

Task Queue Implementation Pattern

Ekaterina Gonina (Author), Jike Chong (Shepherd), UC Berkeley ParLab

Name

Task Queue

Problem

Some applications can be broken down into variable length tasks to be executed concurrently. How do we efficiently load balance the tasks among processing elements in such applications?

Context

Many problems can be expressed as a set of tasks where the number of tasks is input-dependent. Tasks denote an independent unit of work that can be executed in parallel with other tasks. Some examples of such application include branch-and-bound algorithms (see **Branch-and-Bound** pattern) and graph traversal algorithms (see **Graph Algorithms** pattern). These applications have large amounts of tasks and large compute demands that we need to exploit for efficient parallel execution. In such applications, the number of tasks and sometimes the length of each task can vary dynamically. In order to effectively parallelize such applications and exploit concurrency, we must take care of issues that influence scalability. Good parallel scaling requires load balancing among the participating processors. Task queue is a well-known mechanism that is primarily designed to address the load imbalance problem [1].

Task Queue is defined as a mechanism to synchronously distribute a sequence of tasks among parallel threads of execution. The global problem is broken down into tasks and the tasks are enqueued onto the queue. Parallel threads of execution pull tasks from the queue and perform computations on the tasks. The runtime system is responsible for managing thread accesses to the tasks in the queue as well as ensuring proper queue usage (i.e. dequeuing from an empty queue should not be allowed).

Tasks may come from using loop-level parallelism (see **Loop-Parallelism** pattern) where a set of iterations of the loop can represent a task. These tasks have nice properties of statically-known lengths and can be expressed simply as loop iteration indices. A broader category of algorithms use the task-parallelism model (see **Task Parallelism** pattern) to represent parallel entities of execution. In this model tasks may be of varying lengths and need more meta-data to be represented in the queue. For example tree- or graph-traversal problems are typical in this scenario – a task is defined as traversing outgoing edges of a vertex. Given a vertex we traverse its edges to find its children that are new tasks to be enqueued. The execution time of these tasks varies with the number of outgoing edges for a vertex.

Intelligent task scheduling using a task queue can yield other important benefits such as data locality. By scheduling tasks on processing elements whose memory contains the data that is used by the task, we can increase locality and thus performance of the application (for more information on exploiting locality see **Memory Parallelism** pattern).

Forces

1. *Centralized vs Distributed* – in order for each parallel thread of execution to efficiently access the task queue data structure, it must live in a predefined space. Centralized task queues are simpler to implement, however they can quickly become a bottleneck as the number of processing elements increases. Distributed task queues require more overhead, however they are able to scale to much larger number of processing elements.
2. *Eager vs Lazy scheduling*. – load balancing enforcement can happen either periodically, for example every time a task is assigned to a processor, (eager approach) or only in the case when one processor becomes idle (lazy approach). The eager scheduling can provide better load balancing at the expense of extra overhead, while lazy scheduling might not yield as good of load balance, but requires less overhead.

Solution

Implementing a Task Queue in software involves the following steps:

1. *Determine Task Abstraction*
2. *Determine Queue Structure*
 - a. *Centralized vs Distributed*
 - b. *Determine queue operations*

1. *Determine Task Abstraction*

Depending on the structure of the application, after breaking it down into tasks using patterns in the Computational and Algorithm Strategy levels, we can define the notion of “task”. Tasks denote independent execution units.

For example in a graph search problem (a **Graph Algorithms** pattern) we can create tasks using the **Task Parallelism** pattern, where we define a task as a set of operations on a vertex of the graph. A task abstraction in this case can be the vertex ID. In another example, we might have a loop of element-wise multiplication when computing a dot-product of two vectors in **Dense Linear Algebra** application. Using the **Loop Parallelism** pattern, we decompose this process into tasks, where a task is a set of iterations. In this case, tasks can be abstracted as a starting and ending iteration indices.

2. Determine Queue Structure

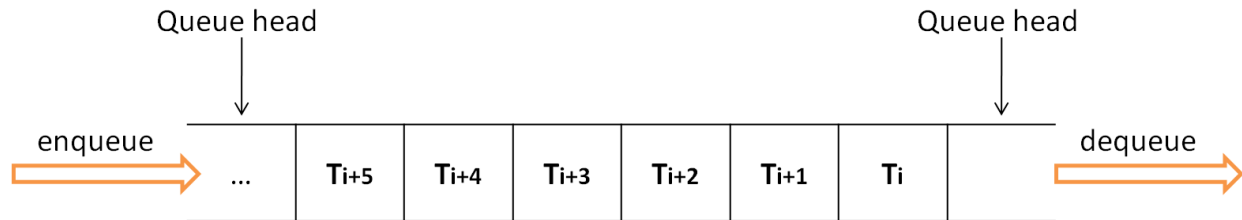


Figure 1, an illustration of a Task Queue. Tasks (T_i) are enqueued at the Queue tail position and tasks are dequeued at the Queue head position.

Figure 1 shows a sample illustration of a task queue. The queue is a First-Come-First-Serve (FIFO) data structure, the tasks get executed in order they were put in the queue. There are two basic operations, enqueue – adding a task to the queue, and dequeue – removing a task from the queue. Tasks get enqueued at the end of the queue (at the queue *tail*). Tasks get dequeued at the front of the queue (at the queue *head*). In the context of using the queue data structure to manage *data* instead of tasks, please refer to **Shared Queue** pattern.

There are two alternatives to implementing the task queue – in software or hardware. The software approach is platform independent, however, software managed task queues present a high task managing overhead, therefore, tasks must be large enough to compensate for parallel overhead of managing them in the queue. If the underlying hardware platform supports implementing the queue in hardware or already has an existing task queue structure to manage tasks among parallel processors (see Carbon [1]), it might be more efficient to use it rather than implementing the task queue in software. However, using hardware-managed task queue restricts the portability of the application, since it will require re-implementation for a different platform that might not support hardware task queues.

a. Centralized vs Distributed

When implementing the task queue on a parallel platform, a decision needs to be made whether to keep the queue on one processor or distribute it among all the processors. The first alternative is to have a centralized queue where the queue is kept on one processor and other processors acquire tasks from that processor. The other alternative is to implement a distributed queue, where each processor has a local queue it works on. In this model some extra synchronization might be needed in order to assign tasks to processors in case a local task queue becomes empty.

The centralized queue is easiest to implement, one processor is responsible for keeping track of all tasks and assigning them to processors. However, the task queue may become a bottleneck with increasing number of processors and not see efficient scaling. To determine if centralized queue is a good approach, we need to consider three factors: task size, latency to acquire a task and the number of processors. If the task size is large and the overhead of acquiring a task by a processor is

small and the number of processors is small, then centralized queue is an efficient mechanism to use. However, if the number of processors increases, or the size of the task is small compared to the latency of acquiring a task, the centralized task queue will not be a good choice for a task managing mechanism.

The distributed queue model scales much better to larger number of processors, however it may require more overhead of assigning tasks to processors. For example, if processor 0 needs a task to execute and it does not have any tasks in its local queue, it may have to call an extra function to obtain a task to work on. This overhead can be significant if the amount of computation is not large enough to hide it.

It is more difficult to enforce global ordering on the tasks if we are using a distributed queue. In order to make sure tasks get executed in a particular order globally, extra synchronization is needed. If this is an important consideration for the application, centralized queue may be a favored approach, since global task order enforcement is easier to implement in that case. Assigning priorities to tasks can be a method to enforce ordering on tasks in the queue. For example, tasks that have dependencies on another task in the computation can be assigned a lower priority such that they will be executed *after* the task they have a dependency on.

In order to gain performance in memory accesses, it is essential to exploit locality as much as possible. Scheduling tasks on processors that generated those tasks is one approach to achieve better locality. For example during a tree traversal, it is likely that after computing the “parent” task a processor generates “children” tasks, that share common data (such as parent information). Since this information will be present in the local memory on that processor, it is efficient to schedule the “children” tasks on that processor thereby exploiting locality.

Distributed Task Stealing

One method for implementing a distributed task queue is distributed task stealing. In this model, each processor has its own queue, it enqueues its local tasks to this queue and operates on them, keeping the computation local. Task enqueueing and dequeueing always happens at the *head* of the queue. When a processor finishes executing, it looks for a new task in its queue. If there is not a new task in its own queue the processor *steals* a task from another processor. Task stealing is always performed at the *tail* of the queue.

This method has the following advantageous properties:

1. It has been shown to be provably efficient both in terms of execution time and space usage [1]
2. By scheduling the child tasks on the same processor as the parent task we get better cache locality since it supports significant data sharing between parent and child tasks.

Hierarchical Task Queueing

Hierarchical Task Queueing is a hybrid approach between distributed and a centralized queue. In this model each processor has a small private queue which it uses to enqueue and dequeue its tasks. When the private queue fills up, the tasks are moved to a global queue. If the private queue becomes empty, the processor gets tasks from the global queue.

This model usually performs worse than Distributed Task Stealing queue because it does not exploit locality as well. However, if the amount of available parallelism is limited or the amount of local memory is small, hierarchical task queues can sometimes perform better [1].

b. Determine Queue Operations

After determining the structure of the queue, we need to define the basic operations:

1. *Init* – initialize the queue. If the queue is distributed, it needs to be properly initialized on every processor.
2. *Enqueue* – the procedure for adding tasks to the queue. Need to make sure not to exceed capacity.
3. *Dequeue* – the procedure for removing task from the queue. Need to make sure not to dequeue from an empty queue. Pre-fetching can be an important optimization here to hide latency.

Invariant

1. Tasks cannot be enqueued onto a full queue and dequeued from an empty queue
2. All tasks generated by decomposing the application are processed and they are processed only once

Examples

1. A Sample Task Queue Implementation

The following Python code shows a sample implementation of a basic centralized task queue. This sample code was obtained from [2].

```
import threading
from Queue import Queue

class TaskQueue(Queue):

    def __init__(self):
        Queue.__init__(self)
        self.all_tasks_done = threading.Condition(self.mutex)
        self.unfinished_tasks = 0

    def _put(self, item):
        Queue._put(self, item)
```

```

        self.unfinished_tasks += 1

def _get():
    Queue._get(True)
    self.unfinished_tasks -= 1

def task_done(self):
    self.all_tasks_done.acquire()
    try:
        unfinished = self.unfinished_tasks - 1
        if unfinished <= 0:
            if unfinished < 0:
                raise ValueError('task_done() called too many times')
            self.all_tasks_done.notifyAll()
        self.unfinished_tasks = unfinished
    finally:
        self.all_tasks_done.release()

def join(self):

    self.all_tasks_done.acquire()
    try:
        while self.unfinished_tasks:
            self.all_tasks_done.wait()
    finally:
        self.all_tasks_done.release()

```

The above code defines basic functionality of the task queue. TaskQueue uses the generic Queue object in Python. The functions are as follows:

`_init_` - initializes the queue to have no tasks. `threading.Condition(self.mutex)` initializes threads, they are controlled by the `self.mutex` lock. Initially all threads are blocked.

`_put` - the enqueue operation. Enqueues a task defined as `item` onto the queue. Increments the total number of unfinished tasks in the queue.

`_get` - the dequeue operation. Dequeues a task from the head of the queue. The Boolean parameter in `Queue._get()` indicates if the operation is blocking or non-blocking. In this example, `True` indicates the `_get` operation as blocking - the control returns to the thread only when it successfully dequeues a task, otherwise it blocks on this call.

`task_done` - For each `get()` used to dequeue a task, a subsequent call to `task_done()` by a thread tells the queue that the processing of the task is complete. Decrements the number of unfinished tasks.

`join` - blocks until all items in the queue have been processed, i.e. when the count of unfinished tasks in the queue is zero.

We can modify this code to implement a distributed queue. Instead of allocating one queue, each processor calls the `_init()` routine, thus each processor has a local queue to work with. The basic operations stay the same, except for the `_get` function. Instead of simple `Queue._get()`, we need to make sure that if the local queue is empty, the processor gets a task from the tail of the queue of a different processor. The python-like pseudo code for this is as follows:

```
def _get():
    if Queue._get(False)
        self.unfinished_tasks -= 1
    else:
        getTaskFromPE()
def getTaskFromPE():
    peNum = getPENum()
    queue = getQueue(peNum)
    queue._getFromTail(True)
```

`getTaskFromPE()` gets a processor ID number using a specific mechanism (`getPENum()`), such as determining which processor has the most number of tasks enqueued in its queue, and returns a task from the tail of the queue for that processor (`getFromTail()`).

2. Using the Task Queue

Simple use of Task Queue

Below is a simple program to illustrate the use of the use of TaskQueue interface shown above. The program simulates a queue of tasks that get processed by threads (by performing dummy arithmetic operations).

```
import sys

def worker():
    name = threading.currentThread().getName()
    while True:
        x = q.get()
        sys.stdout.write('%s\t%d\n' % (name, x))
        if x <= 1:
            sys.stdout.write('!\n')
        elif x % 2 == 1:
            q.put(x * 3 + 1)
        else:
            q.put(x // 2)
        q.task_done()

q = TaskQueue()
numworkers = 4
```

```
for i in range(numworkers):
    t = threading.Thread(target=worker)
    t.setDaemon(True)
    t.start()

for x in range(1, 50):
    q.put(x)

q.join()

print 'Computation Done.'
```

`worker()` – invoked on each thread, processes a task from the queue. If the value of the task (x) is greater than one, the thread increments the task by either $(x*3+1)$ or $(x/2)$ depending on if the value of x is odd or even, enqueues x back onto the queue and marks the task as done. If x is less than or equal to 1, the task is complete. The main loop of this function continues until there are no tasks to be processed.

The main body of the program initialized the TaskQueue object and 4 threads to work on tasks in the queue. It starts the threads execution loop and populates the queue with tasks containing values from 1 to 50. Each thread invokes the `worker()` function, and processes tasks in the queue. `q.join()` call blocks until there are no tasks in the queue. Once there are no tasks in the queue (i.e. all values in the tasks are less than or equal to 1) the `q.join()` unblocks and the computation ends.

Note that the implementation is independent from the task queue structure, the worker just calls `q.get()` and gets a task to work on. The mechanisms for retrieving the tasks are hidden behind this call.

Breadth First Search (BFS) on a Graph [3]

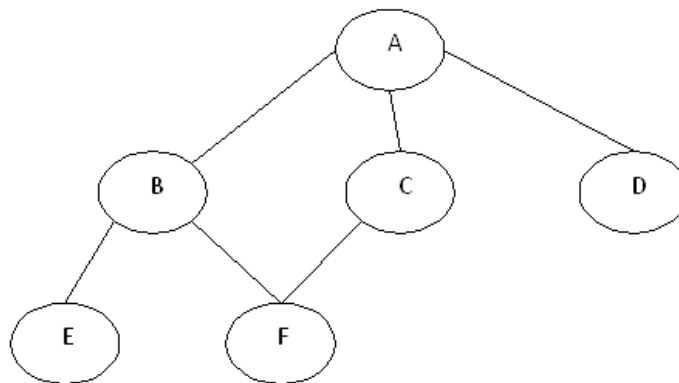


Figure 2. A simple Graph

Breadth First Search (BFS) is one of the most common ways to traverse a graph in a search process (see **Graph Algorithm Pattern** for details). The problem of BFS is as follows:

Given a graph $G=\{V,E\}$, find a node V that contains value X .

For example, Figure 2 illustrates a simple graph that we need to traverse to find the node with a particular value (say, F). BFS traverses the graph from root to all the nodes by looking at a level of the graph at a time. For the graph in Figure 2, BFS traversal would traverse the nodes in the order of: {A, B, C, D, E, F}. It starts at root A, traverses all its children {B, C, D}, then moves down to the next level and traverses nodes {E, F}.

Algorithmic Steps

1. **Step 1:** Push the root node in the Queue.
2. **Step 2:** Loop until the queue is empty.
3. **Step 3:** Remove the node from the Queue.
4. **Step 4:** If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

The following Java-like pseudo-code shows the implementation of the BFS algorithm:

```
public void processNode(TaskQueue q, const int value) {  
  
    Node n = null;  
    while(n = (Node)q.dequeue() != null) { //blocked until this thread  
                                        //gets a node to work on  
  
        if(n.getValue() != value) {  
            Node child=null;  
            while((child=getUnvisitedChildNode(n))!=null) {  
                child.visited=true;  
                q.enqueue(child);  
            }  
        } else {  
            return; //found the value, return  
        }  
    }  
}  
  
public void bfs(Graph G, int val){  
  
    TaskQueue q = new TaskQueue(); //create a task queue  
    q.enqueue(G.rootNode);  
  
    Thread[] threads = new Thread[10];  
  
    for(i = 0; i<10; i++) {  
        threads[i].processNode(q, val);  
    }  
    join(threads); //join threads after execution  
    return;  
}
```

This code allocates 10 threads and calls `processNode()` function on each. Each thread takes a node off of the queue, checks the value, and if it is not the value we're looking for, puts the children of the node onto the queue. The process repeats on each thread until the queue is empty or the value is found.

Known Uses

Carbon – Carbon is a hardware managed task queue created by Intel [1]. It supports various task abstractions and queue operations. Carbon performs close to ideal task queue and outperforms software-implemented task queues when compared on various benchmarks.

Cilk++ – Cilk++ is a general-purpose programming language designed for multithreaded parallel programming originally developed at MIT [4]. Cilk++ employs the distributed task stealing task queue abstraction (called “work-stealing”) to schedule and load balance tasks among threads.

CUDA – Nvidia’s CUDA programming environment [5] uses task queue to schedule work on hardware threads. Work is specified in a CUDA kernel (set of tasks) and the runtime system schedules the tasks on the hardware threads.

Related Patterns

Upper levels

Computational/Structural

1. *Graph Algorithms* – in graph traversal and search algorithms it is often convenient to express each node to search as a task. Task queue is a useful abstraction to efficiently implement such search algorithms.
2. *Branch-and-Bound* – in branch-and-bound applications the problem is expressed in terms of searching a solution space to make a decision or to find an optimal solution. The solution involves dividing the search space into sub-spaces and efficiently bounding and pruning the search spaces. A task can be the evaluation of sub-spaces, which are generated dynamically during this process, we can schedule these tasks using the task queue pattern.

Algorithm Strategy

1. *Task Parallelism* – task parallelism defines tasks as units of work that can be executed in parallel. Task queue can be used to schedule these tasks to be executed on parallel threads of execution.
2. *Recursive Splitting* – recursive splitting divides a problem into sub-problems recursively, each sub-problem can be represented as a task. We can use task queue to schedule these tasks generated by recursive splitting on parallel threads of execution.
3. *Graph Partitioning* – graph partitioning divides a problem represented as a graph into sub-problems by using graph partitioning algorithms. Each partition can be further subdivided into subproblems until we reach the optimal granularity. Task queue can be used to schedule these tasks on parallel threads of execution.

Same level

1. *Shared Queue* – if the queue data structure is to be shared among processors to transfer *data* instead of tasks, shared queue pattern describes the mechanisms for accomplishing this task.
2. *Loop-level Parallelism* – the loop iterations can represent tasks that can be scheduled using the Task Queue pattern.
3. *MasterWorker* – Tasks can be scheduled to execute on worker threads using the task queue. The master process creates tasks and manages their scheduling if it owns the task queue (in a centralized queue model for example).
4. *Memory parallelism* – the memory parallelism pattern describes ways to take advantage of locality in a computation. In the task queue pattern it is important to schedule tasks that share the same data on the same processor, this intersects with the memory parallelism pattern.

Lower level

1. *Thread Pool* – task queue can be implemented using a thread where each task is assigned to a thread in a thread pool.
2. *Mutual Exclusion* – parallel threads' accesses of tasks in the queue need to be managed in order to guarantee correct order of accesses. Mutual exclusion can be used to manage threads' accesses of the queue.

References

1. Kumar, S., Hughes, C. J., and Nguyen, A. 2007. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. *SIGARCH Comput. Archit. News* 35, 2 (Jun. 2007), 162-173.
2. Recipe 475160: TaskQueue - <http://code.activestate.com/recipes/475160/>
3. Introduction to Graph with Breadth First Search (BFS) and Depth First Search (DFS) Traversal Implemented in JAVA, <http://www.codeproject.com/KB/java/BFSDFS.aspx>
4. Cilk++ Solution Overview. <http://www.cilk.com/multicore-products/cilk-solution-overview>
5. CUDA Programming Guide, http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf