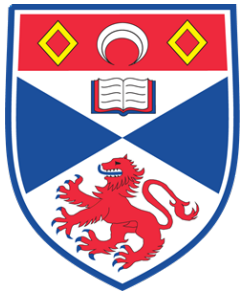
University of St Andrews

CS4204 Coursework 1

Parallel Patterns

Author:
150008022

April 27, 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.

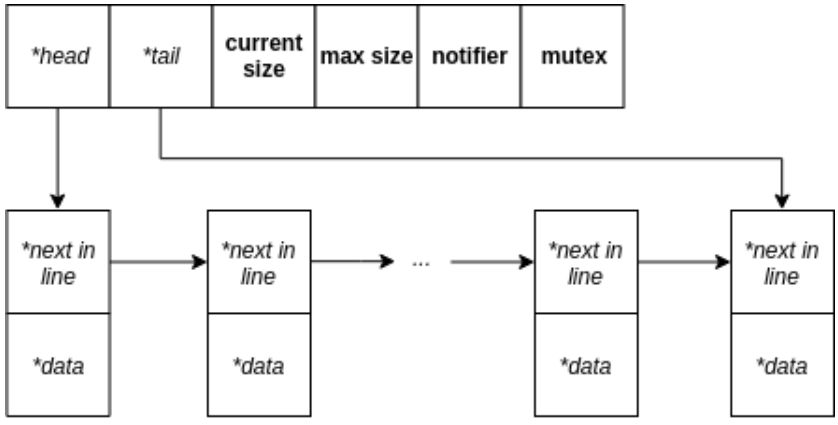


Figure 1: Queue structure, showing head and tail pointers, and each member of queue referencing the next.

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.

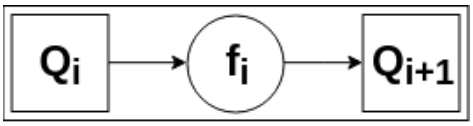


Figure 3: Single step without input queue Q_i, output queue Q_{i+1} and a function f_i.

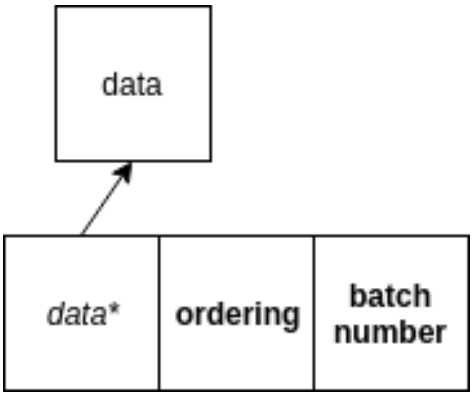


Figure 4: Structure of a task.

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 (Figure 4). 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.

The task wrapper was also extended to provide ordering and batch number values which could be used for recovering the input ordering once pro-

cessed by multiple worker threads.

Once satisifed with the single worker per step implementation, the ability to have multiple worker threads per step was then implemented. 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 simple 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.

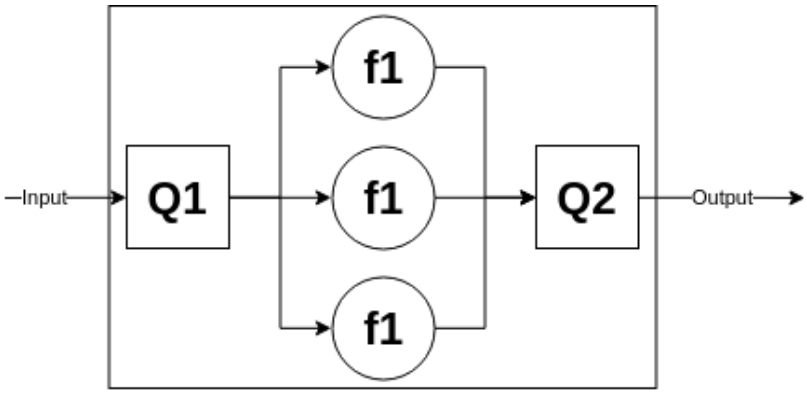


Figure 5: Step with multiple workers.

The step 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 multiple each input by two, and the 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 6. The tests were also run with valgrind to check that no memory leaks were occurring (figure 7).

```
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 6: 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 7: Valgrind output when running step test.

3 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 8 shows how a pipeline was implemented for this submission. An array of functions are submitted on creation of the pipeline, along with the number of worker threads that should run for each stage in 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.

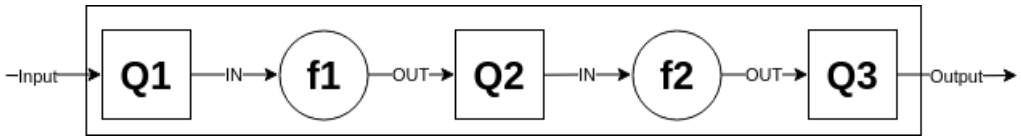


Figure 8: Pipeline abstraction, where Q signifies a blocking queue, and f signifigies a function being applied.

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 9 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).

[1] 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.

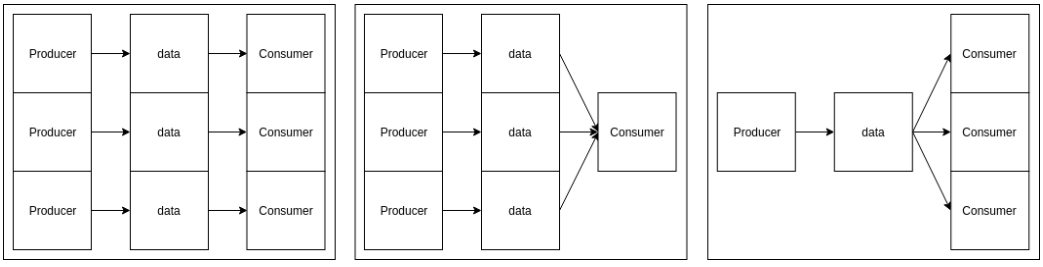


Figure 9: Scenarios when creating queues.

4 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.

As mentioned earlier, `Step`s already provided a method of having multiple worker threads processing from the same input queue, and the pipeline API already contained a way of constructing steps and submitting tasks. Therefore the farm API simply wrapped a pipeline instance with a single step and multiple worker threads.

Conclusion

References

[1] 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.