CS4204 Concurrency and Multi-Core Architectures P2 -

Parallel Libraries (Para-Pat)

26/03/2024 - 230033234

1. Introduction

The following practical involves designing a parallel pattern library, **‘Para-Pat’**, in C/C++ using pthreads. The library is intended to provide a high-level interface for parallelising code through Farm and Pipeline patterns, abstracting away complexities like thread management and task queues. The main objective is to allow users to incorporate these patterns into their C/C++ code, where the farm pattern distributes tasks among multiple worker threads and the Pipeline pattern divides the workload into stages with dedicated workers.

Performance evaluation was conducted using the provided examples, **fibonnaci.c** and **convolution.cpp**, by measuring runtimes under configurations with varying input, allowing us to explore the impact of granularity on parallelism.

1.1 Completion Checklist

Section	Functional Status
Part 1	**Attempted, Fully Functional**
Implementing the parallel farm pattern.	Attempted, Fully Functional
Part 2	**Attempted, Fully Functional**
Implementing the parallel pipeline pattern.	Attempted, Fully Functional
Part 3	**Attempted, Fully Functional**
Give a comparison with another library (FastFlow)	Attempted, Fully Functional
Part 4	**Attempted, Fully Functional**
Evaluate ParaPat with Convolution example.	Attempted, Fully Functional
Evaluate ParaPat with Fibonacci example.	Attempted, Fully Functional

Table 1: Checklist, showing the completion status of the project.

2. Design and Implementation (~860 words)

2.1 Architecture

The Para-Pat library was designed to provide a flexible and also scalable architecture for utilising parallel patterns. The library consists of several key components:

- **Pipeline** - The class that represents the overall pipeline and is responsible for managing the stages, connecting them, and executing the pipeline.
- **Stage** - An abstract class that defines the structure of a stage (Farm or Pipe) in the pipeline.
- **Farm** - A child of the Stage class that creates a specified number of worker threads, each responsible for executing a given worker function (using WorkerWrapper) in parallel.
- **Pipe** - A child of the Stage class that creates a single worker thread for executing the worker function.
- **Worker** - Represents an individual worker within a stage. It has an input queue, one or more output queues (these are normally pointers to the input queues of future stages), and stores the worker function that performs the actual work on tasks.

Furthermore, the **Task** and **Result** structs are used to represent the input and output data for the workers respectively (they are simply wrappers for the void* data used to store the worker functions data).

2.2 Key Design Choices

In terms of aiming to abstract away the complexities of thread management, task queues, and locking mechanisms (with the use of structs and classes), an important decision was to use thread-safe queues **('ThreadSafeQueue')** for inter-thread communication and the handling of tasks/results. This was done with the use of '**pthread_mutex-t',** ensuring thread safety and preventing race conditions when multiple threads access shared resources concurrently.

To allow for the propagation of tasks between the stages, the **Worker** class has a vector of output queues. This vector of output queues consists of all the input queues in the next stage, for direct linking of queues. The **Pipeline** class is responsible for connecting the stages by linking the output queues of one stage to the input queues of the next stage. It also handles the execution of the pipeline, distributing tasks to the workers in the first stage, and collecting the results from the final stage's workers. This design choice allows for efficient round-robin distribution of tasks to the next stage, whilst minimising the chance of contention as we are not using **only** single output queue (but multiple in parallel) between stages, improving load balancing and overall performance.

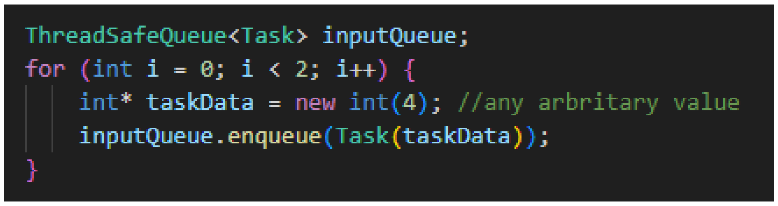
*(We initially wrote a **StageManager** class (now commented out) which had initially attempted to centralise task distribution between stages in the pipeline. However, the approach was later removed due to massive overheads and contention issues introduced, which did not reduce overall execution time. The code is kept in the project for proof of work.)*

2.3 Implementation Details & Workflow

2.3.1 Task Generation and Pipeline Execution

The main entry point is typically the **main()** function of the file that the user intends to use the library with. The user can create instances of the desired parallel patterns (Farm, Pipe, or a combination of both) and then add them to a **Pipeline** object.

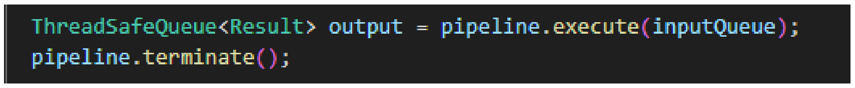
Tasks are generated by the user and enqueued into a ThreadSafeQueue<Task> object, which represents the input queue for the pipeline. Here's an example of how tasks might be generated and enqueued:



```
ThreadSafeQueue<Task> inputQueue;
for (int i = 0; i < 2; i++) {
    int* taskData = new int(4); //any arbitrary value
    inputQueue.enqueue(Task(taskData));
}
```

Figure 1: Creating the input queue to pass into the pipeline.

Once the tasks are enqueued (stored in void* data), the **Pipeline** object's **execute()** method is called, passing in the **'inputQueue'** as an argument. This method initiates the pipeline execution process.



```
ThreadSafeQueue<Result> output = pipeline.execute(inputQueue);
pipeline.terminate();
```

Figure 2: Executing the pipeline with the 'inputQueue' and then terminating.

(A 'main.cpp' file has been written to demonstrate further use of the library. We have created an unnecessarily complex parallel pattern for demonstration of robustness and testing purposes.)

2.3.2 Stage Connectivity

Within the **execute()** method, the **Pipeline** class first connects the stages by linking the output queues of one stage to the input queues of the next stage. This is achieved through the **connectStages()** method, which iterates over the stages and calls the **connectWorkers()** method on each stage, passing in the workers from the next stage.

After the stages are connected, the **execute()** method starts distributing tasks to the workers in the first stage. This is done by dequeuing tasks from the **inputQueue** and enqueuing them into the input queues of the first stage's workers in a round-robin fashion.

Once all tasks from the **inputQueue** have been distributed, an end-of-stream (EOS) task is enqueued for each worker in the first stage. This EOS task signals the workers to propagate the EOS signal to the subsequent stages, ensuring that all tasks are processed before completion.

We isolate the terminate() function from the execute() function incase the user desires to input more queues of tasks. The program can handle multiple input queues, it only terminates once the shutdown tasks are sent through.

2.3.3 Worker Task Processing

All **Worker** objects implement the run() method for processing tasks from the input queue. The method continuously checks the input queue for available tasks, when a task is dequeued, the worker checks its type:

- **Valid Task:** If valid (not an EOS or shutdown signal), the worker executes the associated worker function with the task's data (void* data) as input. The result of the worker is then enqueued to the next stage's workers' input queues in a round-robin fashion.
- **EOS Task:** If the task is an EOS task, the worker propagates the EOS signal by enqueuing it to all workers in the next stage. This ensures that the EOS signal is properly propagated through the pipeline.
- **Shutdown Task:** If the task is a shutdown signal, the worker sets its **stopRequested** flag to **true**, indicating that it should initiate the shutdown process. The shutdown task is also enqueued to all workers in the next stage to propagate the shutdown signal.

If no task is available in the input queue, the run() method yields the CPU using **'sched_yield()'** to reduce busy waiting.

2.3.4 Result Collection

After the pipeline execution is initiated, the **execute()** method waits for the tasks to be processed by the final stage's workers. It collects the results from the output queues of the final stage's workers and enqueues them into a **ThreadSafeQueue<Result>** object, which represents the output queue of the pipeline.

Once all expected results have been collected, the execute() method returns the output queue, allowing the user to dequeue and process the results (type conversion from void* data back to original datatype).

2.3.5 Termination and Cleanup

We run the **pipeline.terminate()** method, enqueuing shutdown tasks for all workers in the first stage, propagating the shutdown signal through the pipeline. It then waits for all worker threads to join (using **pthread_join()**) before returning. Several class destructors will eventually be run, cleaning up allocated resources.

3. Performance Analysis

3.1 Methodology

To evaluate the performance of the Para-Pat library, we conducted experiments on the provided examples: **'fibonacci_parapat.cpp'** and **'convolution_parapat.cpp'**. The main performance metric was **average execution time**, and the experiments were carried out on a multi-core system (AMD Ryzen 7 5800X 8-cores), with varying configurations of parallel patterns and input sizes.

To obtain an accurate measurement of the execution time, we used the **'get_current_time()'** function provided to record the start and end times of the parallel sections. The average execution time is the average of **five** attempts.

3.2 Results & Analysis

3.2.1 Convolution Example

Input: 1024x1024 images, 50 images

Where Stage1(n) -> Stage2(n) represents the order ***read_image_and_mask() -> process_image()***

Configuration	Average Execution Time (s)	Speed Up
Sequential Execution (Baseline)	10.7391	1.00x
2-Stage Pipeline (Pipe -> Pipe)	10.1928	1.05x
Number of Parallel Workers = 2		
Nested Farm within Pipeline (Farm(2) -> Farm(2))	5.7078	1.88x
Hybrid Nested Configuration 1 (Farm(2) -> Pipe)	10.5562	1.02x
Hybrid Nested Configuration 2 (Pipe -> Farm(2))	5.16782	2.08x
Number of Parallel Workers = 3		
Nested Farm within Pipeline (Farm(3) -> Farm(3))	4.1397	2.59x
Hybrid Nested Configuration 1 (Farm(3) -> Pipe)	10.3152	1.04x
Hybrid Nested Configuration 2 (Pipe -> Farm(3))	3.7703	2.84x

Number of Parallel Workers = 4		
Nested Farm within Pipeline (Farm(4) -> Farm(4))	3.7956	2.83x
Hybrid Nested Configuration 1 (Farm(4) -> Pipe)	10.927	0.98x
Hybrid Nested Configuration 2 (Pipe -> Farm(4))	2.9832	3.60x
Number of Parallel Workers = 5		
Nested Farm within Pipeline (Farm(5) -> Farm(5))	3.1061	3.46x
Hybrid Nested Configuration 1 (Farm(5) -> Pipe)	11.6106	0.92x
Hybrid Nested Configuration 2 (Pipe -> Farm(5))	2.7579	3.89x
Number of Parallel Workers = 6		
Nested Farm within Pipeline (Farm(6) -> Farm(6))	3.6739	2.92x
Hybrid Nested Configuration 1 (Farm(6) -> Pipe)	12.3469	0.87x
Hybrid Nested Configuration 2 (Pipe -> Pipe -> Farm(6))	2.321	4.63x
Number of Parallel Workers = 7		
Nested Farm within Pipeline (Farm(7) -> Farm(7))	2.7329	3.93x
Hybrid Nested Configuration 1 (Farm(7) -> Pipe)	12.2478	0.88x
Hybrid Nested Configuration 2 (Pipe -> Farm(7))	2.1549	4.98x
Number of Parallel Workers = 8		
Nested Farm within Pipeline (Farm(8) -> Farm(8))	2.8554	3.76x
Hybrid Nested Configuration 1 (Farm(8) -> Pipe)	12.8112	0.84x
Hybrid Nested Configuration 2 (Pipe -> Farm(8))	2.0991	5.11x

Table 2: Showing the Average Execution Time and Speed Ups for the ‘Convolution’ example.

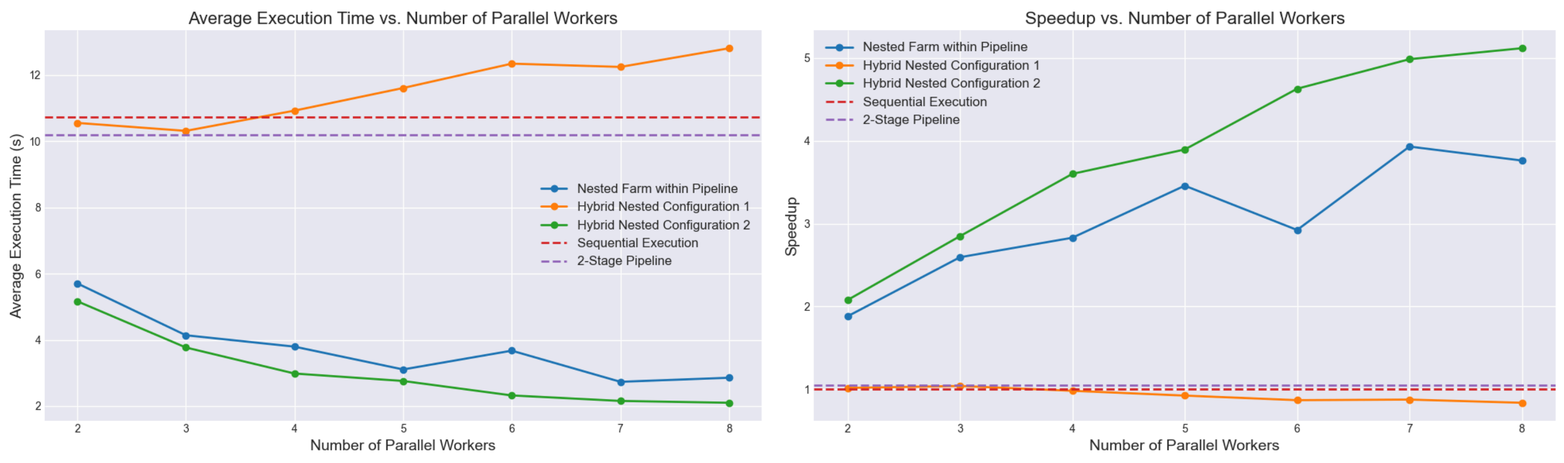


Figure 3: A graph showing the trends in Average Execution Time/Speedup vs Number of Parallel Workers.

The results of this experiment can be seen in the figure 3 above. Using the sequential execution as a baseline of 10.7391 seconds, the 2-stage pipeline configuration shows a slight improvement with a speedup of **1.05x**. In terms of the **Nested Farm within Pipeline** configuration, as the number of parallel workers increases, the execution time decreases, and the speedup improves. The **best performance** is achieved with **7** workers, resulting in a speedup of **3.93x**. However with, 8 workers, the speedup slightly decreases to 3.76x, indicating potential overhead/contention issues when higher thread counts are reached.

The **Hybrid Nested Configuration 1** shows no significant improvement over the sequential execution. As the number of workers increases, the speedup can be seen to decrease, this could be due to the coarse-grained locking that the **ThreadSafetyQueue** uses. There is increased contention as several workers in the first stage contend with access to the single input queue in the second stage, slowing down execution time. Perhaps implementing a lock-free/wait-free approach could fix this.

The **Hybrid Nested Configuration** has been shown to demonstrate the best overall performance and scalability, where increasing the number of workers consistently improves the speedup, reaching **5.11x** with 8 workers.

3.2.2 Fibonacci Example

Input: Fibonacci(900090000), 25 Tasks (iterations)

Where Stage1(n) -> Stage2(n) represents the order ***payload1() -> payload2()***

Configuration	Average Execution Time (s)	Speed Up
Sequential Execution (Baseline) ***(Ran on C compiler)***	38.1009	1.00x
2-Stage Pipeline (Pipe -> Pipe)	40.2357	0.95x
Number of Parallel Workers = 2		
Nested Farm within Pipeline (Farm(2) -> Farm(2))	22.1412	1.72x
Hybrid Nested Configuration 1 (Farm(2) -> Pipe)	40.7495	0.94x
Hybrid Nested Configuration 2 (Pipe -> Farm(2))	21.9197	1.74x
Number of Parallel Workers = 3		
Nested Farm within Pipeline (Farm(3) -> Farm(3))	18.2088	2.09x
Hybrid Nested Configuration 1 (Farm(3) -> Pipe)	42.2899	0.90x
Hybrid Nested Configuration 2 (Pipe -> Farm(3))	20.8372	1.83x
Number of Parallel Workers = 4		
Nested Farm within Pipeline (Farm(4) -> Farm(4))	20.0909	1.90x
Hybrid Nested Configuration 1 (Farm(4) -> Pipe)	44.3034	0.86x
Hybrid Nested Configuration 2 (Pipe -> Farm(4))	21.5218	1.77x
Number of Parallel Workers = 5		
Nested Farm within Pipeline (Farm(5) -> Farm(5))	17.4074	2.19x
Hybrid Nested Configuration 1 (Farm(5) -> Pipe)	43.9787	0.87x
Hybrid Nested Configuration 2 (Pipe -> Farm(5))	21.8264	1.75x
Number of Parallel Workers = 6		
Nested Farm within Pipeline (Farm(6) -> Farm(6))	20.4241	1.87x
Hybrid Nested Configuration 1 (Farm(6) -> Pipe)	46.6236	0.82x
Hybrid Nested Configuration 2 (Pipe -> Pipe -> Farm(6))	22.5989	1.69x
Number of Parallel Workers = 7		
Nested Farm within Pipeline (Farm(7) -> Farm(7))	22.8568	1.67x
Hybrid Nested Configuration 1 (Farm(7) -> Pipe)	49.0131	0.78x
Hybrid Nested Configuration 2 (Pipe -> Farm(7))	24.9329	1.53x
Number of Parallel Workers = 8		
Nested Farm within Pipeline (Farm(8) -> Farm(8))	25.2292	1.51x
Hybrid Nested Configuration 1 (Farm(8) -> Pipe)	49.4192	0.77x
Hybrid Nested Configuration 2 (Pipe -> Farm(8))	24.5682	1.55x

Table 3: Showing the Average Execution Time and Speed Ups for the ‘Fibonacci’ example.

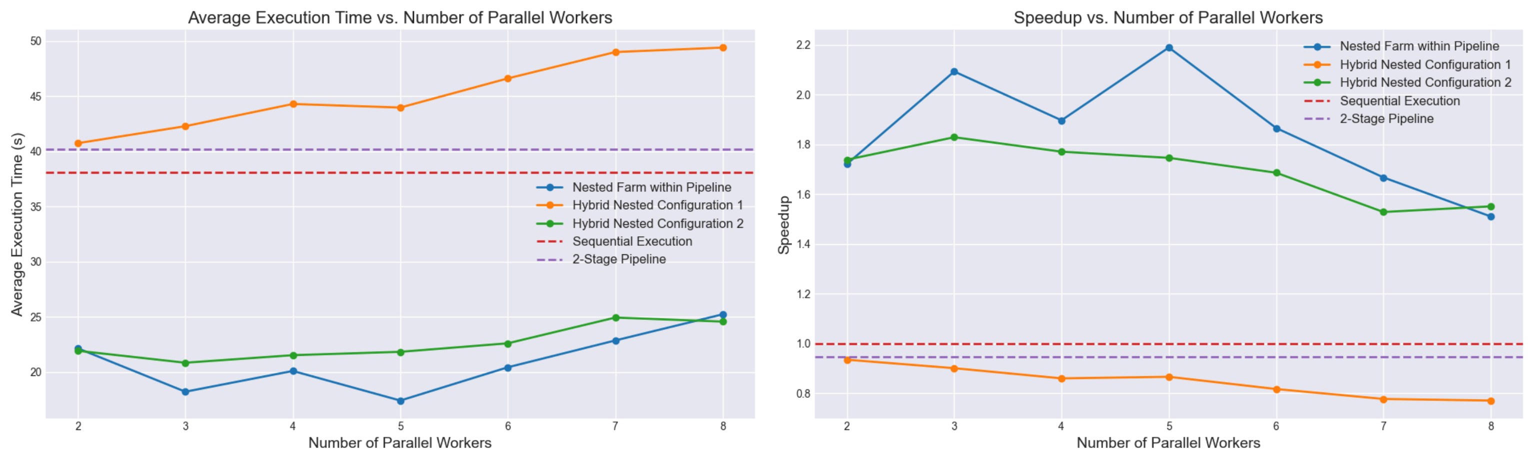


Figure 4: A graph showing the trends in Average Execution Time/Speedup vs Number of Parallel Workers.

Looking at Figure 4, the sequential execution has an average execution time of 38.1009 seconds, whereas the 2-stage pipeline configuration shows a slight slowdown compared to sequential execution, with a speedup of 0.95x. The **Nested Farm within Pipeline** performance improves as the number of parallel workers increases, reaching a maximum speedup of 2.19x with 5 workers. However, with some fluctuation, the speedup decreases with more workers, indicating potential load imbalances (causing fluctuation), and increased contention between threads, causing the slowdown.

The **Hybrid Nested Configuration 1** configuration performs poorly compared to the sequential execution, with the speedup scaling inversely to the number of workers. As mentioned previously, as the number of workers increases in Stage 1, there is a greater chance of contention as more workers are attempting to gain access to the single input queue of the **Pipe** in Stage 2.

In terms of the **Hybrid Nested Configuration 2**, the configuration shows some improvement over the sequential execution, with speedups ranging from **1.53x** to **1.83x**. However, speedup does not consistently improve with the increase in workers, suggesting limited scalability.

Overall, the convolution example has been proven to demonstrate better scalability and performance improvements compared to the Fibonacci example, when using Para-Pat. In terms of configurations, the **Hybrid Nested Configuration 2 (Pipe -> Farm(N))** has been shown to perform exceptionally well for the convolutional example, achieving significant speedups as we increase parallel workers. On the other hand, the Fibonacci example shows limited speed-ups and scalability across all configurations.

4. Comparison with Another Library

4.1 Overview

To ensure a comprehensive evaluation of the Para-Pat library, we will compare it with the FastFlow C++ parallel programming framework [1]. The comparison between Para-Pat and FastFlow will focus on three main aspects: **design, architecture, and performance**.

4.2 Design and Architecture Comparison

Both Para-Pat and FastFlow can provide high-level parallel patterns to simplify the development of parallel applications. However, one can argue their differences can be broken down into several components:

- **Model/Approach:** The Para-Pat library follows a task-based model, where we encapsulate the data into ‘Task’ structs, and it is passed between stages in the pipeline by **Worker** threads. FastFlow follows a stream-based programming model, where nodes process input tasks and produce output tasks, these approaches are the same.

- **Pattern Implementation:** The Para-Pat library has implemented farm and pipeline patterns with C++, with the help of pthreads. The main aim is to allow these patterns to be modular and easily composable, which allows for easy code integration. On the other hand, FastFlow provides a wide range of parallel patterns, including the ones present in Para-Pat, but also map, reduce, stencil, and more. TaskFlow implements these patterns using C++ templates, with greater optimisations. In terms of extensibility, FastFlow also allows users to define custom patterns, Para-Pat may require modifications to the core library code to do this.

- **Synchronisation Mechanisms:** Our library uses pthread mutexes to synchronise and coordinate between threads, with the help of thread-safe queues to allow communication between stages and workers. On the other hand, FastFlow uses lock-free queues and atomic operations for synchronisation, reducing contention and therefore improving scalability. Our library uses round-robin to distribute tasks, but it appears FastFlow uses a work-stealing scheduler to balance the load among workers.

4.3 Performance Comparison

To compare the performance of Para-Pat and FastFlow, we used inspiration from the documentation and the '/tests' (examples of how to use the library) folder of the FastFlow library allowing us to incorporate FastFlow. developing our example file: **'fibonacci_ff.cpp'**. We modified these files to use FastFlow patterns instead of para-pat and measured the execution time and speedup achieved.

- *Input: Fibonacci(900090000), 20 Tasks (iterations)*
- *Where Stage1(n) -> Stage2(n) represents the order* ***payload1() -> payload2()***

	Para-Pat		FastFlow		
Configuration	**Average Execution Time (s)**	**Speed Up**	**Average Execution Time (s)**	**Speed Up**	**FastFlow Relative Speedup**
Sequential Execution (Baseline) ***(Ran on C compiler)***	**38.1009**	**1.00x**	-	-	**-**
2-Stage Pipeline (Pipe -> Pipe)	40.2357	0.95x	21.814	1.75x	**1.84x**
Number of Parallel Workers = 2					
Nested Farm within Pipeline (Farm(2) -> Farm(2))	22.1412	1.72x	11.8800	3.21x	**1.87x**
Hybrid Nested Configuration 2 (Pipe -> Farm(2))	21.9197	1.74x	21.5252	1.77x	**1.02x**
Number of Parallel Workers = 3					
Nested Farm within Pipeline (Farm(3) -> Farm(3))	18.2088	2.09x	8.996	4.24x	**2.03x**
Hybrid Nested Configuration 2 (Pipe -> Farm(3))	20.8372	1.83x	21.4262	1.78x	**0.97x**
Number of Parallel Workers = 4					
Nested Farm within Pipeline (Farm(4) -> Farm(4))	20.0909	1.90x	8.1431	4.69x	**2.47x**
Hybrid Nested Configuration 2 (Pipe -> Farm(4))	21.5218	1.77x	21.8141	1.75x	**0.99x**
Number of Parallel Workers = 5					
Nested Farm within Pipeline (Farm(5) -> Farm(5))	17.4074	2.19x	7.1224	5.35x	**2.44x**
Hybrid Nested Configuration 2 (Pipe ->	21.8264	1.75x	23.408	1.63x	**0.93x**

Farm(5))					
Number of Parallel Workers = 6					
Nested Farm within Pipeline (Farm(6) -> Farm(6))	20.421	1.87x	6.6729	5.71x	**3.05x**
Hybrid Nested Configuration 2 (Pipe -> Farm(6))	22.5989	1.69x	24.1712	1.58x	**0.93x**
Number of Parallel Workers = 7					
Nested Farm within Pipeline (Farm(7) -> Farm(7))	22.8568	1.67x	5.8148	6.54x	**3.92x**
Hybrid Nested Configuration 2 (Pipe -> Farm(6))	24.9329	1.53x	24.546	1.55x	**1.01**
Number of Parallel Workers = 8					
Nested Farm within Pipeline (Farm(8) -> Farm(8))	25.2292	1.51x	8.7957	4.33x	**2.87x**
Hybrid Nested Configuration 2 (Pipe -> Farm(8))	24.5682	1.55x	25.7266	1.48x	**0.95x**

Table 4: Showing the Average Execution Time and Speed Ups for the 'Fibonacci' example.

The FastFlow implementation provides significant speed-ups on average, with the highest speed-up being with the **Nested Farm within Pipeline** with 7 workers (**3.92x more than ParaPat).** However, Para-Pat seems to have an advantage in most of the **Hybrid Nested Configuration 2 (Pipe -> Farm(N))** where it appears that potential overheads in the FastFlow have resulted in minor slowdowns, compared to the significant boosts that FastFlow offers in the other configurations.

5. Evaluation and Conclusion

The Para-Pat library can be seen to successfully implement the Farm and Pipeline patterns, with the use of pthreads, providing a high-level interface for parallel code. The performance analysis section, with the given examples, demonstrates the potential speedups when using these patterns, particularly in scenarios with coarse-grained parallelism and minimal contention.

For the convolution example, the **Hybrid Nested Configuration 2 (Pipe -> Farm(N))** was shown to have the best performance, achieving up to a **5.11x** speedup with 8 parallel workers. However, the Fibonacci example showed more of a limited scalability across all configurations, with a maximum speedup of **2.19x** with 5 workers. This shown discrepancy highlights the importance of workload granularity, and how contention and load imbalances can affect performance.

Compared to the FastFlow parallel programming framework, Para-Pat takes advantage of a task-based programming model, implementing parallel patterns using pthreads, with synchronisation relying on mutex locks and thread-safe queues. Whilst FastFlow offers a wider range of patterns and optimisations, however, Para-Pat's simplicity and modularity, allow for a quicker integration into existing C++ code.

The performance comparison with FastFlow has revealed that, for the Fibonacci example, FastFlow consistently outperformed Para-Pat across various worker configurations, potentially due to its work-stealing scheduler and the lock-free synchronisation that it uses instead. This suggests that FastFlow may be better suited for scenarios with more fine-grained parallelism, where there is increased contention.

5.1 Future iterations

Overall, the Para-Pat library can be shown to provide an easy foundation for parallelising code, using just the Farm and Pipeline patterns. Future iterations could focus on more advanced task scheduling, lock-free synchronisation, and extending the support for more parallel patterns such as Stencil, etc. Perhaps even automated performance tuning, where the number of workers used in stages dynamically changes based on runtime feedback could be an interesting development.

6. References

[1] http://calvados.di.unipi.it/ (FastFlow)
[2] https://www.it.uu.se/research/upmarc/events/120928/Aldinucci.pdf