CS4204: Concurrency and Multi-core Architecture

Matriculation Number: 150002503

March 21, 2021

Contents

Abstract

The objective of this practical was to develop skills in task-based parallelism, work-stealing and load balancing [1]. The requirements of this practical were to implement a parallel divide and conquer skeleton using pthreads and analyse the performance of the skeleton. The parallel skeleton had to exhibit features such as load balancing and random work stealing and allow for tasks to manage their dependencies. It was required that the skeleton be developed in C/C++. Benchmarks were provided for the performance analysis. Details on how to compile and run this code can be found in the `README.md` file in `src/`.

1 Introduction

Parallelism is a hardware artefact and is about making things happen at the same time [2]. This differs from concurrency, which is a programming abstraction or the illusion of independent threads of execution [2]. An example of concurrency would be two threads being hyper-threaded on the same physical core. Here no parallelism is taking place as neither thread executes at the same time. An example of parallelism would be two threads executing simultaneously on separate physical cores. Parallelism and parallel programs have become more popular as multi-core architectures have become more widespread. Common consumer laptops now likely have multi-core architectures. There are many ways to achieve parallel programs and these come in the form of parallel patterns. These patterns include a task farm, pipeline, workpool and divide and conquer. A skeleton is the implementation of a parallel pattern. A skeleton can be evaluated for its cost and performance relative to other skeletons. For example, how does a task farm skeleton compare to that of a pipeline skeleton for matrix multiplications. Evaluating the performance of the skeletons of patterns is important as certain patterns are often much better suited to specific problems than others. Another key aspect of parallel programming is the threshold value for a problem. The threshold value is the value in which the problem is better solved sequentially rather than in parallel. This is a useful technique to limit the number of tasks created and reducing parallel overheads.

The focus of this practical is on the divide and conquer parallel pattern. The divide and conquer pattern is an example of task-based parallelism where tasks can create other tasks. Dependencies will exist between tasks and tasks can also be executed in parallel. The divide and conquer pattern can make use of explicit and implicit parallelism. Explicit parallelism is when the programmer or skeleton has control over the scheduling and parallelism [3]. Implicit parallelism identifies potential parallel tasks [3]. This practical uses explicit parallelism as the skeleton decides when to create tasks and spawn workers. However, the actual scheduling of threads on cores is left to the runtime which is not controlled by the skeleton. The divide and conquer skeleton needs four functions: a function to divide a problem into subproblems, a function to combine the results of sub problems, a function to check whether a threshold has been met and a function that returns a base result once a threshold has been met. A requirement is that subproblems are independent of each other (meaning there is not data-dependencies present).

Load balancing is a technique used to maximise worker usage and minimise idle workers. This is usually employed in a non-shared task pool skeleton. There are different load balancing scheme and the focus of this practical is on work stealing. A work stealing scheme is where workers will always add tasks to their own task pool. Once a work is idle it can steal a task from another worker's task pool. Work stealing has the advantage of little overhead and reduced chance of worker overload compared to other schemes, such as work-pushing. Work stealing can be implemented using perfect random stealing or perfect hierarchical stealing. Perfect random stealing works by choosing a random worker to steal from. Perfect hierarchical stealing has the workers in a tree structure and workers will first steal from their subtree before going to their parent node. This has the benefit of completing work that a task depends on quicker. Perfect random stealing is used in this practical.

2 Design & Implementation

This design can be split into three sections. The public interface to the skeleton, a worker and a task. The data structures and control flow of each are discussed below.

2.1 Skeleton Interface

The public interface for the skeleton code was inspired by the paper A Divide and Conquer Parallel Pattern Implementation for Multicores [4]. The skeleton was defined in the header `dac.h`. The skeleton code requires four lambda functions that each define the divide, combine, base and threshold steps described above along with the problem itself. A user would provide these four functions and the problem. Then the user would then call `compute()` so that the skeleton can calculate the result. Finally, the use would call `getResult()` to obtain the result of the solved problem. This public interface is useful as it only requires the user to define the types of the problem and result and the four lambda functions required. The parallelism is completely hidden from the user. Examples of the lambda functions can be found in `fib.h`, `mergesort.h` and `qsort.h`.

The skeleton interface also has fields to control the worker pool. The worker pool is a vector of workers. A worker is created for each physical core on the machine that is running this skeleton.

On a call to `compute()` the skeleton will instantiate the worker pool and create a root task. Each worker will be given a reference to the worker pool so that they may perform work stealing. The root task is pushed to one worker's work queue. Now all the workers in the work pool are started and the skeleton waits for them to terminate. At this point the root task will contain the result to the problem and the result is passed back to the skeleton code.

2.2 Task

The Task class represents a unit of work in the divide and conquer pattern. Each task has a problem and a result. If the task creates subproblems then a vector of results is created. Once all child tasks have completed the task can combine the results from the results vector into a single result. If the task itself is a subproblem then it can post its result into its parent's result vector. Each task has a pointer to its parent task. The flag `isRoot` controls whether a task is a subproblem or not. Only one task can be the root task of a problem (the root of the hierarchy tree). Two flags control the state of a task: `isWaiting` is set when a task is waiting for its child tasks to complete and `isComplete` is set once a result has been obtained by a task, either from the base function or combining the results from its children. The hierarchy of tasks can be seen in figure 1

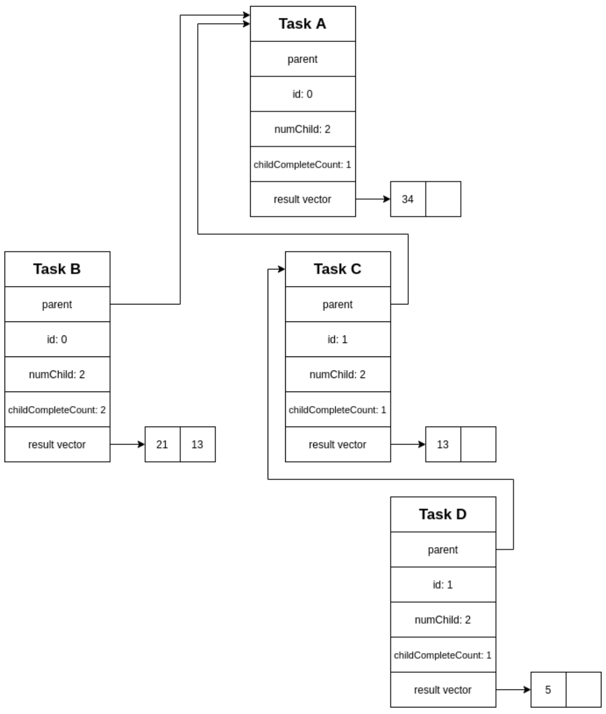


Figure 1: Diagram of task hierarchy. Note each task is in a different state in their lifecycle.

Figure 1 shows four tasks in different stages in their lifecycle. Task A has two children B and C, one of which has complete (but not gone out of scope yet). Task B has complete and has posted its result to its parent, task A. It will no longer be added back to a work queue and will be deallocated once it goes out of scope. Task C is in a similar state to task A, however, its that has complete has already gone out of scope. Task C is waiting on task D. Task D is waiting on one child as well. It is shown that through the pointer to a task's parent the dependencies between the tasks is maintained. This is important as tasks cannot complete until all their child tasks have complete.

2.3 Worker Pool

The worker class represents the workers in the worker pool for the skeleton. The design was inspired by lecture 10. Each worker has its own thread-safe work queue and reference to the worker pool. The worker pool is a vector of worker, instantiated by the skeleton, as described above. Along with the work queue each worker has a copy of the four lambda functions passed to the skeleton so that it can perform the functionality required. The general layout of the worker pool can be seen in figure 2.

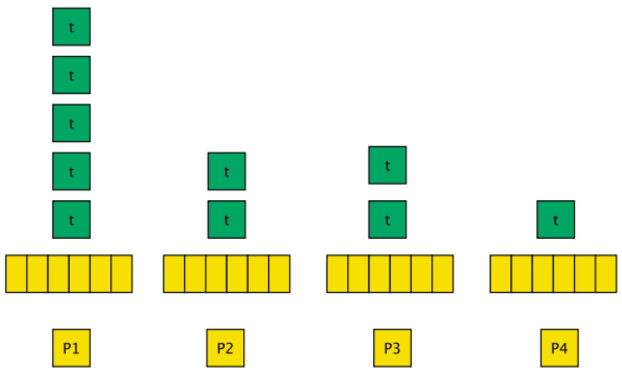


Figure 2: Diagram of the worker pool. Each worker has its own work queue. Note, each worker also has a copy of the four lambda functions passed to the skeleton code. Image taken from lecture 10.

The worker class has two significant functions. The first is `work()` which controls how the worker obtains its next task or terminates. The second is `solveTask()` which controls how a task is solved.

The function `work()` controls the lifetime of a worker. A worker is live and incomplete until the root task has complete on any of the workers. Once the root task has been set to complete it is added back to a work queue. Then a worker will pull the root task from the work queue, either the same worker or a different worker via work stealing. This worker will notices the root task has complete. It will then set itself to complete and exit. Once one worker has set itself to complete all other workers will be notified and they will exit as well.

If none of the workers have complete yet then a worker will continue to pull tasks from their own work queue and solve them using `solveTask()`. Once a worker has an empty work queue it will attempt to steal work from another worker. The worker will shuffle a vector of indexes to ensure random stealing. It will then iterate the worker pool looking to steal work. If a worker in the worker pool has completed by this point then the original worker will also complete. Otherwise the original worker will attempt to steal work. The original worker will continue to attempt to steal work until it successfully does so or a worker is set to complete. This control flow is shown in figure 3.

The function `solveTask()` controls the lifetime of a task. If a task is not added back to the work queue then it goes out of scope and is deallocated. This is safe as the task hierarchy only points to the parent, which are guaranteed to have a longer lifetime than its child tasks. Initially, the worker

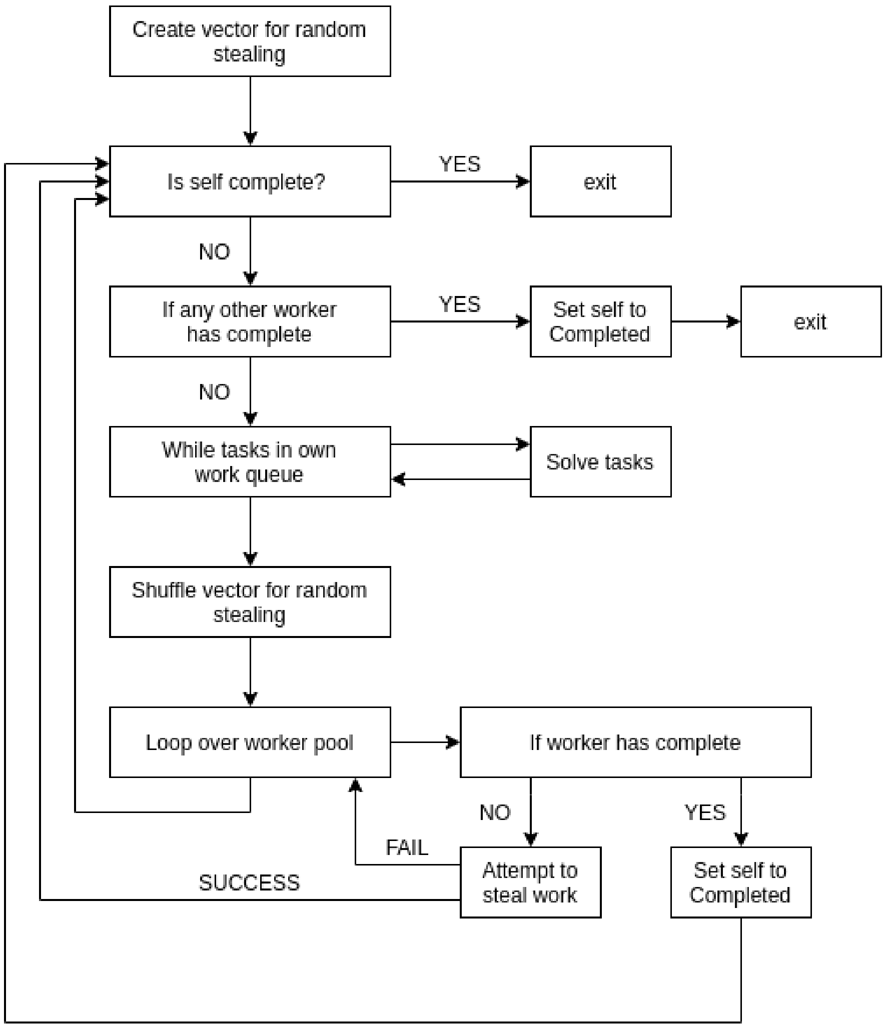
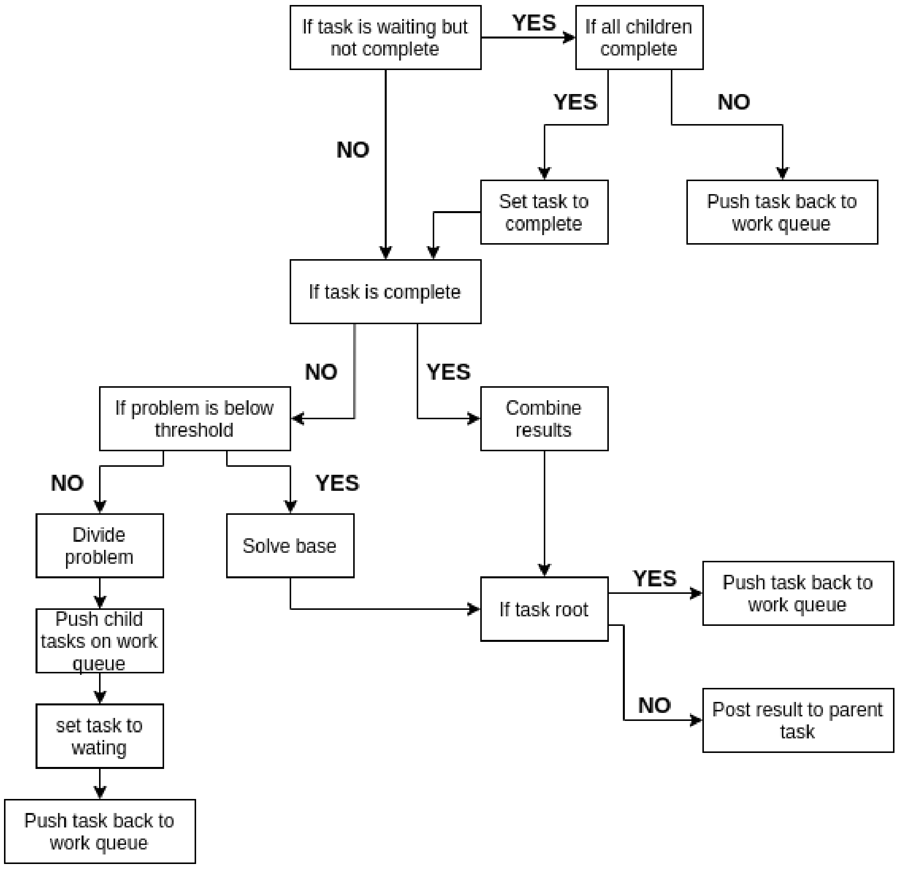


Figure 3: Diagram showing the control flow for the function `work()`.

checks if a task is waiting on any child tasks. If the task is waiting then it checks if all the children have complete. If all the children of the task have complete then the task is marked complete. If task is still waiting on a child then it is pushed back to the work queue. Then the worker checks if a task has been complete. If a task has complete then the results from its children can be combined. If the task itself a sub task then it posts its result to its parent. If the task is the root task then it is pushed back to the work queue. As noted above, this will cause all the worker to exit as the problem has been solved.

If a task is not complete nor waiting on child tasks then it needs to be solved. The worker checks whether the task's problem is below the threshold. If the task's problem is below the threshold then the problem is solved using the base function and the task is considered complete and dealt with using the same means as a complete task. If the task's problem is above the threshold then the problem is divided in to sub problems. The worker creates a child task for each sub problem and pushes them to it's work queue. Finally, the worker will set the task to waiting and push it back to its work queue. This is shown in figure 4.

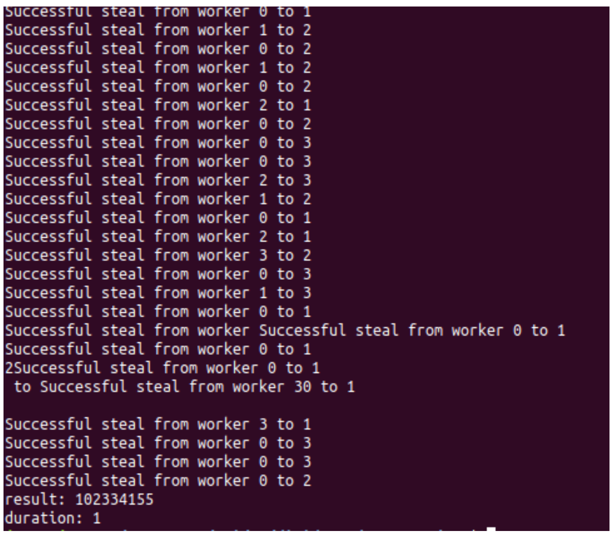


If task is waiting but not complete
YES
If all children complete
YES
NO
NO
Set task to complete
Push task back to work queue
If task is complete
NO
YES
If problem is below threshold
Combine results
NO
YES
Divide problem
Solve base
Push child tasks on work queue
If task root
YES
Push task back to work queue
set task to wating
NO
Post result to parent task
Push task back to work queue

Figure 4: Diagram showing the control flow for the function `solveTask()`.

2.4 Load Balancing

The load balancing scheme implemented is random work stealing. This scheme is fairly aggressive and amounts to a lot of stealing when the number of tasks and number of workers is similar. Work stealing is reduced as the volume of tasks increases. Another thing to note is that work is always stolen from the head of a work queue. Thereby stealing child tasks before a parent task is stolen. Stealing from the front of a queue instead of the back of a queue should improve performance as task dependencies should be resolved sooner. Figure 5 shows the skeleton printing to console when a work steal occurs. The program ran is a Fibonacci problem where $n = 40$ and the threshold is 30.



```
Successful steal from worker 0 to 1
Successful steal from worker 1 to 2
Successful steal from worker 0 to 2
Successful steal from worker 1 to 2
Successful steal from worker 0 to 2
Successful steal from worker 2 to 1
Successful steal from worker 0 to 2
Successful steal from worker 0 to 3
Successful steal from worker 0 to 3
Successful steal from worker 2 to 3
Successful steal from worker 1 to 2
Successful steal from worker 0 to 1
Successful steal from worker 2 to 1
Successful steal from worker 3 to 2
Successful steal from worker 0 to 3
Successful steal from worker 1 to 3
Successful steal from worker 0 to 1
Successful steal from worker Successful steal from worker 0 to 1
Successful steal from worker 0 to 1
2Successful steal from worker 0 to 1
 to Successful steal from worker 30 to 1

Successful steal from worker 3 to 1
Successful steal from worker 0 to 3
Successful steal from worker 0 to 3
Successful steal from worker 0 to 2
result: 102334155
duration: 1
```

Figure 5: Output of skeleton showing work stealing occurring. This highlights the aggressiveness of the stealing.

3 Experiment

3.1 Design

An experiment was designed to show the performance of the skeleton with varying numbers of cores supplied. The threshold for each experiment was constant. Each run was timed using the chrono library in c++. Each benchmark was ran three times at a specific number of cores and the average of the three runs was recorded. The fib problem was ran from Fib(0) to Fib(39) with a threshold value of 10. The two sorting problem ran from a list of 10,000 elements to 1,000,000 elements at a step size of 10,000. The threshold for both the sorting problems was 2000, which was the value suggested in [4]. The number of cores tested was one, two, four and eight.

The sequential benchmarks provided were converted into C++ so that the chrono module could be used to time each benchmark. The Fibonacci benchmark was also changed so that Fib(0) = 0 instead of Fib(0) = 1 as it was originally.

3.2 Results

Figure 6 shows the performance of the skeleton code for the merge sort problem. The sequential benchmark is the blue series. As can be seen from figure 6 running the skeleton with one or two cores does not improve the performance of a merge sort at all. This is likely because the overhead of the skeleton outweighs the performance gain of any parallelism. This is expected, especially for one core where there is no gain in parallelism. Running the skeleton with four or eight cores shows a significant improvement in the performance of the merge sort. In these instances the performance gain from the parallelism is greater than the overhead of the skeleton. In a larger problem it is expected that the performance gain of the skeleton, for any number of cores greater than one core, would be more pronounced.

Figure 7 shows the performance of the skeleton code for the quick sort problem. The sequential

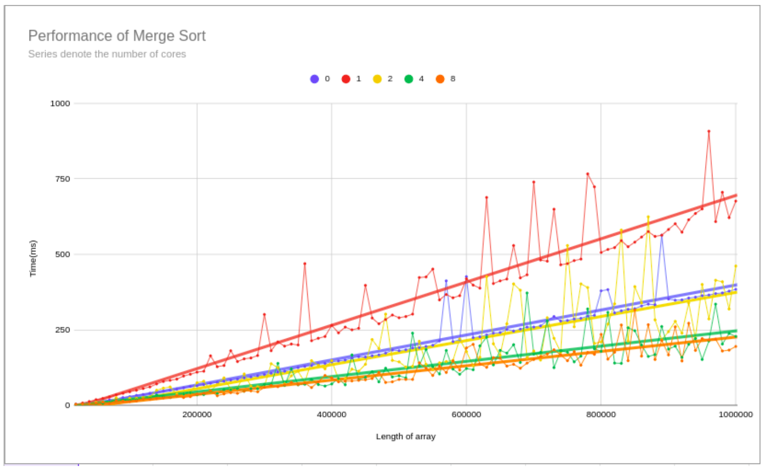


Figure 6: Graph displaying the performance results of the merge sort problem. Note, core 0 is the sequential benchmark provided.

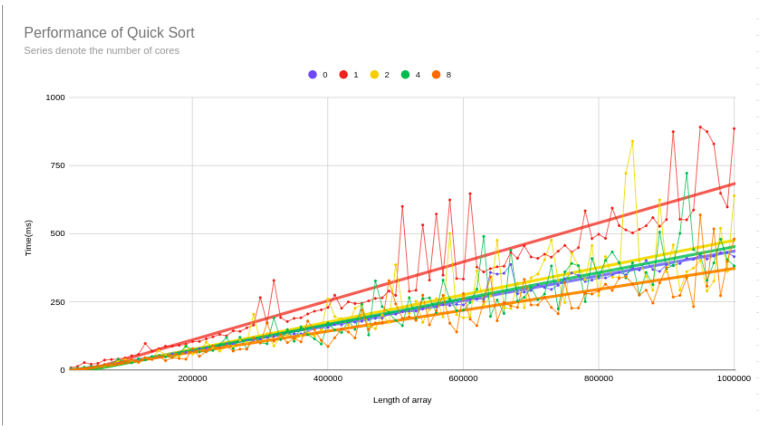


Figure 7: Graph displaying the performance results of the quick sort problem. Note, core 0 is the sequential benchmark provided.

benchmark is the blue series. Figure 7 shows that the only configuration of the skeleton to outperform the sequential benchmark is the eight core configuration. A configuration of two and four cores have a similar performance to the sequential benchmark. This was not an expected result and it shows that the overhead of two and four cores still outweighs any gain in parallel computation. An eight core configuration does outperform the sequential benchmark, which was to be expected.

The reason why certain skeleton configurations do not outperform the sequential benchmarks for both quick sort and merge sort can be explained by a number of causes. Firstly, the thread safe work queue are locking queues guarded by a single lock. From the first practical it is known that this is not the best performing implementation of a thread safe queue. It would have been significantly better to use a lock-free implementation. A locking implementation was used as it was easiest to ensure correctness while the skeleton code was being developed. Another cause for the lack of speedup is that level of parallelism, i.e. the threshold level. The threshold was set to 2000 for both sorting problems. However, it is clear that this threshold could have been higher for merge sort and much higher for quick sort before it would adversely effect the performance of the skeleton. A preliminary experiment should have been conducted to get the threshold level optimal. Finally, a larger data size would have been more expressive. This experiment makes the sort behaviour look linear as the data size is very small. A larger data size would have shown the expected O(nlogn) behaviour and a more pronounced difference between the sequential benchmark and the skeleton configurations.

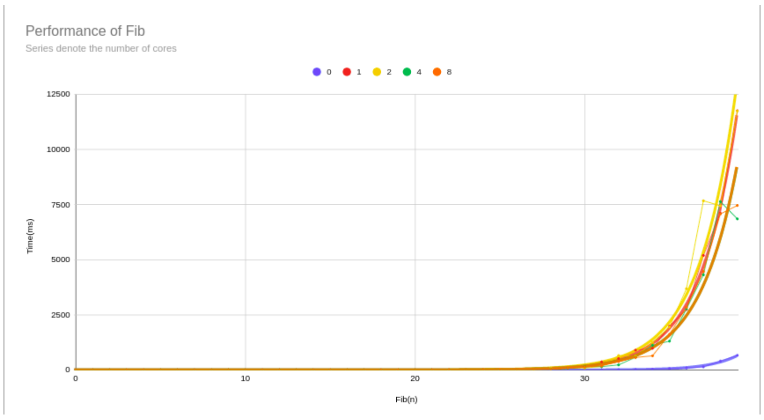


Figure 8: Graph displaying the performance results of the Fibonacci problem. Note, core 0 is the sequential benchmark provided.

Figure 8 shows the performance of the skeleton code for the Fibonacci problem. The sequential benchmark is the blue series. This problem was entirely different to the sorting problems. The sorting problems have behaviours of O(nlogn) whereas the behaviour of the Fibonacci problem is exponential. There is, however, a linear solution to the Fibonacci problem using memoization. This technique is used in the base function for the Fibonacci problem. The threshold value was set to ten so for all values up to ten the skeleton should have been, and was, identical to the sequential benchmark. It can be seen that up to a value of Fib(30) there is no difference in the sequential benchmarks and the skeleton configurations. From Fib(30) onward the exponential behaviour is clearly seen in the skeleton configurations. It appears that the sequential benchmark also exhibits exponential behaviour, however, the overhead from the skeleton clearly gives the exponential behaviour a much larger coefficient. Considering that a linear solution for the Fibonacci problem exists it is unlikely that a parallel solution would be employed.

4 Conclusion

In conclusion, the objective of this practical to develop skills in task-based parallelism, work-stealing and load balancing was achieved. This is evident in the requirements of the practical being met. A critical analysis of the divide and conquer implementation was given.

References

[1] Practical Specifications,
https://studres.cs.st-andrews.ac.uk/CS4204/Practicals/Practical2/CS4204-Practical2.pdf

[2] Lecture 1: Introduction
https://studres.cs.st-andrews.ac.uk/CS4204/Lectures/CS4204-L1-Intro.pdf

[3] Lecture 10: Divide and Conquer
https://studres.cs.st-andrews.ac.uk/CS4204/Lectures/CS4204-L10-DC.pdf

[4] A Divide and Conquer Parallel Pattern Implementation for Multicores, M Danelutto et al https://dl.acm.org/doi/pdf/10.1145/3002125.3002128?fbclid=IwAR0BW5V69q3riFZCufD5Tdy6ABGkSESfV5gDZBatBoDbQ3iY4Au7jnZhars