CS4204 W11 Practical
150015752
23th April 2019

Overview

The assignment of the practical is to develop C library that implements parallel patterns such as Farm and Pipeline using Pthreads. The goals can be broken down to these points:

- **Concurrent queue**: Implement queue from which each worker of a stage in pipeline pulls data concurrently.
- **Farm pattern**: The farm must allow the user to specify the number of workers and the worker function to be executed in parallel.
- **Pipeline pattern:** The pipeline must allow the user to declare a number of stages that themselves call user-defined pipeline functions.
- **Evaluation**: Evaluate performance of the farm pattern as a function of number of workers and size of data to process.

The practical implements all the features, evaluates the performance using a 3D graph and also implements pipelines that can have other pipelines as stages.

Running

To build the library from scratch, these commends have to be run in order:

```
gcc -o Farm.o -c Farm.c

gcc -o ConcQueue.o -c ConcQueue.c

gcc -o Stage.o -c Stage.h

gcc -o Worker.o -c Worker.c

gcc -o Pipeline.o -c Pipeline.c

gcc -o parapat.o -c parapat.h

ar rcs parapat.a ConcQueue.o Stage.o Worker.o Farm.o Pipeline.o parapat.o
```

To create an executable with the library, the `libparapat.a` file has to be in the main directory and this should be executed:

```
gcc myfile.c -L. -lparapat -lpthread -o myfile-exe
```

To run the testing, execute testing-exe.

```
./testing-exe
```

Design and Implementation

Queue

Concurrent queue requires atomic enqueueing and dequeueing. To achieve the atomicity, 2 pthread mutex locks are utilized, one for each operation. This is in order to be able to enqueue data into queue even when another thread is dequeueing and vice versa. To enqueue an element, the head mutex lock is acquired, the next element is established as the head, the former head element is retrieved and the mutex is unlocked. Similar process is established when dequeueing, except an element is pushed into a queue, rather than retrieved. The concurrent queue implementation also employs a use of divider element. The divider element is always in the queue and allows for a simplification of queue manipulations, because head and tail nodes do not need to be checked for NULL. Therefore, when dequeueing, and additional check has to be made for whether the dequeued element is not the divider element. If yes, then it is enqueued back and the next element is retrieved.

Interface of queue is defined in the *Queue.h* file and the concurrent queue implementation is in the *ConcQueue.c* file.

Pattern Design

Despite using functional language, a problem such as implementing parallel patterns is suited well for object-oriented approach. For example, the farm pattern consists of multiple workers to which data is handed to from a queue. These workers can be separate objects, and the farm another separate object that contains refrences to these workers. To achievie such functionality in C, structs are defined for each class. These structs hold necessary fields for each object in the code. Since a pipeline can contain different types of stages, such as farm or a single stage, these stages should have a common structure to inherit from. This structure is the *Stage* struct. The *Stage* struct contains necessary information for each stage, that is:

- **Stage name**: A substage is recognized by its stage name.
- **Input queue**: Concurrent queue for input data.
- **Output queue**: Concurrent queue for output data.

Each stage requires these fields for processing of data. To recognize different stages, each *Stage* struct has a stage name which specifies the stage type. To allow for functionality of this stage's struct, the original struct is cast to this stage's struct, such as *Farm* or *Worker* according to its stage name. Each of the stages is defined in its own C file, with an appropriate header. The header contains definition for the stage's struct and definitions for each public function of the stage. Each stage also contains appropriate *init* and *destroy* functions to malloc or free up memory of stage structs.

This implementation allows for a much more modular code that suits the nested nature of these parallel patterns.

Farm Pattern

Each *Farm* struct is initialized by a function for the workers and a number of workers to create. Each *Worker* struct is then initialized by this function. If the worker is a part of a farm, memory is allocated for *pthread_t* ID of the worker. The ID is then used for creating and joining thread on which this worker operates. If ID is set to NULL, the worker is recognized as a single stage.

The original function passed to the worker is wrapped in a wrapper function that runs the function in a while loop. For each iteration of the while loop, an element is dequeued from the input queue and passed as argument to the original function. The result of this function is then sent to the output queue. The while loop finishes, when the next element in the loop is end-of-stream element or NULL. Previous versions employed a *peek()* function which would look at a next element in the queue and only retrieve it if it's not end-of-stream element, however, this was an incorrect approach as there could be interleaving operations between peeking and dequeing of an element. Therefore, the peeking was thrown away and only the dequeing was left in. If a dequeued element is end-of-stream element, it is passed back to the input queue, in order for the farm collector to retrieve it. The wrapper function is then passed to the *pthread_create* function when creating a new thread.

The *Farm_run()* function runs all workers and the *Farm_collect()* function joins threads of all workers, therefore the function blocks until all data in the input queue has been processed and each worker is done. The former of these functions basically serves as the emitter, and the latter as the collector.

An alternative to this implementation would be to have emitter and collector as another stages, but this would raise the complexity and a decision has been made to keep the C code as simple as possible, but still modular.

Pipeline Pattern

Pipeline consists of a number of stages which are executed in order, however, each stage can run in parallel. This project implements pipeline that can have a farm, a worker, or another pipeline as a single stage. The number and the order of these stages is user-defined. The pipeline pattern is implemented in the *Pipeline.c* file. Because the *Pipeline* struct inherits from the *Stage* struct, a pipeline can also be a stage in another pipeline. Each stage is added to the end of a pipeline by the *Pipe_addStage()* function which also creates and assigns necessary input and output queues for each stage. The stages are kept in a *Stage* array in the *Pipeline* struct along with the number of stages. Once the pipeline is built, *void** array data is pushed into the pipeline input queue by the *Pipe_putToQueue()* function followed by the end-of-stream element. The pipeline is then executed by the *Pipe_run()* function that determines correct stage type based on each stage's name and executes an appropriate *run()* function. This approach basically immitates polymorphism. Output of the pipeline can then be retrieved into a *void** array by the *Pipe_getOutput()* function.

This approach creates a more modular code that also allows for an alternative to polymorphism.

A number of examples of how to initialize stages, add them to a pipeline and execute the pipeline are in the *testing.c* file.

Evaluation

Evaluation has been done for a single farm. I did not see a reason to evaluate any other pattern. Single worker is basically just a non-parallel execution and a pipeline can consist of a number of workers or farms, but these can be evaluated separately. The farm provides clearest view of parallel performance. To demonstrate, the farm pattern has been ran for exponentially raising number of input elements and linearly raising number of workers. Function passed to the workers was calculating a large number in Fibonacci sequence and then passing result into the output queue. The highest number of input elements was 2^{19}. The number of workers was incremented by 1 from 1 to 20. The result of the calculation can be seen in the Figure 1 below. Clearly, a single worker has the worst performance of more than 100s on the largest input dataset. Two workers can finish the same dataset almost twice as fast. With every added worker this number lowers but not by such large amount and the best performance of the largest dataset is actually achieved by 8 workers. After this number, the performance is more or less stabilized, and even starts to raise a bit towards the end. This might indicate congestion by a large number of threads trying to acquire a lock on the queues.

The graph also indicates that a change in dataset size yields similar results for farms with different number of workers higher than four.

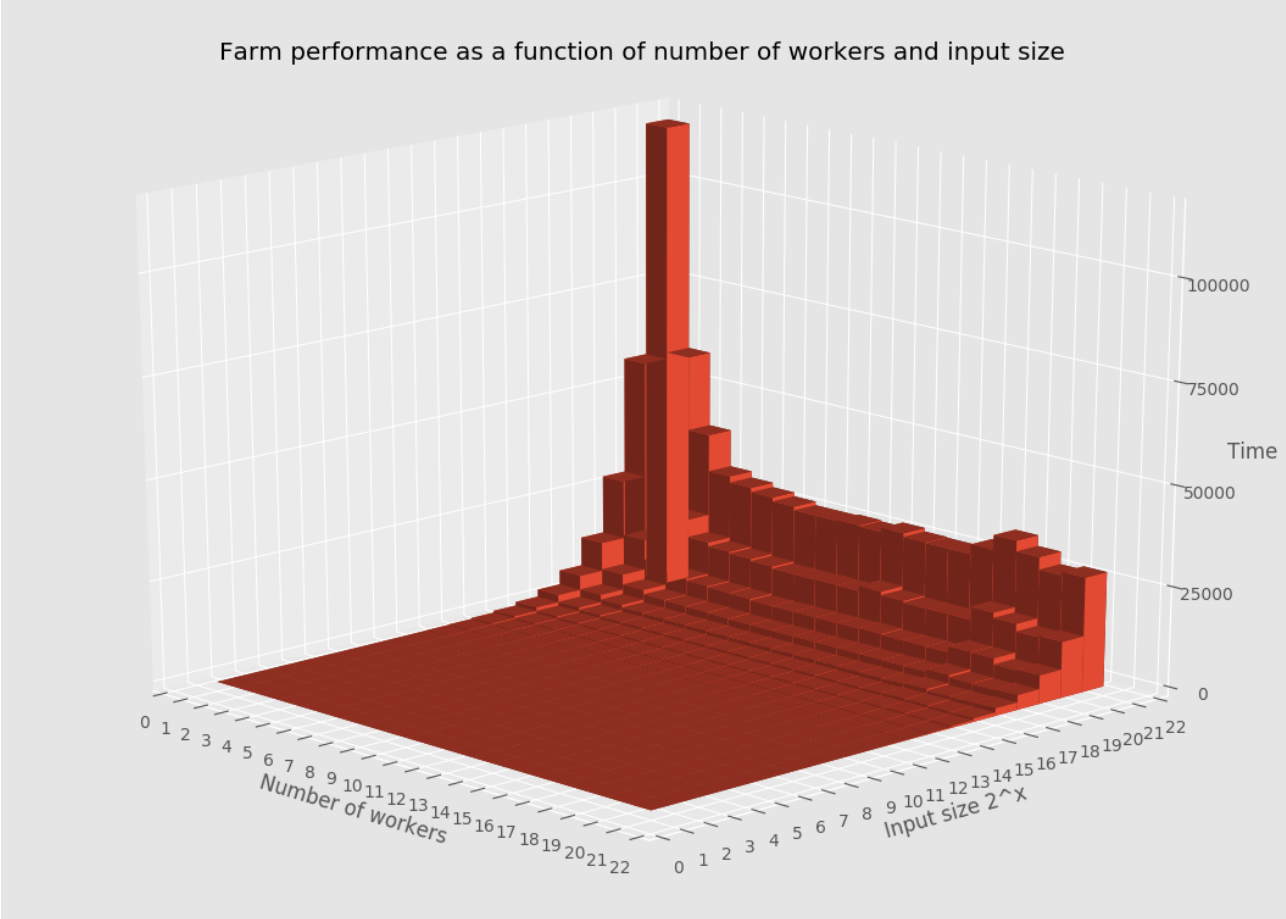

Figure 1: 3D graph of farm performance

Testing

Testing has been done using a number of test functions inplemented in the *testing.c* file that can be run by executing the *testing-exe* file.

Conclusion

This practical implements both the farm pattern and the pipeline pattern, tests them using a number of test functions and provides evaluation of the farm pattern's performance. To implement either of these patterns, a concurrent queue had to be developed first. Pipelines are also able to contain another pipelines as stages. All the designing choices and implementation are explained in the design section. For additional details, the code is thorougly commented. Evaluation presents a clear view of various farm pattern configurations in a 3D graph and analyses the results. The most difficult task of this practical was to implement the farm pattern, and to do so in C, with limited options for debugging. Given more time, I'd implement nested patterns and provide a more thorough evaluation of the project.