2020

Lock-free Batch Queue

User Manual

Eial Abo Yonis, Aia Wakid

Contents

Overview

Lock-free batch queue (BQ) is an extension to the regular Lock-free queue (MSQ), BQ can handle a sequence of operations in a batch, provides faster execution for operation sequences.

BQ demonstrates a significant performance improvement of up to 16x (depending on the batch lengths and the number of threads), compared to previous queue implementations.
BQ provides the regular Enqueue/Dequeue, also provides two new functions: FutureEnqueue/FutureDequeue which save the Enqueue/Dequeue operation aside from the queue as a pending operation, calling these functions will make the batch operation to be added later to the queue.

These two functions are the ones who make BQ more faster than MSQ, so its highly recommended to use the regular Enqueue/Dequeue operations only if we want the operation to change the queue immediately, and to use instead in the other cases the Future operations: if we can delay the operation a little bit or if we have a sequence of a relevant Enqueue operations we can put them in a batch operation to insert them to the queue together.

To insert a pending operation (Future operation) to the queue we can use the evaluate function.

*note than executing the functions Evaluate, regular Enqueue/Dequeue will add all the pending operations to the queue.

For more info. Look at the Functions section.

Compilation and Time measurements

Compilation

To use BQ data structure in your code you should include BQ.hpp by adding the next line code:

#include “BQ.hpp”

To compile BQ you should run the next command:

g++ -pthread BQ.cpp -std=c++11 -latomic

Time measurements

To make Time compression between BQ and MSQ run the next command line in your shell:

./Measurements x y

x – batch length.

y – how many times to repeat the test.

This test will run tests for different mount of threads on each MSQ and BQ.

For each thread we run a test for 2 seconds and count how many million operations is done to the data structure, we repeat this test y times and calculate the average of the y runs.

Batch length is x for BQ tests.

In MSQ tests we pick randomly Enqueue/Dequeue.

In BQ tests we pick randomly Enqueue/Dequeue/FutureEnqueeu/FutureDequeue.

To run Measurements, you need the next files: BQ.cpp, BQ.hpp, MSQ.cpp, MSQ.hpp, OpsTest.cpp.

After running the script, we will get the file: results.csv

Running main.py in the same directory with results will make 2 graphs:

1. How many million operations both BQ and MSQ can do in 1 second for different amount of running threads, this will be in file called: x_long_batch_Graph.png (x is batch length)
2. BQ_throughput/MSQ_throughput for different amount of running threads, this will be saved in file called: x_throughput.png (x is batch length).

Data Structures

Struct Future

```
Struct Future{
        void* result;
        bool isDone;
}
```

We get a future object as a return value of the two functions FutureEnqueue and FutureDequeue.

Every future object has two fields:

Bool isDone is a flag that represents that the FutureEnqueue/FutureDequeue that we called has Enqueued/Dequeued the Queue.

Void* result holds the return result of calling FutureDequeue if and only if isDone is true.

Functions

Init

void Init();

This function initializes the free-lock batch queue data structure, it should be called once before calling any other function.

Init_thread

void Init_thread();

This function initializes the calling thread, it should be called only once by each thread.

Enqueue

void Enqueue(void*);

This function gets a pointer to an item and add it to the end of the queue.

*note: this function will also execute all the pending FutureEnqueue/FutureDequeue that has been saved until this function has been called.

Dequeue

void* Dequeue();

This function removes the first item in the queue and returns it.

*note: this function will also execute all the pending FutureEnqueue/FutureDequeue that has been saved until this function has been called.

FutureEnqueue

Future* FutureEnqueue(void*);

This function gets a pointer to an item and save it aside to be added to the queue later.

The function returns a Future object o, o.isDone will be updated to be true when the item will be added to the queue.

FutureDequeue

Future* FutureDequeue();

This function saves that a Dequeue operation should be done later.

The function returns a Future object o, o.isDone will be updated to be true when the Dequeue operation is done, only after that o.result will save the value of item that we have dequeued from the queue.

Evaluate

void* Evaluate(Future*);

given a Future object o, which we get as a return value from the functions FutureEnqueue and FutureDequeue as mentioned above, this object represents an Enqueue/Dequeue operation that has been saved aside the queue and it didn't affect it yet.

Executing the function Evaluate on the Future object will run the Enqueue/Dequeue operation which o represents on the queue.

The function will update o->isDone and o->result will be the return value of the function.

*note: this function will also execute all the pending FutureEnqueue/FutureDequeue that has been saved until this function has been called.