2020

Lock-free Batch Queue

Programmers guide

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 that will influence the queue immediately, 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.
Unlike standard operations, deferred operations need not be applied to the shared queue before their responses occur. When a future method is called, its details are recorded locally together with previous deferred operations that were called by the same thread. A Future object is returned to the caller, who may evaluate it later.
Deferred operations allow to apply pending operations in batches: BQ delays their execution until the user explicitly evaluates a future of one of them or calls a standard method. When that happens, all pending operations of the same thread are gathered to a single batch operation. This operation is then applied to the shared queue. Afterwards, the batch execution is finalized by locally filling the futures' return values. This mechanism reduces synchronization overhead by allowing fewer accesses to the shared queue, as well as less processing in the shared queue – thanks to the preparations performed locally by the initiating thread during the run of each future operation, and the local pairing of applied futures with results following the batch execution.
Whenever a deferred enqueue operation is called, the executing thread appends its item to a local list. This way, when the thread has to perform a batch operation, the list of nodes to be linked to the shared queue's list is already prepared.
The key to applying all operations of a batch at once to the shared queue, is to set up a moment in which the state of the queue is "frozen". Namely, we establish a moment in which we hold both ends of the queue, so that we know its head and tail, and its size right before the batch takes effect.
This way we can unambiguously determine the queue's shape after applying the batch, including its new head and tail. We achieve a hold of the queue's ends by executing a batch operation in stages, according to the following scheme.
The thread first creates an announcement describing the required batch operation. An announcement is an auxiliary object used to announce an ongoing batch operation, so that other threads will not interfere with it but rather help it complete. Then, the thread modifies the shared queue's head to point to the created announcement. This marks the head so that further attempted dequeues will help the batch execution to be completed before executing their own operations. Now we hold one end of the queue.
Next, the initiating thread or an assisting thread links the list of items, which the initiating thread has prepared in advance, after the shared queue's tail. This determines the tail location after which the batch's items are enqueued. Thus, now we hold both ends of the queue, as required. We then update the shared queue's tail to point to the last linked node.
As a last step that would uninstall the announcement and finish the batch execution, we update the shared queue's head. It is possible that during the execution of the required enqueues and dequeues the queue becomes empty and that some of the dequeues operate on an empty queue and return NULL. We make a combinatorial observation that helps quickly determine the number of non-successful dequeues. This number is used to determine the node to which the queue's head points following the batch execution. By applying this fast calculation, we execute the batch with minimal interference with the shared queue, thus reducing contention.

Data Structures

Struct Future

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

Future object represents a pending operation (FutureEnqueue/FutureDequeue), it contains a result, which holds the return value of the deferred operation that generated the future object (for dequeues only, as enqueue operations have no value) and an isDone Boolean value, which is true if and only if the deferred computation has been completed.

Struct Node

```
Struct Node{
        Void* item;
        std::atomic<Node*> next;
}
```

Node object represents an element in the queue (the queue is a list of nodes), each Node object contains item (the value that was enqueued to the queue) and a next pointer which points to the next element in the queue.

Struct BatchRequest

```
struct BatchRequest{
        Node* firstEnq;
        Node* lastEnq;
        unsigned int enqsNum;
        unsigned int deqsNum;
        unsigned int excessDeqsNum;
}
```

A BatchRequest object is prepared by a thread that initiates a batch, and consists of the details of the batch's 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.

Struct PtrCnt

```
struct PtrCnt{
        Node* node;
        uintptr_t cnt;
}
```

A PtrCnt object contains a Node* pointer which points to an element in the queue and contains a counter.

We use this struct to create two pointers: to the first and last elements in the queue, we save in the counter the number of the Enqueues/Dequeued we succeeded to make.

Struct Ann

```
struct Ann{
        BatchRequest batchReq;
        std::atomic <PtrCnt> oldHead;
        std::atomic <PtrCnt> oldTail;
}
```

When a thread wants to make a batch operation, he creates an Announcement object and add it to the start of the queue to “tell” the other threads that I am making a batch operation.

Any thread who Faces an Ann object in the queue (even if it’s not his Ann), will execute the batch operation.

Ann object contains a batchReq object which contains all the info. To execute the batch, also it contains two pointers oldHead and oldTail that will be updated to the Head and the Tail of the queue before the batch has been executed.

Union PtrCntOrAnn

```
union PtrCntOrAnn{
        PtrCnt ptrCnt;
        struct {
                uintptr_t tag;
                Ann* ann;
        } wrap;
}
```

PtrCntOrAnn which is a 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.

We use this union as a pointer to the Head of the queue since ptrCnt is a points to the elements of the queue and since we could face an Ann object at the start of the queue if there is a batch operation running.

When we approach the head of the queue we check if the tag is 1 we conclude that the Head is an Ann Otherwise, it contains a PtrCnt (the tag overlaps PtrCnt .node, whose least significant bit is 0 since it stores either NULL or an aligned address).

Struct FutureOp

```
struct FutureOp{
        Type type;
        Future* future;
}
```

FutureOp object contains Future object (explained above) and contains the future type: Enqueue/Dequeue.

Struct ThreadData

```
struct ThreadData{
        queue<FutureOp> opsQueue;
        Node* enqsHead;
        Node* enqsTail;
        unsigned int enqsNum;
        unsigned int deqsNum;
        unsigned int excessDeqsNum;
}
```

ThreadData object hold local data for each thread.

First, the pending operations details 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.

Global variables

PtrCntOrAnn SQHead

This variable points to the first element in the shared queue, it's a union of PtrCnt and announcement.

PtrCnt points to the regular elements of the queue (Nodes), and the counter contains the number of the successful dequeues that has been done so far.

Since the the threads enter in their running time announcements to the beginning of the shared queue then the first element could be announcement and not node, therefore SQHead can be announcement.

PtrCnt SQTail

This variable points to the end of the shared queue, the counter contains the number of the successful enqueues that has been done so far.

Interface Functions

Init

void Init();

This function initializes the free-lock batch queue data structure by creating a dummy node and update the SQHead and SQTail to point to the dummy node.

Init_thread

void Init_thread();

This function initializes the calling thread, by initializing its fields.

Enqueue

void Enqueue(void* item);

This function gets a pointer to an item and add it to the end of the shared queue by checking if the thread has future operations, if so we add the new item as a future operation by calling the function FutureEnqueue, and then calling the function Evaluate that will execute all the pending (future) operations including Enqueuing the new item.

If the thread has no future operations we add the new item immediately to the shared queue by calling the EnqueueToShared function.

Dequeue

void* Dequeue();

This function Dequeues an item from the shared queue, the function works in the same way of the function Enqueue as explained above.

FutureEnqueue

Future* FutureEnqueue(void* item);

This function makes a future Enqueue operation and adds its info to the thread data by adding the item to the local queue by calling AddToEnqsList, and adding a representing FutureOp to opsQueue queue by calling RecordOpAndGetFuture function, and updating threadData.enqsNum.

FutureDequeue

Future* FutureDequeue();

This function a future Dequeue operation by updating the local thread data: adds a representing FutureOp to opsQueue queue and update threadData.deqsNum and threadData.excessDeqsNum.

Evaluate

void* Evaluate(Future* future);

This function gets a future object and executes it (if its not evaluated yet, we check that by checking future.isDone) by calling ExecuteAllPending which executes all the future operations that the current thread holds.

Implementation Functions

EnqueueToShared

void EnqueueToShared(void* item);

This function gets a new item and adds it to the shared queue (at the end of the queue).

The variable tailAndCnt in the function represents the tail of the queue, in the function we check if tailAndCnt doesn't hold the real tail value, then this means that there is another thread changing the queue, we check if the head holds an announcement this means that another thread is trying to make a batch operation, we help him and by Evaluating the future operations by calling ExecuteAnn function.

We update the tailAndCnt to hold the queue tail value and do the loop again to add the item (if in the current iteration tailAndCnt holds the right value).

DequeueFromShared

PtrCnt HelpAnnAndGetHead();

This function Dequeues an item from the shared queue, by calling HelpAnnAndGetHead function we get the first item in the queue which we want to dequeue (if the head of the queue holds announcement HelpAnnAndGetHead will execute it and return a node value, not announcement).

We dequeue the item from the head of the queue and return its value and update the head of the queue.

HelpAnnAndGetHead

PtrCnt HelpAnnAndGetHead();

This function returns the first Node item in the queue (not announcement), we do a loop and check if the head holds an announcement, we executes it by calling ExecuteAnn function.

First time we find a node item in the beginning of the queue we return it.

ExecuteBatch

Node* ExecuteBatch(BatchRequest req);

This function creates an announcement object that represents the given BatchRequest object, and executes the batching operation by calling ExecuteAnn fuction and returns the old head of the queue.

Before calling ExecuteAnn, we add the announcement object to the head of the shared queue, so if another thread “saw” the announcement object he will “help us” by executing the batch.

ExecuteAnn

void ExecuteAnn(Ann* ann);

This function Executes the given announcement object which represents a batchRequest.

ann->oldTail is updated only after the BatchRequest has been Executed, therefore in the beginning of the function we check if ann->oldTail is not NULL, this means that another thread have executed the given announcement therefore we break the loop.

In the main loop we get the tail of the queue and add after it all the pending Enqueue by making the tail points to ann->batchReq.firstEnq which points to the first item from the future items.

After that we call UpdateHead function to execute the future dequeues.

AddToEnqsList

void AddToEnqsList(void* item);

This function gets an item (future Enqueue) and adds it to the end of the list which have all the future enqueues for this thread.

threadData.enqsHead and threadData.enqsTail points to the begin and the end of the list as mentioned in Data Structures section.

PairFuturesWithResults

void PairFuturesWithResults(Node* oldHeadNode);

This function updated the FutureOp which the thread holds in opsQueue queue, we make an iteration to update all of opsQueue Nodes, in the loop we hold currentHead which make iteration and points to the shared queue.

If the current Operation in opsQueue is ENQ type, we continue to the next iteration.

Else it means that the current operation is DEQ, if there is no more items to dequeue from the shared queue or we reached nextEnqNode which represents the items we added to the shared queue in the current batch, therefore we don't have an item to dequeue, so we update op.future->result to be NULL.

Else, we update op.future->result to have currenthead->item (from the shared queue) and update currentHead to point to currentHead->next (next item in the queue).

In both cases we update op.future->isDone to be true.

ExecuteDeqsBatch

std::pair<unsigned int, Node*> ExecuteDeqsBatch();

This function executes a deques batch, in the iteration we calculate how many successful dequeues we can do (successful dequeues is <= threadData.deqsNum, and also <= number of elements in the queue).

Depending on the number of successful dequeues we point to the element that should be the new head of the queue (newHeadNode in the code), calculated in the loop.

In the end of the function we update SQHead to point to newHeadNode, means that we dequeued all the elements between the old SQHead and newHeadNode.

PairDeqFuturesWithResults

void PairDeqFuturesWithResults(Node* oldHeadNode, unsigned int successfulDeqsNum);

This function updates the FutureOp that the thread saves in his local data in opsQueue queue.

We make iteration with the given successfulDeqsNum steps, in each iteration we hold a pointer currentHead which starts pointing to the given oldHeadNode and in each iteration we update currentHead to point to the next item (iteration on the shared queue).

In each iteration we pop FutureOp from the opsQueue and updates its result to be currentHead->item (value from the shared queue).

After that we make iteration for the number of the excess dequeues and update FutureOp result to be NULL. In both cases we update FutureOp->isDone to be true.

Helping Functions

UpdateHead

void UpdateHead(Ann* ann);

This function updates the head after executing an announcement (which executes a batch), in this function we update the SQHead, by updating it we skip number of Nodes which means we are dequeuing them.

We calculate the number of successful dequeues (successfulDeqsNum in the code): the number of the dequeues in the announcement minus the excess ones.

Note that ann->oldHead, ann->oldTail are the old head and tail of the shared queue before the announcement has been executed.

Therefore, the new head of the queue should be the the node with distance of successfulDeqsNum from the old head ann->oldHead.

We get this node by calling GetNthNode function which makes iteration as the number we give to it (successfulDeqsNum in this case) and start the iteration from the node we give it to it (ann->oldHead), in each iteration we update the pointer to the next node.

Notice that old queue size (the size before we executed the announcement) is <= successfulDeqsNum then we know that all the nodes in the old queue will be dequeued and the new head of the queue will be one of the nodes of the new nodes (that were added in the announcement batch), therefore to save time we call GetNthNode with ann->oldTail node and to make successfulDeqsNum – oldQueueSize iterations, since we started the function from ann->oldTail that means we already skipped all the old nodes.

GetNthNode

Node* GetNthNode(Node* node, unsigned int n);

This function returns the node with distance n from the given node, by doing n iterations and every iteration we update the pointer to take the next node.

RecordOpAndGetFuture

Future* RecordOpAndGetFuture(Type);

This function makes a future and returns it, and also make a FutureOp that represents the future with the given type, and add it to the thread's opsQueue.

ExecuteAllPending

void ExecuteAllPending();

This function executes a batch, if all the pending operations are Dequeues we call ExecuteDeqsBatch and then PairDeqFuturesWithResults, else (means there is at least one Enqueue) we call ExecuteBatch and then PairFuturesWithResults.

In this function we also update the local thread data (enqsHead=enqsTail= NULL since we added all the elements to the shared queue).