

Shared Queue Pattern

Problem

How can concurrently running units of execution efficiently share a queue data structure without race condition?

Context

In many parallel algorithms, a queue is very common data structure that is to be shared among multiple units of execution (UE). For example, many graph traversal algorithms maintain a queue that contains a set of vertexes that need to be traversed in the next span. Parallel graph traversal involves efficient sharing of the queue among concurrent UEs. Also, a task queue is commonly used as a scheduling and load-balancing framework, which implements the Master/Worker pattern.

Forces

- **Simple protocols vs. increased concurrency:** Simple concurrency-control protocols employing a single synchronization construct are easy to implement correctly and hence are less error prone. However, if concurrency-control protocols encompass too much of the shared queue in a single synchronization construct, they will increase the chances that UEs will remain blocked waiting to access the queue and will limit available concurrency. On the contrary, if the protocols are finely tuned, they will increase the available concurrency but such tuning requires more synchronization constructs getting complicated, thus more error-prone.
- **Single-queue abstraction vs. multiple-queue abstraction:** Depending on the memory hierarchy of the underlying hardware and on the number of UEs, maintaining a single queue can cause excess contention and increase parallel overhead. Solutions may need to break with the single-queue abstraction and use multiple or distributed queues.

Solution

Implementing efficient shared queues can be tricky. Appropriate synchronization must be utilized to avoid race conditions but at the same time careful consideration should be given to the way that synchronization is performed. Inefficient synchronization leads to unnecessarily blocking among UEs resulting in poor performance. If possible, consider using distributed queues to eliminate performance bottlenecks.

If it is the case that one must implement a Shared Queue, it can be done as an instance of the Shared Data pattern:

(1) Define the abstract data type (ADT)

An abstract data type is a set of values and the operations defined on that set of values.

In the case of a queue, the values are **ordered lists of zero or more objects** of some type (e.g., integers or task IDs). The operations on the queue are **put** (or enqueue) and **take** (or dequeue).

(2) Choose the concurrency-control protocol. Since the queue will be accessed concurrently, we must define a concurrency control protocol to ensure safe access of the queue. As recommended in the Shared Data pattern, the simplest solution is **to make all operations on the ADT exclude each other**. In addition, we have to decide how the queue will be accessed. First we must decide what happens when a take is attempted on an empty queue or a put is attempted on a full queue. If a thread has to wait in case of an empty queue or a full queue, we call it **blocking queue**. If the access to the queue is immediately returned on such cases, we call it **nonblocking queue**. Also, for better performance of shard queue accesses, there are some choices to be made such as whether to notify all the threads or only the thread that is waiting for the queue on blocking queue, and whether to use **nested locks** or not.

(3) Consider using distributed shared queues. Again, if possible use distributed shared queues, for a centralized shared queue may cause performance bottleneck.

To make this discussion more concrete, we will consider the queue in terms of two specific problems: a queue to hold tasks in a Master/Worker algorithm and a queue to holds vertex IDs in breadth-first graph traversal. Most of the solutions and implementations presented here can be found in Patterns for Parallel Programming[1].

(1) Defining ADT for the queue: Figure 1 illustrates Java implementation of the queue used in Master/Worker algorithm. Linked list structure is defined in class *Node* and *put* and *take* is defined. One can define the task queue as simple as an array that only contains integer values that represent task IDs. In any case, putting a new item exclusively accessing the head and taking the item exclusively accessing the tail, or last, is the common feature of a shared queue. Figure 2 shows CUDA implementation of the queue used in breadth first graph traversal. An array of integer is defined and only the *put* operation is defined because we assume SIMD parallel execution of UEs when taking the vertex IDs in the queue. When each UE takes the vertex IDs, evenly distributed disjoint set of vertex IDs in the queue are assigned to each UE by CUDA runtime framework, leaving no need for explicit *take* operation to be defined for this specific problem.

(2-1) Defining all operations on the ADT to exclude each other: As discussed in Shared Data pattern, the easiest solution is to ensure that the operations are executed serially. One can make the operation as a synchronized method [2] in case of Java implementation. In other languages, one might implement synchronized method by using her own lock that ensures mutual exclusion of the method. Because it is not possible for two invocations of synchronized methods on the same object to interleave, when one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

```

public class SharedQueue1
{
    class Node //inner class defines list nodes
    { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
    }

    private Node head = new Node(null); //dummy node
    private Node last = head;

    public synchronized void put(Object task)
    { assert task != null: "Cannot insert null task";
      Node p = new Node(task);
      last.next = p;
      last = p;
    }

    public synchronized Object take()
    { //returns first task in queue or null if queue is empty
      Object task = null;
      if (!isEmpty())
      { Node first = head.next;
        task = first.task;
        first.task = null;
        head = first;
      }
      return task;
    }

    private boolean isEmpty(){return head.next == null;}
}

```

Figure 1. Nonblocking queue that ensures that at most one thread can access the data structure at one time

```

typedef struct _VertexQueue
{
    int *Queue;
    int *head;
} VertexQueue

VertexQueue q;
cudaMalloc((void**)&(q.Queue), sizeof(int)*MAX_SIZE);
cudaMalloc((void**)&(q.head), sizeof(int)*1);

void put (int vertexID)
{
    int index = atomicAdd(q.head, 1);
    q.Queue[index] = vertexID
}

```

Figure 2. Shared Queue using atomicAdd() in CUDA

(2-2) Blocking/Nonblocking queue: What should be done on an empty queue or on a full queue needs to be defined. For the task queue used in the Master/Worker algorithm, the policy depends on how termination will be handled by. Suppose, for example, that all the tasks will be created at startup time by the master. Then an empty task queue will indicate that the UE should terminate, and we will want the take operation on an empty queue to return immediately with an indication that the queue is empty. That is, we want a non-blocking queue. Another possible situation is that tasks can be created dynamically and that UEs will terminate when they receive a special “poison pill” task. Then the appropriate behavior might be for the take operation on an empty queue to wait until the queue is nonempty, That is, we want a block-on-empty queue.

The queue operations defined in Figure 1 is nonblocking queue, where it returns null if queue is empty. A straightforward implementation is to enforce mutual exclusion as described in Implementation Mechanisms. Note that we put *synchronized* keyword on the *take* and *put* methods, making them not to interleave each other. In the code, *isEmpty* method is private, and only invoked inside a synchronized method. Thus it need not be synchronized. (If it were public, it would need to be synchronized as well.)

The blocking version of the shared queue is shown in Figure 3. In this version, the take operation is changed so that a thread trying to take from an empty queue will wait until it is filled rather than returning immediately with null. The waiting thread needs to release its lock that it acquired when it entered the synchronized method and needs to reacquire it before trying again. In Java, this is implemented in the *wait* and *notify* methods. Similar primitives are available with POSIX threads (Pthreads) [3], and techniques for implementing this functionality with semaphores and other basic primitives can be found in [4]. The changes that have been made from Figure 1 is highlighted with red boxes in Figure 3.

We added to the *put* method a call to *notifyAll*. However, this implementation has a performance problem in that unnecessary notification will be sent even when there is no blocked thread. One way to handle this problem is to keep track of the number of

blocked threads and only perform a *notifyAll* when the number is greater than zero.

```
public class SharedQueue2
{
    class Node
    { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
    }

    private Node head = new Node(null);
    private Node last = head;

    public synchronized void put(Object task)
    { assert task != null: "Cannot insert null task";
      Node p = new Node(task);
      last.next = p;
      last = p;
      notifyAll();
    }

    public synchronized Object take()
    { //returns first task in queue waits if queue is empty
      Object task = null;
      while (isEmpty())
      {try{wait();}catch(InterruptedException ignore){}}
      { Node first = head.next;
        task = first.task;
        first.task = null;
        head = first;
      }
      return task;
    }

    private boolean isEmpty(){return head.next == null;}
}
```

Figure 3. Blocking queue that ensures at most one thread can access the data structure

(2-3) More efficient concurrency-control protocols: Note that the shared queue in Figure 1 or 2 enforces a thread that is trying to execute the *take* method to be blocked if there is a thread that is executing the *put* method. Careful examination of the operations in our non-blocking shared queue shows that the *put* and *take* are noninterfering since they do not access the same variables: The *put* method modifies the reference *last* and the *next* member of the object referred to by *last*. The *take* method modifies the value of the task member in the object referred to by *head.next* and the reference *head*. Since these are non-interfering operations, each operation need not to be blocked when the other operation is executing. Therefore we can use one lock for *put* and a different lock for *take*. This solution is shown in Figure 4. Note that the *put* and *take* methods are now no

more defined with the synchronized keyword. Instead, it contains a synchronized block with its own lock object. Now that they have different lock objects, they do not block unnecessarily, increasing the performance.

```
public class SharedQueue3
{
    class Node
    { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
    }

    private Node head = new Node(null);
    private Node last = head;

    private Object putLock = new Object();
    private Object takeLock = new Object();

    public void put(Object task)
    { synchronized(putLock)
      { assert task != null: "Cannot insert null task";
        Node p = new Node(task);
        last.next = p;
        last = p;
      }
    }

    public Object take()
    { Object task = null;
      synchronized(takeLock)
      { if (!isEmpty())
        { Node first = head.next;
          task = first.task;
          first.task = null;
          head = first;
        }
      }
      return task;
    }
}
```

Figure 4. Nonblocking shared queue that uses separate locks so they can proceed concurrently

(2-3) Nested locks for blocking queue: It is tricky to implement a blocking queue based on the queue that uses separate locks for the *take* and *put* method. The trickiness comes from the fact that *wait*, *notify*, *notifyAll* methods on an object can only be invoked within a block synchronized on that object. Also, if we have optimized the invocations of *notify* as described before then *w*, the count of waiting threads, would be accessed in both *put*

and *take*. Therefore, we use *putLock* both to protect *w* and to serve as the lock on which a taking thread blocks when the queue is empty.

```
public class SharedQueue4
{
    class Node
    { Object task;
      Node next;

      Node(Object task)
      {this.task = task; next = null;}
    }

    private Node head = new Node(null);
    private Node last = head;
    private int w;
    private Object putLock = new Object();
    private Object takeLock = new Object();

    public void put(Object task)
    { synchronized(putLock)
      { assert task != null: "Cannot insert null task";
        Node p = new Node(task);
        last.next = p;
        last = p;
        if(w>0){putLock.notify();}
      }
    }

    public Object take()
    { Object task = null;
      synchronized(takeLock)
      { //returns first task in queue, waits if queue is empty
        while (isEmpty())
        { try{synchronized(putLock){w++; putLock.wait();w--;} }
          catch(InterruptedException error){assert false;}}
        { Node first = head.next;
          task = first.task;
          first.task = null;
          head = first;
        }
      }
      return task;
    }

    private boolean isEmpty(){return head.next == null;}
}
```

Figure 5. Blocking shared queue that uses separate nested locks

Code is shown in Figure 5. Notice that *putLock.wait()* in *take* will release only the lock on *putLock*, so a blocked thread will continue to block other takers from the outer block

synchronized on *takeLock*. This is okay for this particular problem. Compared to the nonblocking queue, this scheme continues to allow putters and takers to execute concurrently, except when the queue is empty.

Another issue to note is that this solution has nested synchronized blocks. Nested synchronized blocks should always be examined for potential deadlocks. In this case, there will be no deadlock since *put* only acquires one lock, *putLock*.

(3) Distributed shared queues: As the number of UEs that access the shared queue increase, it will be likely that the contention to the centralized shared queue become a hot spot and degrade the performance significantly. As an example of distributed queues, a complete package to support fork/join programs using a pool of threads and a distributed task queue in the underlying implementation is presented in [1]. The idea of the example is to associate a nonblocking distributed queue with each thread rather than using a single central task queue. When a thread generates a new task, it is put in its own task queue. When a thread executes a task, it first tries to take one from its own queue. If it is empty, it randomly chooses another thread and tries to steal one from that threads' queue. As each task tries to take a new task first from its own task and there is no single hot spot, it eliminates the performance degradation.

CUDA supports atomic operation on shared memory as well as on global memory. As shared memory is local to each core in GPU, we can implement a local distributed queues using atomic operation on shared memory. Figure 6 illustrates this. Instead of passing shared queue head in global memory (*q.head* in Figure 2) to *atomicAdd*, pass local queue head defined in shared memory (*localQ_head* in Figure 6).

```
__shared__ int localQ[MAX_SIZE];
__shared__ int localQ_head;
__shared__ int globalQ_index;

// put vertexID into the local queue
int index = atomicAdd(&localQ_head, 1);
localQ[index] = vertexID;

// Master thread in a thread block obtains the index of global queue
// where the local queue contents need to be copied to
if (threadIdx.x == 0)
    globalQ_index = atomicAdd(q.head, localQ_head);
__syncthreads();

// Copy the local queue content to the global queue
if (threadIdx.x < localQ_head)
    q.Queue[globalQ_index + threadIdx.x] = localQ[threadIdx.x];
}
```

Figure 6. Local distributed queue in CUDA using *atomicAdd* on shared memory

CUDA framework as mentioned before implicitly handles load-balancing with its

runtime. This is achieved at the expense of following CUDA programming model where any results obtained in the local shared memory need to be copied to the global memory to be re-used in the subsequent kernels. To follow this, we copy the local queue content to the global memory in Figure 6. The second `atomicAdd()` obtains the index of global of queue. However, this mutual exclusion to global memory is only performed by a master thread in a thread block, which results in very little contention.

Example

Fibonacci number computing

We will use an example that has shown in [1]. The example computes in parallel Fibonacci number of a certain number n . As shown in Figure 7 and Figure 8, if this number n is smaller than some threshold, we compute Fibonacci number in serial execution as it will be more efficient considering the parallel execution overhead.

The `run` method recursively decompose the task by creating new tasks and execute them in parallel by invoking the `fork` method on each created task. The `fork` method is defined in `Task` class shown in Figure 11 and note that it is a part of the distributed queue package shown in Figure 12. The package is a much simplified version of the FJTask package [5], which uses ideas from [6].

The `fork` method put the caller task into the task queue of its associated thread. Thread-pool has been created in main before Fibonacci number computation is executed (Figure 7). A thread goes over the loop shown in Figure 9 until it receives an ending signal (*poison task*). In the loop, the thread takes a task from its own task queue and, if it is empty it tries to steal one from the other thread's task queue. The `steal` method of a `TaskRunner` class which extends class `Thread` is shown in Figure 10.

```

//Performance-tuning constant, sequential algorithm is used to
//find Fibonacci numbers for values <= this threshold
static int sequentialThreshold = 0;

public static void main(String[] args) {
    int procs; //number of threads
    int num; //Fibonacci number to compute
    try {
        //read parameters from command line
        procs = Integer.parseInt(args[0]);
        num = Integer.parseInt(args[1]);
        if (args.length > 2)
            sequentialThreshold = Integer.parseInt(args[2]);
    }
    catch (Exception e) {
        System.out.println("Usage: java Fib <threads> <number> "+
            "[<sequentialThreshold>]");
        return;
    }

    //initialize thread pool
    TaskRunnerGroup g = new TaskRunnerGroup(procs);

    //create first task
    Fib f = new Fib(num);

    //execute it
    g.executeAndWait(f);

    //computation has finished, shutdown thread pool
    g.cancel();

    //show result
    long result;
    {result = f.getAnswer();}
    System.out.println("Fib: Size: " + num + " Answer: " + result);
}
}

```

Figure 7. A program that computes Fibonacci number

```

public class Fib extends Task
{
    volatile int number; // number holds value to compute initially,
                        //after computation is replaced by answer
    Fib(int n) { number = n; } //task constructor, initializes number

    //behavior of task
    public void run() {
        int n = number;

        // Handle base cases:
        if (n <= 1) { // Do nothing: fib(0) = 0; fib(1) = 1 }
        // Use sequential code for small problems:
        else if (n <= sequentialThreshold) {
            number = seqFib(n);
        }
        // Otherwise use recursive parallel decomposition:
        else {
            // Construct subtasks:
            Fib f1 = new Fib(n - 1);
            Fib f2 = new Fib(n - 2);

            // Run them in parallel:
            f1.fork();f2.fork();
            // Await completion;
            f1.join();f2.join();

            // Combine results:
            number = f1.number + f2.number;
            // (We know numbers are ready, so directly access them.)
        }
    }

    // Sequential version for arguments less than threshold
    static int seqFib(int n) {
        if (n <= 1) return n;
        else return seqFib(n-1) + seqFib(n-2);
    }

    //method to retrieve answer after checking to make sure
    //computation has finished note that done and isDone are
    //inherited from the Task class. done is set by the executing
    //(TaskRunner) thread when the run method is finished.
    int getAnswer() {
        if (!isDone()) throw new Error("Not yet computed");
        return number;
    }
}

```

Figure 8. Fib object recursively decomposes the task and forks the sub-tasks

```

//Main loop of thread. First attempts to find a task on local
//queue and execute it. If not found, then tries to steal a task
//from another thread. Performance may be improved by modifying
//this method to back off using sleep or lowered priorities if the
//thread repeatedly iterates without finding a task. The run
//method, and thus the thread terminates when it retrieves the
//poison task from the task queue.
public void run()
{ Task task = null;
  try
  { while (!poison.equals(task))
    { task = (Task)q.takeLast();
      if (task != null) { if (!task.isDone()){task.invoke();}}
      else { steal(null); }
    }
  } finally { active = false; }
}

//Looks for another task to run and continues when Task w is done.
protected final void taskJoin(final Task w)
{ while(!w.isDone())
  { Task task = (Task)q.takeLast();
    if (task != null) { if (!task.isDone()){ task.invoke();}}
    else { steal(w);}
  }
}
}

```

Figure 9. A threads' main loop executing a task in task queues

```

//Attempts to steal a task from another thread. First chooses a
//random victim, then continues with other threads until either
//a task has been found or all have been checked. If a task
//is found, it is invoked. The parameter waitingFor is a task
//on which this thread is waiting for a join. If steal is not
//called as part of a join, use waitingFor = null.
void steal(final Task waitingFor)
{ Task task = null;

  TaskRunner[] runners = g.getRunners();
  int victim = chooseToStealFrom.nextInt(runners.length);
  for (int i = 0; i != runners.length; ++i)
  { TaskRunner tr = runners[victim];
    if (waitingFor != null && waitingFor.isDone()){break;}
    else
    { if (tr != null && tr != this)
      task = (Task)tr.q.take();
      if(task != null) {break;}
      yield();
      victim = (victim + 1)%runners.length;
    }
  } //have either found a task or have checked all other queues

  //if have a task, invoke it
  if(task != null && ! task.isDone())
  { task.invoke(); }
}

```

Figure 10. A thread's steal method which randomly steals a task from the other thread's task queues

```

public abstract class Task implements Runnable
{
    //done indicates whether the task is finished
    private volatile boolean done;
    public final void setDone(){done = true;}
    public boolean isDone(){return done;}

    //returns the currently executing TaskRunner thread
    public static TaskRunner getTaskRunner()
    { return (TaskRunner)Thread.currentThread(); }

    //push this task on the local queue of current thread
    public void fork()
    { getTaskRunner().put(this);
    }

    //wait until this task is done
    public void join()
    { getTaskRunner().taskJoin(this);
    }

    //execute the run method of this task
    public void invoke()
    { if (!isDone()){run(); setDone(); }
    }
}

```

Figure 11. Task class which Fib class is extended from.

```

class TaskRunnerGroup
{ protected final TaskRunner[] threads;
  protected final int groupSize;
  protected final Task poison;

  public TaskRunnerGroup(int groupSize)
  { this.groupSize = groupSize;
    threads = new TaskRunner[groupSize];
    poison = new Task(){public void run(){assert false;}};
    poison.setDone();
    for (int i = 0; i!= groupSize; i++)
      {threads[i] = new TaskRunner(this,i,poison);}
    for(int i=0; i!= groupSize; i++){ threads[i].start(); }
  }

  //start executing task t and wait for its completion.
  //The wrapper task is used in order to start t from within
  //a Task (thus allowing fork and join to be used)
  public void executeAndWait(final Task t)
  { final TaskRunnerGroup thisGroup = this;
    Task wrapper = new Task()
    { public void run()
      { t.fork();
        t.join();
        setDone();
        synchronized(thisGroup)
          { thisGroup.notifyAll();} //notify waiting thread
      }
    };
    //add wrapped task to queue of thread[0]
    threads[0].put(wrapper);
    //wait for notification that t has finished.
    synchronized(thisGroup)
    { try{thisGroup.wait();}
      catch(InterruptedException e){return;}
    }
  }

  //cause all threads to terminate. The programmer is responsible
  //for ensuring that the computation is complete.
  public void cancel()
  { for(int i=0; i!= groupSize; i++)
    { threads[i].put(poison); }
  }

  public TaskRunner[] getRunners(){return threads;}
}

```

Figure 12. The TaskRunnerGroup class that forms a fork/join package using a thread-pool based on distributed task queues

Related Patterns

- Shared Data: Shared Queue pattern is an instance of Shared Data pattern.
- Master/Worker: Shared Queue pattern is often used to represent the task queues in algorithms that use the Master/Worker pattern.
- Fork/Join pattern: Thread-pool-based implementation of Fork/Join pattern is supported by this pattern.

References

[1] Timothy Mattson, Beverly Sanders, Berna Massingill, "Patterns for Parallel Programming", Addison-Wesley, 2004

[2] Synchronized Methods,

<http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html>

[3] David Butenhof, "Programming with POSIX Threads", Addison-Wesley, 1997

[4] Gregory Andrews, "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison-Wesley, 2000

[5] Doug Lea, "A Java fork/join framework", In proceeding of the ACM 200 conference on Java Grande, page 36-43, ACM Press, 2000

[6] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, Yuli Zhou, "Cilk: An efficient multithreaded runtime system", Journal of Parallel and Distributed Computing, 37(1): 55-69, August 1996

Authors

- Original Author: Timothy Mattson
- Revised: Youngmin Yi