Module code: CS4204
Matriculation number: 200017941

Practical 2: Parallel Patterns

Overview

In this coursework, I have implemented a parallel pattern library which provides functionality for a farm and a pipeline pattern. These patterns make use of a queue to store the tasks and it provides thread-safe access to each worker executing in parallel. These patterns have been tested using the two supplied examples: *example.c* and *convolution.c*. In *example.c*, a farm pattern has been utilised since this would be the best way to parallelise the calculation of the Fibonacci sequence. Using a pipeline pattern wouldn't make sense because there are no separate stages in its execution as it is just one function that does the calculation. On the other hand, *convolution.c* utilises the pipeline pattern due to there being different sequential stages in its execution. Each of these sequential stages are basically functions to be pipelined hence the choice to use the pipeline pattern. Additionally, for each of the provided examples, I have conducted an experiment evaluating the execution of the code with and without a parallel pattern. The purpose of this is to highlight the effect of parallel patterns in speeding up the execution of tasks.

Design and Implementation

Task Queue

The queue has been implemented as a *TaskQueue struct* and it stores the necessary information regrading the queue itself such as its head and tail. It also stores *pthread_mutex_t* and *pthread_cond_t* variables which are necessary for the queue to be able to handle multiple threads accessing the queue simultaneously. Each element in the queue is a *TaskData struct* which acts as the input data for the worker function. Alongside this, it stores a pointer to the next task data in the queue. This is necessary since queues store data in an orderly manner and this information is not maintained in *TaskQueue* as it only needs to maintain a pointer to the retrievable task data.

The implementation of the queue uses a coarse-grained locking mechanism to handle concurrent access which is simply a *mutex* lock and unlock at the beginning and end of the operation functions, namely *put_task()* and *get_task()* which push and pop task data. Another important feature in the implementation is the condition variable mentioned above. This is mostly relevant for the pipeline pattern where the input queue of the next stage in the pipeline depends on the output queue of the previous stage. Therefore, whenever the previous stage pushes a task data into the shared queue, this needs to be signalled to the next stage and it should proceed to pop this task data. This is achieved by having a *pthread_cond_signal()* call in *put_task()* after a task data has been pushed. Then in *get_task()*, before the queue is accessed, a call is made to *pthread_cond_wait() which* receives this signal and can then proceed to pop the task data. If this wait is not implemented the nit would lead to a segmentation error since NULL would be passed to the thread retrieving a task data. It is important to note that this call in get_task() is within a while loop which essential does a busy wait so that its continuously checks for the availability of task data.

Worker Tasks

Each task that a worker receives is encapsulated in the *WorkerTask struct.* It contains the function to execute in parallel, the input queue containing the input parameters to the function and the output queue to store the return values from the function. These are the tree main components which make

up each task. An important point to note is that *WorkerTask* contains a union of four function pointers, each representing a function given in the examples that need to be parallelised. These functions include: *fib()*, *get_image_name()*, *read_image_and_mask()* and *process_image()*. The reason for this design decision is so that the function pointer for each function takes into consideration the parameter and return types which are different for each function.

Farm Pattern

The farm is designed to have a single input queue and a single output queue which is shared by all the workers.

The implementation for this contains two parts. The first is a function to create the farm which instantiates the *WorkerTask* data structure and begins the multi-threaded execution, passing the *WorkerTask* data structure to each thread executing in parallel. The second part is the wrapper function which is executed in each thread. This function is responsible for popping a task data form the input queue, calling the worker function with the popped task data and pushing the returned value to the output queue. This is done repeatedly until the input queue is empty which causes the thread to terminate. After all the threads have terminated, the output queue is then retuned.

Pipeline Pattern

The pipeline is designed to have three stages, corresponding to the three main functions in *convolution.c* which need to be executed in order. It has four queues in total which include the main input queue to the pipeline, the main output queue from the pipeline, an intermediate queue connecting the first stage to the second stage and another intermediate queue connecting the second stage to the third stage.

The implementation for this again contains two parts. The first is a function to create the pipeline which follows the same logic as before. For each stage, a *WorkerTask* data structure is instantiated before passing it to the wrapper function for that stage which is executed in a separate thread. It is important to note that the input and output queues for the stages will overlap so the same queues are used when instantiating the *WorkerTask* data structure. For example, the output queue of the first stage and the input queue of the second stage are the same. The second part is the wrapper function for each of the stages which all follow the same logic. The only difference is that since the threads depend on each other, the condition variable comes into play. This wasn’t the case for the farm pattern since each thread is completely independent. So, whenever a call is made to *get_task()*, if the queue is empty, then a busy wait is done until the queue is non-empty as previously discussed. Another important point to note is that the termination condition for the second and third stages are different. The first two stages each have a flag associated with it which marks whether that stage has completed its execution. Therefore, instead of just checking that the input queue is empty to terminate the stage, it also checks whether the relevant flag has been marked.

Performance Evaluation

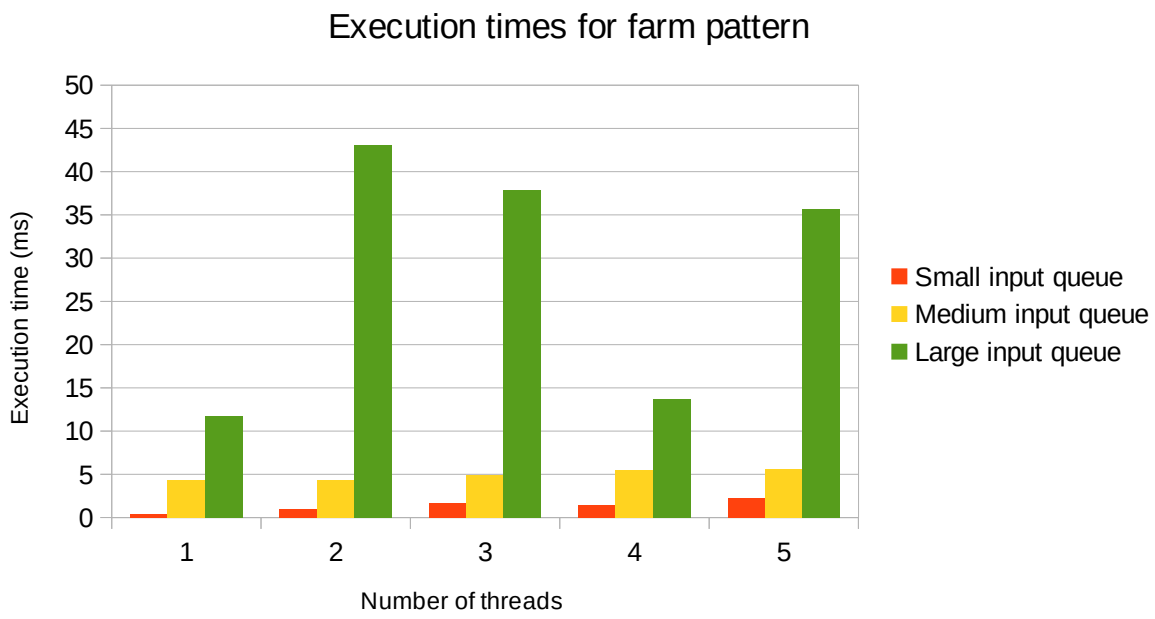
In order to evaluate the performance of the parallel implementations, I ran a series of tests for both the given example. For each test run, I varied the number of threads and the size of the input queue.

example.c

For example.c, I used three different input queue sizes: small which has 10 tasks, medium which has 100 tasks and large which has 1000 tasks. For each input size, I ran the test with threads ranging from 1 to 5. Below are the results from running the tests.

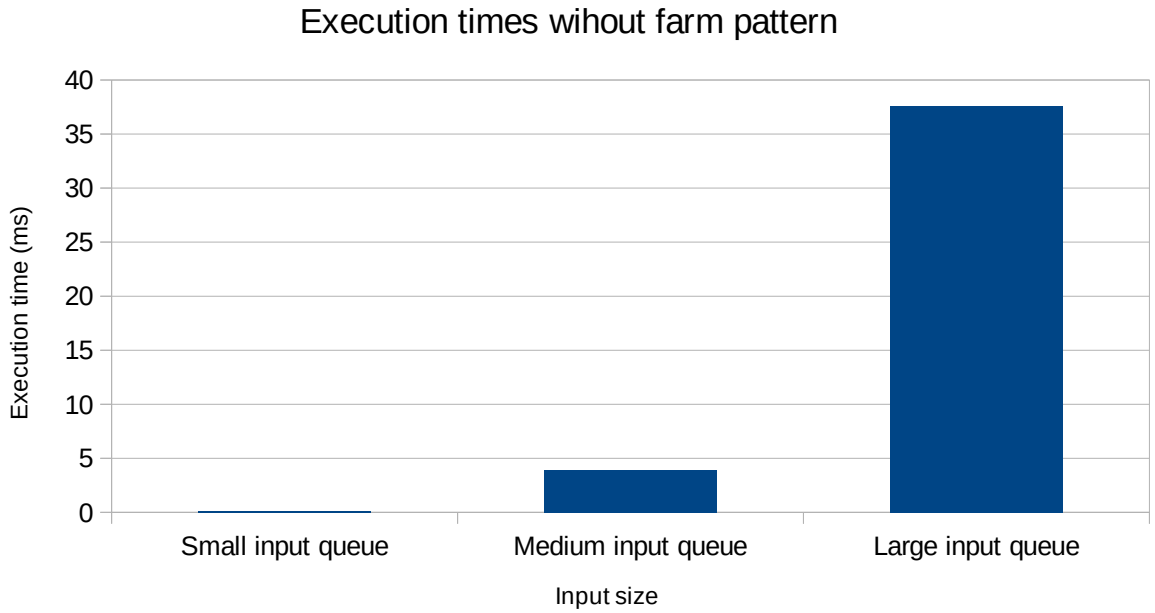
Execution using the farm parallel pattern:

Number of threads	Small input queue (10 tasks)	Medium input queue (100 tasks)	Large input queue (1000 tasks)
1	0.433	4.359	11.787
2	1.022	4.299	43.119
3	1.721	4.926	37.848
4	1.4	5.438	13.738
5	2.281	5.614	35.714



Execution without using the farm parallel pattern:

Small input queue (10 tasks)	Medium input queue (100 tasks)	Large input queue (1000 tasks)
0.106	3.888	37.614



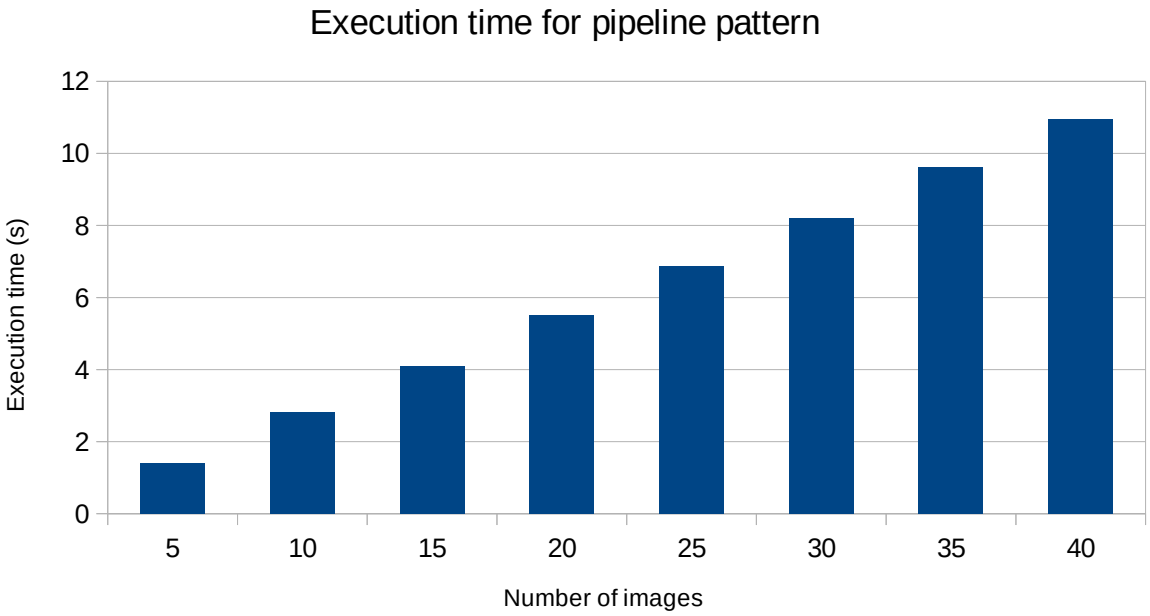
It can be observed that the execution times recorded are unusual. Increasing the number of threads should in theory decrease the time taken to complete executing all the tasks, but there seems to be some inconsistencies. Additionally, using the parallel pattern should result in a performance improvements but this doesn't seem to be the case for the most part.

convolution.c

For *convolution.c*, the number of images range from 5 to 40. Below are the results from running the tests.

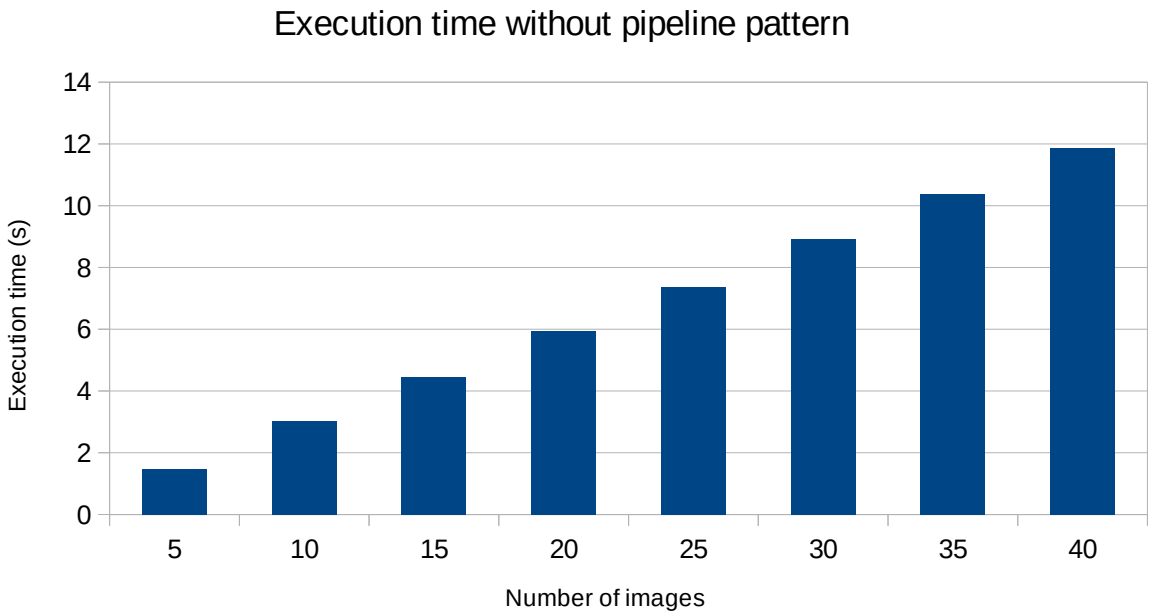
Execution using the pipeline parallel pattern:

Number of images	Execution time (ms)
5	1.396
10	2.817
15	4.099
20	5.507
25	6.878
30	8.206
35	9.607
40	10.954



Execution without using the farm parallel pattern:

Number of images	Execution time (ms)
5	1.485
10	3.019
15	4.452
20	5.926
25	7.35
30	8.915
35	10.387
40	11.879



Using the parallel pattern does seem to result in performance improvements but this is very minor. This is because the pipeline isn't implemented correctly since the next stage in the pipeline is only executed after the previous stage has completed its execution which can be observed from the trace output.