Multicore Programming

Executors

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Bureau 414C

Master 1 computer science – Semester 8

Motivation

- Express multitask operations without writing the thread logic.
- Rely on *thread pools* to limit thread management overhead (since Java 5).

Outline

Generality

Thread Pool

Summary and References

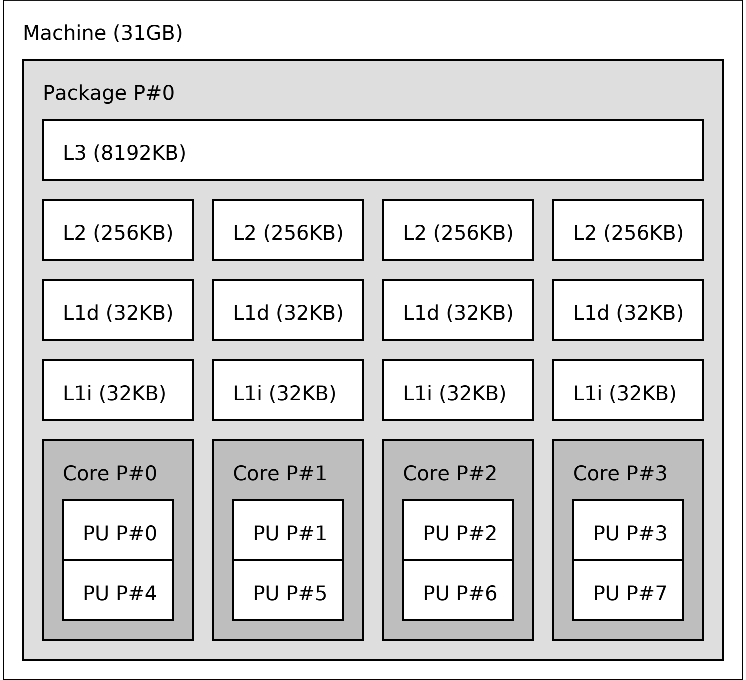
Outline

Generality

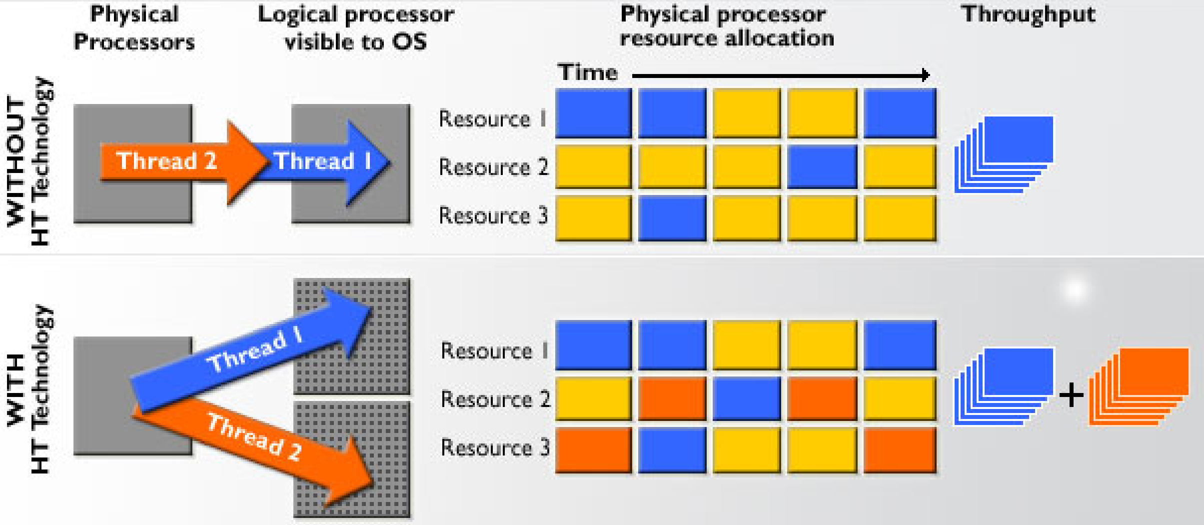
- Multicore Architecture
- Classic Threads
- Problems with Threads

Thread Pool

Summary and References


Machine (31GB)
Package P#0
L3 (8192KB)
L2 (256KB)
L2 (256KB)
L2 (256KB)
L2 (256KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1d (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
L1i (32KB)
Core P#0
PU P#0
PU P#4
Core P#1
PU P#1
PU P#5
Core P#2
PU P#2
PU P#6
Core P#3
PU P#3
PU P#7

How Hyper-Threading Technology Works



Outline

Generality
- Multicore Architecture
- Classic Threads
- Problems with Threads

Thread Pool

Summary and References

Thread API

Interface `Runnable`:

```
void run()
```

Class `Thread`:

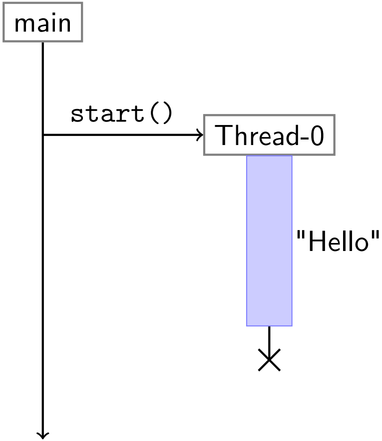
```
Thread()
Thread(Runnable command)
static Thread currentThread()
void join() // synchronous
void run() // synchronous
static void sleep(long millis)
void start() // asynchronous
```

Example of Thread Creation

```java
class ExaRun implements Runnable {
    public void run() {
        System.out.println("Hello");
    }
}
new Thread(new ExaRun()).start();
new Thread(() -> System.out.println("Hello")).start();
```

From now on, we avoid `new Thread()` and `start`, except when implementing an executor.

Thread Creation



Outline

Generality
- Multicore Architecture
- Classic Threads
- Problems with Threads

Thread Pool

Summary and References

Problem 1: C10k Problem

As a motivational example, a server dealing with multiple requests:

- 1 connection: easy without thread
- 10/100 connections: easy with threads, possible without
- 1k connections: technical with threads, difficult without
- 10k connections: problem with only threads, easy with Erlang (green threads)

Thread Costs

Maximum number of concurrent threads:

- around 1 MB for each stack for recursive calls
- at most 10k with 10 GB of RAM

Maximum rate of thread creations:

- around 0.1 ms per creation
- costly creation and destruction

Problem 2: Code Clarity

Low-level thread logic is:

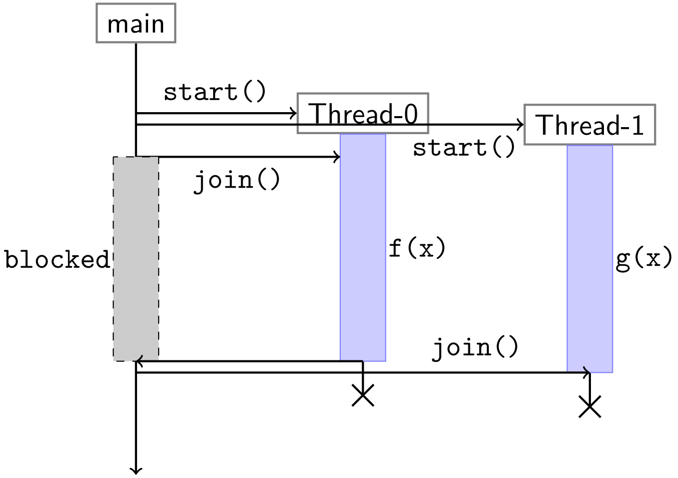
- hard to reason with (difficult to debug)
- heavy to write (lot of synchronizations)

Example of Boilerplate Thread Code

```
int y = f(x);
int z = g(x);
System.out.println(y + z);
```

```
Result result = new Result();
Thread t1 = new Thread(() -> { result.left = f(x); });
Thread t2 = new Thread(() -> { result.right = g(x); });
t1.start();
t2.start();
t1.join();
t2.join();
System.out.println(result.left + result.right);
```

Thread Synchronization



Outline

Generality

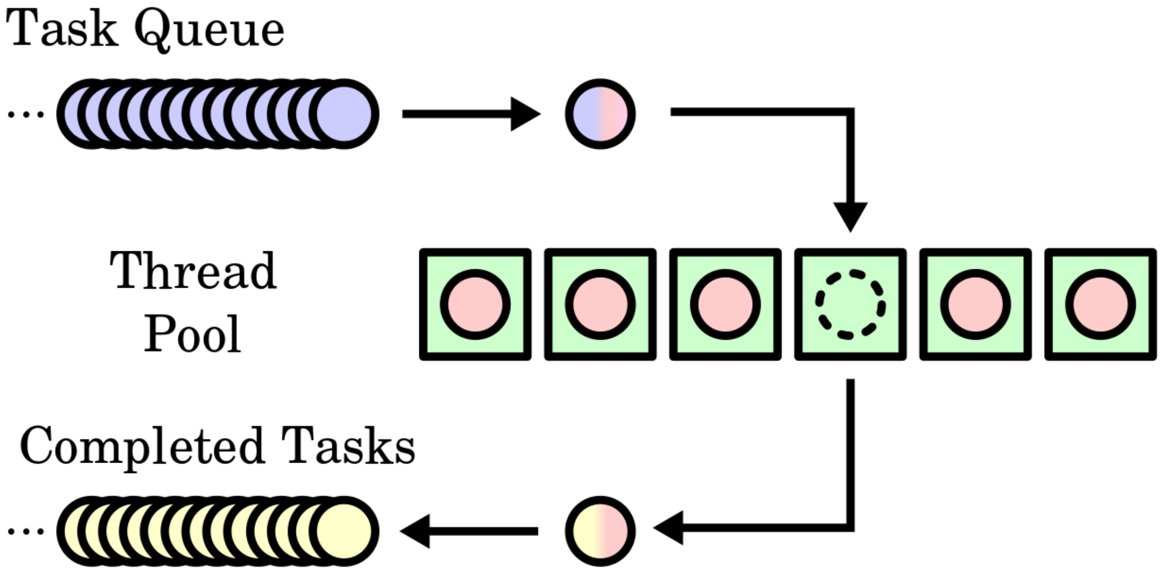
Thread Pool
- Overview
- Future and Task Interface
- Executors Interface
- ForkJoinPool: a Special Thread Pool

Summary and References

Solution

Thread pool:

- one or a few long-running threads per core with several tasks to execute sequentially
- reuse threads (limited number of threads and fewer creations)
- provide features for synchronization
- alleviate both cost and clarity problems


Task Queue
...
Thread
Pool
Completed Tasks
...

Preemption

- Each thread can be interrupted by the operating system for concurrency (to let others have a time-share of a CPU core).
- A task cannot be paused and goes back in the queue to let another task executes.

Outline

Generality

Thread Pool

- Overview
- Future and Task Interface
- Executors Interface
- ForkJoinPool: a Special Thread Pool

Summary and References

Future

- Placeholder (like Optional) for a result that will be computed later.
- Similar to promises.
- Inspired by the conflict between RPC and message passing paradigm:
  - RPC Remote Procedure Call: easy to program with (close to sequential programming) but synchronous (blocking).
  - Message passing Asynchronous but harder to reason about.
- Futures/promises can be seen as messages to future self.

Future<V>

Class Future:

```
boolean cancel(boolean mayInterruptIfRunning)
V get() // synchronous
V get(long timeout, TimeUnit unit) // synchronous
boolean isCancelled()
boolean isDone()
```

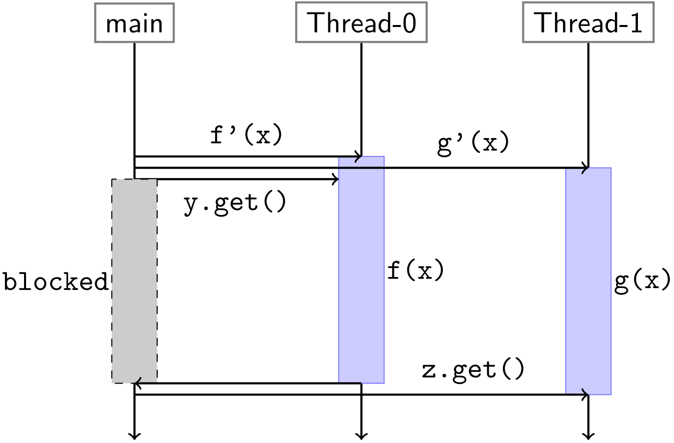
Example of Code Clarity

```
int y = f(x);
int z = g(x);
System.out.println(y + z);
```

Parallel version with asynchronous functions (`f'` and `g'`):

```
Future<Integer> y = f'(x);
Future<Integer> z = g'(x);
System.out.println(y.get() + z.get());
```

Thread Synchronization



Task API

A *task* is either:

- an action: interface `Runnable` defining the method `void run()`
- or a function: interface `Callable<V>` defining the method `V call()`

Outline

Generality

Thread Pool

- Overview
- Future and Task Interface
- Executors Interface
- ForkJoinPool: a Special Thread Pool

Summary and References

Executor

Interface `Executor`:

```
void execute(Runnable command) // asynchronous
```

Example:

```
Executor executor = anExecutor();
executor.execute(new RunnableTask());
executor.execute(() -> { processing(); });
```

Limitation: not possible to check task completions.

ExecutorService

Interface:

```
Future<?> submit(Runnable command) // asynchronous
Future<V> submit(Callable<V> callable) // asynchronous
List<Future<V>> invokeAll(Collection<Callable<V>> callables) // sync.
```

Alternative invocation methods with timeout (preferable when possible).

Example

```
int y = f(x);
int z = g(x);
System.out.println(y + z);
```

```
Future<Integer> y = executor.submit(() -> f(x));
Future<Integer> z = executor.submit(() -> g(x));
System.out.println(y.get() + z.get());
```

Executors

Convenience class to easily create an `ExecutorService`:

```
Executors.newSingleThreadExecutor();
Executors.newFixedThreadPool(10);
Executors.newCachedThreadPool();
```

Different types of thread pools:

- **single thread** execution of a single task at a time
- **fixed thread** the maximum number of threads is fixed (threads are not reclaimed)
- **cached thread** expandable thread pool (suitable with many short-lived tasks)

More customized executors may be created by instantiating class `ThreadPoolExecutor`.

Pool Termination

Methods from interface `ExecutorService` for termination:

```
boolean awaitTermination(long timeout, TimeUnit unit)
void shutdown()
List<Runnable> shutdownNow()
```

Effects:

awaitTermination blocks until all tasks are terminated (with timeout)

shutdown prevent the executor from accepting new tasks

shutdownNow as `shutdown` but cancel waiting tasks and try interrupting executing tasks (return unfinished tasks)

Java: Task Interruption

- Interrupting a thread is done cooperatively.
- The method `interrupt` is called on a given thread.
- The task executed by this thread must regularly check if it has been interrupted with `Thread.interrupted()`.
- Alternatively, the task frequently calls a method that may throw `InterruptedException`.

Complete Example

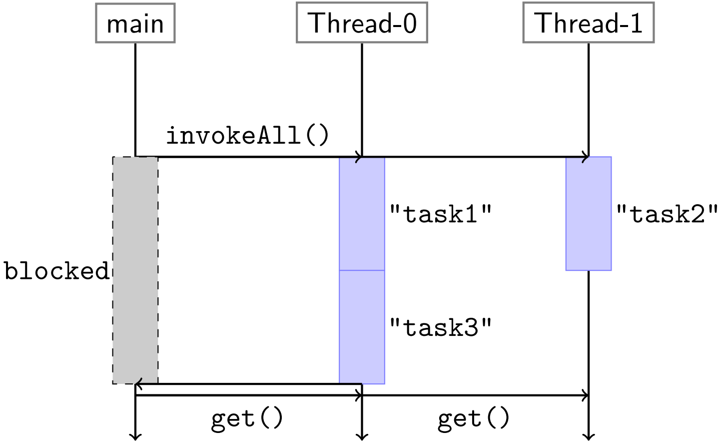
```
List<Callable<String>> callables = Arrays.asList(
    () -> "task1",
    () -> "task2",
    () -> "task3");

List<Future<String>> results =
    executor.invokeAll(callables);

executor.shutdown();
executor.awaitTermination(1, TimeUnit.SECONDS)

results.stream()
    .map(future -> future.get(1, TimeUnit.SECONDS)).
    .forEach(System.out::println);
```

Thread Synchronization



ScheduledExecutorService – part 1

To specify the execution of a task in the future (asynchronous):

```
Future<?> schedule(Runnable command, long delay, TimeUnit unit)
Future<V> schedule(Callable<V> callable, long delay, TimeUnit unit)
```

ScheduledExecutorService – part 2

To specify that a task must be repeated (every few time units or with a minimum delay between each termination and start):

```
Future<?> scheduleAtFixedRate(Runnable command,
    long initialDelay, long period, TimeUnit unit)
Future<?> scheduleWithFixedDelay(Runnable command,
    long initialDelay, long delay, TimeUnit unit)
```

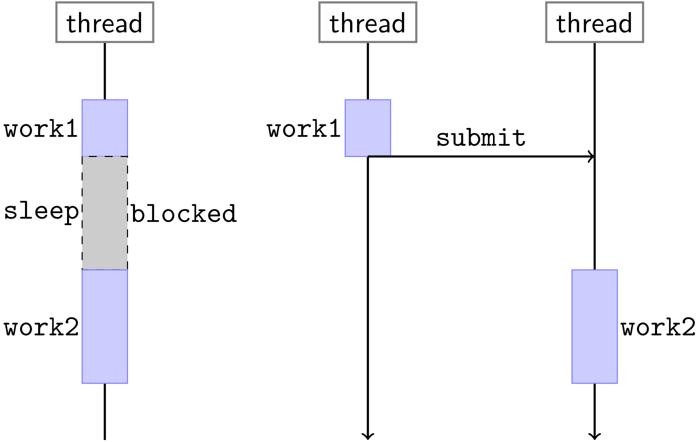
Scheduled versions of most previous methods exist for `Executors`. Moreover, class `ScheduledThreadPoolExecutor` allows creating a customized scheduled executor.

Avoiding Sleeps

```
work1();
Thread.sleep(10000);
work2();
```

```
work1();
ScheduledExecutorService scheduler =
    Executors.newScheduledThreadPool(1);
scheduler.schedule(work2, 10, TimeUnit.SECONDS);
scheduler.shutdown();
```

Sequence Diagram



Other Non-Covered Features

- daemon and non-daemon threads
- `ExecutionException`, `RejectedExecutionException`
- `ThreadFactory`, `RejectedExecutionHandler`
- `ThreadLocal`, `ThreadGroup`, `ThreadInfo`

Outline

Generality

Thread Pool

- Overview
- Future and Task Interface
- Executors Interface
- ForkJoinPool: a Special Thread Pool

Summary and References

A Special Executor

- Designed for tasks that can be decomposed recursively (forked and then joined): `ForkJoinTask`.
- Rely on a work-stealing algorithm: when a thread is free, it steals tasks from other threads.
- Used by parallel streams and `Arrays.parallelSort()`.

ForkJoinPool

Same behavior as `Executor` for `execute` (asynchronous execution without result) and as `ExecutorService` for `submit` (asynchronous execution with a future).

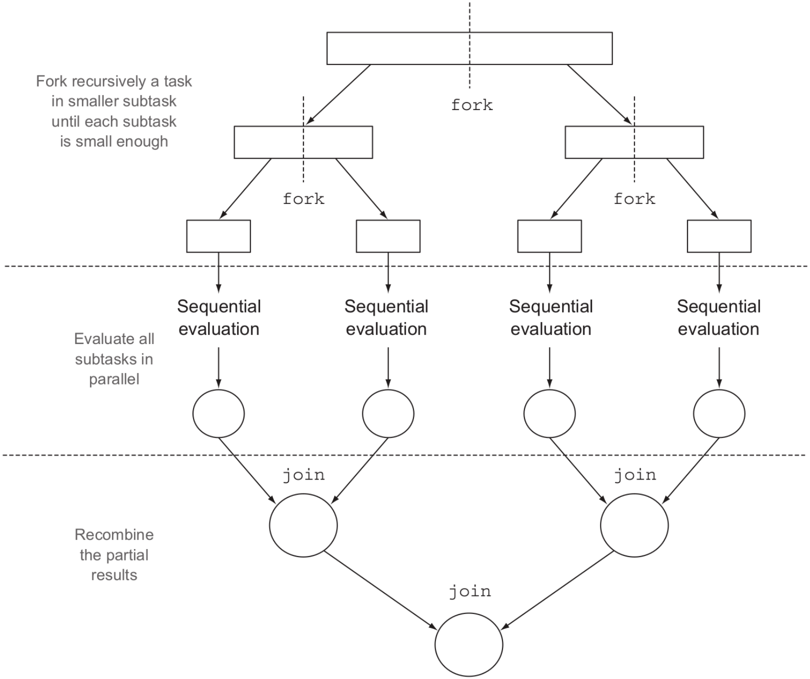
Tasks can be submitted synchronously:

```
T invoke(ForkJoinTask<T> task) // synchronous
```

A static common fork/join pool is available:

```
static ForkJoinPool commonPool()
```

Numerous possible customizations.


Fork recursively a task in smaller subtask until each subtask is small enough
fork
fork
fork
Sequential evaluation
Sequential evaluation
Sequential evaluation
Sequential evaluation
Evaluate all subtasks in parallel
join
join
Recombine the partial results
join

ForkJoinTask<V>

Asynchronous execution:

```
ForkJoinTask<V> fork() // asynchronous
V join() // synchronous
```

Synchronous execution:

```
V invoke() // synchronous
static Collection<ForkJoinTask<?>>
    invokeAll(Collection<ForkJoinTask<?>> tasks) // synchronous
static void invokeAll(ForkJoinTask<?>... tasks) // synchronous
```

Concrete implementations of abstract class `ForkJoinTask`:

- `RecursiveTask<V>` with one method to implement: `V compute()`
- `RecursiveAction` with one method to implement: `void compute()`

Task Division Algorithm

Divide work until it is small enough:

```
if (currentPortion() <= THRESHOLD)
  // do the work directly
else
  // split current work into two pieces
  // fork a piece (incurring additional recursive splits)
  // execute the other piece (incurring additional recursive splits)
  // wait for the result of the first piece
  // combine the results
```

Alternative for the division (with a blocking thread):

```
  // split current work into two pieces
  // invoke the two pieces (incurring additional recursive splits)
  // wait for both results
  // combine the results
```

Example with a RecursiveTask<V>

```
protected Double compute() {
    if (length <= THRESHOLD)
        return computeSequentially();

    int half = length / 2;

    RecTask leftTask = new RecTask(half);
    leftTask.fork();

    RecTask rightTask = new RecTask(length - half);
    Double rightResult = rightTask.compute();

    Double leftResult = leftTask.join();
    return leftResult + rightResult;
}
```

Example with a RecursiveAction

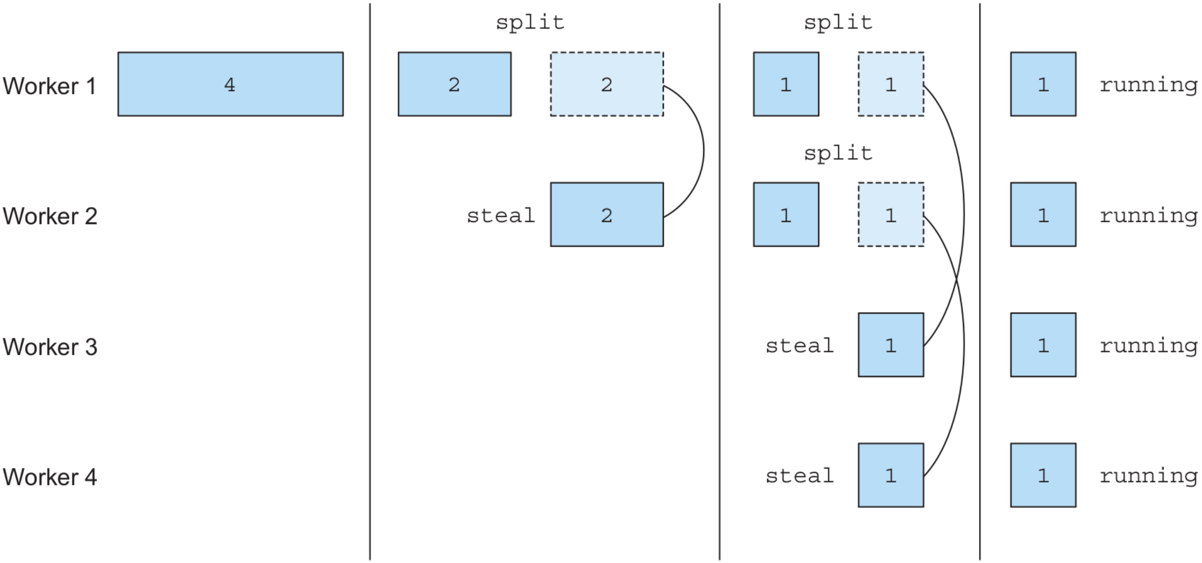
```
protected void compute() {
    if (length < THRESHOLD) {
        computeSequentially();
        return;
    }

    int half = length / 2;

    invokeAll(new RecAction(half),
              new RecAction(length - half));
}
```

Submission Example to ForkJoinPool

```
Long res = new ForkJoinPool()
    .invoke(new RecTask(1_000_000));
new ForkJoinPool()
    .invoke(new RecAction(1_000_000));
```



Worker 1
Worker 2
Worker 3
Worker 4
4
split
2
2
steal
2
split
1
1
split
1
1
steal
1
steal
1
1
running
1
running
1
running
1
running

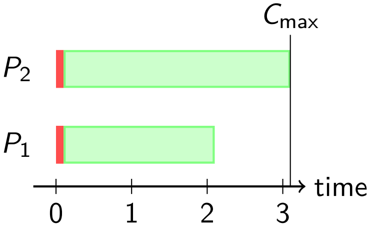
Threshold Selection

Select threshold by testing and measuring performance:

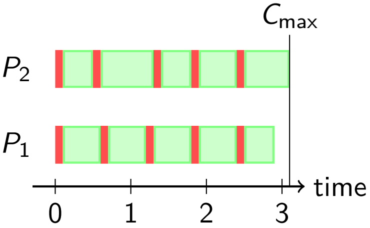
- Too much divisions (low `THRESHOLD`) leads to task management overhead: fine grain.
- Not enough divisions (high `THRESHOLD`) leads to work imbalance among the threads (some will finish earlier than others): coarse grain.

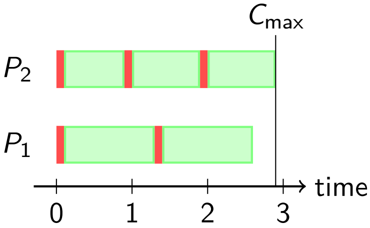
Work Imbalance

Coarse grain:



Fine grain:





Complete Example (Sequential Part)

```
public class RecPiTask extends RecursiveTask<Double> {
    static long THRESHOLD = 1_000_000;
    long MC;

    public RecPiTask(long MC) {
        this.MC = MC;
    }

    protected Double computeSequentially() {
        Supplier<double[]> supplier = () -> new double[] {
            ThreadLocalRandom.current().nextDouble(),
            ThreadLocalRandom.current().nextDouble() };
        return Stream.generate(supplier)
            .limit(MC)
            .filter(x -> x[0] * x[0] + x[1] * x[1] < 1)
            .count() * 4. / MC;
    }
```

Complete Example (Parallel Part)

```
    protected Double compute() {
        if (MC <= THRESHOLD)
            return computeSequentially();

        long half = MC / 2;

        RecPiTask leftTask = new RecPiTask(half);
        leftTask.fork();

        RecPiTask rightTask = new RecPiTask(MC - half);
        Double right = rightTask.compute();

        Double left = leftTask.join();
        return (half * left + (MC - half) * right) / MC;
    }
}
```

Complete Example (Invocation Part)

```
long MC = 100_000_000;
// Sequentially
Double pi1 = new RecPiTask(MC)
    .computeSequentially();
// In parallel
Double pi2 = ForkJoinPool.commonPool()
    .invoke(new RecPiTask(MC));
```

Outline

Generality

Thread Pool

Summary and References

Thread Pools

- Support for concurrency in Java has evolved and continues to evolve.
- Thread pools are generally helpful but can cause problems when many tasks are blocking.
- The fork/join framework lets you recursively split a parallelizable task into smaller tasks, execute them on different threads, and then combine the result of each subtask in order to produce the overall result.

Official Documentation

- Documentation of package `concurrent`
- Documentation of interface `ExecutorService`
- Documentation of class `Executors`
- Documentation of class `ForkJoinPool`
- Documentation of class `ForkJoinTask`
- Java Tutorial on Executors