May Column: Multi-threaded Algorithm Implementations

By George T. Heineman
June 18, 2009 | Comments: 2

This is the seventh of a series of monthly columns in the blog associated with the **Algorithms in a Nutshell** book, published October 2008 by **O'Reilly Media, Inc.**

The past six blog entries (which you can find at the end of **last month's column**) described in greater detail algorithms drawn from the chapters of our book. In the remaining entries for this year, we will "branch out" and cover some new territory, or at least provide a different perspective to existing results from the book. At the end of last month's column, I promised to explore the sorting algorithm IntroSort but that discussion will have to wait until a future blog entry because I got caught up instead with the notion of multithreading. In computer science a thread can be considerably more "light-weight" than an operating system process. With the increasing support for and popularity of multi-core processing chips, the use of threads has the potential to take advantage of these multi-core chips with only minimal change to the underlying algorithm implementation.

In this column we explore how to use threads in Java and C, on both Windows machines and Unix architectures. For this column we focus on the following examples:

- **Multi-threaded QuickSort** in C with a single helper thread
- **Multi-threaded QuickSort** in Java with a single helper thread
- How to **generalize** such a solution to support an arbitrary number of helper threads
- **Multi-threaded ConvexHull** in Java with a single helper thread
- An example of when threading does not help: **Multi-threaded Nearest-Neighbor**
- **Lessons Learned**

All of the code provided for this month's Blog is new. While developing this code, I actually decided to implement a new ConvexHull algorithm known as QuickHull, but that discussion will have to wait until next Month's June Blog entry.

Download May Code Samples

You can download the code samples described in this column from the *code.zip* file found at **code.zip** (206,753 bytes). The following examples were tested on a standard Windows desktop computer running **Eclipse Version 3.4.1**. The **JDK version** is 1.6.0_13. In fact, the *code.zip* file is actually an exported Eclipse project, which means you will be able to easily load this month's code into your Eclipse workspace. Should you choose to compile and execute these examples outside of Eclipse, simply ensure that your CLASSPATH variable is properly configured to use the compiled sources of the ADK.

To bring this month's code into Eclipse, simply unzip the code.zip file found at code.zip. Once you have unzipped the files, in Eclipse choose to create a New Java Project. For the "Project name" enter "May 2009". Then choose the "Create project from existing source" radio button and browse to the location where you unzip'd the code.zip file. Make sure you select the directory named "May_2009". Then click Finish. If you have already imported the JavaCode project from the ADK into this Eclipse workspace, then the code should compile cleanly; if you have not, then you will need to tell this Eclipse project about the location of the compiled Jar file for the ADK in which the implemented algorithms can be found.

Multi-threaded QuickSort in Java with a single helper thread

QuickSort is able to decompose a larger sorting problem into two smaller subproblems.

```
/**
 * Straight quicksort.
 *
 * @param ar           array to be sorted.
 * @param cmp          comparison function.
 * @param left         lower bound index position  (inclusive)
 * @param right        upper bound index position  (inclusive)
 */
void quickSort(void **ar, int(*cmp)(const void *,const void *),
               int left, int right) {
  if (left < right) {
    int p = partition(ar, cmp, left, right);
    quickSort(ar, cmp, left, p-1);
    quickSort(ar, cmp, p+1, right);
  }
}
```

Fig. 1: QuickSort implementation in C

The two subproblems *sort ar[left, pi-1]* and *sort ar[p+1, right]* are independent problems and, theoretically, can be solved at the same time. It has long been hoped that sophisticated compilers would be able to detect such situations and automatically generate code to construct and execute threads. Until such time, however, we have to take care of this work manually.

Let's first start by constructing a solution with a single helper thread that will be used to complete one of these quickSort subtasks. However, thread resources are precious and we must be careful when we use extra threads. Should we use it on a large subproblem or only for small ones? Let's decide to use a helper thread to solve a subproblem if (a) there currently is no helper thread working; and (b) the requested number of elements to sort is below a specific threshold value. We can experimentally evaluate different threshold values to see what might be a good choice. Rather than choose an absolute threshold value, we choose to set the threshold value to N/R where N is the number of elements to be sorted and R is a user-specified ratio value. Thus, when R=1, the threshold value is set to N and the program tries to always solve subproblems with the helper thread, should it be available. When R=MAXINT then the threshold value is set to ZERO and therefore the program never uses a helper thread. We constructed an experiment to run 20 trials of sorting N={65536, 131072, 262144, 524288, and 1048576} random 26-character strings using values of R from 1 to 50; there is one final run for R=MAXINT. The following graphs show the results of execution on a Linux box with four Dual-Core AMD Opteron(tm) Processor chips.

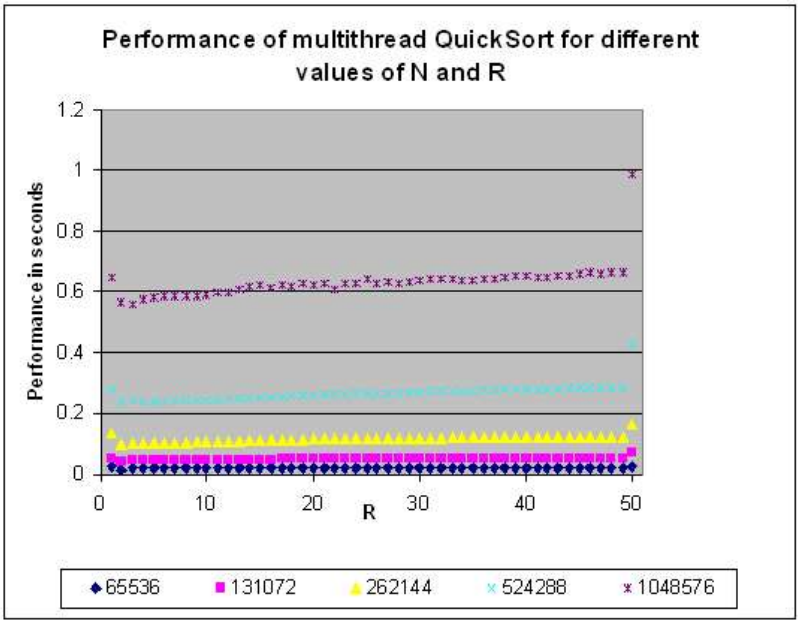

Fig. 2: Performance of Multi-Thread QuickSort for varying N and R.

Reading the graph from left to right, you can see that the first data point (R=1) reports the performance that tries to immediately begin using the helper thread while the last data point (R=51) reports the result when no helper thread is ever used. When we compute the "speedup factor" from time T1 to a smaller time T2, we use the equation T1/T2. Simply using an extra thread shows a speedup factor of 1.19 for 65,536 strings to 1.52 for 1,048,576. This is a very nice return on investment for a small programming change!

n	Speedup of Multi-thread to no thread
65,536	1.19
131,072	1.32
262,144	1.26
524,288	1.51
1,048,576	1.52
2,097,152	1.36
4,194,304	1.61
8,388,608	1.38

Table 1: Speedup for (ratio=1) vs. (ratio=MAXINT).

Returning to **Fig. 2**, you can see that the best improvement occurs near where R=2, and as N increases in size, it appears that the best result appears when R=3 or R=4. My interpretation is that there is a built-in overhead to using threads, and one shouldn't automatically dispatch a new thread of execution without some assurance that the primary thread will not have to wait and "block" until the helper thread completes its execution. It seems the "sweet spot" is near where R=4.

Another way to view the results from **Fig. 2** is to compare the ratio of performance as compared against R=1 (always initiating a helper thread when available). As you can see in **Fig. 3**, there is an initial performance benefit for small R (such as R=2, 3 or 4) but over time as R increases the benefit becomes smaller. Indeed for large enough n (greater than 131,072, for example), the actual performance of delaying a threaded subproblem only for "small enough" problems actually becomes worse than simply multithreading at every opportunity. The final point (R=51) in **Fig. 3** reflects the numeric values from **Table 1**.

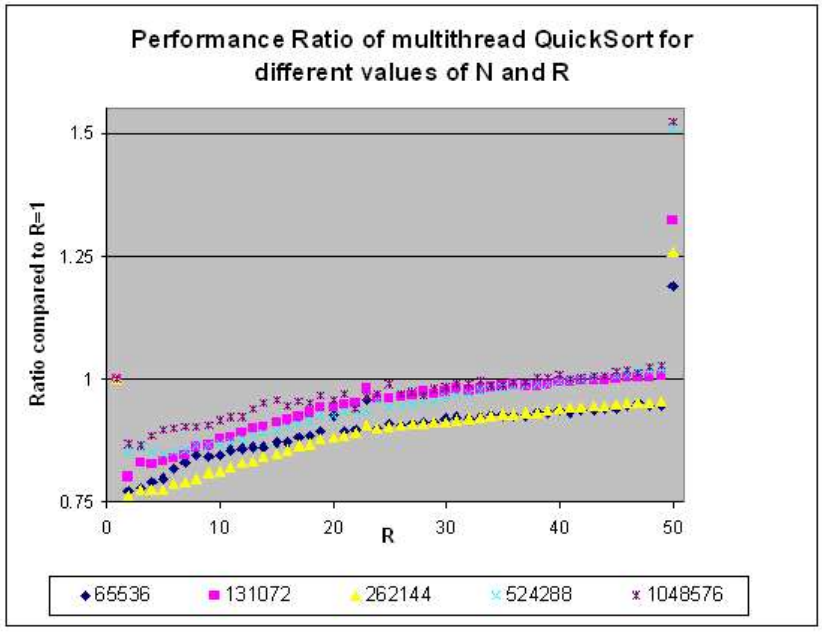

Fig. 3: Comparing the performance ratio of multi-thread QuickSort with different N and R values.

So let's see the solution. The fully coded solution is available in the `multithreadQsort.c` file found in this month's blog code file. First we show the modified `quickSort` routine, which we label as `quickSort2` . This code executes within a primary thread. `helpRequested` has the value 0 when no helper thread is currently processing a subproblem. For each of the two subproblems, there is a simple check to see whether the primary thread should continue the recursive `quickSort2` function call. As you can see, the separate helper thread is dispatched to compute a subproblem only if (a) no help is currently requested and (b) the size of the subproblem is smaller than the specified threshold value. This logic is applied to both the sorting of the left sub-array as well as the sorting of the right sub-array.

To dispatch the helper thread to solve a subproblem, the primary thread sets the value of `helpRequested` to 1 after specifying to the helper the range of the sub-array which it is being asked to sort (either ar [left, p-1] or ar [p+1, right]).

```
/**
 * Quicksort that delegates to helper thread the sorting of a
 * sub-array should a thread be available and if the problem
 * size is small enough to warrant such an action.
 *
 * @param ar            array being sorted
 * @param cmp           comparison function to use
 * @param helper        context for helper thread
 * @param left          lower bound index position  (inclusive)
 * @param right         upper bound index position  (inclusive)
 */
void quickSort2(void **ar, int(*cmp)(const void *,const void *),
                THREAD_INFO *helper, int left, int right) {
  int p,n;

  if (left < right) {
    p = partition(ar, cmp, left, right);
    n = p - left;

    /* If already requested help or problem too big, recurse. */
    if (helpRequested || n >= helper->threshold) {
      quickSort2(ar, cmp, helper, left, p-1);
    } else  {
      /* initialize helper variables and ask for help. */
      helper->left = left;
      helper->right = p-1;

      helpRequested = 1;
    }

    n = right - p;
    if (helpRequested || n >= helper->threshold) {
      quickSort2(ar, cmp, helper, p+1, right);
    } else {
      /* initialize helper variables and ask for help. */
      helper->left = p+1;
      helper->right = right;

      helpRequested = 1;
    }
  }
}
```

Fig. 4: QuickSort implementation in C

To understand how the helper thread completes its task, we need to look at the `quickSort1` function which executes within the helper thread, as shown in **Fig. 5**. This thread runs until it is told that it is done, during which it checks to see if any help has been requested. Once `helpRequested` is "raised" (or otherwise set to 1) it immediately launches the single-threaded implementation of `quickSort` which was shown in **Fig. 1**. When it completes, it sets `helpRequested` back to 0 to announce it is once again free.

Using Pthreads, the popular POSIX standard, the helper thread will first wait until all threads are ready to go (at the first `pthread_barrier_wait` invocation) and then, once done, it will wait a second time until all threads have synchronized to completion. It is worth pointing out the

condition for the while loop in `quickSort1` ; it would be incorrect to simply check `!done` since there may be a race condition that occurs if the primary thread decides that its final subtask needs to be performed within a helper thread. In this case, it is technically possible that both `done` and `helpRequested` are non-zero at the same time. For this reason, the while loop must check for either `!done` or helpRequested.

```
/**
 * Helper thread executes single-thread QuickSort.
 *
 * @param arg    thread context
 */
void *quickSort1(void *arg) {
  THREAD_INFO *context = (THREAD_INFO *) arg;

  pthread_barrier_wait(&barrier);

  /** Tight spin loop. Only do work if requested. */
  while (!done || helpRequested) {
    if (helpRequested) {
      quickSort(context->ar, context->cmp,
                context->left, context->right);

      helpRequested = 0;
    }
  }

  pthread_barrier_wait(&barrier);
}
```

Fig. 5: QuickSort implementation in C, continued...

The primary thread executes the `quickSortEntry` function as shown in **Fig. 6**. It synchronizes with the helper thread and then executes `quickSort2` as shown in **Fig. 4**. Once this function returns, `quickSortEntry` disables the helper thread and waits for its completion.

```
/**
 * Entry point for primary thread.
 *
 * Once synchronized, launches the primary sort routine.
 *
 * @param arg    thread context
 */
void quickSortEntry(void *arg) {
  THREAD_INFO *context = (THREAD_INFO *) arg;

  /** Wait until all threads ready to go. */
  pthread_barrier_wait(&barrier);

  /** When we get here, all threads are synchronized.
   * numElements is a global storing the number of elements. */
  quickSort2(context->ar, context->cmp, context->helper,
             0, numElements-1);

  /** Stop Helper thread. and wait for all threads before exit. */
  done=1;
  pthread_barrier_wait(&barrier);
}
```

Fig. 6: QuickSort implementation in C, continued...

Please read the `multithreadQsort.c` file for full details on the `THREAD_INFO` data structure and how threads are properly initialized and launched.

Multi-threaded QuickSort in Java with a single helper thread

We can replicate the one-helper solution in Java, as shown in **Fig. 7**. This time, the elements being sorted are all integers rather than randomized 26-character strings used earlier.

Because of the simplified threading model in Java, the code is simpler. Whenever `qsort2` determines that the number of elements to be sorted is smaller than the threshold and the helper is not working, it spawns a new thread. When this new thread completes its execution, it resets `helpRequested` back to **false**.

```
/**
 * Sort using quicksort method with separate helper thread.
 * Both left and right and bounded by [0, ar.length).
 * @param left     The left-bounds within which to sort
 * @param right    The right-bounds within which to sort
 */
public void qsort (final int left, final int right) {
  qsort2 (left, right);

  // wait until helper is done (if it is still executing).
  while (helpRequested) { }
}

/**
 * Sort using quicksort method with separate helper thread.
 */
public void qsort2 (final int left, final int right) {
  if (right <= left) { return; }

  // partition
  int p = pi.selectPivotIndex (ar, left, right);
  final int pivotIndex = partition (left, right, p);

  // If helper working or problem too big, continue with recursion
  int n = pivotIndex - left;
  if (helpRequested || n >= threshold) {
    qsort2 (left, pivotIndex-1);
  } else {
    helpRequested = true;

    // complete in separate thread
    new Thread () {
      public void run () {
        qsort2 (left, pivotIndex - 1);
        helpRequested = false;
      }
    }.start();
  }
```

```
  // If helper working or problem too big, continue with recursion
  n = right - pivotIndex;
  if (helpRequested || n >= threshold) {
    qsort2 (pivotIndex+1, right);
  } else {
    // complete in separate thread
    helpRequested = true;

    new Thread () {
      public void run () {
        qsort2 (pivotIndex+1, right);
        helpRequested = false;
       }
    }.start();
  }
}
```

Fig. 7: Multi-threaded QuickSort implementation in Java

The code properly works since only the primary thread reads the `helpRequested` variable or sets it to **true**, while the helper thread only resets the value of `helpRequested` back to **false**. It is true that certain timing considerations could cause the primary thread to miss an opportunity to launch the helper thread (because `qsort2` could have completed and the thread may not have yet reset `helpRequested`). However, such a situation will not lead to either deadlock or livelock, so we can safely use this simpler implementation. When we execute this code on the same Unix machine as earlier, we witness similar performance trends.

If we wish to use additional helper threads, we have to change the underlying solution we have shown for both C and Java. The next section shows our solution using Java.

How to generalize such a solution to support an arbitrary number of helper threads.

The code from **Fig. 7** cannot easily be extended to multiple threads because we need to add bookkeeping to remember which threads are executing, and we have to avoid spawning either too many threads or deadlocking. The `algs.model.multithread.array.QuickSort` implementation in this month's blog code base shows how this works.

The primary change is that we introduce an integer variable, `helpersWorking` which records the number of active helper threads working. Because multiple threads must access this variable we use a *mutex object*, which ensures mutually-exclusive access to this shared variable. **Fig. 8** shows the `qsort2` implementation.

```
/**
 * Multi-threaded quicksort method entry point.
 * Both left and right and bounded by [0, ar.length).
 * @param left     The left-bounds within which to sort
 * @param right    The right-bounds within which to sort
 */
private void qsort2 (final int left, final int right) {
  if (right <= left) { return; }

  // partition
  int p = pi.selectPivotIndex (ar, left, right);
  final int pivotIndex = partition (left, right, p);

  // are all helper threads working OR is problem too big?
  // Continue with recursion if so.
  int n = pivotIndex - left;
  if (helpersWorking == numThreads || n >= threshold) {
    qsort2 (left, pivotIndex-1);
  } else {
    // otherwise, complete in separate thread
    synchronized(helpRequestedMutex) {
      helpersWorking++;
    }

    new Thread () {
      public void run () {
        // invoke single-thread qsort
        qsortN (left, pivotIndex - 1);
        synchronized(helpRequestedMutex) {
           helpersWorking--;
        }
      }
    }.start();
  }

  // are all helper threads working OR is problem too big?
  // Continue with recursion if so.
  n = right - pivotIndex;
  if (helpersWorking == numThreads || n >= threshold) {
    qsort2 (pivotIndex+1, right);
  } else {
    // otherwise, complete in separate thread
    synchronized(helpRequestedMutex) {
      helpersWorking++;
    }

    new Thread () {
      public void run () {
        // invoke single-thread qsort
        qsortN (pivotIndex+1, right);
        synchronized(helpRequestedMutex) {
          helpersWorking--;
        }
      }
    }.start();
  }
}
```

Fig. 8: Multi-threaded QuickSort implementation in Java with multiple helper threads

Whenever a thread is spawned, the `helpersWorking` variable is incremented, and the thread itself will decrement this same value upon completion. Using the mutex variable, `helpRequestedMutex` and the ability in Java to **synchronize** a block of code for exclusive access, this implementation safely updates the `helpersWorking` variable. As with our original C implementation with a single helper, `qsort2` invokes the single-threaded `qsortN` method within

its helper threads. This ensures that only the primary thread is responsible for spawning new threads of computation.

For this design, why is it so important to prevent helper threads from spawning new helper threads? Should this be allowed to happen, then the "first" helper thread would have to synchronize with these "second" threads so the "second" threads would only begin to execute after the "first" helper thread had properly partitioned the array. Implementing this logic will prove challenging.

Let's now consider how well this multi-helper thread solution works when compared to the earlier single-helper thread solution. In order to compare like-minded executions with each other, the following graphs compare the Java single-helper solution against the Java multi-helper solution. Also the QuickSort algorithms are going to be sorting random floating point numbers from the range [0, 1]. We have several parameters that we consider:

- Size N of the array being sorted: This falls in range {65,536 to 1,048,576}
- Ratio threshold N/R below which a helper thread computes a sub-problem. We experimented with values of R in the range {1 to 20} and also MAXINT.
- Number of helper threads available: we experimented with 1 to 9 helper threads
- Partition method to use: we tried both "select a random element" and "select the rightmost element".

The parameter space thus amounts to 1,890 unique combination of these parameters. The `MultiThreadDriver` Java file contains the code to generate results for these experiments. For convenience, we include an Excel spreadsheet file `ComputationResults.xls` within this month's blog entry.

In all of the ensuing graphs, we use the "select rightmost element" as the partition algorithm. In general, we have found there is a noticeable performance slowdown of about 5% in using the random number generator across all experiments.

If R=1, then the algorithm chooses to initiate a helper thread whenever it can. The graph in **Fig. 9** shows there is a speedup factor of 1.61 for N=1,048,576 when there may be one helper thread. However, for all values of N, adding more threads actually worsens the situation. Thus with up to 9 helper threads for N=1,048,576 the improvement converges on an improvement of about 1.49.

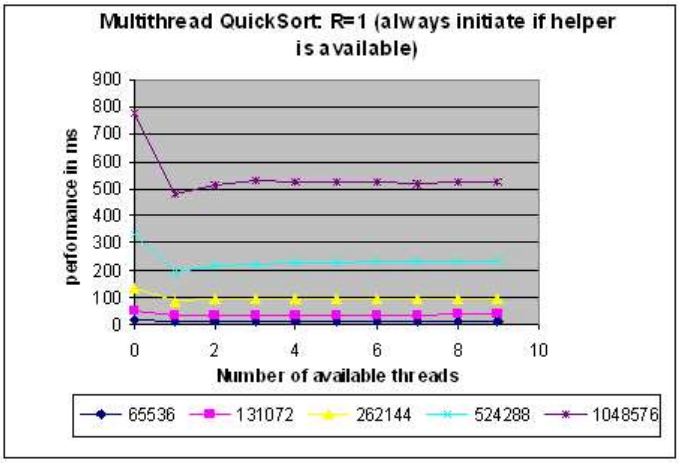

Fig. 9: Multi-threaded QuickSort : R=1 and up to 9 helper threads.

Now consider the "sweet spot" we had identified earlier, namely for R=4. The execution performance is graphed in **Fig. 10**. The results are quite favorable. For N=1,048,576 one sees continuous improvement, converging on a final speedup factor of 2.3. Thus using threads judiciously has cut the performance time of QuickSort by more than half.

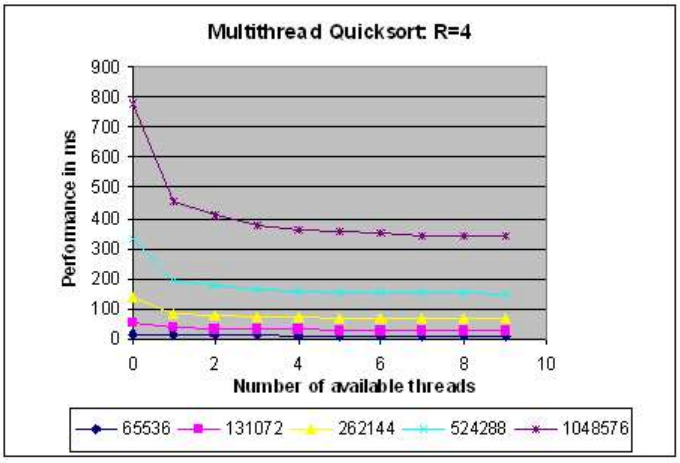

Fig. 10: Multi-threaded QuickSort : R=4 and up to 9 helper threads.

Research in speedup factors for parallel algorithms shows there are inherent limitations to how much extra threading or extra processing will actually help a specific algorithmic implementation. Under these restrictions, I feel quite satisfied in producing a multi-threaded QuickSort implementation that achieved a speedup factor greater than 2. The primary reason we were able

to achieve this speedup is that the individual subproblems of recursive QuickSort are entirely independent, and there will be no contention for shared resources by the multiple threads. If other problems share this same characteristic, then they too should be able to benefit from our multi-threaded approach. We do not have far to look to find a candidate problem.

Multi-threaded ConvexHull in Java with a single helper thread

Given a set of two-dimensional points P in the Cartesian plane, the *convex hull* is the smallest convex shape that fully encloses all points in P. **Fig. 11** shows the computed convex hull for a small sample points set. One can verify that a point **x** should **not** belong on the convex hull by simply finding three points in P that form a triangle that contains **x.** For example, the point **6** can be eliminated since it is enclosed within the triangle formed by points **4**, **5** and **7**. The convex hull is the clockwise ordering of points labeled **L0** through **L7**.

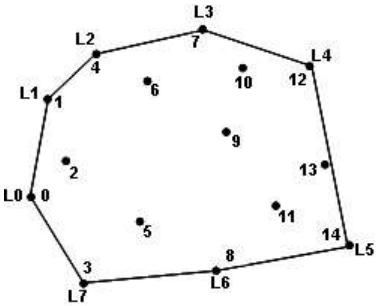

Fig. 11: Computed Convex Hull for a sample of points.

Using a brute force approach, one can generate all possible triangles from the set P and eliminate all points from P that fall within a triangle. As shown in our book, such a brute force algorithm exhibits $O(n^4)$ performance. Surely we can do better than this!

The implementation of the Convex Hull Scan algorithm is listed in **Fig. 12**.

```
/**
 * Use Andrew's algorithm to return the computed convex hull for
 * the input set of points.
 *
 * Points array must have at least three points to be meaningful.
 * If it does not, then the sorted array is returned as the "hull".
 *
 * This algorithm will still work if duplicate points are found in
 * the input set of points.
 *
 * @param points     a set of (n ≥ 3) two dimensional points.
 */
public IPoint[] compute (IPoint[] points) {
  // sort by x-coordinate (and if ==, by y-coordinate).
  int n = points.length;
  new HeapSort ().sort(points, 0, n-1, IPoint.xy_sorter);
  if (n < 3) { return points; }

  // Compute upper hull by starting with leftmost two points
  PartialHull upper = new PartialHull(points[0], points[1]);
  for (int i = 2; i < n; i++) {
    upper.add(points[i]);
    while (upper.hasThree() && upper.areLastThreeNonRight()) {
      upper.removeMiddleOfLastThree();
    }
  }

  // Compute lower hull by starting with rightmost two points
  PartialHull lower = new PartialHull(points[n-1], points[n-2]);
  for (int i = n-3; i >=0; i--) {
    lower.add(points[i]);
    while (lower.hasThree() && lower.areLastThreeNonRight()) {
      lower.removeMiddleOfLastThree();
    }
  }

  // remove duplicate end points when combining.
  IPoint[] hull = new IPoint[upper.size()+lower.size()-2];
  System.arraycopy(upper.getPoints(),
     0, hull, 0, upper.size());
  System.arraycopy(lower.getPoints(),
     1, hull, upper.size(), lower.size()-2);

  return hull;
}
```

Fig. 12: Implementation of Convex Hull Scan

The key idea of this algorithm is to break the problem into two: Compute the upper partial hull and then the lower partial hull. Once these are computed, they are fused together to form the actual convex hull. This algorithm first sorts the points by x-coordinate from left to right (breaking ties using the y-coordinate). It then "walks" through the points from left to right to form the upper hull and then walks again through the points from right to left to form the lower hull. One can readily see that these two partial hulls can be computed at the same time since they are independent problems.

The two inefficiencies in this implementation (which we will call the ADK implementation) are: (1) the use of HeapSort where QuickSort would be noticeably faster; and (2) the waste of the invocations to getPoints() of the partially computed hulls. If you review the code, you will see that this method creates and populates an array of points which represents the partial hull, only to simply copy this array into the computed solution, `hull`. We will address both of these inefficiencies even as we work to involve multiple helper threads into the implementation.

Note that we can only use multiple threads once the initial points are sorted, and we must synchronize and wait for both threads to complete before we fuse the two partial results together again. The code to complete this task is shown in **Fig. 13**.

In our original ConvexHullScan implementation we used a HeapSort algorithm to sort the array of points. The code in this month's blog now uses a multi-threaded Quick Sort implementation where the elements being sorted are two-dimensional points in the Cartesian plane. We have to create a new class to handle situations where the elements being sorted do not automatically provide a comparable method to compare to elements. You can review the `QuickSortExternal` class to see how this is done. For the implementation shown below, we choose to use QuickSort using our favorite threshold ratio of 4, with a number of helper threads.

```
public IPoint[] compute (final IPoint[] points) {
    // sort by x-coordinate (and if ==, by y-coordinate).
    final int n = points.length;

    // sort with available threads, using R=4 sweet spot.
    QuickSortExternal  qs =
      new QuickSortExternal  (points, IPoint.xy_sorter);
    qs.setPivotMethod(qs.lastSelector());
    qs.setNumberHelperThreads(numThreads);
    qs.setThresholdRatio(4);

    // trivial cases can return now.
    if (n < 3) { return points; }

    // from this point on, we only use two threads.
    final PartialHull upper =new PartialHull(points[0],points[1]);
    final PartialHull lower =new PartialHull(points[n-1],points[n-2]);

    Thread up = new Thread() {
      public void run() {
        // Compute upper hull by starting with leftmost two points
        for (int i = 2; i < n; i++) {
          upper.add(points[i]);
          while (upper.hasThree() && upper.areLastThreeNonRight()) {
            upper.removeMiddleOfLastThree();
          }
        }
      }
    };

    Thread down = new Thread() {
      public void run() {
        // Compute lower hull by starting with rightmost two points
        for (int i = n-3; i >=0; i--) {
          lower.add(points[i]);
          while (lower.hasThree() && lower.areLastThreeNonRight()) {
            lower.removeMiddleOfLastThree();
          }
        }
      }
    };

    // start both threads and wait until both are done.
    up.start();
    down.start();
    try {
      up.join();
      down.join();
    } catch (InterruptedException ie) {
      System.err.println("Multithreaded execution interrupted.");
    }

    // remove duplicate end points when combining. Transcribe the
    // partial hulls into the array return value.
    IPoint[] hull = new IPoint[upper.size()+lower.size()-2];
    int num = upper.transcribe (hull, 0);
    lower.transcribe (hull, num-1, lower.size() - 2);

    return hull;
}
```

Fig. 13: Multi-threaded implementation of Convex Hull Scan

As you can see in the **Fig. 13**, we are restricted to using only two threads for constructing the two partial hulls, even though we may have used a larger number of threads during the QuickSort phase of the algorithm. At the end of this method, we avoid the wasteful allocation of unneeded arrays by modifying the `PartialHull` class (included just within this month's blog code) which simply transcribes into the `hull` solution the points from the partial hulls.

As shown in the following graphs, we indeed witness a speedup; adding a large number of helper threads only helps the sorting step of the algorithm, however, since the problem can only be subdivided into two concurrently executing sub-tasks. However, the speedup is substantial; recall we replaced the use of HeapSort by using QuickSort (this alone should introduce a speedup factor of about 2 given the performance benefit of QuickSort over HeapSort); but this alone is not enough to account for the truly noticeable speedup shown in **Table 2**. Clearly the use of multiple threads for this algorithm has a noticeable and great speedup.

n	Speedup of one-helper thread to no thread
65,536	6.41
131,072	8.66
262,144	9.93
524,288	11.07
1,048,576	12.35

Table 2: Speedup for Multi-thread ConvexHull with QuickSort over implementation of convex hull in the ADK.

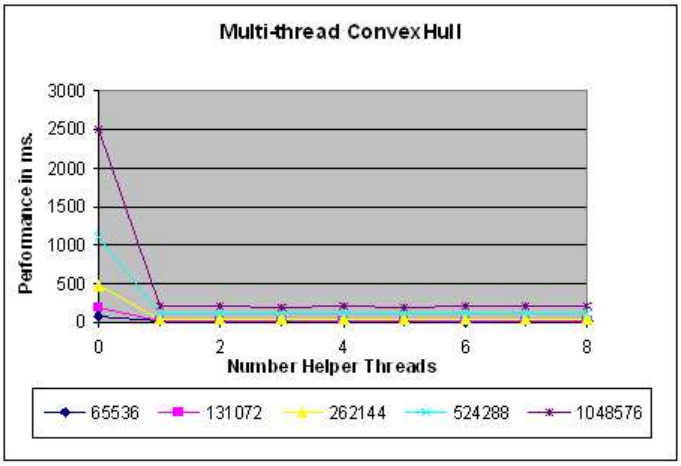

Fig. 14: Multi-threaded performance of Akl-Toussaint heuristic.

We also include a multi-threaded implementation of the Akl-Toussaint heuristic, which pre-processes the initial set of points to eliminate from contention those points which are contained within the extreme quadrilateral formed by the extreme left, right, top and bottom points of this set. In our multi-threaded implementation, we use just two separate threads, each processing half of the points; during testing, we found that the increased communication and programming burden of trying to use more than two threads is just not worthwhile. Naturally, then, the resulting speedup of about 2.58 is the best we were able to achieve.

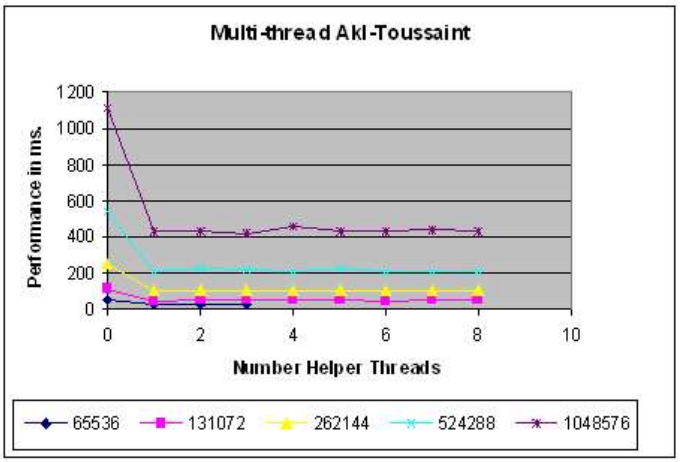

Fig. 15: Multi-threaded performance of Akl-Toussaint heuristic.

We should be excited that we were able to rather easily double the performance of two sample algorithms. Because threading appears to be so powerful, it is important to identify situations where the use of multiple threads is exactly the wrong approach to take.

An example of when threading does not help: Multi-threaded Nearest-Neighbor

In the nearest neighbor problem in which you are given a set of points P in a d-dimensional plane and you must answer queries of the form "what point in P is closest to point **x** using Euclidean distance?" A brute force algorithm simply scans all *n* points in P and finds the closest one to **x**. Perhaps we can achieve arbitrary improvements in speedup simply by throwing more threads at the brute force solution. Unfortunately, the performance experiment shown in **Fig. 16** tells another tale. For small values of N (such as 512) with up to eight helper threads, the multi-threaded version is about 45 times **slower** than the single-threaded version. Only when there are 32,768 points does the multi-threaded implementation out-perform its single-threaded counterpart, but this benefit doesn't last long. As you can see, the benefit lasts only with a high number of dimensions, and even then the speedup plateaus at about 1.2 regardless of how many threads are used. Only with 262,144 points are we able to achieve a consistent speedup of between 1.3 and 1.9. Clearly, simply throwing eight threads at the problem will not make this problem eight times faster.

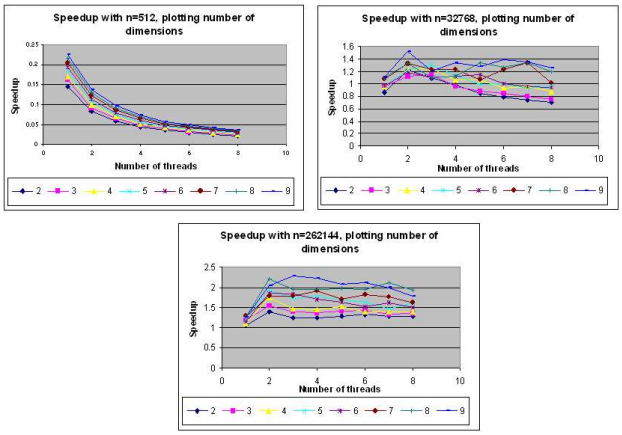

Fig. 16: Multi-threaded performance of Nearest Neighbor brute force algorithm.

Why does this problem not benefit from multiple threads? The essential difference is that the sub-problems computed by the multiple threads are not independent. Each of K threads is given an N/K subset of the initial N points and computes the nearest point from within that subset. Once this computation is complete, the smallest distance of these K computed points must then be computed. The details of the implementation are shown in **Fig. 17.**

```
public IMultiPoint nearest (IMultiPoint x) {
  final double[] xraw = x.raw();

  // start thread for each subset
  BruteForceThread[] threads=new BruteForceThread[numThreads];
  int size= points.length/numThreads;
  int offset = 0;

  for (int t = 0; t < threads.length - 1; t++) {
    threads[t] = new BruteForceThread(points, xraw, offset, size);
    threads[t].start();
    offset += size;
  }

  // remainder computed specially.
  threads [threads.length-1] =
    new BruteForceThread(points, xraw, offset, points.length-offset);
  threads [threads.length-1].start();

  // wait until all done, and compute min along the way
  double minValue = Double.MAX_VALUE;
  int bestIndex = -1;
  for (int t = 0; t < threads.length; t++) {
    try {
      threads [t].join();
    } catch (InterruptedException e) {
        System.err.println ("Multi Thread Brute Force interrupted.");
    }

    if (threads[t].best < minValue) {
      minValue = threads[t].best;
      bestIndex = threads[t].bestIndex;
    }
  }

  return results[bestIndex];
}
```

Fig. 17: Multi-threaded implementation of Nearest Neighbor brute force algorithm.

Starting the threads is not the problem; each `BruteForceThread` takes the information and processes different subset of points to locate the best point (i.e., the one closest to the target search point). The problem arises in processing the results of these threads, since they must all complete before we can compare the different results. The code fragment from **Fig. 17** chooses to wait for each thread, in order, until all threads are processed and the shortest result has been found. This situation creates a bottleneck which prevents the speedup from achieving its theoretic maximum.

As shown in Chapter 9, an alternative algorithm requires only O(log n) time if a *kd-tree* (see pp. 280-282 of our book) is constructed from the initial points. The pseudo-code for this algorithm is shown in **Fig. 18**.

```
nearest (T, x)
  n = find parent node where x would have been inserted
  min = distance from x to n.point
  better = nearest (T.root, min, x)
  if (better found) {  return better } else { return n.point }
end

nearest (node, min, x)
  d = distance from x to node.point
  if (d < min)  then
    min = d; result = node.point
  dp = perpendicular distance to node
  if (dp < min)  then
    // note two recursive invocations which could be parallelized
    pt = nearest (node.above, min, x)
    if (distance from pt to x < min)   then
      result = pt; min = distance from pt to x
    pt = nearest (node.below, min, x)
    if (distance from pt to x < min)   then
      result = pt; min = distance from pt to x
  else
    // note just a single recursive invocation here
    if (node is above x)  then
      pt = nearest (node.above, min, x)
    else
      pt = nearest (node.below, min, x)
```

```
    if (pt exists)  then return pt
  return result
end
```

Fig. 18: Pseudo-code for Nearest Neighbor queries

This recursive function, at times, invokes two recursive calls (when the computed perpendicular distance to the node is less than the current minimum value). Couldn't these two sub-task invocations be executed in parallel to improve performance? It turns out there are two reasons that prevent such a strategy from being helpful. First, these two tasks will compute two nearest neighbor results from which the actual nearest must be chosen; this bottleneck is the same as the one that appears in **Fig. 17**. Second, when do we use the helper threads? When the problem size is "small enough" as we did with QuickSort? Judging the size of a problem is difficult without adding extra state and computation to the problem. After extensive testing we found the only situation where there was an improved speedup was when a single helper thread is used only the very first time that the double recursive invocation occurs; when it occurs, thereafter only the single-thread solution is attempted. In the graphs below, we show the performance for N=4,096 and N=262,144 over a number of dimensions. In the graphs shown in **Fig. 19**, we compare the performance times of single-threaded KD-tree nearest neighbor vs. the one-helper thread KD-tree nearest neighbor. We also record the amount of time that we spend actually waiting for threads to complete execution. For data sets higher than 14 dimensions (N=4,096) or 13 dimensions (N=262,144) the one-helper threaded implementation finally outperforms its single-threaded counterpart for a speed up of about 1.35 (or 1.55 for N=262,144). It is interesting to note that the computed waiting time forms an increasingly large percentage of the total time as both the number of points and dimensions increases.

The code for the multi-threaded implementation is a bit too long to provide "as is" within this blog, so I would ask you to look at the code in both packages `algs.model.multithread.nearestNeighbor.onehelper` and `algs.model.multithread.nearestNeighbor.smallhelpers` .

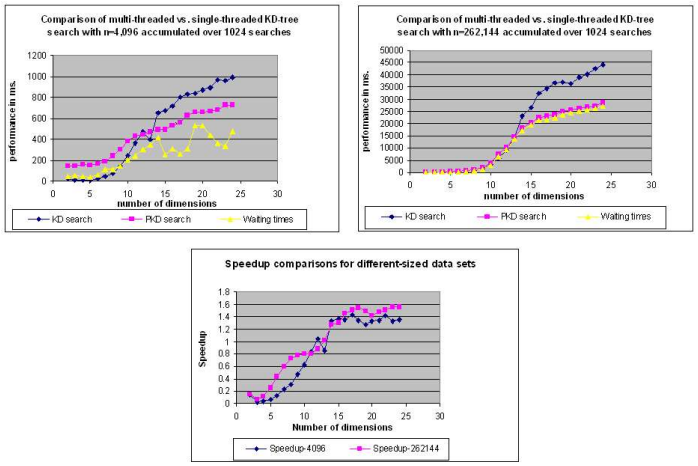

Fig. 19: Comparing multi-threaded Nearest Neighbor (PKD) with single-threaded (KD).

Lessons Learned

Using multiple threads to improve the performance of single-threaded algorithms is challenging. One must pay careful attention that the semantic meaning of the multiple-threaded implementation has not changed; at the same time, one must strive to avoid deadlock. In general, I would stress the following points:

- Only truly independent sub-problems should be solved by separate threads
- Most serial algorithms cannot achieve theoretic maximal speedup because only part of the algorithm can be parallelized among multiple threads
- Don't try to find solutions with as many threads as possible. In many cases, the best empirical results I had were with one helper thread, and this held true on laptops with dual-core chips and Linux boxes with multiple dual-core chips

Ok, so this blog contained a lot of material, which just goes to show how interesting these algorithms are! Full details of all Java classes and C functions are found in the code.zip repository associated with this Blog.

Next Column

In next Month's June column, we will investigate the QuickHull sorting algorithm which did not make it into the actual book itself. Until next time, we hope you take the opportunity to investigate the numerous algorithms in the Algorithms in a Nutshell book as well as to explore the examples provided in the ADK.

Algorithms in a Nutshell
George T. Heineman, Gary Pollice, Stanley Selkow

Tags: algorithms

You might also be interested in:

2 Comments

By Anonymous Coward on December 5, 2009 9:07 AM

In your Java quicksort I do really fail to see what would guarantee that swap made by spawned threads are visible to someone launching the quicksort.

For example, say at the beginning we have: 8,7,6,5,4,3,2,1

First thread swaps to 4,3,2,1,8,7,6,5 and spawns to threads. Those two threads shall correctly see, respectively, 4,3,2,1 and 8,7,6,5.

But there are zero guarantee that any change made by these two spawned threads shall be visible by an observer as far as I understand the Java memory model and your code.

The only way to make it work would be to use AtomicReferenceArray and not []. But the trials/examples I see are using [].

It may work on some VM but it's not guaranteed to work. I don't see/understand how the synchronization on the [] is done.

By big bad bombastic bob on June 4, 2010 1:36 AM

nice example. With multi-core machines you NEED threaded solutions to take advantage of the extra horsepower. I've recently done something similar (an 'N thread' solution) using "work units". Surfing for other multi-thread quicksort solutions led me here. Distributed calculations like 'dnetc' and 'seti@home' use work units also, and it seems to be an effective way of scheduling and NOT deadlocking. Waiting for a work unit you can always do another work unit (and in fact my algorithm does this). On a quad-core machine with 4 worker threads you would expect to see about 1/3 execution time over single-thread, primarily because of the single-threaded 'partition()' process. A DFT or FFT is also a good candidate for threading.