Lecture 17
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)
  – Execute 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
    ```java
    int 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!
    ```java
    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!
  - **Common approach**
    - Create executor for each call to sum
    - Feed tasks to executor in course of computing sum
    - When no more tasks are needed, shut own executor and await its termination
- **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
  – Generate tasks from recursive calls!
  – Execute tasks concurrently
  – Considerations
    • Tasks should be independent
    • Works best if algorithms are \textit{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

```java
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-recursive
Parallelizing Depth-First Search

- Task launching
  ```java
  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
  ```java
  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
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
• How to parallelize?
  – Turn each “base case” computation into a task
  – Recursive calls still handled sequentially
Parallelized Quicksort Code from ParallelQuickSortNew.java (1)

- Task definition
  ```java
  private class PQSTask implements Runnable {
    private int elts[];
    private int i;
    private int j;

    public PQSTask (int elts[], int i, int j) {
      this.elts = elts;
      this.i = i;
      this.j = j;
    }

    public void run () {
      IntArraySortUtils.swap (elts, i, j);
    }
  }
  ```
Parallelized Quicksort Code from ParallelQuickSortNew.java (2)

- Segment-sorting routine

```java
public void parallelQuickSortSegment(int[] elts, int first, int size) {
    if (size == 2) {
        exec.execute(new PQSTask (elts, first, first+1));
    } else if (size > 2) {
        int pivotPosition =
        IntArraySortUtils.partitionSegment(elts, first, size);
        parallelQuickSortSegment (elts, first, pivotPosition-first);
        parallelQuickSortSegment (elts, pivotPosition+1, first+size-1-pivotPosition);
    }
}
```
Parallelized Quicksort Code from ParallelQuickSortNew.java (3)

- Sort driver

```java
public void sort (int[] elts) {
    int NUMTHREADS = Runtime.getRuntime().availableProcessors();
    exec = Executors.newFixedThreadPool(NUMTHREADS);
    parallelQuickSortSegment (elts, 0, elts.length);
    exec.shutdown();
    try {
        exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    } 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 is smaller than this limit

• How to determine number of threads?
  – Recall formula: \( N_{\text{threads}} = N_{\text{CPU}} \cdot U_{\text{CPU}} \cdot (1 + W/C) \)
  – For sorting, \( W/C \) is (very) low
  – In this case, \( N_{\text{threads}} = N_{\text{CPU}} + 1 \) (or 2) is a good idea
  – To compute \( N_{\text{CPU}} \) in Java, use `Runtime.getRuntime().availableProcessors()`

• How to determine sequential task limit?
  – If sorting \( k \) elements, set size limit to \( k / N_{\text{threads}} \)
  – E.g.: if \( k \) is 1,000, \( N_{\text{threads}} = 3 \), then sequential task limit is 333
    • If sorting \( \leq 333 \) elements, do so sequentially
    • Otherwise, do so in parallel
Tuned Parallelized Quicksort Code: ParallelQuickSortNewTunable.java

- Task is redefined:
  ```java
  private class PQSTask implements Runnable {
      private int elts[];
      private int first;
      private int size;
      ...
      public void run () {
          IntArraySortUtils.quickSortSegment (elts, first, size);
      }
  }
  ```

- So is segment sorting routine
  ```java
  public void parallelQuickSortSegment (int[] elts, int first, int size) {
      if (size <= THRESHOLD) {
          exec.execute(new PQSTask (elts, first, size));
      } else {
          ...
      }
  }
  ```

- Result: much better performance!