COMP 322: Fundamentals of Parallel Programming

Lecture 17: Abstract vs. Real Performance — an "under the hood" look at HJlib

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/

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.)

```
finish(() -> {
  final HjPhaser ph = newPhaser(SIG_WAIT);
  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
    A(0); doWork(1);    // Shared work in phase 0
    signal();
    B(0); doWork(100); // Local work in phase 0
    next(); // Wait for T2 to complete shared work in phase 0
    C(0); doWork(1);
  });
   asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
     A(1); doWork(1);    // Shared work in phase 0
     next(); // Wait for T1 to complete shared work in phase 0
     C(1); doWork(1);
     D(1); doWork(100); // Local work in phase 0
   });
}); // 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)
}
```

```
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())
  }
}
```

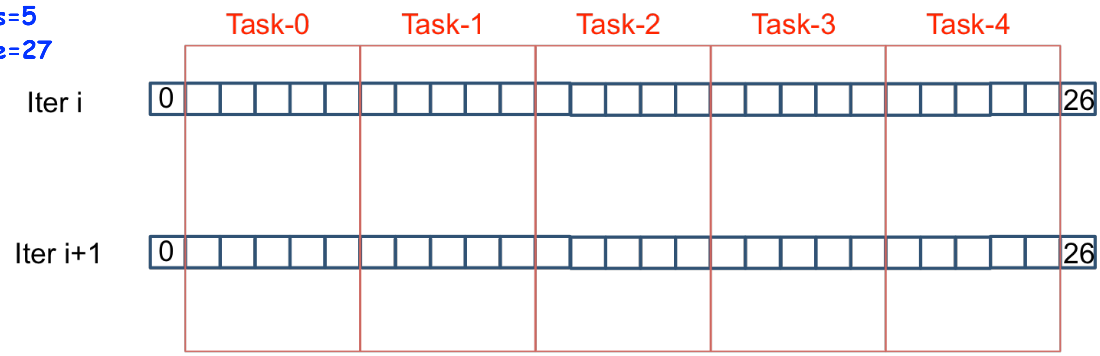

Lab 6 Solution - Cholesky with DDFs

```
/**
 * Triggers execution of s3ComputeStepBody (possibly asynchronously).
 */
public void s3ComputeStep(
    final int tileSize, final int numTiles,
    final Point pointTag, final Map<Point, Object> dataStore) {

    final int k = pointTag.read(0);
    final int j = pointTag.read(1);
    final int i = pointTag.read(2);

    double[][] l1Block = null;
    final double[][] l2Block;
    final double[][] aBlock = readDataItem(dataStore, newPoint(j, i, k));

    if (i == j) {   // Diagonal tile.
        l2Block = readDataItem(dataStore, newPoint(i, k, k + 1));
        l1Block = l2Block;
    } else {   // Non-diagonal tile.
        l2Block = readDataItem(dataStore, newPoint(i, k, k + 1));
        l1Block = readDataItem(dataStore, newPoint(j, k, k + 1));
    }

    final double[][] s3Result = s3ComputeStepBody2(
        tileSize, numTiles, pointTag, dataStore,
        aBlock, l1Block, l2Block);

    storeDataItem(dataStore, newPoint(j, i, k + 1), s3Result);
}
```

```
/**
 * Triggers execution of s3ComputeStepBody (possibly asynchronously).
 */
public void s3ComputeStep(
    final int tileSize, final int numTiles,
    final Point pointTag, final Map<Point, Object> dataStore) {

    final int k = pointTag.read(0);
    final int j = pointTag.read(1);
    final int i = pointTag.read(2);

    HjFuture<double[][]> aBlockF = readDataItem(dataStore, newPoint(j, i, k));
    HjFuture<double[][]> l2BlockF = readDataItem(dataStore, newPoint(i, k, k + 1));
    HjFuture<double[][]> l1BlockF = readDataItem(dataStore, newPoint(j, k, k + 1));

    final HjDataDrivenFuture<double[][]> s3Future = newDataDrivenFuture();
    storeDataItem(dataStore, newPoint(j, i, k + 1), s3Future);

    asyncAwait(aBlockF, l1BlockF, l2BlockF, () -> {
        double[][] l1Block = null;
        final double[][] l2Block;
        final double[][] aBlock = aBlockF.get();

        if (i == j) {   // Diagonal tile.
            l2Block = l2BlockF.get();
            l1Block = l2Block;
        } else {   // Non-diagonal tile.
            l2Block = l2BlockF.get();
            l1Block = l1BlockF.get();
        }

        final double[][] s3Result = s3ComputeStepBody2(
            tileSize, numTiles, pointTag, dataStore,
            aBlock, l1Block, l2Block);

        s3Future.put(s3Result);
    });
}
```

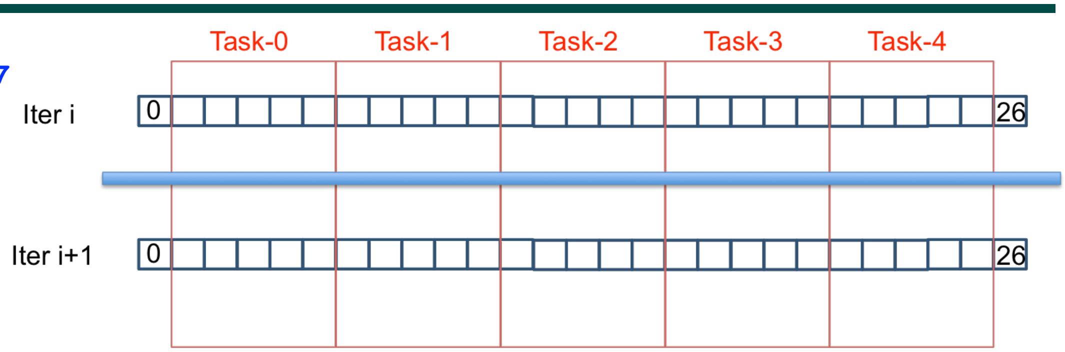
Iterative Averaging with Chunking

Num tasks=5
Array size=27



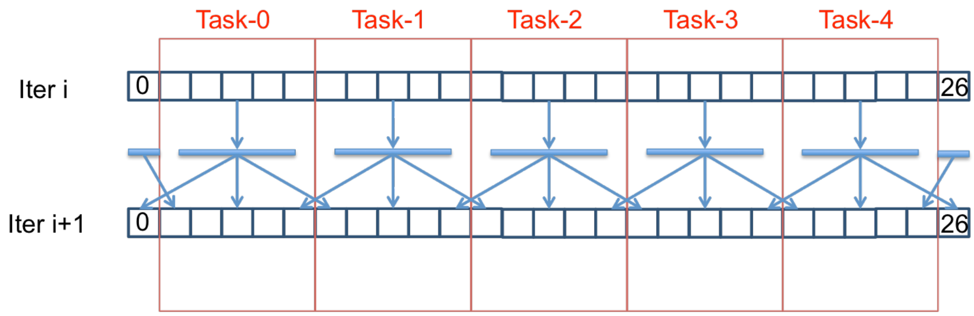
Chunked Iterative Averaging with Barriers

Num tasks=5
Array size=27



Chunked Iterative Averaging with Phasers

Num tasks=5
Array size=27



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

```
for (int j = startIncJ; j <= endIncJ; j++) {
    myNew[j] =(myVal[j - 1] + myVal[j + 1]) / 2.0;
}
```

to

```
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

HJ-lib source program is a standard Java 8 program

javac Foo.java

Java compiler

Java compiler translates Foo.hj to Foo.class, along with calls to HJ-lib with lambda parameters (async, finish, future, etc)

Foo.class

All the "magic" happens here!

java Foo

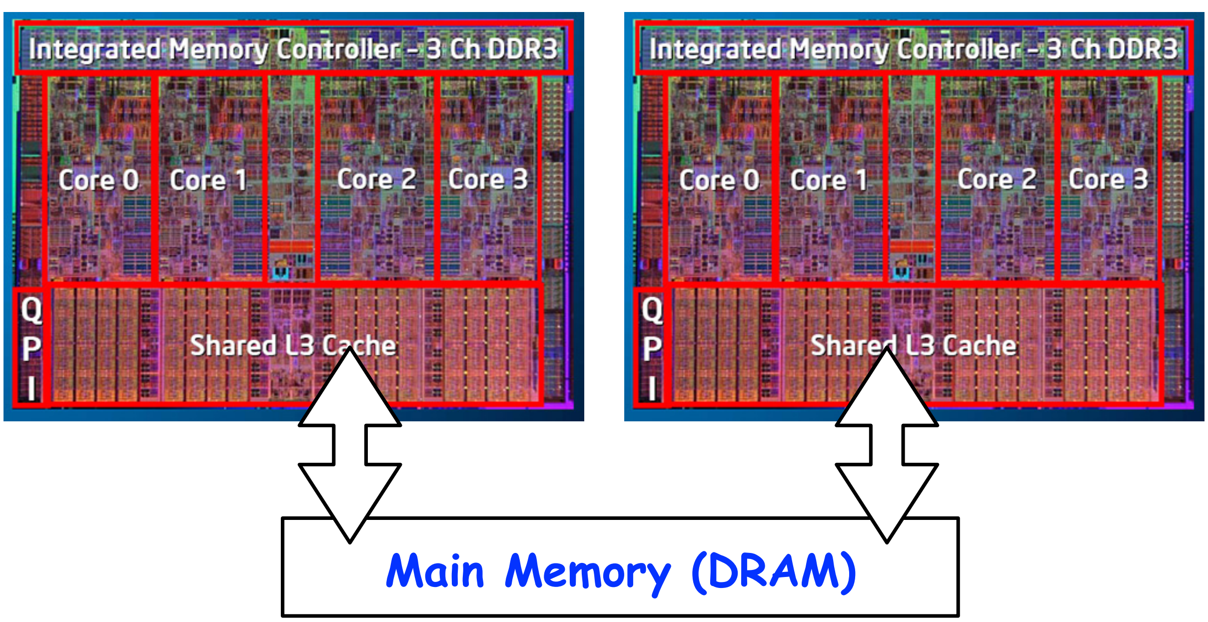
HJ-lib Runtime Environment = Java Runtime Environment + ***HJ-lib libraries***

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

HJ-lib Program Output

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

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

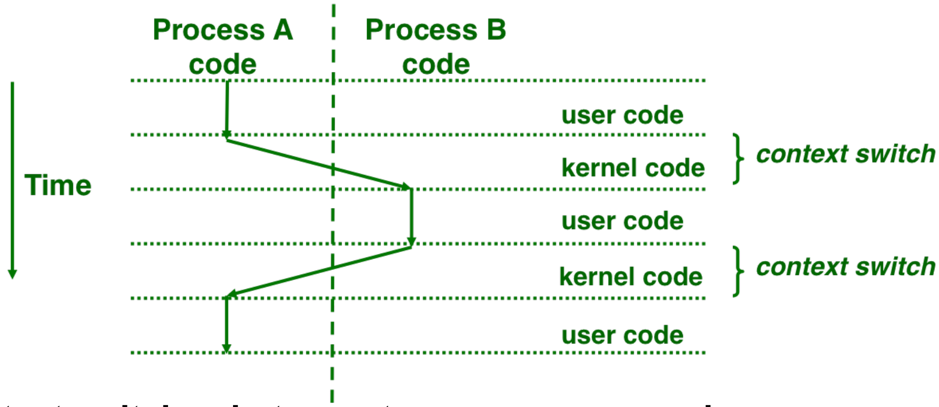


An example compute node with two quad-core Intel Xeon (CPUs, for a total of 8 cores/node (NOTS has 16 cores/node)

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!

Source: COMP 321 lecture on Exceptional Control Flow (Alan Cox, Scott Rixner)

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

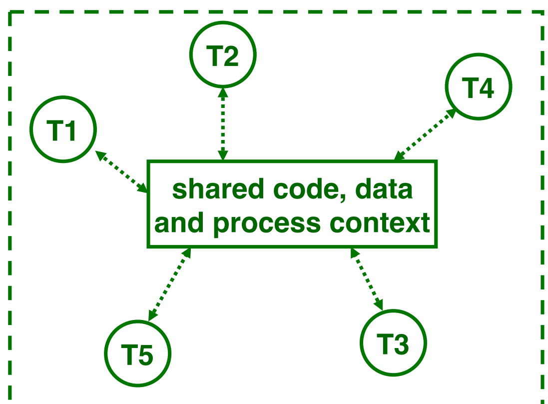
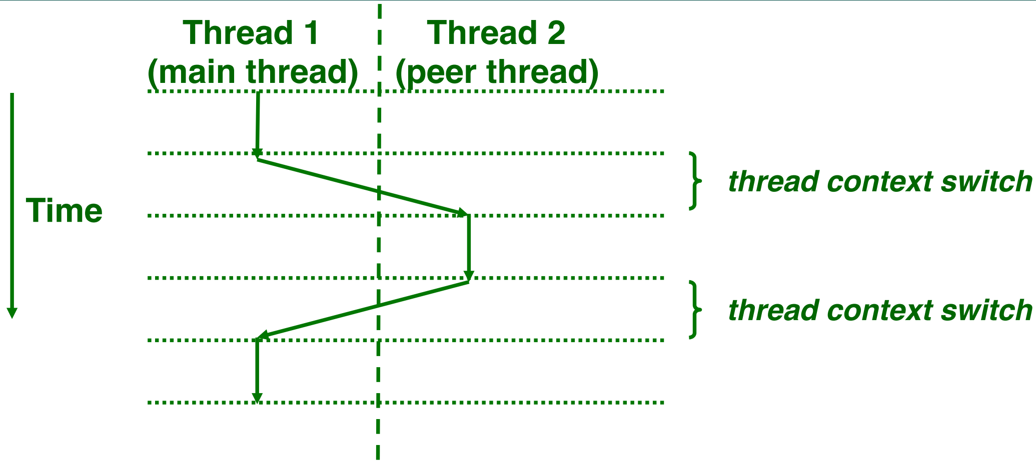


Figure source: COMP 321 lecture on Concurrency (Alan Cox, Scott Rixner)

Thread-level Context Switching on the same processor core

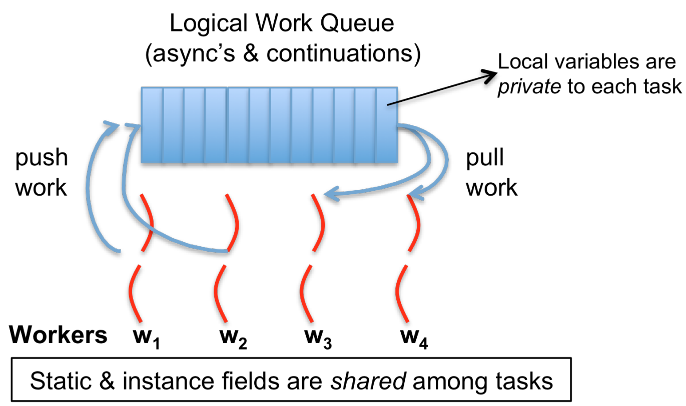


- Thread context switch is cheaper than a process context switch, but is still expensive (just not "very" expensive!)
- It would be ideal to just execute one thread per core (or hardware thread context) to avoid context switches

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)

HJ-Lib Tasks & Continuations
Worker threads
Operating System
Hardware cores



- 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

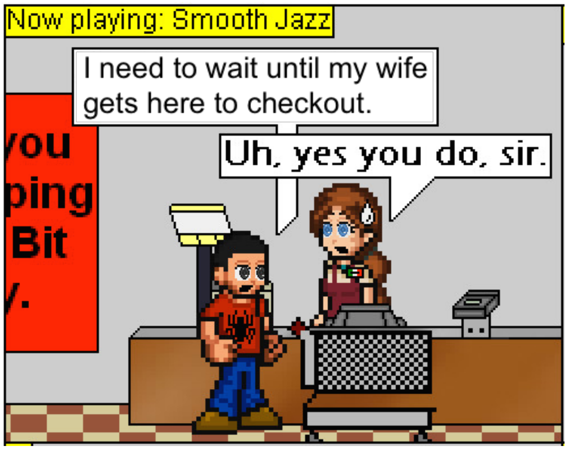
Image sources: http://www.deviantart.com/art/Randomness-20-178737664,
http://www.wholefoodsmarket.com/blog/whole-story/new-haight-ashbury-store

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 …

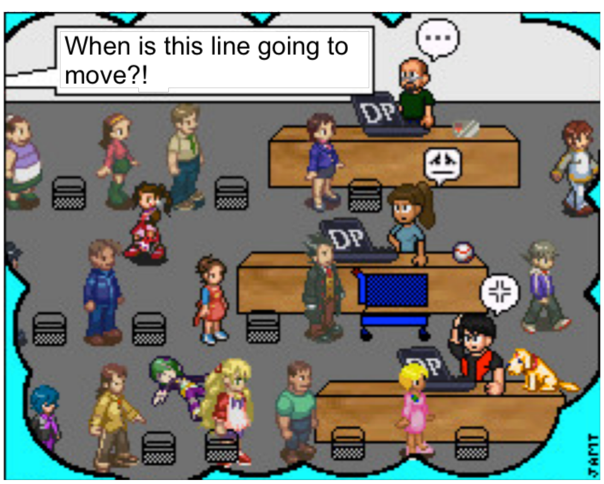


. . .

- 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)



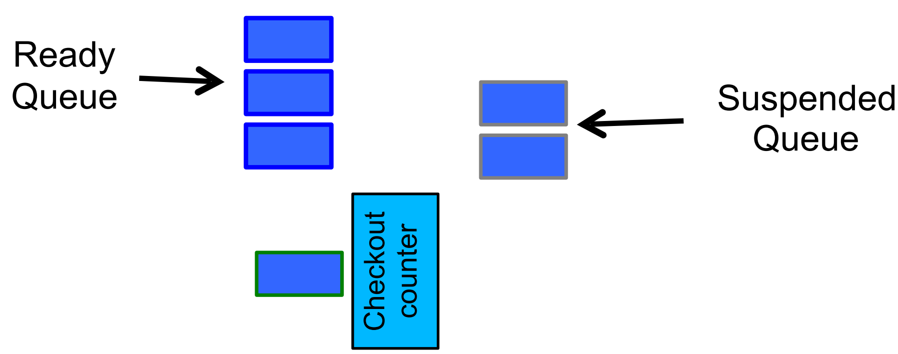
- Creating too many worker threads can exhaust system resources (OutOfMemoryError), and also leads to context-switch overheads when blocked worker threads get unblocked
  - Context-switching in checkout counters stretches the analogy — maybe assume that there are 8 keys to be shared by all active checkout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

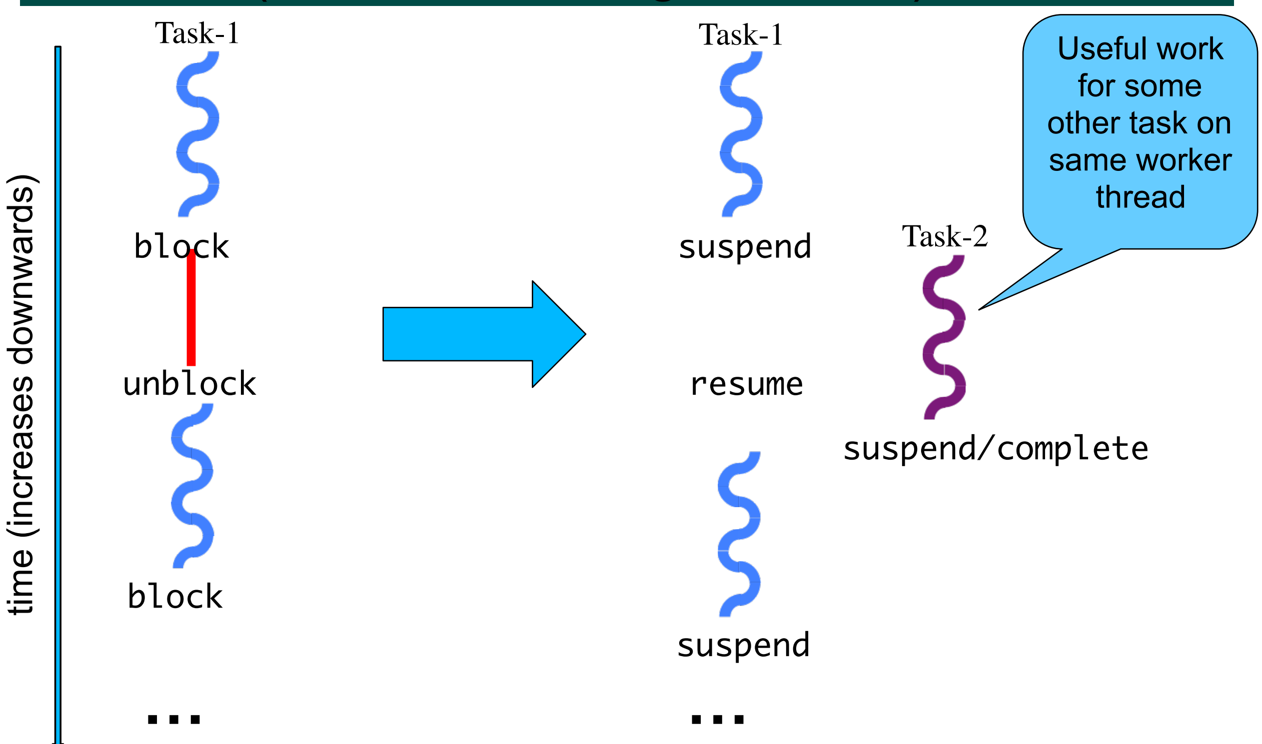
- A continuation is one of two kinds of program points
  - — The point in the parent task immediately following an **`async`**
  - — The point immediately following a *blocking* operation, such as an end-**`finish`**, future **`get`**(), or barrier
- Continuations are also referred to as task-switching points
  - — Program points at which a worker may switch execution between different tasks (depends on scheduling policy)

```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```

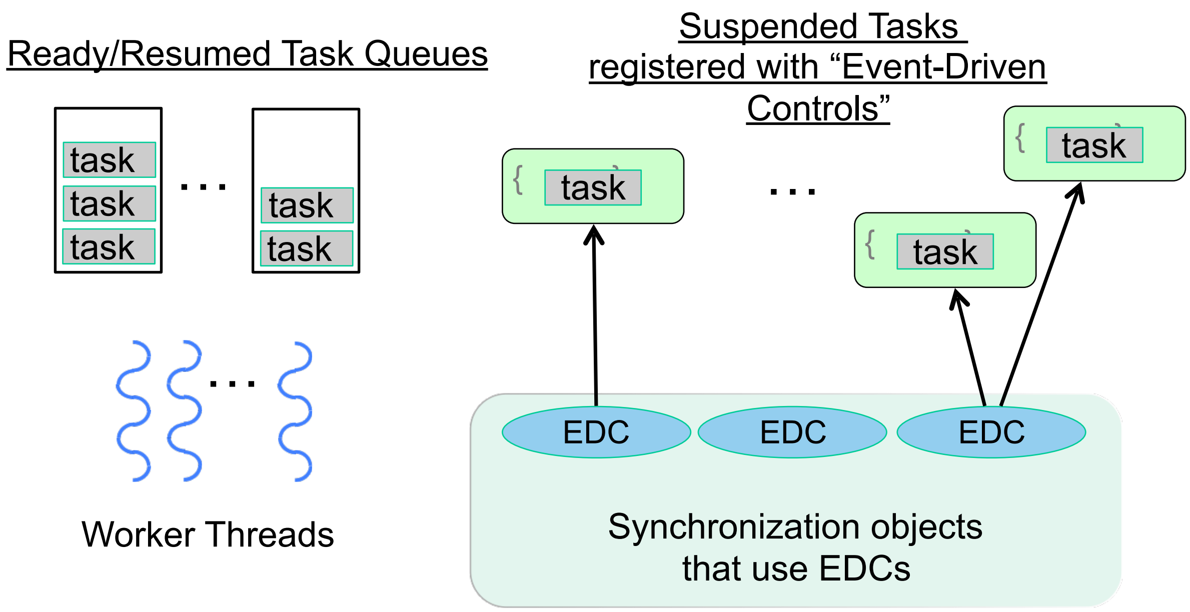
Continuations

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

Cooperative Scheduling (view from a single worker)



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