University of St Andrews

CS4204 Coursework 2

# Parallel Patterns

*Author:*
150008022

April 28, 2019

# Goal

To implement and evaluate a library for parallelising C programs, using PThreads, locks, and queues.

# 1 Blocking Queue

The implementation of parallel task handlers would have to accumulate their outputs and buffer their inputs in a thread safe way, and so a blocking queue was necessary. `queue.h` provides an interface for a queue to be used by other components, and `queue.c` provides an implementation.

The queue is a linked list structure, where each node in the list has a reference to the element next in line. The queue structure tracks the last in line for adding new nodes, and the first in line for removing nodes. A conditional variable is used to notify threads waiting to add or remove elements that the size of the queue has changed, and a normal mutex is used to ensure that only one node is able to perform the addition or removal at a time (effectively creating a monitor). Figure 1 shows a queue struct with a series of node structs.

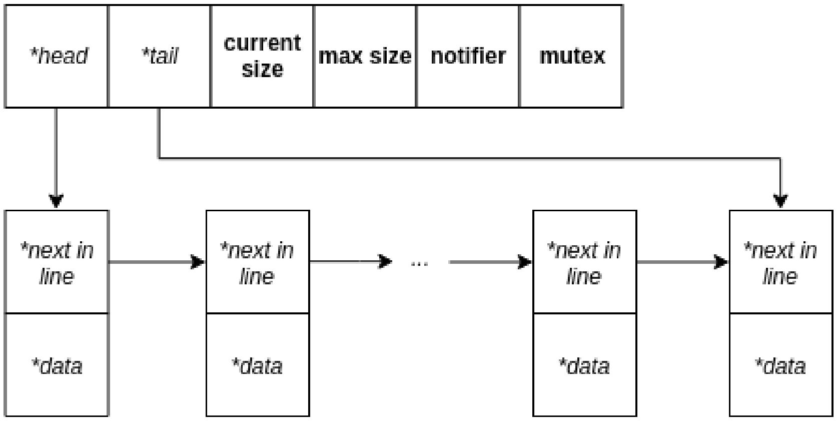
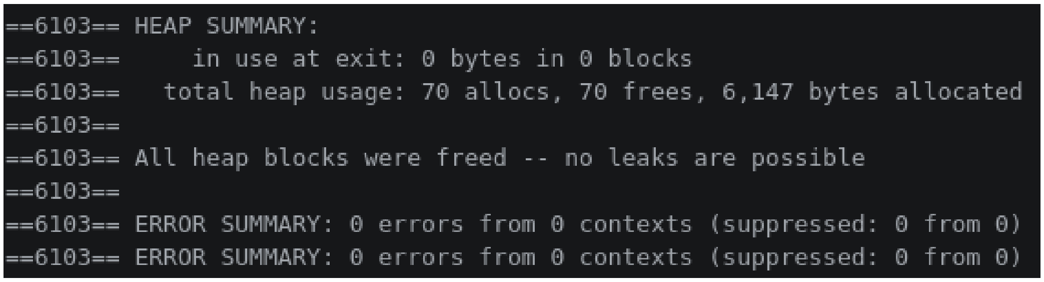


Figure 1: Queue structure, showing head and tail pointers, and each member of queue referencing the next.

The blocking queue was tested for correctness by writing a simple test program that would create a queue, and spawn a large number of threads. Half of which would enqueue and the other dequeue elements and print what they did and when they were finished. If the queue was not thread safe, then threads would either block permanently due to the loss of elements in the list during insertion and removal, multiple threads would dequeue the same element, and elements could be left over despite an even number of inserts and removes. Whilst this is clearly not an exhaustive test, it was enough to provide confidence in the implementation. Use `make queuetest` then `./queuetest` to compile and run the tests. Figure 2 shows the output of `queuetest` when run using `valgrind --leak-check=full -v ./queuetest`.



```
==6103== HEAP SUMMARY:
==6103==     in use at exit: 0 bytes in 0 blocks
==6103==   total heap usage: 70 allocs, 70 frees, 6,147 bytes allocated
==6103==
==6103== All heap blocks were freed -- no leaks are possible
==6103==
==6103== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==6103== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 2: Valgrind output when running queue test.

## 2 Steps

The project was started with the intention of implementing nested parallel patterns. This required defining components that would be able to integrate both with pipelines and parallel farms. This component is referred to as a `Step`. Figure 3 shows visualises a step, with an input queue, a function to apply, and an output queue to place data once processed.

The step API is listed in `step.h`, and provides three methods: Step creation, destruction, and shutdown signalling. Step creation requires providing an input and output queue (that can't be the same queue), as well as the function to apply which could take any number of parameters and return anything.

Originally, each step would spawn a single worker thread on creation which would run a function that would continuously poll the input queue for data to process. Upon receiving `NULL`, the worker thread terminates. The shutdown signal method adds NULL to the given steps input queue in order to trigger this shutdown. If multiple steps share an input queue then a NULL would have to be supplied for each consuming worker thread to ensure they all terminated correctly.

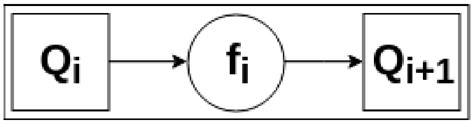


Figure 3: Single step without input queue $Q_i$, output queue $Q_{i+1}$ and a function $f_i$.

## 3 Filter

Inspired by the Java 8 Streams API (and since it is a useful feature), I also wanted to include the ability for a step to filter data. An example of filters in pipelines is a graphics engine removing features that are out of view from the pipeline to avoid unecessary processing.

Since NULL would terminate the worker thread for a step, I needed to differentiate between a filter function returning NULL (i.e. data was filtered) and the termination signal NULL. This was done by wrapping each instance of input data to the queues in a Task structure. A NULL task would be used as the termination signal. If a step was specified to be a filter, and its function outputted NULL, then the task would not be forwarded. This allowed NULL data to be passed through the pipeline also, and so developers using the library would not have to worry about their return data terminating the pipeline. Instead, they would return NULL from any function they wanted to act as a filter.

## 4 Ordering

If inputs were to be processed in parallel, then there are many situations where ordering would have to be restored to the outputs. This was achieved by grouping inputs into `Batch`s, as shown in figure 4. When the user submits an array of data to be processed, an array of tasks are created for each data point in the order of the original data. Then the batch is created to track the number of tasks, the number that are completed, and the number that remain after filtering has occurred.

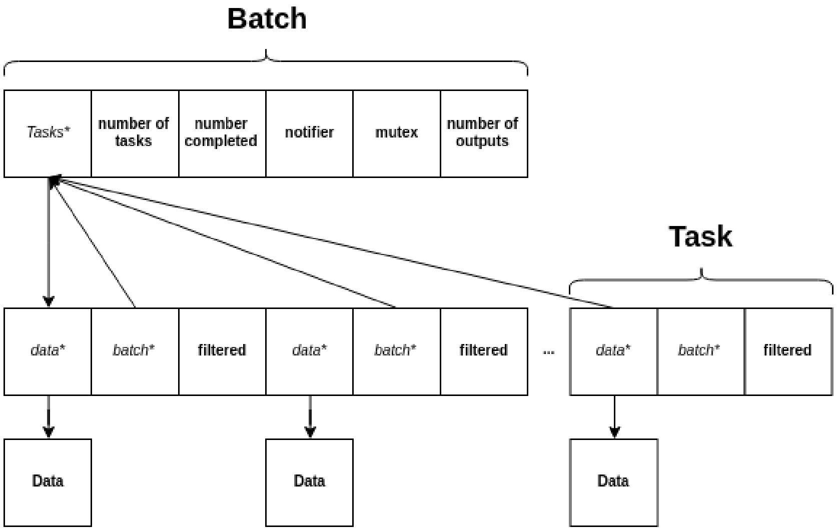


Figure 4: Structure of a task and batch.

When a filter step receives a NULL output, it informs its batch that it is completed. Since tasks could be completed on multiple threads, this increment had to be performed in a thread safe manner. Initially compare and swap was used, as experiments from the previous practical had proven that this would perform better in a multithreaded environment (i.e. less wait times). CAS was replaced with a mutex as it was necessary to notify any consumers of the pipeline that a batch was completed and the results were ready to extract.

In order to differentiate between completion due to being filtered and actual completion, a filtered flag was passed to the completed count incrementing function which if false would also increment the number of outputs. Steps would also know if they were the final step, and would also mark unfiltered tasks completed if they were final.

Tracking the actual number of ouputs allowed for the consumer of the pipeline to allocate only the memory they needed rather than allocating a buffer the same size as the inputs. The results were returned as a `Result` struct, which contained the number of results, and a pointer to an array of the results.

After incrementing the number of completed tasks, if the batch was completed then the number of outputs would be calculated using the number tasks marked as filtered, and any threads waiting on batch completion would be notified using the conditional variable.

# 5 Parallel Farm

The parallel farm pattern involves a pool of worker threads that remain idle until assigned a task by a coordinator. Once a worker thread completes its task, it then returns to being idle until a new task is assigned. Parallel farms are useful as they avoid the overhead involved in creating threads for each new input.

Since the `Step` struct already contained a single worker thread, it was simple to extend this to allow for a variable number of worker threads. Figure 5 shows a step that has three worker threads each applying a function to inputs from the same queue and placing the return values in the same queue. This allowed steps to operate as parallel farms. It was noted that since each thread would share the same input queue, increasing the number of worker threads would increase the level of contention when trying to dequeue and enqueue tasks due to the blocking queues locking mechansim. [1] provide a compare-and-swap queue implementation that could be used to reduce wait times when dequeing and enqueing to the input and output queues.

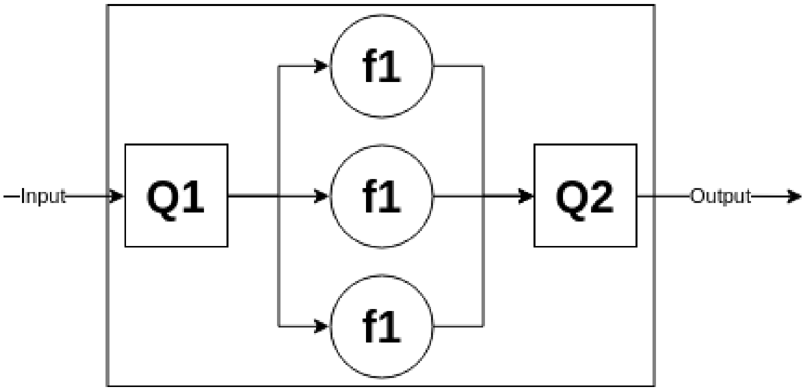


Figure 5: Step with multiple workers.

The parallel farm implementation and filter functionality was tested by iteratively building steps with a variable number of worker threads and a variable number of inputs and checking that the correct outputs were received. The normal step test would multiply each input by two, and the

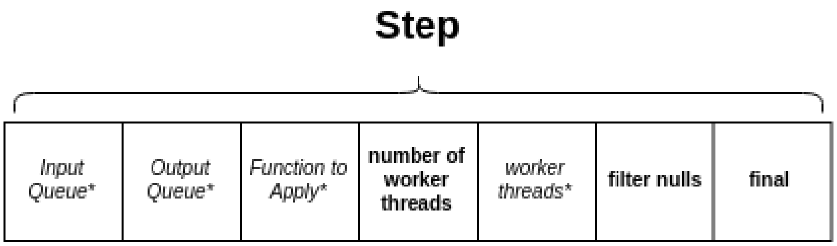
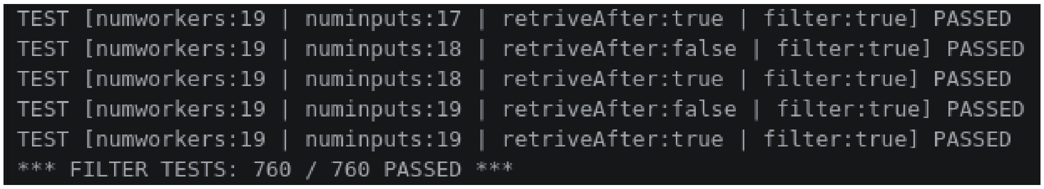


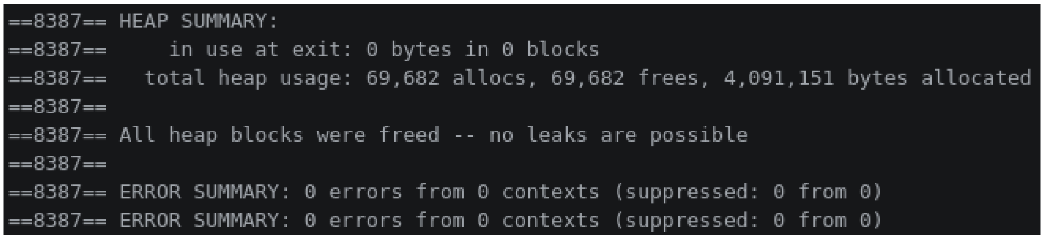
Figure 6: Components of `struct Step`.

filter test would remove even numbers from the input. Use `make steptest` then `./steptest` to compile and run the test. The output from an example test run is shown in figure 7. The tests were also run with valgrind to check that no memory leaks were occurring (figure 8).



```
TEST [numworkers:19 | numinputs:17 | retriveAfter:true | filter:true] PASSED
TEST [numworkers:19 | numinputs:18 | retriveAfter:false | filter:true] PASSED
TEST [numworkers:19 | numinputs:18 | retriveAfter:true | filter:true] PASSED
TEST [numworkers:19 | numinputs:19 | retriveAfter:false | filter:true] PASSED
TEST [numworkers:19 | numinputs:19 | retriveAfter:true | filter:true] PASSED
*** FILTER TESTS: 760 / 760 PASSED ***
```

Figure 7: Output of filter step tests using combinations of 1 to 19 worker threads and 0 to 19 inputs, where output retrieval is performed before and after signalling the worker threads to terminate.



```
==8387== HEAP SUMMARY:
==8387==     in use at exit: 0 bytes in 0 blocks
==8387==   total heap usage: 69,682 allocs, 69,682 frees, 4,091,151 bytes allocated
==8387==
==8387== All heap blocks were freed -- no leaks are possible
==8387==
==8387== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==8387== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 8: Valgrind output when running step test.

# 6 Pipeline

The pipeline pattern is where a series of functions are applied to some input in a given order. Often the analogy of a conveyor belt is used, as inputs can flow continuously and functions can be applied simultaneously to inputs that are at different stages in the pipeline.

Figure 9 shows how a pipeline was implemented for this submission. An array of functions are submitted on creation of the pipeline. The pipeline interface allows for inputs to be added to a queue, where they will wait to be processed, and for outputs to be polled from the outgoing queue.

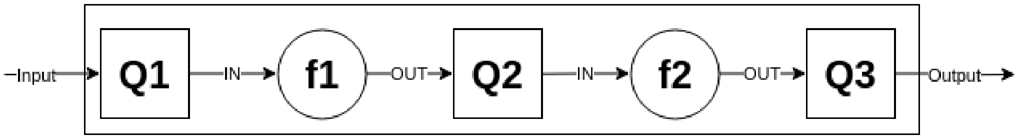


Figure 9: Pipeline abstraction, where Q signifies a blocking queue, and f signifigies a function being applied.

The pipeline was defined to consist of a series of steps. The thread for step $i$ would poll queue $Q_i$ for an input $x$ to process, compute $f(x)$, then add the output to queue $Q_{i+1}$. $Q_0$ and $Q_{n+1}$ are the input and output queues made accessible by the pipeline interface, where $n$ is the number of functions in the pipeline. The number of queues needed can be calculated using the number of functions $n + 1$.

One of the difficult design decisions was setting how to manage the size of the queues that would buffer data between each step in the pipeline. Figure 10 shows three situations: equal producers and consumers, more producers thatn consumers, and more consumers than producers. The relative speed of the producers and consumers would also have to be considered (e.g. how quickly the producer produces). The processing time of the pipeline would be dependent on the speed of the slowest step if queues are assumed to always have data to consume at each step (i.e. no waiting).

[2] describe and try to tackle the problem of choosing these buffer sizes. They conclude that in a homogeneous pipeline (where each step takes the same time), equal buffer sizes at each step is efficient. They also provide optimised buffer size allocations for pipelines where each step takes incrementally longer, for example.

I opted for the size of the input queue at each step to be equal to the number of consumer threads at the next step. This would at least ensure that as long as the producer equalled or outpaced the consumer, the the consumer would never have to wait for data to process. This could be wasteful in terms of memory however if the producer is unable to fill the queue due to the consumer being quicker.

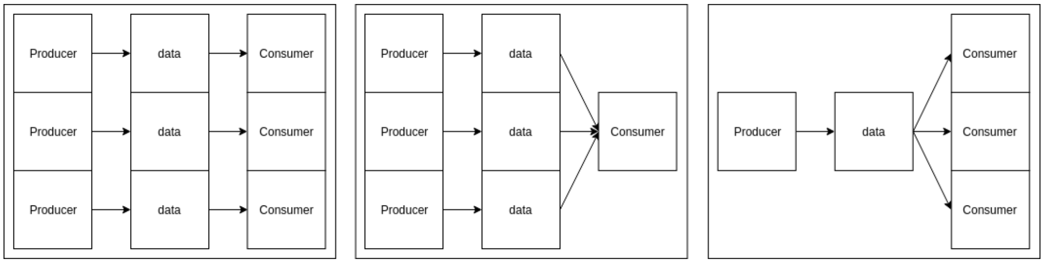


Figure 10: Scenarios when creating queues.

The pipeline allowed for data to be submitted in batches, and would return each batch in the order it was submitted. Consumers of the pipeline API could request for the next batch, which would block until it was available by using the `Batch` conditional variable mentioned earlier.

# 7 Nested Parallel Patterns

Since parallel farms were implemented as steps with multiple worker threads and pipelines were constructed using steps, it was simple to nest parallel farms within the pipelines. On creation of the pipeline, users submit the array of functions to apply, and an array that would define how many threads should be used at each step. This allowed for pipelines to be constructed like that in figure 11.

Tests were again written to ensure that these nested structures worked correctly by increasing the number of worker threads, number of inputs, and number of batches submitted to the pipeline, using the same functions that were used to test the parallel farms. `make pipetest;./pipetest` compiles and executes these tests. Figure 12 shows the output of `pipetest` when run using valgrind.

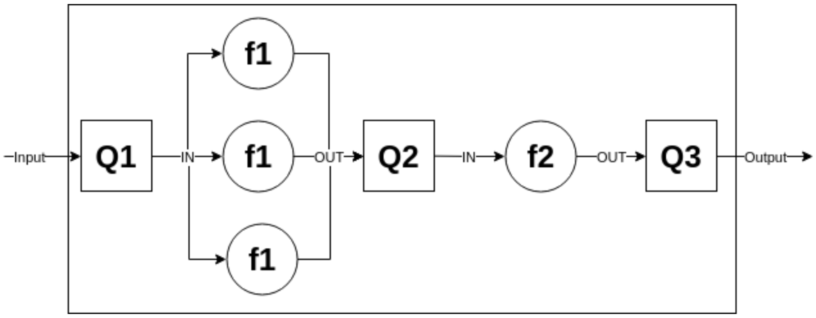
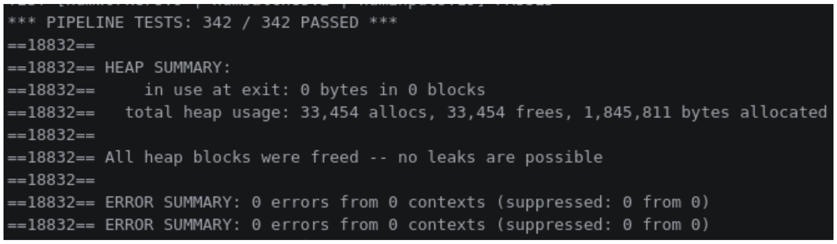


Figure 11: Pipeline with nested parallel farm at step one.



```
*** PIPELINE TESTS: 342 / 342 PASSED ***
==18832==
==18832== HEAP SUMMARY:
==18832==     in use at exit: 0 bytes in 0 blocks
==18832==   total heap usage: 33,454 allocs, 33,454 frees, 1,845,811 bytes allocated
==18832==
==18832== All heap blocks were freed -- no leaks are possible
==18832==
==18832== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==18832== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 12: Output of valgrind when running `pipetest`.

# 8 Experiment

## 8.1 Method

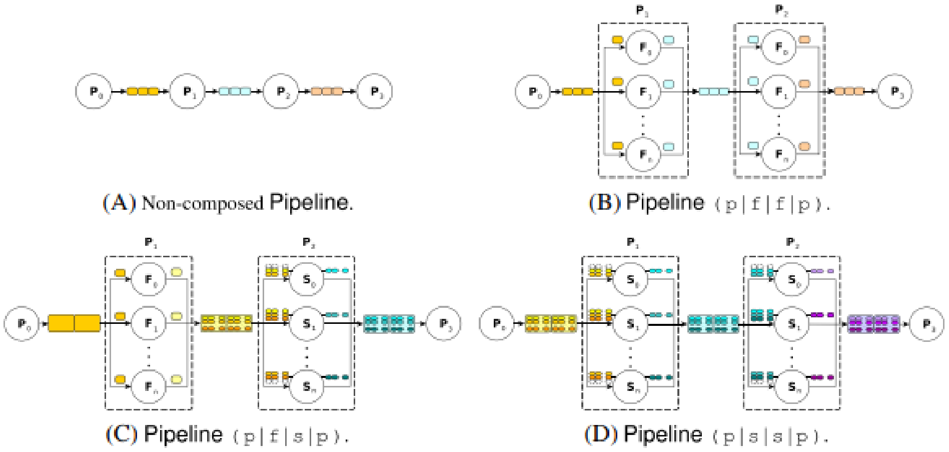
The provided convolution program was going to be used for evaluation as it was clearly appropriate to use a pipeline when processing the images iteratively. Unfortunately I was unable to get the C library to compile with `gpp`.

Instead, computation was simulated by varying sleep times for each function. Three phases of the geometry phase in a graphics pipeline were used in order to test the effectiveness of the implementation and compare it to a serial computation: clipping, transformation, and rasterisation. The first step removes objects outside of the visual volume from being processed later, the second involves scaling the content to the given viewport, and the last produces pixel data for each point in the resulting image.

Transformation was assumed to take the shortest time since it is a simple vector multiplication, clipping the next longest, then finally rasterisation. The processing time ratio adopted was 2:1:3. Each task was assumed to be able to be parallelised.

The evaluation test is based on the experiments in [3] which used different pipeline compositions as shown in figure 13. GRPPI was also used as an inspiration for the API design of this submission, so it seemed appropriate to evaulate it using a similiar experiment. Three stages were used, which would each either have 1 or 3 worker threads, written as `p` (pipe) if 1 and `f` (farm) if 3. Using the ratio discussed earlier, the relative processing delays were scaled up by factors of 10 from $1\mu s$ to $1000\mu s$. The configurations are summarised in table 1. For each configuration in each row of the table, the computation was also performed single threaded for comparison. The initialisation and teardown times were included in the pipeline execution times, which was expected to result in a cutoff point where the multithreaded pipeline became more efficient.

Larger data sets would have been used, but due to time constraints this was not possible. Input sizes greater than 100 began causing a considerable slowdown for both the linear and pipeline tests.



(A) Non-composed Pipeline.

(B) Pipeline ⟨p|f|f|p⟩.

(C) Pipeline ⟨p|f|s|p⟩.

(D) Pipeline ⟨p|s|s|p⟩.

Figure 13: Composed pipeline structures from [3].

| **Steps** | | | **Delay (micro seconds)** | | | Number of Inputs per Batch |
|---|---|---|---|---|---|---|
| *Step 1* | *Step 2* | *Step 3* | *Step 1* | *Step 2* | *Step 3* | |
| P | P | P | 2,20,...2000 | 1,10,...1000 | 3,30, ... 3000 | 1, 10, 100 |
| P | P | F | 2,20,...2000 | 1,10,...1000 | 3,30, ... 3000 | 1, 10, 100 |
| F | F | P | 2,20,...2000 | 1,10,...1000 | 3,30, ... 3000 | 1, 10, 100 |
| F | F | F | 2,20,...2000 | 1,10,...1000 | 3,30, ... 3000 | 1, 10, 100 |

Table 1: Configurations for experiment.

The expected results in order of lowest execution time were expected to be:

1. F—F—F
2. P—P—F
3. P—P—P and F—F—P
4. Serial

Since the fully parallelised approach would have no bottlenecks, it was expected to be the quickest as the number of inputs grew. The final step took the longest, therefore it was then expected that having only that step parallelised would perform similiarly to being fully parallel. P—P—P and F—F—P were expected to perform equally well due to sharing the bottleneck of the final step.

## 8.2 Results

Figure 14 shows the execution times of the different pipeline configurations and the serial implementation when the lowest delay was set to $1ns$. This was the configuration used in the first experiment, which showed that nanoseconds was far too small a delay. Figure 15 shows the same but with the lowest delay set to $1\mu$s and the number of inputs extended.

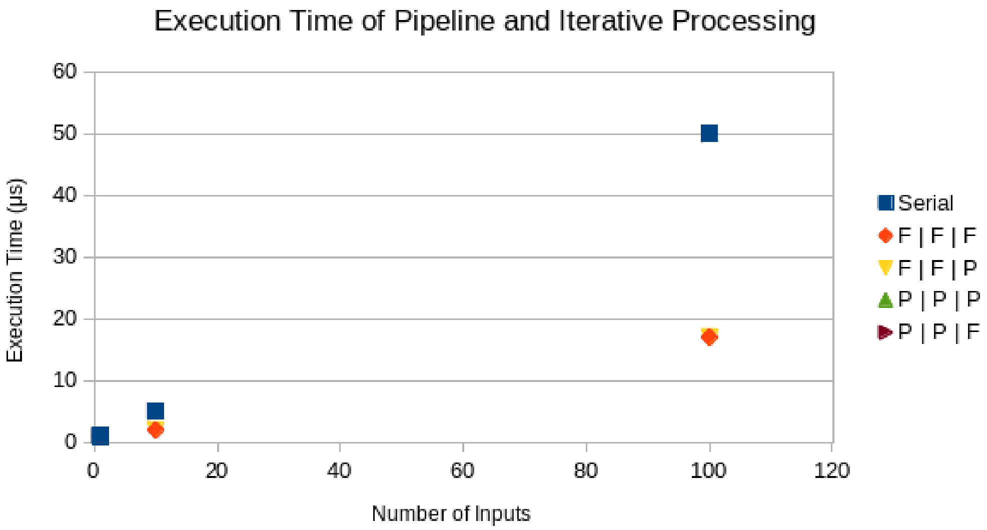


Figure 14: Execution times when base delay is $1\mu s$.

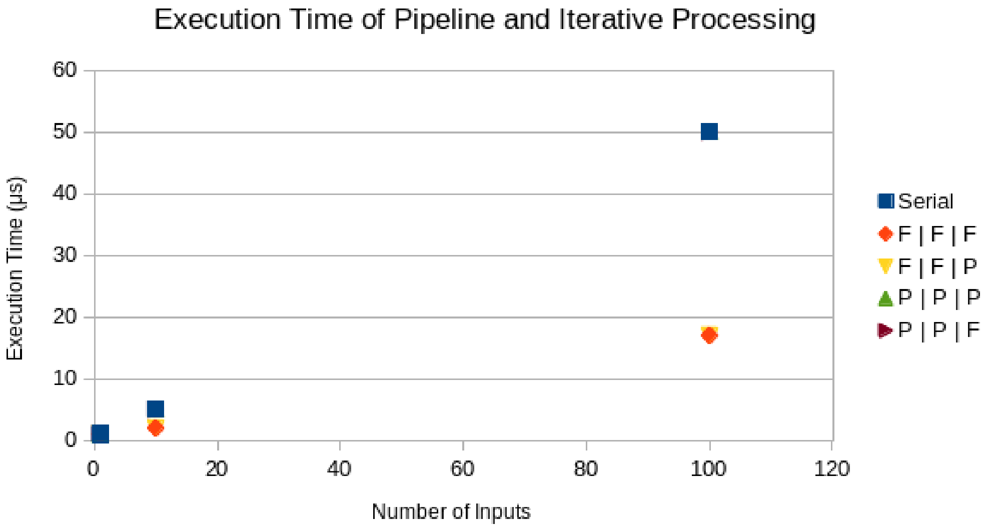


Figure 15: Execution times when base delay is $1\mu s$.

# 9 Evaluation

The fully parallel pipeline (F—F—F) performed better than the serial approach by far.

## 10 Conclusion

The implementation provides an API for building parallel farms, pipelines, and pipelines with nested parallel farms. The ability to filter inputs is also implemented in a generic way, as well as preservation of ordering in outputs. Tests are provided to prove safety and completeness of components, which are each organised and encapsulated in an attempt to apply OOD in C.

The practical greatly improved my understanding of multithreading in C and parallel design patterns, and also made me aware of the variety of APIs that provide a way to integrate these patterns into existing systems. Given more time, I would have like to reduce the memory footprint and the large number of system calls used in the implementation, which were likely the main contributor to the execution time of tests.

## References

[1] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A practical non-blocking queue algorithm using compare-and-swap. pages 470 – 475, 02 2000.

[2] A. Benoit, J. Nicod, and V. Rehn-Sonigo. Optimizing buffer sizes for pipeline workflow scheduling with setup times. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 662–670, May 2014.

[3] David Astorga, Manuel F. Dolz, Javier Fernández, and José García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, pages e4175–n/a, 05 2017.