Parallel Programming Hands-On

Nathan Bronson
nbronson@cs.stanford.edu

Goals

- Parallelize a Java application
  - Work as a team
  - Tackle one phase, or several
- Perform experiments
  - Small and large data sets
  - Dual-core Intel workstations – 1 per person
  - Eight-core (64 HW thread) Sun blades – 1 per team
- Present your results (tomorrow)
  - 10 minutes + 5 minutes of Q&A
  - Explanation is more important than speedup

But first … some Java details

- How does one run code on a separate thread?
  - Manually spawn a new thread (`java.lang`, since 1.0)
    - Extend `Thread`, overriding `run()`
    - Implement `Runnable`, pass it to `Thread` constructor
  - Use a thread pool (`java.util.concurrent`, since 1.5)
    - Implement `Runnable`, pass to `Executor.execute()`
    - *Lower overhead, less work for the operating system*
  - Use a thread pool to create a `Future`
    - Implement `Runnable` or `Callable`, pass to `ExecutorService.submit()`
    - *Task can return a value to the caller, caller can wait for task*

An example future

```
final int x = 10;
int y;
Callable<Integer> func = new Callable<Integer>() {
   public Integer call() {
      return x + 1;
   }
};
Future<Integer> fut = ThreadPool.instance.submit(func);

// wait for func to be run, and grab the result
int result = fut.get();

// note that only final vars are accessible inside the
// anonymous Callable instance, so call() can not
// access y
```

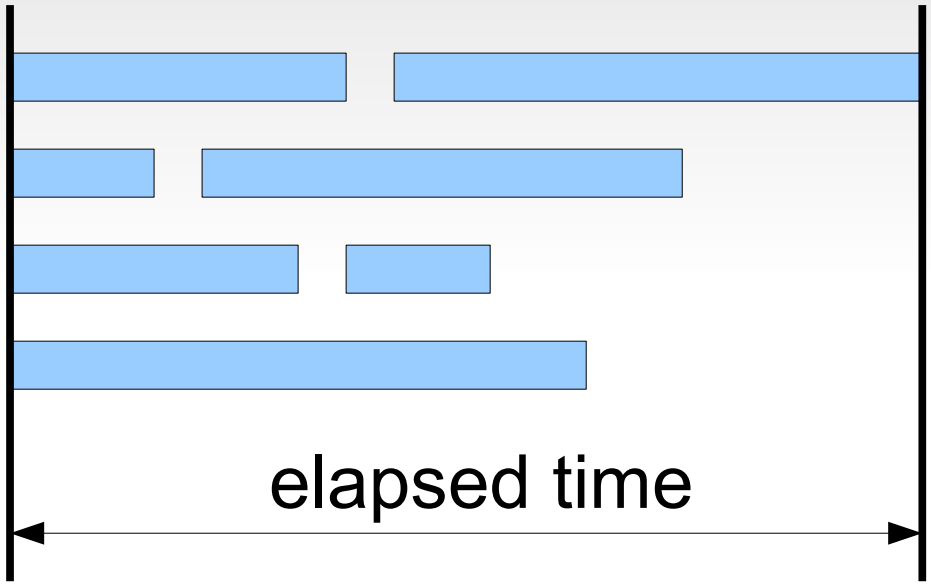
An example future (with exceptions)

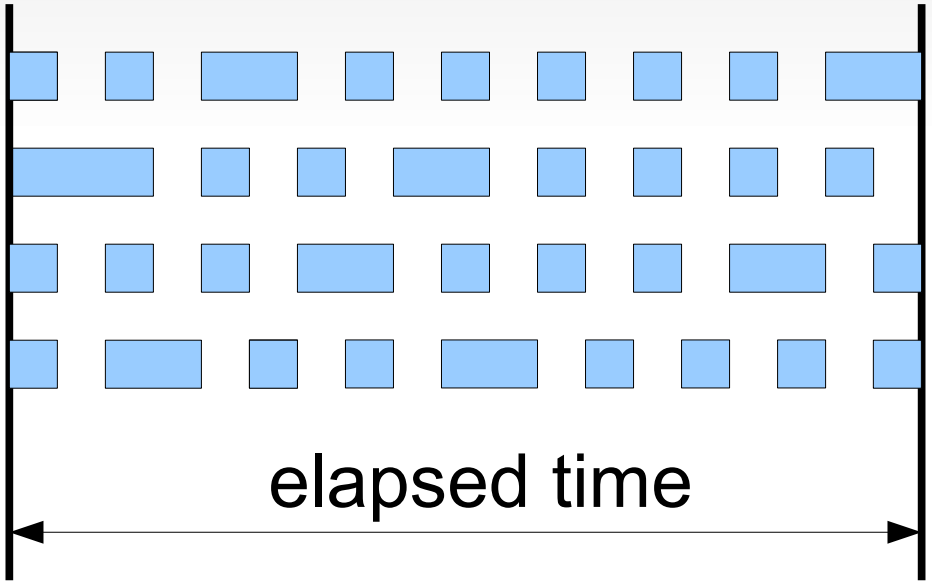
```
final int x = 10;
int y;
Callable<Integer> func = new Callable<Integer>() {
   public Integer call() {
      return x + 1;
   }
};
Future<Integer> fut = ThreadPool.instance.submit(func);

// wait for func to be run, and grab the result
int result;
try {
   result = fut.get();
} catch (Exception exc) {
   throw new RuntimeException(”unexpected”, exc);
}
```

Overheads vs. work imbalance

- Too few subtasks, slowest thread finishes late





- Too few subtasks, too much bookkeeping overhead

A complete example

```
public static int findCount(
        final String str, final Pattern pat) {
    final Matcher matcher = pat.matcher(str);
    int result = 0;
    while (matcher.find()) {
        ++result;
    }
    return result;
}

public static int findCount(
        final List<String> strs, final Pattern pat) {
    int result = 0;
    for (final String s : strs) {
        result += findCount(s, pat);
    }
    return result;
}
```

findCount(): parallel version

```
public static int findCountP(
        final List<String> strs, final Pattern pat) {
    List<Future<Integer>> futs = ...

    while (..not done..) {
        List<String> chunk = ...
        ...populate chunk...

        futs.add(ThreadPool.instance.submit(
            ...sequential findCount(chunk, pat)
        )
    }

    int result = 0;
    for (Future<Integer> f : futs) {
        result += f.get();
    }
    return result;
}
```

findCount(): details

```
/** A parallel implementation of {@link #findCount(List,Pattern)}. */
public static int findCountP(final List<String> strs, final Pattern pat) {
    // each subtask is a sequential findCount, returning an int
    final List<Future<Integer>> futs = new ArrayList<Future<Integer>>();

    // Our goal is to have about R times as many tasks as there are
    // available threads, to balance the division with work imbalance.  A
    // few tests show that R=4 is a reasonable value for this data set.
    final int procs = Runtime.getRuntime().availableProcessors();
    final int chunkSize = Math.max(1, strs.size() / (4 * procs));

    final Iterator<String> iter = strs.iterator();
    while (iter.hasNext()) {
        // Chunk based on some proxy for the amount of work.  This might be
        // the number of lines, character counts, or some mixture (we use
        // line count).
        final List<String> chunk = new ArrayList<String>(chunkSize);
        while (chunk.size() < chunkSize && iter.hasNext()) {
            chunk.add(iter.next());
        }

        // enqueue one task to the thread pool, storing the future for later
        futs.add(ThreadPool.instance.submit(new Callable<Integer>() {
            public Integer call() throws Exception {
                return findCount(chunk, pat);
            }
        }));
    }

    // build up the total result by adding the subresults
    int result = 0;
    for (final Future<Integer> f : futs) {
        final int taskResult;
        try {
            taskResult = f.get();
        }
        catch (final Exception xx) {
            throw new RuntimeException("unexpected", xx);
        }
        result += taskResult;
    }

    return result;
}
```

Running ParListExample

Launch Applications::Accessories::Text Editor

Open `busyrank/src/archss/example/ParListExample.java`

Launch Applications::Accessories::Terminal

```
> cd ~/busyrank
> ./build
> java -server -cp classes archss.example.ParListExample
```

The app: BusyRank

- A simple document search engine
  - Single-threaded Java
  - ~600 lines
  - Only handles data sets that fit in memory
- Four phases
  - Read files from disk
  - Parse Unicode and split into words
  - Construct an index
  - Answer queries

Project directory structure

- `busyrank/` – Top level
  - `small/` – (Symlink to) small data set + test queries
  - `large/` – (Symlink to) large data set + test queries
  - `seqsrc/` – Source code for BusyRank
    - Code is in the `archss` package, `seqsrc/archss` dir
  - `src/` – Initially this is the same as `seqsrc`
    - Suggestion: modify the code here (or create more source dirs)

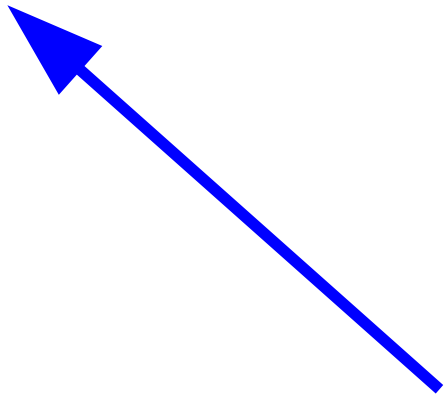
Project helper scripts

- `run` *`DATADIR [SRCDIR]`*
  - Builds the program (default is from `src`, but can use any dir)
  - Runs it, with query results going to a directory in `/tmp`
  - Verifies that the search results are correct
- `build` *`[SRCDIR [CLASSESDIR]]`*
  - Builds the program (default is from `src`, but can use any dir)
  - Writes class files to the `classes` by default
- `sunssh`
  - Works like `ssh`, connects to a Niagara T2 hosted by Sun
- `sync_to_sun`
  - Copies the entire `busyrank` directory to the T2 (use with care!)

BusyRank on my single-core laptop

```
> cd ~/busyrank
> ./run small
Compiling 10 source files from /home/nbronson/busyrank/src using /usr/bin/javac
Preheating the buffer cache
Loading all files under small/data
   664648    640000  file load:         0.66 sec (  0.64 CPU, 1.0 utilization)
  5215634   5160000  file parse:        5.22 sec (  5.16 CPU, 1.0 utilization)
  3721939   3690000  create index:      3.72 sec (  3.69 CPU, 1.0 utilization)
  1954007   1920000  search:            1.95 sec (  1.92 CPU, 1.0 utilization)
 11574073  11420000  total:            11.57 sec ( 11.42 CPU, 1.0 utilization)

real    0m11.675s
user    0m11.251s
sys     0m0.257s
Query results match expected
>
```



Speedup is an improvement in the the **total elapsed time**, here 11.57 seconds (11,574,073 microseconds)