Solution to Worksheet #16:
Critical Path Length for Computation with Signal Statement

Compute the WORK and CPL values for the program shown below. (WORK = 204, CPL = 102). How would they be different if the signal() statement was removed? (CPL would increase to 202.)

```java
1. finish(() -> {
2.   final HjPhaser ph = newPhaser(SIG_WAIT);
3.   asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
4.     A(0); doWork(1); // Shared work in phase 0
5.     signal();
6.     B(0); doWork(100); // Local work in phase 0
7.     next(); // Wait for T2 to complete shared work in phase 0
8.     C(0); doWork(1);
9.   });
10.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
11.    A(1); doWork(1); // Shared work in phase 0
12.    next(); // Wait for T1 to complete shared work in phase 0
13.    C(1); doWork(1);
14.    D(1); doWork(100); // Local work in phase 0
15.   });
16. }); // finish
```
Data-Driven Futures - Common Pitfall

void foo(Map<String, DDF> store) {
    finish {
        DDF fooDdf = new DDF()
        async {
            bar(store)
            fooDdf.put(1)
        }
        println("Spawned async");
        store.put("foo", fooDdf)
    }
}

void bar(Map<String, DDF> store) {
    DDF barDdf = new DDF()
    DDF fooDdf = store.get("foo")
    async await (foo) {
        barDdf.put(1 + fooDdf.get())
    }
    store.put("bar", barDdf)
}

Lab 6 Solution - Cholesky with DDFs

void foo(Map<String, DDF> store) {
    finish {
        DDF fooDdf = new DDF()
        store.put("foo", fooDdf)
        async {
            bar(store)
            fooDdf.put(1)
        }
        println("Spawned async");
    }
}

void bar(Map<String, DDF> store) {
    DDF barDdf = new DDF()
    store.put("bar", barDdf)
    DDF fooDdf = store.get("foo")
    async await (foo) {
        barDdf.put(1 + fooDdf.get())
    }
}
Iterative Averaging with Chunking

Num tasks=5
Array size=27

<table>
<thead>
<tr>
<th>Iter i</th>
<th>Task-0</th>
<th>Task-1</th>
<th>Task-2</th>
<th>Task-3</th>
<th>Task-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter i+1</th>
<th>Task-0</th>
<th>Task-1</th>
<th>Task-2</th>
<th>Task-3</th>
<th>Task-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>

Chunked Iterative Averaging with Barriers

Num tasks=5
Array size=27

<table>
<thead>
<tr>
<th>Iter i</th>
<th>Task-0</th>
<th>Task-1</th>
<th>Task-2</th>
<th>Task-3</th>
<th>Task-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iter i+1</th>
<th>Task-0</th>
<th>Task-1</th>
<th>Task-2</th>
<th>Task-3</th>
<th>Task-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
</tbody>
</table>
Lab 6 Iterative Averaging - Missing Speedup?

- Many of you got the correct code but were missing speedup
- The reason is that the computation was *memory bound*
  - Memory access was dominating computation time, did not get benefits from parallelism
- The following change helps observe the (near perfect) speedup
  ```java
  for (int j = startIncJ; j <= endIncJ; j++) {
      myNew[j] = (myVal[j - 1] + myVal[j + 1]) / 2.0;
  }
  ```
  to
  ```java
  for (int j = startIncJ; j <= endIncJ; j++) {
      myNew[j] = Math.log(Math.exp((myVal[j - 1] + myVal[j + 1]) / 2.0));
  }
  ```
HJ-lib Compilation and Execution Environment

Java 8 IDE

```
Foo.java
```

```
javac Foo.java
```

```
Java compiler
```

```
Foo.class
```

```
java Foo
```

```
HJ-lib Runtime Environment = Java Runtime Environment + HJ-lib libraries
```

```
HJ-lib Program Output
```

```
All the "magic" happens here!
```

```
HJ runtime initializes m worker threads (value of m depends on options or default value)
```

```
HJ Abstract Performance Metrics, HJ-Viz output (all enabled by appropriate options)
```

Looking under the hood — let’s start with the hardware

An example compute node with two quad-core Intel Xeon (CPUs, for a total of 8 cores/node (NOTS has 16 cores/node)
Next, how does a process run on a single core?

Processes are managed by OS kernel
- Important: the kernel is not a separate process, but rather runs as part of some user process

Control flow passes from one process to another via a context switch

Context switches between two processes can be very expensive!

What happens when executing a Java program?

- A Java program executes in a single Java Virtual Machine (JVM) process with multiple threads
- Threads associated with a single process can share the same data
- Java main program starts with a single thread (T1), but can create additional threads (T2, T3, T4, T5) via library calls
- Java threads may execute concurrently on different cores, or may be context-switched on the same core
Thread-level Context Switching on the same processor core

• Thread context switch is cheaper than a process context switch, but is still expensive (just not “very” expensive!)

• It would be ideal to just execute one thread per core (or hardware thread context) to avoid context switches

Figure source: COMP 321 lecture on Concurrency (Alan Cox, Scott Rixner)

Now, what happens in a task-parallel Java program (e.g., HJ-lib, Java ForkJoin, etc)

• Task-parallel runtime creates a small number of worker threads, typically one per core

• Workers push new tasks and “continuations” into a logical work queue

• Workers pull task/continuation work items from logical work queue when they are idle (remember greedy scheduling?)
Task-Parallel Model: Checkout Counter Analogy

- Think of each checkout counter as a processor core


Task-Parallel Model: Checkout Counter Analogy

- Think of each checkout counter as a processor core
- And of customers as tasks

source: http://www.deviantart.com/art/Randomness-20-178737664
All is well until a task blocks …

- A blocked task/customer can hold up the entire line
- What happens if each checkout counter has a blocked customer?

source: http://viper-x27.deviantart.com/art/Checkout-Lane-Guest-Comic-161795346

Approach 1: Create more worker threads (as in HJ-Lib’s Blocking Runtime)

- Creating too many worker threads can exhaust system resources (OutOfMemoryError), and also leads to context-switch overheads when blocked worker threads get unblocked
  - Context-switching in checkout counters stretches the analogy — maybe assume that there are 8 keys to be shared by all active checkout counters?

source: http://www.deviantart.com/art/Randomness-5-90424754
Blocking Runtime (contd)

- Examples of blocking operations
  - End of finish
  - Future get
  - Barrier next

- Blocks underlying worker thread, and launches an additional worker thread

- Too many blocking constructs can result in lack of performance and exceptions
  - java.lang.IllegalStateException: Error in executing blocked code! [89 blocked threads]
  - Maximum number of worker threads can be configured if needed
  - HjSystemProperty.maxThreads.set(100);

Approach 2: Suspend task continuations at blocking points (as in HJ-Lib’s Cooperative Runtime)

- Task actively suspends itself and yields control back to the worker
- Task’s continuation is stored in the suspended queue and added back into the ready queue when it is unblocked
- Pro: No overhead of creating additional worker threads
- Con: Complexity and overhead of creating continuations

Cooperative Scheduling: http://en.wikipedia.org/wiki/Computer_multitasking#Cooperative_multitasking
Continuations

- A continuation is one of two kinds of program points
  - The point in the parent task immediately following an `async`
  - The point immediately following a `blocking` operation, such as an `end-finish`, `future get()`, or barrier

- Continuations are also referred to as task-switching points
  - Program points at which a worker may switch execution between different tasks (depends on scheduling policy)

```java
1. finish { // F1
2. async A1;
3. finish { // F2
4. async A3;
5. async A4;
6. }
7. S5;
8.}
```

NOTE: these are “one-shot” continuations, unlike continuations in functional programs that can be called multiple times

Cooperative Scheduling
(view from a single worker)

Useful work for some other task on same worker thread
HJ-lib’s Cooperative Runtime

Any operation that contributes to unblocking a task can be viewed as an event e.g., task termination in finish, return from a future, signal on barrier, put on a data-driven-future, ...

Why are Data-Driven Tasks (DDTs) more efficient than Futures?

- Consumer task blocks on get() for each future that it reads, whereas async-await does not start execution till all Data-Driven Futures (DDFs) are available
  — An “asyncAwait” statement does not block the worker, unlike a future.get()
  — No need to create a continuation for asyncAwait; a data-driven task is directly placed on the Suspended queue by default

- Therefore, DDTs can be executed on a Blocking Runtime without the need to create additional worker threads, or on a Cooperative Runtime without the need to create continuations