the for-loops are set to private using the `private()` clause. For `loop1` the private variables are i, j and for `loop2` are i, j, k

As both `loop1` and `loop2` are now parallelised, the `schedule()` clause option will be added to the directive of each routine. The effects of each option to the performance of the parallel code will be investigated and the best option for each routine will be selected. This is described in more detail in Section 3.

3 Experiments

The main aim of this work is to investigate how each scheduling option affects the performance of the particular code. Therefore it is desirable to measure the achieved speed up for in parallelising the code compared to the sequential form for each of the loop routines. In order to do so, first the best scheduling option for each routine needs to be established.

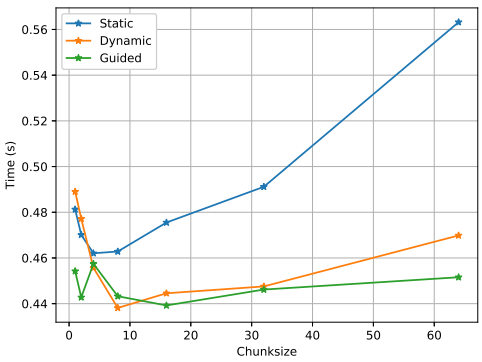
3.1 Finding best scheduling option for each loop routine

3.1.1 Setting up the experiment

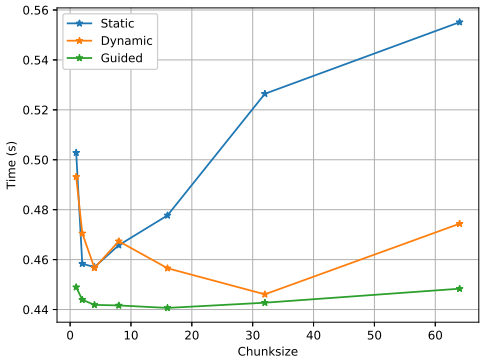
In order to be able to find the best scheduling option for each loop routine, one has to test all available options and check which one performs better by measuring the execution time. The strategy followed was to set the number of threads constant to 4 throughout the duration of the experiment. The scheduling option was varied at each experiment. The options available were the kind of scheduling, which was consisted of `STATIC, DYNAMIC, GUIDED` and the chunksize that could take values of `1, 2, 4, 8, 16, 32, 64`. Therefore each combination of `(kind, chunksize)` was tested and the execution time of each option was recorded for a constant number of threads set to `OMP_NUM_THREADS=4`.

As changing the scheduling option every time would require us to modify the code and re-compile, this wouldn't also allow us to run the test multiple times. In order to ease the process of recording the execution time for each scheduling option, a bash script was created. OpenMP provides a scheduling option called `RUNTIME` which if used, once the code is compiled, allows us to change the scheduling option from the runtime using `OMP_SCHEDULE` enviroment variable. This doesn't require us to re-compile the code. For example if on the runtime `OMP_SCHEDULE=STATIC,4`, then that would set the scheduling kind for each routine to `STATIC` with chunksize of 4.

Consequently, the scheduling clause for each loop routine inside the directive is set to `RUNTIME`. Then the bash script is responsible to go through all the available scheduling options mentioned above, by changing `OMP_SCHEDULE`, each time recording the execution time and validation result for each loop routine. Validation results are compared with the ones from the sequential version to test that the code returns the same values and hence ensure correctness of the calculations. Execution times are used to plot graphs of the execution time versus the chunksize for each kind of scheduling, using python.



(a) Results from running the program once for each schedule option



(b) Average results from running the program 10 times for each schedule option

Figure 1: Execution time for 1000 repetitions of loop1 routine versus the chunksize for the STATIC,n DYNAMIC,n and GUIDED,n schedules when running the program once and 10 times.

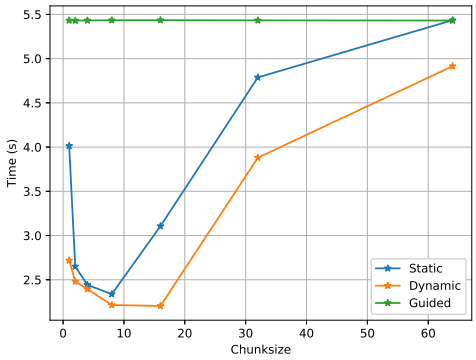
3.1.2 Results

After running the experiment once for each schedule option, the results for `loop1` routine shown in Figure 1a were obtained indicating that the best scheduling option is `DYNAMIC,8` but it can be observed that `GUIDED,16` is very close too. However each time the experiment was repeated, there were variations in each execution time hence the best scheduling option was changing, making it unclear which one was the best. Therefore, the same experiment was performed but this time running each scheduling option 10 times recording the execution time of each. Then the average time was calculated and plotted as shown in Figure 1b. The figure suggests that the best scheduling option for `loop1` is `GUIDED,16`, and it is the one selected for future tests.

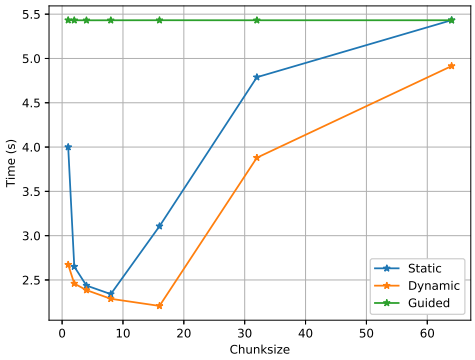
As mentioned in Section 2.1.1, as the outer loop of `loop1` routine iterates from $i = [0...N)$, the inner loop iterates from $j = (i...N)$. Therefore, the work that needs to be done at every iteration decreases as $i \to N$, yielding to unbalanced code. Consequently, as the iteration chunks for `STATIC` scheduling are given in the same order to each thread, in a round-robin order by thread number, it is expected that as each chunksize increases, each iteration chunk will become more unbalanced. This effect will lead to slower execution times as more overheads are introduced, which is confirmed by observing Figure 1.

For `DYNAMIC` schedules, iteration chunks are issued according to the needs of the threads.[5] In other words, after each thread finishes with its current iteration chunk, it is being handed another one. A same approach is followed for `GUIDED` scheduling with the difference that each thread gets a large chunk at the beginning and smaller subsequent chunks afterwards.[5] Therefore, these two options are expected to perform better for unbalance code compared to `STATIC` which is indeed the case as shown in Figure 1.

Observing closer how unbalance the code is, it can be seen that at each iteration of i, j decreases by 1. Therefore, the inner loop has to perform one less iteration every time the outer index increases. Hence the work that inner loop has to execute every time decreases uniformly by one and does not widely vary. For `GUIDED` scheduling means that the way each iteration chunk will be assigned to each thread will be more efficient compared to `DYNAMIC`. As the size of each chunk decreases, then threads that have started with larger size chunks will be then assigned smaller ones compared to the others. `DYNAMIC` scheduling won't be able to schedule execution in this way, but will only be able to assign chunks to threads on a first-come-first-served basis. Concluding, for this kind of problem, `GUIDED` scheduling seems to be more suitable.



(a) Results from running the program once for each schedule option



(b) Average results from running the program 10 times for each schedule option

Figure 2: Execution time for 1000 repetitions of loop2 routine versus the chunksize for the STATIC,n DYNAMIC,n and GUIDED,n schedules when running the program once and 10 times.

While workload in `loop1` routine is not widely unbalanced, this is not the case for `loop2`. As mentioned in Section 2.1.2, as $i \to N$, the amount of work in the inner loops depends on `jmax[i]`. Consequently, most of the work is concentrated in the first 30 iterations of i, while in the rest is sparse. Following this, `GUIDED` scheduling will under-perform no matter the chunksize. This is confirmed by Figure 2.

As the workload is widely unbalance, there are no significant changes from running the experiment for each scheduling option once or 10 times therefore the Figures 2a and 2b are identical. However, from running the experiment just once, it is quite unclear which scheduling option is better as `DYNAMIC,8` and `DYNAMIC,16` are very close. The picture becomes more clear when the experiment is repeated for 10 times as `DYNAMIC,16` seems the option to go with for `loop2`. It is expected that a dynamic scheduling option would work best as this is the most useful option when iterations have widely varying loads.[3] Surprisingly, `STATIC` option also performs quite well as Figure 2.1.2 shows that it follows quite closely the `DYNAMIC` option. While `STATIC` can perform best for load balanced loops, it can also perform well for imbalanced ones but with added overheads.[3]

Commenting on the chunksize, it is obvious from Figures 1 and 2 that the larger it becomes, the worse the performance becomes. This is also the case for smaller chunksizes as each time we are running through the schedule code more overheads are absorbed.[5]

3.2 Measuring speed up for the best scheduling option for each loop routine

3.2.1 Setting up the experiment

After the best scheduling option was selected for each loop routine, as described in Section 3.1, the speed up of the parallelised code using the best scheduling option compared to sequential version for a variable number of threads is desired.

In order to measure the speed up, each loop routine is set to the best scheduling option and the code is compiled. Therefore, scheduling options are set constant throughout the duration of the experiment. Each time the number of available threads is changing using the `OMP_NUM_THREADS` environment variable and the execution time is measured. The number of available threads tested was `1, 2, 4, 6, 8, 16`.

To make the test reproducible and easy to carry out again, a bash script is created. Each time

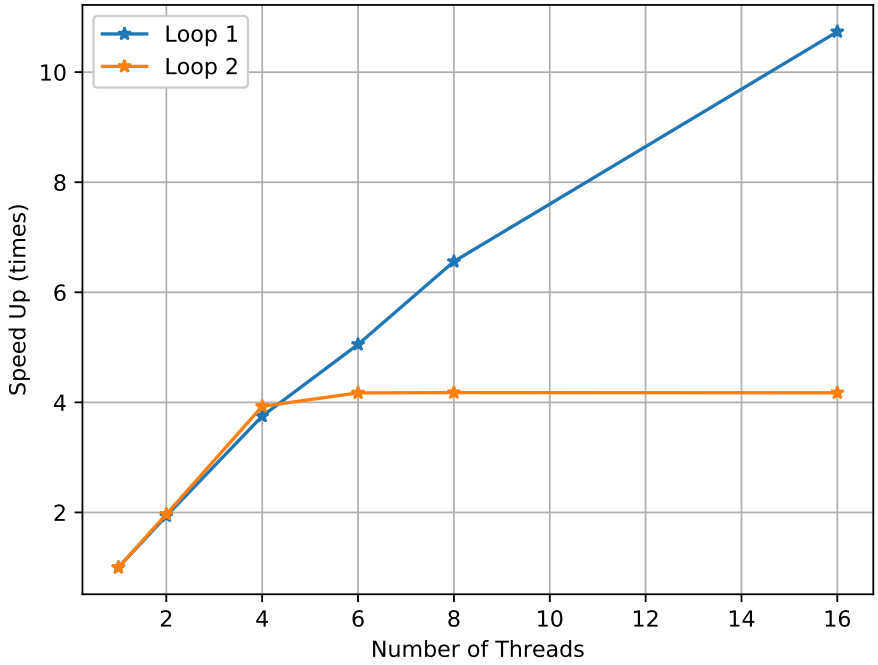


Figure 3: Speed up achieved from running the best scheduling option for each loop routine varying the number of threads.

is responsible to set `OMP_NUM_THREADS` to the next value, run the code and record the execution time and validation results for each option. Validation results are compared with the ones from the sequential version to test that the code returns the same values and hence ensure correctness of the calculations. Execution times are used to plot graphs of the achieved speed up versus the number of threads available for each loop routine, using python.

3.2.2 Results

With the best scheduling options being set to `GUIDED,16` and `DYNAMIC,16` for `loop1` and `loop2` routines respectively the aim is to measure the achieved speed up for every routine by varying the number of threads available. The experiment was performed as described in Section 3.2.1. Each time the program was run for 10 times and the average speed up for each number of threads was recorded in order to make results more robust. The results are shown in Figure 3.

For `loop1` the achieved speed up increases linearly as the number of threads increases. For a number of threads between 1 and 4 there are no significant overheads which suggests that the achieved speed up is directly proportional to the number of threads. However, as the number of threads increases, the overheads increase as well reducing the amount of achieved speed up.

On the other hand, `loop2` achieves a proportional speed up for up to 4 threads but then it gets saturated. Consequently, no additional benefit exists for using more than 4 threads. This is due to the nature of the routine it self, as it contains very unbalanced loads. As most of the work lies at the beginning of the first 30 iterations and the chunksize for this scheduling is set to 16, then 4 threads are enough to carry out the first 64 iterations. Therefore, the rest won't contribute significantly as they will mostly deal with empty loads. However, based on this hypothesis, if we were to change the chunksize for the current scheduling option and set it to smaller, then we should be able to achieve higher speed ups when more threads are available. To test this, the same test was

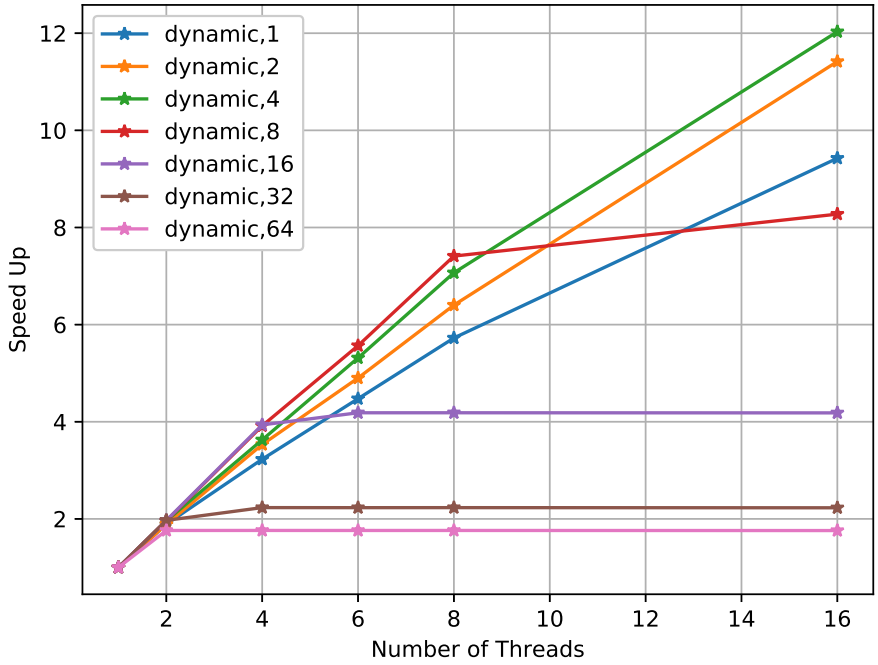


Figure 4: Speed up achieved from running dynamic scheduling option for loop2 routine, varying the number of threads for different chunksizes.

carried out this time varying the chunksize of `loop2` scheduling option and each time testing it for a variable number of threads. The results are plotted in Figure 4 that confirms our hypothesis. As we can observe from the figure, the performance of `loop2` can be improved by x3 further compared to the previous best selection, achieving an overall x12 speed up when executed on 16 threads. We see also that as the chunksize increases, speed up saturates even faster compared to before. When chunksize is set to 32 or 64 the performance is quite comparable but with quite more overheads for the 64 case. As the chunksize is reduced, the saturation effect diminishes and is not visible for sizes less than 4. Consulting the figure above, the best choice that maximises the performance of the `loop2` routine is the `dynamic,4` which is now the best selected scheduling option for it.

4 Conclusion

This work investigated how the scheduling options offer by OpenMP can impact the performance of a parallelised code. Firstly, for not widely varying loads, `GUIDED` option seems the option to go as is less expensive compared to `DYNAMIC`. On the other hand, if the problem is widely unbalanced, `DYNAMIC` option performs better, but also `STATIC` seems to do well but with added overheads.

Consequently, this shows that the selection of scheduling option can significantly affect the performance of the parallelised code and therefore should not be set naively. Moreover, not always more threads mean more speed up. As more threads might be able to offer better speed up, they can also come with large overheads. In addition, it needs to make sure that if large number of threads is required, they will all have something to do. Additionally, the process of tuning the right scheduling option and the number of threads to execute the work is not always straight forward. This is due to the fact that the number of threads might be dependent to the chunksize set by the scheduling option. Consequently, deciding the scheduling option along with the number of threads

might be iterative process instead of a one time straightforward process.

Finally, for future work, it would be nice to see what would happen if we could do the experiments by also parallelising the inner loops and create nested parallelism. This would allow to investigate the impact in performance from parallelising unbalance loads. As we have seen how `STATIC` can perform in unbalanced code, it would be interesting also to see how the other two scheduling options can perform in balanced code and if they can outperform the former.

References

[1] M. Bull. Lecture 1: Concepts. 2018.

[2] M. Bull. Lecture 2: Openmp fundamentals. 2018.

[3] M. Bull. Lecture 4: Work sharing directives. 2018.

[4] S. Kumar. Fundamental Limits to Moore's Law. *ArXiv e-prints*, November 2015.

[5] P. Lindberg. Performance obstacles for threading: How do they affect openmp code? January 2009.

Appendix

Code Listing

All the details of the code along with the bash and python scripts can be found in GitHub at: `https://github.com/cstyl/tp_assignment1`