CMSC 433 Fall 2013
Michael Hicks
(Slides due to Rance Cleaveland)

UNIVERSITY OF MARYLAND

Lecture 13
Parallelizing Algorithms

Recall

- Concurrency: multiple flows of control
- Parallelism: simultaneous flows of control
- Reasons for concurrency
  - Improvements in throughput, responsiveness
  - Natural fit to application domain
- Reasons for parallelism
  - Performance improvements

Parallelizing Algorithms

- One important topic in parallelism: making existing sequential algorithms run in parallel
  - Existing algorithms often perform common, important tasks (e.g. sorting, searching, depth-first search)
  - Making them more efficient improves application / system performance
- *Tasks* offer a framework for studying parallelization
  - Idea: identify computations within algorithms that can be thought of as tasks (i.e. can be performed independently)
  - Execution these tasks concurrently, BUT …
  - … Tune concurrent execution to make best use of computational resources

Loop Parallelization

- Many sequential algorithms are *iterative* (i.e. use loops)
- Loop parallelization: perform (groups of) iterations in parallel
  - Sequential

```
for (Element e : collection)
  process(e);
```

  - Parallel

```
for (Element e : collection)
  exec.execute (new Runnable() {
    public void() run {
      process(e);
    }
  });
```

- When does this work?
  - Iterations must be independent (i.e. result of one iteration does not depend on the other)
  - Example: adding 1 to each element in an array
    - The result of processing each element is independent of the others
    - They can be made into tasks!

Loop Parallelization (2)

- Variation: grouping several iterations together into tasks
  - Consider summing elements in an array

```
sum = 0;
for (int i=0; i < a.length; i++)
  sum += a[i];
```

  - Iterations are not independent and cannot be made into tasks "as is"
  - However, several tasks can be created!

```
int sum[] = new int[NUMTASKS];  // Ensure initialization to 0
for (int i=0; i < NUMTASKS; i++) {
  exec.execute (new Runnable() {
    public void run () {
      for (j = i*NUMTASKS; j < (i+1)*NUMTASKS; j++)
        sum[i] += a[j];
    });
}
// After termination, sum up sum[i] values
```

- In this case, independent tasks created
  - However, final result depends on collecting results of tasks
  - This requires determining when tasks have terminated!
  - Several ways to do this: termination of executor, barriers, latches, etc.
- **You still have to worry about thread-safety, visibility!!**

Parallelizing Recursion

- Sometimes algorithms are recursive

  Example: depth-first search of a tree
  - Process node
  - Search each subtree
  - If there are no subtrees, return
- Similar ideas to loop parallelization can be used
  - Turn recursive calls into tasks!
  - Execute tasks concurrently
  - Considerations
    - Tasks should be independent
    - Works best if algorithms are *tail-recursive:* recursive calls issued at end

Example (JCIP pp. 182): Depth-First Search

- Tree: object in List<Node<T>>
  - List has only one node (the root)
  - Node methods
    - getChildren(): return list of subtrees (list of nodes)
    - compute(): perform computation on node
- Sequential tail-recursive version

```
public<T> void sequentialRecursive(List<Node<T>> nodes,
Collection<T> results) {
  for (Node<T> n : nodes) {
    results.add(n.compute());
    sequentialRecursive(n.getChildren(), results);
  }
}
```

  - Final operation of any call to `sequentialRecursive()` is the recursive call
  - This operation is therefore tail-recurisve

Parallelizing Depth-First Search

- Task launching

```
public <T> void parallelRecursive(final Executor exec,
                                  List<Node<T>> nodes,
                                  final Collection<T> results) {
  for (final Node<T> n : nodes) {
    exec.execute(new Runnable() {
      public void run() { results.add(n.compute()); }
    });
    parallelRecursive(exec, n.getChildren(), results);
  }
}
```

- Result collection

```
public <T> Collection<T> getParallelResults(List<Node<T>> nodes)
            throws InterruptedException {
  ExecutorService exec = Executors.newCachedThreadPool();
  Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
  parallelRecursive(exec, nodes, resultQueue);
  exec.shutdown();
  exec.awaitTermination();
  return resultQueue;
}
```

Performance Tuning

- The previous examples showed how task boundaries can be defined for parallelization
- However: should every task be run concurrently?
  - There is overhead in task launching
    - Insertion into work queue
    - Retrieval from work queue by worker thread
  - There is only run-time benefit if the final run-time decreases!
- We will study this issue in the context of *parallel sorting*

Recall Quicksort

- A fast sequential sorting algorithm invented by Tony Hoare (Turing Award winner) based on
  - Partitioning
  - Recursion
- quickSortSegment (elts, i, j) sorts elements in segment of array elts starting at i and extending j elements to the right
  - First, partition segment into two subsegments: those less than elts[i] and those greater than elts[i]
    - elt[i] is called the *pivot*
    - Partitioning involves scanning through segment and potentially swapping pivot with other elements
  - Then, recursively sort each of the subsegments

Sequential Quicksort code from IntArraySortUtils.java

```
public static void quickSortSegment (int[] elts, int first, int size) {
  if (size == 2) {
    if (elts[first] > elts[first+1])
      swap (elts, first, first+1);
  }
  else if (size > 2) {
    int pivotPosition = partitionSegment(elts, first, size);
    quickSortSegment (elts, first, pivotPosition-first);
    quickSortSegment (elts, pivotPosition+1, first+size-1-
pivotPosition);
  }
}
```

- (Almost) tail-recursive!
- Since recursive calls work on disjoint parts of the array, these can be made parallel
- To parallelize, can try turning each recursive call into a task

Parallelized Quicksort Code from ParallelQuickSort.java (1)

- Task definition

```
private class PQSTask implements Runnable {

  private final int[] elts;
  private final int first;
  private final int size;
  private final CountDownLatch numUnsorted;

  PQSTask (int[] elts, int first, int size, CountDownLatch numUnsorted) { … }
                  .
  public void run () {
    if (size == 1) numUnsorted.countDown();
    else if (size == 2) {
      if (elts[first] > elts[first+1])
        IntArraySortUtils.swap (elts, first, first+1);
      reduceUnsorted(size);  // Used for termination determination
    }
    else if (size > 2) {
      final int pivotPosition = IntArraySortUtils.partitionSegment(elts, first, size);
      numUnsorted.countDown(); // pivot in correct sorted position
      threadPool.execute(new PQSTask (elts, first, pivotPosition-first, numUnsorted));
      threadPool.execute(new PQSTask (elts, pivotPosition+1, first+size-1-pivotPosition,
numUnsorted));
    }
  }
}
```

Parallelized Quicksort Code from ParallelQuickSort.java (2)

- Sort driver

```
public void sort (int[] elts) {
  int length = elts.length;
  CountDownLatch numUnsorted = new CountDownLatch(length);
  threadPool.execute(new PQSTask (elts, 0, length, numUnsorted));
  try {
    numUnsorted.await();
  }
  catch (InterruptedException e) {}
}
```

Performance

- Parallelized Quicksort is slower (on my two-core machine) than sequential Quicksort!
  - When sorting k elements, on average k/2 tasks will be created
    - k = 10: 5 tasks
    - k = 1,000,000: 500,000 tasks!
  - The overhead of task management overwhelms the gains from parallelism
- Can solve this by coarsening task boundaries (fewer, bigger tasks)

Tuning Parallel Quicksort

- Several different ways to approach this
  - Key point: want to limit number of tasks based on number of CPUs
  - One idea:
    - Determine number of threads to be used
    - Determine size of sorting problem that should be handled sequentially, based on number of threads
    - Only create new tasks when the sorting problem they are handed is bigger than this limit
- How to determine number of threads?
  - Recall formula: $N_{threads} = N_{CPU} * U_{CPU} * (1 + W/C)$
  - For sorting, W/C is (very) low
  - In this case, $N_{threads} = N_{CPU} + 1$ (or 2) is a good idea
- How to determine sequential task limit?
  - If sorting k elements …
  - … set size limit to $k/ N_{threads}$
  - E.g.: if k is 1,000, $N_{threads} = 3$, then sequential task limit is 333
    - If sorting ≤ 333 elements, do so sequentially
    - Otherwise, do so in parallel

Tuned Parallelized Quicksort Code: ParallelQuickSortTunable.java

- Task is redefined:

```
public void run () {
  int sequentialize Target = elts.length / (numCPUs+2);
  if (size <= sequentializeTarget) {
    sequentialSort.sortSegment (elts, first, size);
    reduceUnsorted (size);
  }
  else {
    final int pivotPosition = IntArraySortUtils.partitionSegment(…);
    final int size1 = pivotPosition-first;
    final int size2 = first+size-1-pivotPosition;
    numUnsorted.countDown(); // pivot in correct sorted position
    threadPool.execute(new PQSTask (elts, first, size1, numUnsorted));
    threadPool.execute(new PQSTask (elts, pivotPosition+1, size2,
numUnsorted));
  }
}
```

- Result: parallelized implementation is faster than sequential, in general