About the project

This project implements the queue from the article "BQ: Lock-Free Queue with Batching" by Gal Milman et al.. The article also provides Linearizability Proof.

This queue functions as a regular lock-free queue, allowing thread-safe enqueue and dequeue methods.
In addition, this queue allows using future operations – enqueues and dequeues that can be delayed, for later execution.
Instead of executing each operation one by one, this queue optimizes the execution by reducing the number of atomic operations and executes all operations at once.
Using batching reduces the contention and improve scalability.
According to the article, this queue presents performance improvements of up to 16 times faster than an MSQ (lock-free queue without batching).

This project was implemented using the C++11 programming language.

The project uses the atomic operation **compare_exchange_weak**, as a compare and swap (CAS), which is mentioned in the article. Therefore, it includes the **atomic** library.
To allow the use of a 16-byte CAS, it is mandatory to use the “march=native” flag when compiling with CSL3 sever.

The project adds two additional methods to the implementation in the article regarding the thread data part, which does not change the logic of the queue. One is undoing a future operation, which allow to remove the last future enqueue or future dequeue that was added for later execution and was not executed yet. The second is getting a list (queue) of all the items that was dequeued when executing a batch.

User manual

Who Can Use the Project?

It is recommended to have prior parallel programming knowledge before using this project.
The project is beneficial for programmers that are implementing parallel programs and needs a thread safe queue.

What Solution the Project offers?

This project provides a Batch Queue which is Lock-Free, allowing exploit both batching and lock-free for having better performance compared to other queues for parallel programming.

How to Use the Project?

This project can be used by downloading the project files to your program and including the Queue.h.

Using Examples

One can use it for various possible programs. For example, it can be used in a ~~for~~ multi-cores programs using shared memory, like queue for the post office, doctors, etc.

Programmer Guide

Tutorial

Creating New Queue

Create a new Queue by calling its' constructor:

```
Queue<type>* queue_name = new Queue<type>(n);
```

Where type is the type name (int, double, char, etc.), queue_name is the name of the queue, and n is the number of threads that will be used by the queue at most.

For example, a queue named "queue1", contains integers and uses 16 threads, will be construct this way:

```
Queue<int>* queue1 = new Queue<int>(16);
```

Adding an Item to the Queue

If we would like to add an item to the queue (without batching or future operation), as we would add (or push) in a regular parallel queue, we would use the "Enqueue" method:

```
queue_name->Enqueue(&item, thread_id);
```

Where queue_name is the batch queue that was created before, item is an object of the type of the queue, and thread_id is the identifier of the thread that is calling the method.

For example, following the last example, thread #7 will create a new int with the value 7, and will add it to the queue:

```
int number = 7;
queue1->Enqueue(&number, 6); // queue1 = {7}
```

Removing an Item from the Queue

If we would like to remove an item from the queue (without batching or future operation), as we would remove (or pop) in a regular parallel queue, we would use the "Dequeue" method:

```
queue_name->Dequeue(thread_id);
```

Where queue_name is the batch queue that was created before and thread_id is the identifier of the thread that is calling the method.
This method will return the item that was removed, if the queue is not empty, otherwise it will return NULL

For example, following the last example, thread #2 will remove an item from the queue twice.

```
int* number1 = queue1->Dequeue(1); // *number1 = 7
int* number2 = queue1->Dequeue(1); // number2 = NULL
```

Future Enqueue

If we would like to create a future enqueue, we would use the "FutureEnqueue" method:

```
queue_name->FutureEnqueue(&item, thread_id);
```

Where queue_name is the batch queue that was created before and thread_id is the identifier of the thread that is calling the method.

For example, following the last example, thread #9 will create a new int, with the value 5, and will future-add it to the queue:

```
int number = 5;
queue1->FutureEnqueue(&number, 8); // queue1 = {}
```

Future Dequeue

If we would like to create future dequeue, we would use the "FutureDequeue" method:

```
queue_name->FutureDequeue(thread_id);
```

Where queue_name is the batch queue that was created before and thread_id is the identifier of the thread that is calling the method.

For example, following the last example, thread #9 will future-remove an item from the queue:

```
Future<int>* f = queue1->FutureDequeue(8); // queue1 = {}
```

Evaluate

If we would like to execute the batch, we would use "Evaluate" method.

```
queue_name->Evaluate(future_to_evaluate,thread_id);
```

Where queue_name is the batch queue that was created before, thread_id is the identifier of the thread that is calling the method and future_to_evaluate is the previous future item that was returned from FutureEnqueue or Future Dequeue. This method will execute all the future operations of this thread.

For example, following the last example, thread #9 will evaluate its last future operation:

```
// batch is {ENQ, DEQ}, f->isDone == false
queue1->Evaluate(f, 8); // queue1 = {}
// *(f->Result) == 5, f->isDone == true
```

Accessing f->Result before evaluating may yield an arbitrary result. After evaluating it will have a dequeue operation result, if it is indeed a FutureDequeue's Future.

Another example for the use of evaluate is getting all the items that were dequeued:

```
int num[6] = {1, 2, 3, 4, 5, 6};
Future<int>* f[12];
for (int i = 0; i < 6; i++) {
            f[i] =  queue1->FutureEnqueue(num+i,0);
}
queue1->Evaluate(f[0] ,0);
```

```
for (int i = 0; i < 6; i++) {
            f[i+6] = queue1->FutureDequeue(0);
}
std::queue<int*>* q1 = new std::queue<int*>();
queue1->Evaluate(f[6], q1 ,0); //q1 = {1,2,3,4,5,6}
```

UndoFuture

This method will be used if we would like to cancel the last future operation (only if we did not evaluate it):

```
Queue_name->UndoFuture(thread_id);
```

Where queue_name is the batch queue that was created, thread_id is the identifier of the thread that is calling the method.
If there are no future operations, this method returns immediately.

For example:

```
int number1 = 5, number2 = -6, number3 = 7;
Future<int> f;
f = queue1->FutureEnqueue(&number1, 8); // batch = {E}
queue1->FutureEnqueue(&number2, 8); // batch = {E, E}
queue1->FutureEnqueue(&number3, 8); // batch = {E, E, E}
queue1->UndoFuture(8); // batch = {E, E}
queue1->Evaluate(f, 8); // queue1 = {5, -6}
```

Documentation

Structs

At this part we would describe the data structures we created for this project, for any use of the Queues' Implementation.

Future

{result – T*, isDone – bool}

A Future contains a result , which holds the return value of the deferred operation that generated the future (for dequeues only, as enqueue operations have no return value) and an isDone Boolean value, which is true only if the deferred computation has been completed. When isDone is false, the contents of result may be arbitrary.

Node

{item – T*, next – atomic<Node*>}

A trivial node structure for creating linked list, except for using atomic pointer to the next, allowing the use atomic CAS.

BatchRequest

{firstEnq – Node*, lastEnq – Node*, enqsNum – unsigned int, deqsNum – unsigned int, excessDeqsNum – unsigned int}

A BatchRequest is prepared by a thread that initiates a batch and consists of the details of the batches' pending operations: firstEnq and lastEnq are pointers to the first and last nodes of a linked list containing the pending items to be enqueued; enqsNum, deqsNum, and excessDeqsNum are, respectively, the numbers of enqueues, dequeues and excess dequeues in the batch.

PtrCnt

{node – Node*, cnt – unsigned int}

PtrCnt contains a Pointer to a node, and a counter. A struct for the head/tail of the queue.

Ann

{batchReq – batchRequest, oldHead – PtrCnt, oldTail – PtrCnt}

An Ann object represents an announcement. It contains a BatchRequest instance, with all the details required to execute the batch operation it stands for. Thus, any operation that encounters an announcement may help the related batch operation complete before proceeding with its own operation.

In addition to information regarding the batch of operations to execute, Ann includes oldHead, the value of the head pointer (and dequeue counter) before the announcement was installed, and oldTail , an entry for the tail pointer (and enqueue counter) of the queue right before the batch is applied (i.e., a pointer to the node to which the batch's list of items is linked).

PtrCntOrAnn

{ptrCnt – PtrCnt || {tag – unsigned int, ann – Ann*}}

16-byte union that may consist of either PtrCnt or an 8-byte tag and an 8-byte Ann pointer. Whenever it contains an Ann, the tag is set to 1. Otherwise, ASQHead contains a PtrCnt (the tag overlaps PtrCnt .node, whose least significant bit is 0 since it stores either NULL or
an aligned address).

FutureOp

{type – {ENQ, DEQ}, future – Future*}

This struct note a future operation for the queue, Enqueue or Dequeue.

ThreadData

{opsQueue – Queue<FutureOp>, enqsHead – Node*, enqsTail – Node*, enqsNum – unsigned int, deqsNum – unsigned int, excessDeqsNum – unsigned int}

A threadData array holds local data for each thread. First, the pending operations details are kept, in the order they were called, in an operation queue opsQueue, implemented as a simple local non-thread-safe queue. It contains FutureOp items. Second, the items of the pending enqueue operations are kept in a linked list in the order they were enqueued by FutureEnqueue calls. This list is referenced by enqsHead and enqsTail (with no dummy nodes here). Lastly, each thread keeps record of the number of FutureEnqueue and FutureDequeue operations that have been called but not yet applied, and the number of excess dequeues.

Queue

{ASQHead – atomic<PtrCntOrAnn>, ASQTail – atomic<PtrCnt>, threadData – ThreadData*, numberOfThreads – unsigned int}

Similarly to MSQ, the shared queue is represented as a linked list of nodes in which the first node is a dummy node, and all nodes thereafter contain the values in the queue in the order they were enqueued. We maintain pointers to the first and last nodes of this list, denoted ASQHead and ASQTail respectively (which stand for Atomic Shared Queue's Head and Tail). Batch operations require the size of the queue for a fast calculation of the new head after applying the batch's operations. To this end, Queue maintains counters of the number of enqueues and the number of successful dequeues applied so far. These are kept size-by-side with the tail and head pointers respectively, and are updated atomically with the respective pointers using a double-width CAS. This is implemented using a Pointer and Count object (PtrCnt , that can be atomically modified) for the head and tail of the shared queue.

Methods

EnqueueToShared

EnqueueToShared appends an item after the tail of the shared queue, using two CAS operations, in a similar manner to msq's Enqueue: it first updates ASQTail .node->next to point to a node consisting of the new item, and then updates ASQTail to point to this node. An obstructing operation might enqueue its items concurrently, causing the first CAS (in Line 5) to fail. In this case, EnqueueToShared would try to help complete the obstructing operation, before starting a new attempt to enqueue its own item.

DequeueFromShared

DequeueFromShared. If the queue is not empty when the dequeue operation takes effect, DequeueFromShared extracts an item from the head of the shared queue and returns it; otherwise it returns NULL.

HelpAnnAndGetHead

HelpAnnAndGetHead. This auxiliary method assists announcements in execution, as long as
there is an announcement installed in ASQHead.

ExecuteBatch

ExecuteBatch. ExecuteBatch is responsible for executing the batch. Before it starts doing so, it checks whether there is a colliding ongoing batch operation whose announcement is installed in ASQHead. If so, ExecuteBatch helps it complete. Afterwards, it stores the current head in ann, installs ann in ASQHead and calls ExecuteAnn to carry out the batch.

ExecuteAnn

ExecuteAnn is called with ann after ann has been installed in ASQHead. ann's oldHead field consists of the value of ASQHead right before ann's installation. ExecuteAnn carries out anns's batch. If any of the execution steps has already been executed by another thread, ExecuteAnn moves on to the next step. Specifically, if ann will have been removed from ASQHead by the time ExecuteAnn is executed, ann's execution will have been completed, and all the steps of this run of ExecuteAnn would fail and have no effect.
ExecuteAnn first makes sure that ann's enqueued items are linked to the queue, in the while
loop in Line 39. If they have already been linked to the queue, and the old tail after which they were linked has also been recorded in ann, it follows that another thread has completed the linking, and thus we break out of the loop in Line 43. Otherwise, we try to link the items by performing a CAS operation on the next pointer of the node pointed to by the tail in Line 44. In Line 45 we check whether the items were linked after tail , regardless of which thread linked them. If so, we record tail , to which the items were linked, in ann. Otherwise, we try to help the obstructing

enqueue operation complete in Line 50, and start over with a new attempt to link the batch's items.
The next step is ASQTail 's update in Line 52. There is no need to retry it, since it fails only
if another thread has written the same value on behalf of the same batch operation. Lastly, we call UpdateHead to update ASQHead to point to the last node dequeued by the batch. This update uninstalls the announcement and completes its handling.

UpdateHead

The UpdateHead method calculates successfulDeqsNum. It then determines the new head according to the following optimization: If the number of the batch's successful dequeues is at least the size of the queue before applying the batch, which implies that the new dummy node is one of the batch's enqueued nodes, the new head is determined by passing over successfulDeqsNum − oldQueueSize nodes, starting with the node pointed to by the old tail. Otherwise, it is determined by passing over successfulDeqsNum nodes, starting with the old dummy node. Finally, UpdateHead updates ASQHead (and as in ASQTail 's update, there is no need to retry the CAS).

GetNthNode

This method gets a node and a number, and return the node in the n'th place to the node.

Enqueue

Enqueue checks whether the thread-local operation queue opsQueue is empty. If it is, it directly calls EnqueueToShared. Otherwise, to satisfy EMF-linearizability, the pending operations in opsQueue must be applied before the current Enqueue is applied. Hence, Enqueue calls FutureEnqueue with the required item, which in turn returns a future. It then calls Evaluate with that future. This results in applying all preceding pending operations, as well as applying the current operation.

Dequeue

The implementation of Dequeue is similar to the one of Enqueue. If Dequeue succeeds, it returns the dequeued item, otherwise (the queue is empty when the operation takes effect) it returns NULL.

FutureEnqueue

FutureEnqueue calls AddsToEnqsList, and also updates the local numbers of pending enqueue operations. In addition, FutureEnqueue calls to RecordOpAndGetFuture. A pointer to the Future object encapsulated in the created FutureOp will be returned by the method, so that the caller could later pass it to the Evaluate method.

AddToEnqsList

AddsToEnqsList adds the item to be enqueued to thread's list of items pending to be enqueued. This list will be appended directly to the end of the shared queue's list of nodes when a batch operation is executed by this thread.

This is the reason why these items are stored in a linked list of nodes rather than directly in opsQueue.

RecordOpAndGetFuture

RecordOpAndGetFuture enqueues a FutureOp object representing an enqueue operation to the thread's opsQueue, and return a pointer of this FutureOp object.

FutureDequeue

FutureDequeue updates the local numbers of pending dequeue operations and excess dequeues. The latter is calculated based on Lemma 5.3 on the paper. FutureDequeue then enqueues a FutureOp object representing a dequeue operation to the thread's opsQueue. Like FutureEnqueue, FutureDequeue returns a pointer to a Future object.

Evaluate

Evaluate receives a future and ensures it is applied when the method returns. Future may be evaluated by its creator thread only.
If the future has already been applied from the outset, its result is immediately returned. Otherwise, it calls ExecuteAllPending.
It is possible to call evaluate with a queue parameter, and all the dequeued item in this evaluation is inserted to the queue.

ExecuteAllPending

In this method, all locally-pending operations found in threadData.opsQueue are applied to the shared queue at once. After the batch operation's execution completes, while new operations may be applied to the shared queue by other threads, the batch operation results are paired to the appropriate futures of operations in opsQueue.
If opsQueue consists of at least one enqueue operation, the batch operation's execution and the results-to-futures pairing are accomplished by calling ExecuteBatch and PairFuturesWithResults, respectively. If all pending operations are dequeues, we pursue a different course of action, calling PairDeqFuturesWithResults.

PairFuturesWithResults

PairFuturesWithResults receives the old head. It simulates the pending operations one by one according to their original order, which is recorded in the thread's
opsQueue. Namely, it simulates updates of the head and tail of the shared queue. This is done by advancing nextEnqNode (which represents the value of tail->next in the current moment of the simulation) on each enqueue, and by advancing currentHead on dequeues that occur when the queue in its current state is not empty. The simulation is run in order to set results for future objects related to the pending operations and mark them as done.

ExecuteDeqsBatch

The ExecuteDeqsBatch method first assists a colliding ongoing batch operation if there is any. It then calculates the new head and the number of successful dequeues by traversing
over the items to be dequeued in the loop in Line 152. If there is at least one successful dequeue, the dequeues take effect at once using a single CAS operation in Line 160. The CAS pushes the shared queue's head successfulDeqsNum nodes forward

PairDeqFuturesWithResults

PairDeqFuturesWithResults pairs the successfully-dequeued-items to futures of the appropriate operations in opsQueue. The remaining future dequeues are unsuccessful,
thus their results are set to NULL.

Additional methods/functions:

max

simple max function which returns the greater number of two numbers.

UndoFuture

UndoFuture cancel the last FutureEnqueue or FutureDequeue. It takes out the future dequeue/enqueue from the opsQueue and calculate the excessDeqsNum. If the last operation was "enqueue", it removes the last item from the thread's enqueue list.