University of Edinburgh

MSc High Performance Computing with Data Science

Threaded Programming

Coursework Part 2

Exam No:

B139964

November 30, 2018

Contents

List of Figures

1 Introduction

Parallelising an application most often boils down into parallelising loops. The idea of parallelism lies on the fact that multiple resources are available and consequently can be used in order to reduce the execution time of a problem. In order to reap the benefits of the parallelism, the decomposition of the problem into parallel tasks should be done in such a way that the overheads introduced by parallelism are small and the available resources are fully utilised by balancing the workload.

All loop scheduling methods developed aim to minimise execution time. This is achieved by distributing the load as evenly as possible amongst the available processors where at the same time the number of synchronisation operations required is kept to minimum.[1] Each of these methods however have different strengths and weaknesses. For example, a `static` scheduling method, while it can keep the synchronisation operations at minimum by evenly dividing the number of iterations to each processor, in the case where the load in each loop chunk is irregular, load imbalance occurs and the method struggles. The opposite holds for a `dynamic` scheduling method where the allocation of each iteration chunk at each processor is decided on run-time. Therefore, the load is balanced dynamically at the expense of introducing synchronisation overheads. On the other hand, `guided` scheduling method aims to combine the characteristics of these two approaches by dynamically changing the chunk-size of iterations assigned to each processor on run-time. As this method starts with larger chunks of iterations at the beginning of execution, the overheads from synchronisation are reduced as in `static`. As it progresses and the chunk-size is reduced, load balancing occurs resembling to the behaviour of `dynamic` scheduling. However, this method struggles when the first chunks of iterations contain the largest part of the overall work.

A loop scheduling option that combines characteristics of the scheduling options mentioned above is called `affinity` scheduling. In this method, all the processors are evenly assigned their local iteration space, as in `static` scheduling, where they can all work in order to minimise synchronisation overheads. While working on their local iteration space, the chunk-size of each work within that space is reduced dynamically following the ideas of `guided` scheduling. Once a processor finishes its local work, reaches to the processor with the most available work left and steals a piece of its work simulating the `dynamic` scheduling. This will achieve load balancing while at the same time the synchronisation overheads are reduced.

1.1 Aims and Objectives

In this piece of work we are interested in implementing an affinity loop scheduling algorithm in OpenMP and examine its performance by running it on a number of threads each time using two different applications in the form of two different loops. Moreover, we are interested in comparing the results with the best built-in OpenMP schedule determined in [2].

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 affinity scheduling was implemented, describing the choice of the data structure and the synchronisation between threads is given in Section 2. In Section 3 the experiments performed are described and the results are presented along with an explanation in 4. Finally, in Section 5 the conclusions drawn from the experiments along with possible future work are presented.

2 Implementation

2.1 Choice of Structure

Implementing affinity loop scheduling means that the loop iterations will be split amongst all the available threads where each thread will have its own local work queue. Moreover, once a thread finishes its assigned work, it should be able to get some work from the most loaded thread. In order to do so, a structure that is shared amongst all the threads is used that contains all the necessary information for the threads to do so.

First of all as we are dealing with boundaries, a structure of type `bound_str` is defined in order to host the low and high boundaries. This is done to avoid declaring each boundary separately and create a more readable code.

All the information that will be shared amongst all threads is included in a structure of type `thread_str`. Its first entry `global` is the global boundaries that effectively contains the boundaries of the loop. Moreover the local boundaries of each thread are also contained in the entry called `local`. These values will be determined once and will not change throughout the execution of the program.

In order to be able to perform affinity, each thread needs to publish which is going to be the next work available so that it self or other threads can access it. Consequently, an entry called `next_lo` is also included where each thread stores the lower bound index of the next work available.

Note that since `p` threads can be available, `local` and `next_lo` should be arrays of size `p`. Consequently, the structure also holds the number of threads available and it is determined by the user once executing the program. Moreover, after this value becomes available, the required memory for the aforementioned entries is allocated dynamically before entering the parallel region and de-allocated at the end of the program.

2.2 Affinity Loop Scheduler

2.2.1 Overview

The implemented affinity loop scheduler is divided into two parts. First is the initialisation part where the local iteration space is assigned to each thread. Then follows the actual execution part, where each thread executes its local iteration set and upon completion starts executing work that has been assigned to the other threads. In order to ensure correctness of the code, the two parts of the scheduler are separated with a synchronisation barrier. This is to ensure that before starting execution of any work all the threads have initialised their shared data.

2.2.2 Initialisation

At the beginning of the parallel region the thread id is obtained. Since no affinity is performed yet, each thread is set to the most loaded so that it can start working on its local set.

Therefore, a loop of `N` iterations is split equally between `p` threads. Each thread is assigned a contiguous local set of iterations using:

$$low = i * ceil\left(\frac{N}{p}\right) \tag{1}$$

$$high = (i+1) * ceil\left(\frac{N}{p}\right) \tag{2}$$

where $i, low, high$ is the current thread and the boundaries of the local set respectively. In the case where the total number of iterations `N` is not exactly divisible with the number of threads `p` then the high boundary of the last thread will be set to equal to `N`, receiving a smaller local set of iterations. Note that this values are stored in the `local` entry of the structure according to the thread index. The `next_lo` entry of each thread is initialised with the low value of its local set. Therefore, at the end of the initialisation stage, `p` work queues are created and there are `p` works available.

2.2.3 Execution

The execution part of the affinity scheduler runs until there is no work left for a the thread to execute. Each thread is assigned a work to execute based on which thread is currently the most loaded. Initially, as each thread has its own local work, is assigned to be the most loaded. Therefore, the work assigned to it is the next available work from its local queue. In the case where the thread has finished its local work, it is assigned work from the work queue of the currently most loaded thread. When there is no work left in any of the other threads, the execution reaches to completion.

Assigning Work

As the execution loop is running, each thread is assigned a work based on its most loaded status, noted as `most_loaded`. The work is assigned by obtaining the next available lower-bound from the queue that is indicated by the most loaded status. Let the assigned lower-bound be `clo`. In addition to that, the upper-bound, `chi`, of the assigned work needs to be obtained. This is achieved by effectively calculating the chunk-size of the work as follows:

$$chunk\_size = ceil\left(\frac{high[most\_loaded] - clo}{p}\right) \tag{3}$$

where $high[most\_loaded]$ is the upper bound of the `most_loaded` work queue. Consequently, `chi` is obtained by $chi = clo + chunk\_size$.

However, the assignment of `clo` requires to read the next available lower boundary from the most loaded work queue array and then update its next available work. As this value can be accessed and updated by other threads, we need to ensure that only one thread access it and updates it at a time to avoid duplicate assignments of the same work. Consequently, a critical region is used to protect the read and write of the next available work. Using a critical region, means that the read and write of the next available work is performed by only one thread at a time, independently of what the `most_loaded` status is and therefore each thread is assigned a unique lower bound of a work to execute. Note that the updated next work is set to be equal to the upper bound of the assigned work.

Establishing the Most Loaded Thread

As mentioned above, the accesses to a work queue are decided using the `most_loaded` status. If affinity of the current thread hasn't started yet, its `most_loaded` status is set to be its thread id, therefore it accesses its own local work queue. However, once it has finished processing its local work queue, it must be able to determine from where to take some work. Therefore, from that point on, the thread starts checking which of the other threads is the most loaded, i.e with the most available work left, and then is assigned work from its work queue.

In order to decide which thread is the most loaded, it checks the remaining work of all the other threads and chooses the thread with the highest value. Note that during the evaluation of the remaining work of each thread, the `next_lo` array is accessed. While this is a read-only

operation it doesn't require the use of the critical region to protect the access. In the case however when there is no work left to all the other threads, the `most_loaded` status of the current thread is set accordingly to indicate end of execution.

2.2.4 Alternative Implementation using Locks

As mentioned in Section 2.2.3, in order to synchronise the threads when accessing and updating the next available work, a critical region was used. Using a critical region however means that if one thread tries to access the next available work of the work queue 0 and another of work queue 1 at the same time, then one has to wait for the other to exit the critical region before entering it. However, race conditions will occur only when more than one threads try to update the next available work of the same work queue and not of different ones.

Consequently, in an effort to reduce idle time, threads are synchronised using a more flexible approach called `locks`. As there are `p` locks created, they are used to replace the critical region by each time locking the same region but this time only for threads that are trying to update the next available work of the same work queue. As a result, the rest of the threads are not blocked and have to wait and the idle time is reduced.

Note that since the number of threads remains constant throughout the execution of the program, the number of lock variable remains constant too. For that reason the allocation and de-allocation of the lock variables is performed once in the main program.

3 Experiments

In this part the experiments regarding the implemented affinity loop scheduler are explained and the results are presented. Each experiment is run using two different applications, namely `loop1` and `loop2`, as specified in the previous coursework. The workload in the first loop decreases over the iteration space while in the second loop each iteration might take time O(1) or O(N). A further discussion of the nature of each loop is available in [2].

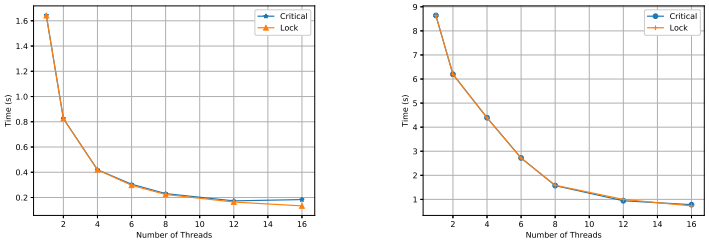
The experiments performed are as follows:

- Experiment 1: Performance comparison of affinity scheduling using critical region and locks for each application.
- Experiment 2: Performance comparison of affinity scheduling using locks and each OpenMP scheduling option using the best chunk-size determined from running on 4 threads, for each application. Note that the best OpenMP scheduling option selected in the previous coursework is also shown here.

Each experiment was carried out 10 times for `1,2,4,6,8,12` and `16` threads and the average times were recorded. In order to obtain accurate timings, the experiments were submitted at the back-end of Cirrus.

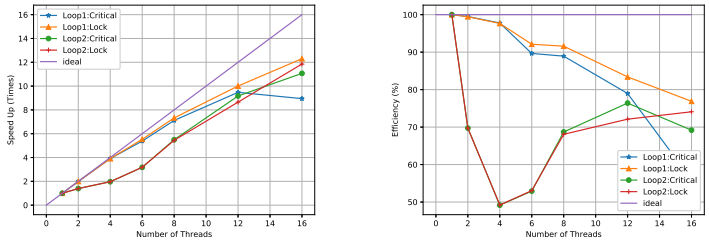
3.1 Experiment 1

This experiment aims to investigate which one of the two versions implemented performs better. In order to do so, the execution time, speed up and efficiency for each application are considered. Speed up (S) is defined as T_1/T_p for each implementation and efficiency as S/p where p is the number of threads. The results are presented in Figures 1 and 2.



(a) Execution Time of Loop1 (b) Execution Time of Loop2

Figure 1: Execution time obtained from running the affinity scheduler using critical region and locks for different number of threads, for both applications



(a) Speed Up (b) Efficiency

Figure 2: Achieved speed up and efficiency obtained from running the affinity scheduler using critical region and locks for different number of threads, for both applications

3.2 Experiment 2

As the affinity scheduling combines the best traits of each OpenMP scheduling option, this experiment aims to investigate how the implemented version of affinity scheduling using locks performs compared to the best available OpenMP scheduling options, determined in coursework, 1 for each application.

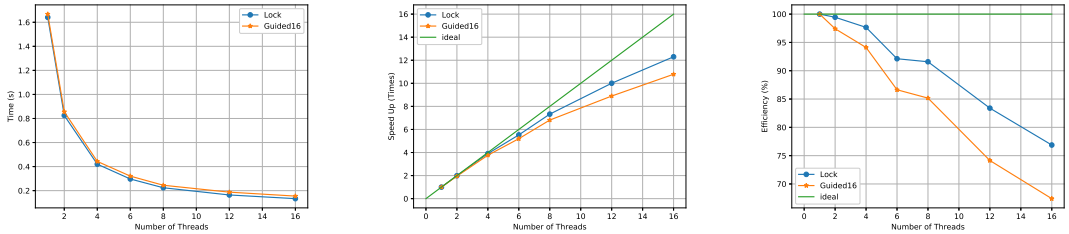
For each OpenMP scheduling option we considered the option that performs better using the chunk-size defined on 4 threads. That is for `loop1` we compare it with `Guided,16` and for `loop2` we compare it with `Dynamic,16`. Initially the best option for `loop2` was chosen to be `Dynamic,16` but after a more in-depth examination it was set to be `Dynamic,4`. Both options are considered here.

For each loop, the execution time, speed up and efficiency are considered. Speed up (S) is defined as T_1/T_p for each implementation and efficiency as S/p where p is the number of threads. The results are presented in Figures 3 and 4.

4 Discussion

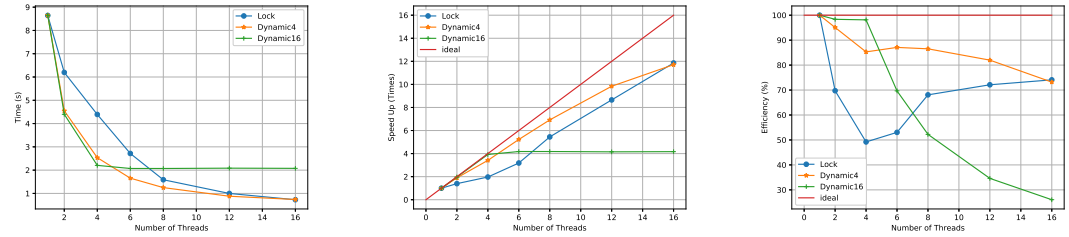
Observing the results obtained in Section 3.1, in terms of execution time presented in Figure 1 we can see that both versions of the affinity scheduler perform the same. Affinity scheduling using locks only seems to outperform affinity scheduling with critical region in terms of execution time in the case of `loop1` when running on 16 threads. This means that the work performed inside the protected region is very small. Therefore, protecting that region will not set the threads trying to enter it into idle for too much time introducing overheads. Consequently, using a critical region will not have a noticeable impact on the execution time of the program.

Observing however Figure 2 we can see that there is some difference in speed up and efficiency



(a) Execution Time (b) Speed Up (c) Efficiency

Figure 3: Execution time, speed up and efficiency obtained from running the affinity scheduler using locks and guided OpenMP scheduling option with the best chunk-size defined on 4 threads. The obtained results are from running each implementation on different number of threads, for loop1.



(a) Execution Time (b) Speed Up (c) Efficiency

Figure 4: Execution time, speed up and efficiency obtained from running the affinity scheduler using locks and dynamic,16 OpenMP scheduling option with the best chunk-size defined on 4 threads. The obtained results are from running each implementation in different number of threads, for loop2. The dynamic,4 option is also shown here.

in each version. As the utilisation of locks will result in reduced synchronisation costs since less threads are idle every time compared to the critical region, increasing the number of threads will result into better speed ups in the former version. Therefore, using locks instead of critical region will result in an implementation that scales up better with the number of threads regardless of the application.

In order to compare the affinity schedule with the available schedules in OpenMP it is worth remembering first how each of the OpenMP schedules works. Here, only `Dynamic` and `Guided` are discussed. The `dynamic` scheduler allocates an equal sized chunk of the overall iterations to each thread on run-time on a first-come-first serve basis. On the other hand, `guided` does the same but the chunks assigned to each thread are reduced over time according to the number of iterations left.

For `loop1`, the affinity scheduler seems to scale better compare to the `guided,16` option. Observing the execution times of each scheduler in Figure 3a the execution time of the affinity scheduler is slightly smaller compared to the `guided`. Moreover, observing Figures 3b and 3c can be seen that both schedulers scale well as the number of threads increasing. As the load of `loop1` is not widely unbalance and it is reduced over the iteration space this is somewhat expected. The reason is that both of them vary the chunk-size over the number of iterations left and they dynamically assign the chunks to each thread counterbalancing any unbalanced workload. However, affinity scheduler works better since at the beginning, before affinity starts there is no synchronisation required between the threads therefore results in better performance for this type of load.

Observing the results of the two schedulers for the `loop2` application in Figure 4 one can see

that the outcomes are different from `loop1`. First of all, the workload of `loop2` is widely unbalanced and most of the work is concentrated at the beginning of the iteration space. As mentioned above, in coursework 1, the best option for `loop2` was initially set to `dynamic,16` but after a more in-depth tuning of the chunk-size considering the number of threads, was changed to `dynamic,4`.

Starting the comparison between affinity scheduling and `dynamic,16` we can say easily that the former performs much better compared to the latter. As most of the workload of the application is concentrated in the first 60 iterations using a dynamic scheduler with chunk-size of 4 will result in few threads to be assigned with most of the work. As the ones that finish their work will have to wait for the others to finish as well, introducing more than 4 threads in this case will not make a difference and the performance will be saturated. On the other hand, using an affinity scheduler will also result in few threads to be assigned with most of the work. However, as the ones that finish their work now will be able to help the ones that are still working, affinity scheduler won't saturate and its performance will keep improving as the number of threads is increased.

By reducing the chunk-size of the dynamic scheduler, we can see how the performance changes and it ends up outperforming the affinity scheduler. As now a smaller chunk-size will be assigned to each thread, the amount of load at the beginning will be partitioned better amongst the threads. Therefore, this scheduler will be able to perform better compared to `dynamic,16` as the number of threads is increased. An important notice here is even-though `dynamic,4` outperforms the affinity scheduler, this was done after a grid-search tuning of the chunk-size was performed. In general, determining the appropriate chunk-size of the scheduler is not an easy think to do and is also application dependent. Affinity scheduler on the other hand, does not require this type of tuning. Moreover, its performance is improved by increasing the amount of threads and it matches the performance of `dynamic,4` at 16 threads.

Moreover, comparing the efficiency of each implementation in Figure 4c we can see that affinity scheduling at the beginning is more inefficient compared to the dynamic scheduler. Even-though it is inefficient for small number of threads, we can see that when the number of threads is increased over 4 threads it starts to become more efficient. In addition, while its efficiency is increasing this is not the case for the dynamic schedulers as it starts to decrease. This is due to the fact that synchronisation overheads start to become significant for the dynamic scheduler.

5 Conclusion

This work implemented and investigated an affinity loop scheduler that combines the best traits of the available OpenMP schedulers. Two versions of the affinity loop scheduler were presented. The first version synchronised threads using a critical region while the second locks. It was found that even-though both implementations were resulted in almost identical execution times for both applications, the second version would scale slightly better with the number of threads. The reason behind that is because using locks would result in smaller synchronisation overheads.

Moreover, the performance of the affinity scheduler for each application was compared to the best scheduling options determined in coursework 1. For the first application a slightly unbalanced load was examined. It was found that affinity scheduler slightly over-performs `guided` scheduler. This is due to the reduced synchronisation overheads at the beginning of the affinity scheduler.

In the case of the second application where a widely unbalanced load was examined, the affinity scheduler was slightly under-performing compared to the `dynamic`, but eventually for large enough number of threads it was able to match its performance. However, `dynamic` was only able to outperform affinity scheduler after an extensive grid-search tuning of the chunk-size parameter, a tuning that affinity scheduler does not need.

References

[1] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[2] C. Stylianou. Threadded programming: Coursework part 1. October 2018.

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`