Synchronization in a Thread-Pool Model and its Application in Parallel Computing*

Masaaki Mizuno Liubo Chen Virgil Wallentine
Department of Computing and Information Sciences
Kansas State University, Manhattan, KS 66506
{masaaki, lch6388, virg}@cis.ksu.edu
Tel: (785) 532-6350 Fax: (785) 532-7353

Keywords: Grid Computation, Synchronization, Thread-pool model

1 Introduction

In this paper, we consider synchronization in a *thread-pool model* and its application in scientific parallel computing. Consider a server system which processes requests from clients, where requests are either to *query* or to *update* a unique shared variable. The server receives a request and passes the request to a job thread. The job thread processes the request and sends a reply back to the client. There are two types of concurrent models to implement such a system: a thread-per-request model and a thread-pool model[2]. In the thread-per-request model, every incoming request causes a new job thread to be spawned to process it. In the thread-pool model, the system maintains a pool (or pools) of pre-spawned job threads. When a new request arrives, the request is passed to a free job thread in the pool. After the request is processed, the thread is returned to the pool to execute another request. The advantages of the thread-pool model include that the system can have a bound on the number of threads and reduce the cost of thread creation. The thread-pool model is often used in embedded systems and web servers. We have developed a synchronization methodology well-suited for the thread-pool model.

Synchronization has been well-studied in operating systems and parallel computing. Almost all such synchronization codes rely on Operating System (OS) primitives that block executing threads for synchronization. Such solutions are valid for the thread-per-request model. However, in the thread-pool model where threads are a limited resource, often such solutions are not efficient or even not acceptable. For example, consider the producers/consumers with a bounded buffer problem found in many textbooks[1]. Suppose that there are 10 buffer entries and that the thread pool maintains less than 10 threads. Then, if the first 10 requests are from consumers, the system becomes deadlocked. In our approach, when an execution needs to be blocked, synchronization code releases and returns the executing thread to the thread pool, rather than blocking the thread, so that the thread can execute another job. This is done by returning from one function and calling another function. As a result, switching from one job to another is done extremely fast. We applied our synchronization approach to grid-computation and obtained good results. The results show that our thread-pool model synchronization approach can speed up many applications, such as web servers and embedded systems.

*If the paper is accepted, Liubo Chen will present the paper.

2 Job objects and thread pool

2.1 Thread pool

In the thread-pool model, each job code is often implemented in a specific method (we call it *execute*()) in a special class (we call such a class a *job* class). A job class implements a generic job interface (we name it *GenericJob*), which declares the abstract method *execute*(). At run time, when the system receives a request, it instantiates an associated job object and places it in the thread pool. In the thread pool, a free thread is assigned to the job object and executes its *execute*() method. When *execute*() terminates, the job object is deleted and the executing thread is returned to the thread pool. Our server example has two job classes, Inquire and Update. Upon an arrival of an inquire (an update) request, the system instantiates a job object of Inquire (Update) and places it in the thread pool.

Many thread pool implementations are available. Some are found in the Web. However, some implementations are very complex. In this paper, we present a simple implementation of a thread pool in Java in a class denoted *ThreadPool*. *ThreadPool* maintains a queue of ready jobs of type *GenericJob*, denoted *readyQueue*. We use JFC's LinkedList class to implement *readyQueue* (therefore, downcasting is required when a job object is dequeued). *ThreadPool* provides two methods:

- **void** *addJob(GenericJob job)*: adds *job* in *readyQueue* and wakes up a job thread if there is any sleeping one. Its implementation is

```
a1:  public synchronized void addJob(GenericJob job) {
a2:  readyQueue.add(job); notify();
a3:  }
```

- *GenericJob getJob(void)*: returns one ready job found in *readyQueue*. If there is no ready job in the queue, it blocks the calling thread. Its implementation is

```
b1:  private synchronized GenericJob  getJob() {
b2:      while (readyQueue.isEmpty()) try {wait();} catch (InterruptedException e){}
b3:      return (GenericJob) readyQueue.removeFirst();
b4:  }
```

Let *threadPool* be an instance of *ThreadPool*. The body of threads managed by the thread pool is:

- c1: **while**(true) {*(threadPool.getJob()).execute();* }

2.2 Synchronization method and job object

Synchronization is performed by a synchronization method. It maintains a queue of *GenericJob* objects (called *waitQueue*) to place waiting jobs. If the synchronization condition is satisfied and the execution can proceed, the synchronization method performs some book keeping job (*i.e.*, changing state, etc.) and *returns true*. If the synchronization condition is not satisfied, the execution must be blocked. In such a case, the synchronization method places the job into *waitQueue* and *returns false*. Whenever state related to synchronization changes and some jobs may need to be awaken, a test is performed; if the synchronization condition is satisfied, it moves a job (jobs) from *waitQueue* to *readyQueue*. Further discussions on synchronization methods are given in the next section.

Each job class implements *GenericJob* and provides the implementation of *execute*(). We divide the execution into phases. The phase begins with 0 and is advanced by one every time the execution passes a synchronization method. To keep track of the current phase, each job object maintains an integer variable *phase*. The phase value is used to determine what part of the code segment the execution resumes the next time when the job object is assigned to a thread. The body of *execute*() is of form:

```
d1:  switch (phase) {
d2:  case 0:
d3:       sequential code for phase 0
d4:       if (¬synchronizatoinMethod(this)) { phase = 1; return; } // job is blocked
d5:  case 1:
d6:                 ...
d7:       }
d8:  }
```

The bodies of all **case** statements have the same structure as that of **case** 0. Note that the return statement in line d4 is for the executing thread to return to line c1 in the thread body. The phase value is updated to the next phase only when the execution returns the thread to the thread pool so that the next time when a thread is assigned to the job, it executes from the next case statement.

3 Application in scientific parallel computation

3.1 Jacobi Iteration

We applied our thread pool implementation to a (non-optimized) Jacobi Iteration on finite differential approximation to partial differential equations for a heat transfer problem. We use the algorithm given in [1]. To compare the performance, we have developed two implementations: thread-per-request implementation (assign one thread to each process) and thread-pool implementation (assign one job object to each process and have threads in the thread pool execute the job objects).

Given a spatial region with an evenly spaced grid of points and values for points on the boundaries of the region, the goal is to approximate the steady-state solution for interior points. In each iteration the new value of a point is set to a combination of the old values of four neighboring points. Assume that the grid is $n * n$ square and that it is surrounded by a square of boundary points. We provide two $n+2$ by $n+2$ matrices: one (named *grid*) to represent the grid and its boundary and another (named *new*) for the set of new values. The 0th and $(n+1)^{st}$ rows and columns store boundary values.

Let PR be the number of the processes. We assign each process a horizontal strip of size $n/PR * n$ to update. To ensure that every process has completed one update phase before any process begins the next one, we use barrier synchronization after each update phase. Each process executes the (pseudo) code given below:

```
double grid[n+2][n+2], new[n+2][n+2];   // index starts with 0
int HEIGHT = n / PR;   // assume PR evenly divides n
double maxdiff[PR];

process worker(id) {
   int firstRow = (id-1) * HEIGHT - 1;
   int lastRow = firstRow + HEIGHT - 1;
   double mydiff = 0.0;
   initialize my strips of grid and new, including boundaries;
   barrier(id);
   for (iters = 1; i <= MAXITERS; i+=2) {
      for (i = firstRow; i <= lastRow; i++) {
         for (j = 1; j <= n; j++)
            new[i][j] = (grid[i-1][j]+grid[i+1][j]+grid[i][j-1]+grid[i][j+1])*0.25;
      }
      barrier(id);
      for (i = firstRow; i <= lastRow; i++) {
```

```
      for (j = 1; j <= n; j++)
        grid[i][j] = (new[i-1][j]+new[i+1][j]+new[i][j-1]+new[i][j+1])*0.25;
    }
    barrier(id);
    for (i = firstRow; i <= lastRow; i++) {
      for (j = 1; j R <= n; j++) mydiff = max(mydiff, abs(grid[i][j]-new[i][j]));
    }
    maxdiff[w] = mydiff;
    barrier(id);
    maximum difference is the max of the maxdiff[*]; break if the condition is met
  }
}
```

3.2 Synchronization

Barrier synchronization maintains an array of boolean (*arrived[PR]*), one entry for each process. When process "id" arrives at the barrier, it sets *arrived*[id] to true. Barrier synchronization for request-per-thread (that is, each process is mapped to a different thread) implementation is given below. It is a **synchronized** method and uses Java synchronization primitives **wait**() and **notifyAll**().

```
e1:  arrived[id] = true; boolean go = true;
e2:  for (i = 0; i < PR; i++) if (! arrived[i]) { go = false; break;}
e3:  if (go) { for (i = 0; i < PR; i++) arrived[i] = false;  notifyAll(); }
e4:  else {try {wait();} catch(InterruptedException e){}}
```

When a thread arrives at the barrier, it sets its own *arrived* entry to true. If at least one entry of *arrived* is false, the thread must be blocked. If all entries of *arrived* are true, the thread changes all entries of *arrived* to false and wakes up all blocked threads.

Barrier synchronization for the thread pool implementation is given below. The differences from the synchronization for the thread-per-request implementation are:

1. The **notifyAll**() method is replaced by moving jobs from *waitQueue* to *readyQueue*.

2. The **wait**() method is replaced by placing the job object in *waitQueue* and returning false (when **wait**() is not called, it returns true).

   Recall that when the synchronization method returns false, *execute*() in the job object returns (d4). That is, the executing thread returns to c1 and fetches a next ready job from *readyQueue*.

More general discussions on how to derive a synchronization method for the thread-pool model from that for the thread-per-request model are given in the full paper.

```
f1:  arrived[id] = true; boolean go = true;
f2:  for (i = 0; i < PR; i++) if (! arrived[i]) { go = false; break;}
f3:  if (go) {
f4:    for (i = 0; i < PR; i++) arrived[id] = false;
f5:    while (! waitQueue.isEmpty()) threadPool.addJob((GenericJob)waitQueue.removeFirst());
f6:    return true;
f7:  } else { waitQueue.add(job); return false; }
```

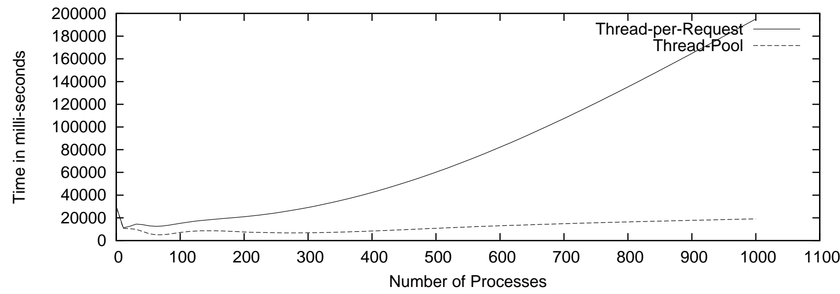

Figure 1: Performance of Thread-Per-Request vs. Thread-Pool

3.3 Performance

We run both implementations on a Linux system with $4 * 450$MHz Xeon processors. The size of the grid is $1000 * 1000$. We change the number of processes from 1 to 1000. In the thread-per-request implementation, the number of threads is equal to the number of processes, and in the thread-pool implementation, the number of job threads is fixed to 10. The results are shown in Figure 1. When the number of processes is small (1-10), there is not much difference in performance. It should be noted that when the number of processes is increased from 1 to 4, the performance in both implementations improves linearly. This is because Linux assigned each thread to a different processor. The differences in performance become clear when the number of processes increases. When the number of processes is 1000, the performance of the thread-pool implementation is over 10 times as good as that of the thread-per-request implementation (19077 msec vs. 195122 msec). In the thread-pool implementation, even though the number of processes becomes 1000, the decrease in performance is less than 50% (about 10386 msec when the number of processes is 20 vs. about 19077 msec when that is 1000). This shows extremely low overhead of job switching and synchronization in the thread-pool implementation combined with our synchronization method.

Note that if we apply the synchronization code for the thread-per-request model in the thread-pool implementation, deadlock will result unless the number of job threads in the thread-pool is equal to or greater than the number of processes. From the above results, we conclude that our synchronization approach can potentially speed up many applications that use the thread-pool model, such as web servers and embedded systems,

4 Conclusion

In our approach of synchronization for the thread-pool model, when a job needs to be blocked for synchronization, the executing thread is released and returned to the thread pool, rather than being blocked. In our performance test in Jacobi Iteration, the thread-pool implementation using our synchronization method shows tremendous speed improvement compared with the thread-per-request implementation.

In the full paper, we will give more general discussions on how to derive a synchronization method for the thread-pool model from that for the thread-per-request model (which is usually found in literature). We will also discuss how to handle synchronization in a loop (in Jacobi Iteration, barrier synchronization appears in a loop, and we actually needed to modify the job object and the job thread implementation slightly to handle such a case).

References

[1] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley, 2000.

[2] D. C. Schmidt and S. Vinoski. Comparing alternative programming techniques for multi-threaded corba servers: Thread pool. *SIGS C++ Report Magazine*, 1996.