CSE201: Advanced Programming

Lecture 19: Thread Pool

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- There are two ways to create your own **Thread** object
  - Implementing the **Runnable** interface
  - Subclassing the **Thread** class and instantiating a new object of that class

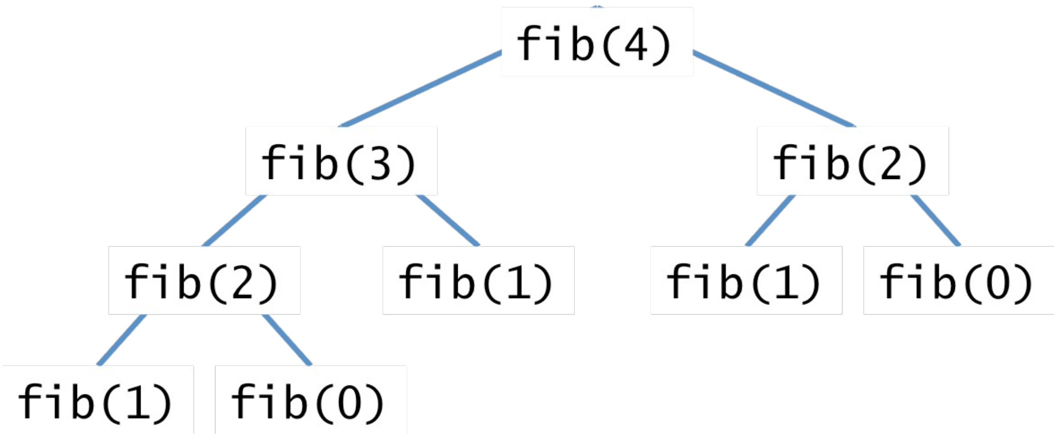
```
public class ArraySum extends Thread {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    @Override
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                          throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum t1 = new ArraySum(array, 0, size/2);
      ArraySum t2 = new ArraySum(array, size/2, size);
      t1.start(); t2.start();
      t1.join(); t2.join();
      int result = t1.getResult() + t2.getResult();
    }
}
```

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                          throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      Thread t1 = new Thread(left);
      Thread t2 = new Thread(right);   // struck through
      t1.start(); t2.start(); right.run();   // t2.start(); struck through
      t1.join(); t2.join();   // t2.join(); struck through
      int result = left.getResult() + right.getResult();
    }
}
```

Changes for using only one thread along with the main thread

- **Multiple inheritance is not allowed in Java** hence if our ArraySum class extends Thread then it cannot extend any other class. By implementing Runnable our ArraySum can easily extend any other class
- **Subclassing is used in OOP to add additional feature**, modifying or improving behavior. If no modifications are being made to Thread class then use Runnable interface
- **Thread can only be started once**. Runnable is better as same object could be passed to different threads
- If just run() method has to be provided then **extending Thread class is an overhead for JVM**

Think Tasks, not Threads

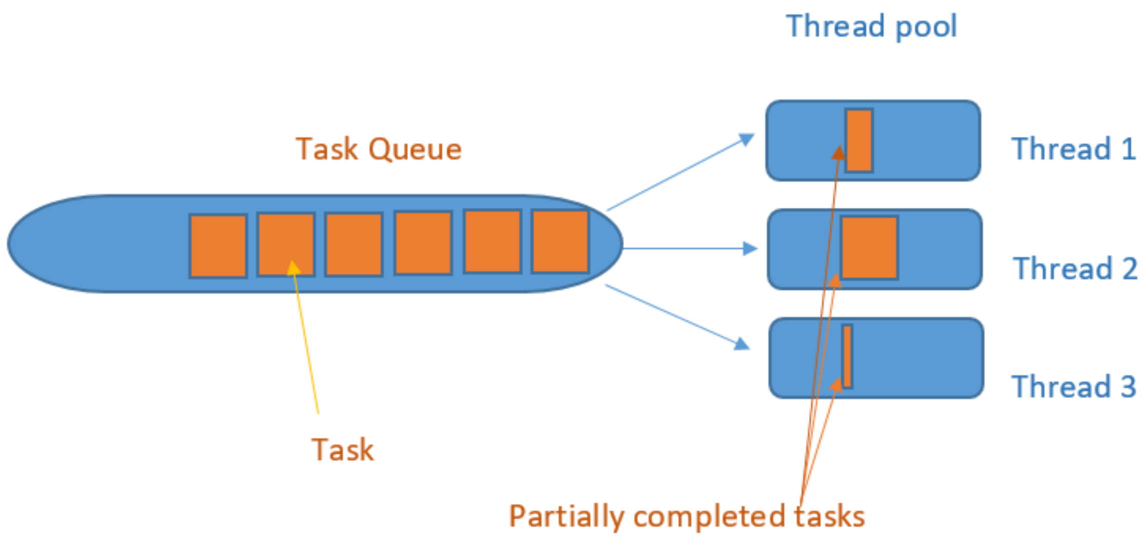


- Tasks are logic unit of work
- Threads are mechanism by which tasks can run asynchronously
- E.g., for calculating Fibonacci number (Lecture 18), each node in this tree represents one task
- Tasks are lightweight than a thread !
  - Why ?

Mapping Tasks to Cores

- Generally
  - # of tasks $\geq$ threads available
  - parallel algorithm must map tasks to threads
  - schedule independent tasks on separate threads (consider computation graph)
  - threads should have minimum interaction with one another

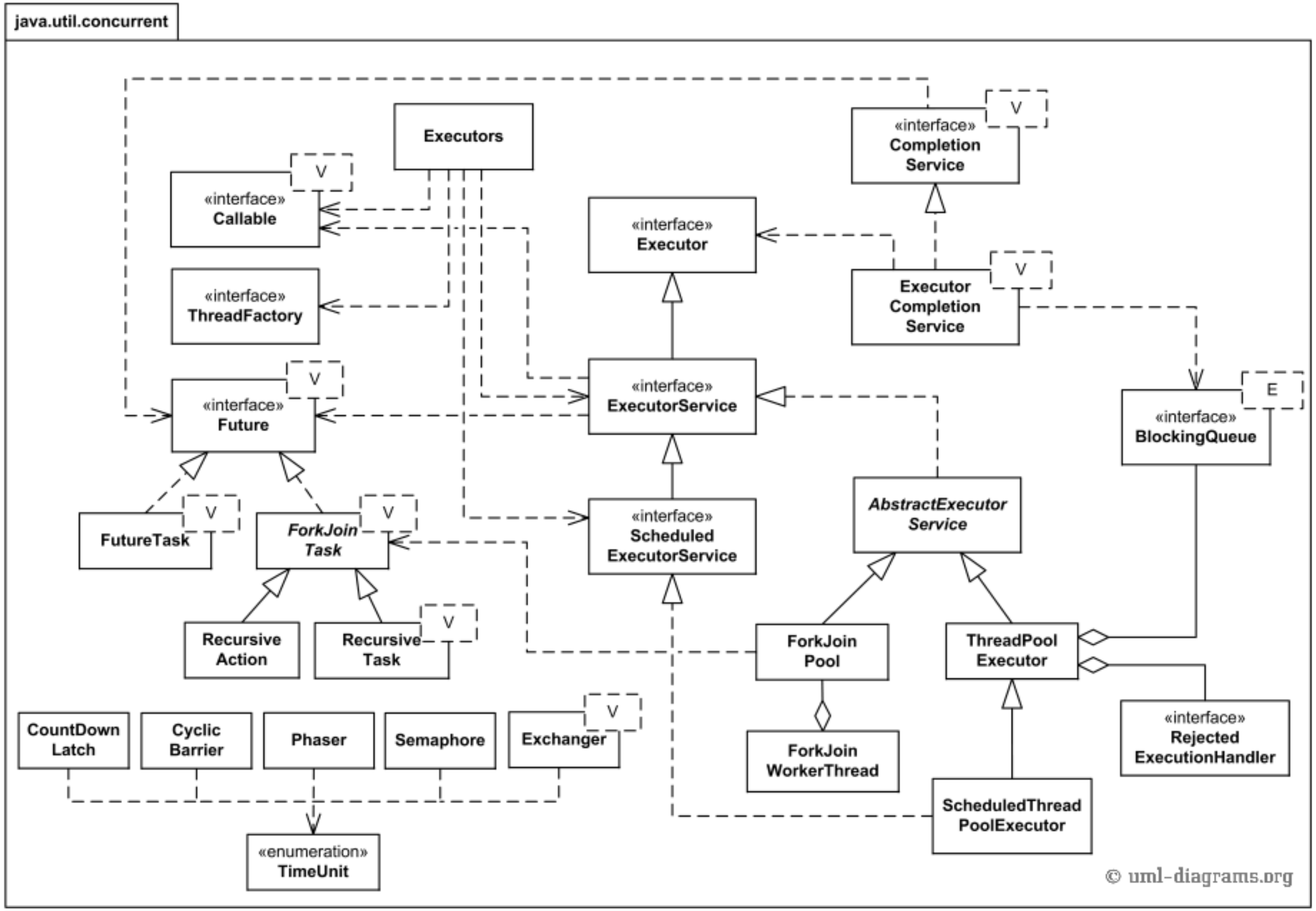
Thread Pool



- Thread-pool consists of a fixed number of threads
  - Provided by the Java runtime
- User application creates “task” rather than threads
- These tasks are added to a task-pool
- Free threads from thread-pool takes out a task from task-pool and execute it

Fig. Source: http://www.geek-programmer.com/what-are-thread-pools-in-java/

Package java.util.concurrent



- Framework for concurrent programming
- In this course we will only introduce a few basic features of this framework

ExecutorService Interface

- An `ExecutorService` is a group of thread objects (thread pool), each running some variant of the following
  - `while (....) { get work and run it; }`
- `ExecutorService` take responsibility for the threads they create
  - User starts and shuts down `ExecutorService`
  - `ExecutorService` starts and shut down threads
- Method `execute(Runnable object)`
  - Accepts task as a Runnable type object that is executed by a thread in thread pool
- Method `shutdown()`
  - Thread pool terminates once all pre-submitted tasks are executed

Executors Class

- Provides factory and utility methods for ExecutorService
- Static method `newFixedThreadPool(int num_threads)`
  - Creates a thread pool that reuses a fixed number of threads for task execution

Let’s Revisit Our Parallel Array Sum

```
public class ArraySum implements Runnable {
    int[] array;
    int sum, low, high;
    public ArraySum(int[] arr, int l, int h) {
        array=arr; sum=0; low=l; high=h;
    }
    //assume array.length%2=0
    public void run() {
        for(int i=low; i<high; i++)
            sum += array[i];
    }
    public int getResult() { return sum; }
    public static void main(String[] args)
                                throws InterruptedException {
      int size; int[] array; //allocated (size) & initialized
      ExecutorService exec = Executors.newFixedThreadPool(2);
      ArraySum left = new ArraySum(array, 0, size/2);
      ArraySum right = new ArraySum(array, size/2, size);
      exec.execute(left); exec.execute(right);
      if(!exec.isTerminated()) {
          exec.shutdown();
          exec.awaitTermination(5L, TimeUnit.SECONDS);
      }
      int result = left.getResult() + right.getResult();
    }
}
```

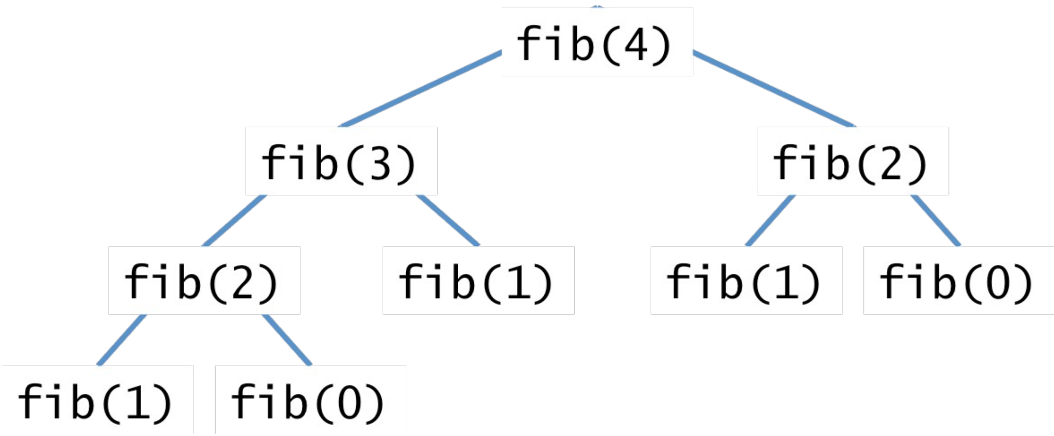
- `ExecutorService` methods:
  - `isTerminated()`
    - Returns true if all tasks are terminated following the shutdown
  - `awaitTermination(long timeout, TimeUnit unit) throws InterruptedExecption`
    - Blocks until all tasks have completed execution after a shutdown request
- Important that you wait for all tasks to terminate after a shutdown request

Let’s Revisit Our Multithreaded Server

```
import java.io.*; import java.net.*;
import java.util.concurrent.*;
public class Server  {
    public static void main(String args[ ])
                                    throws IOException {
        /* create a server socket
           bound to the specified port 1234 */
        ServerSocket me = new ServerSocket(1234);
        /* Server is now listening
           for incoming client’s request */
        ExecutorService exec = Executors.newFixedThreadPool(2);
        while (true) {
           /* Connection is established */
           Socket connection = me.accept();
           System.out.println("Connected");
           Runnable task = new ConnectionHandler(connection);
           /* new Thread(task).start(); */
           exec.execute(task);
        }
    }
}
```

- Rather than creating a new thread for every incoming client connection, we will instead create a new task and submit it to thread pool
    - No other changes to Server.java or Client.java
- Now our server will not go crazy even if several clients are lined up simultaneously

How to Improve Parallel Fibonacci?



- We know that there is a lot of parallelism and hence its not efficient to just create two tasks, i.e., one task for fib(n-1) and another task for fib(n-2)
- Every node in this tree can be computed in parallel
- Recursive divide and conquer application!

ForkJoinPool

- Designed to support a common need
  - Recursive divide and conquer pattern
  - For small problems (below cutoff threshold), execute sequentially
  - For larger problems
    - Define a task for each subproblem
    - Library provides
      - A Thread manager, called a ForkJoinPool
      - Methods to send your subtask objects to the pool to be run, and your call waits until they are done
      - The pool handles the multithreading well
- The “thread manager”
  - Used when calls are made to RecursiveTask’s methods fork(), invokeAll(), etc.

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n, int _r) { n=_n; result=_r; }

}
```

- Step-1
  - `Fibonacci` class should extend the class `RecursiveAction`
  - `RecursiveAction` represents a task that doesn’t return any result

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n, int _r) { n=_n; result=_r; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);



    }



}
```

- Step-2
  - Implement the method "`public void compute()`"
    - Similar to run() method
  - Computes the recursive divide and conquer task
  - Similar to Runnable implementation of Fibonacci, create the two tasks. One for calculating fib(n-1) while the other for calculating fib(n-2)

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n, int _r) { n=_n; result=_r; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();



    }




}
```

- Step-3
  - Start the first task (“left”) **asynchronously**
  - Calling the fork() method on one of the task is similar to calling start() on a thread.
  - However, fork() does not start any new thread but rather adds this task to the task pool
    - Similar to calling execute() from ExecuterService

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n, int _r) { n=_n; result=_r; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        right.compute();


    }




}
```


- Step-4
  - Start the second task ("right") **sequentially**, i.e. on the current thread
  - Why not start this also with fork() ?
    - Not an error and you can definitely do so
    - However, the current thread is already done with current task (the compute() method) hence it can be reused to directly compute the "right" task

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n, int _r) { n=_n; result=_r; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        right.compute();
        left.join();
        this.result = left.result + right.result;
    }




}
```

- Step-5
  - Once the “right” task completes, compute method should wait for all the asynchronous tasks spawned inside it (i.e. “left” task)
  - left.join() is a blocking operation and will return only when “left” has terminated
    - Similar to thread.join() but this waits for a “task” to terminate rather than a “thread”
  - Sum the partial results

Parallel Fibonacci Using ForkJoinPool

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveAction {
    int n, result;
    public Fibonacci(int _n, int _r) { n=_n; result=_r; }

    public void compute() {
        if(n<2) {
            this.result = n;
            return;
        }
        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        right.compute();
        left.join();
        this.result = left.result + right.result;
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        pool.invoke(task);
        int result = task.result;
    }
}
```

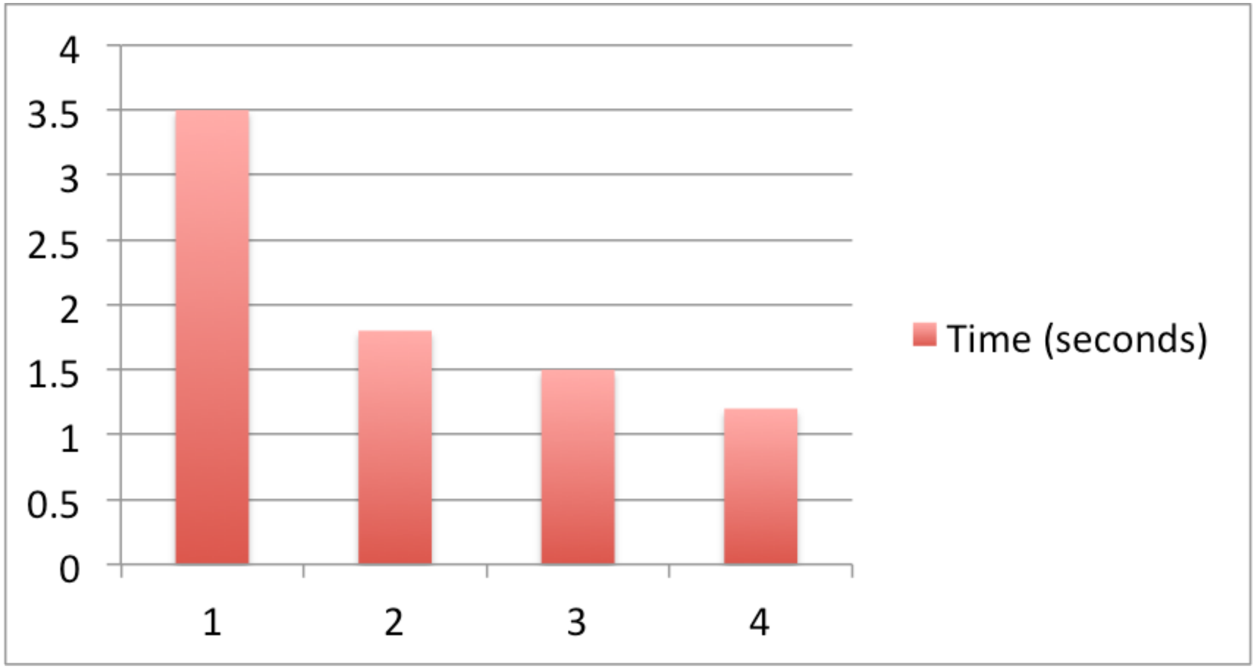

- Step-6
  - Create a ForkJoinPool type thread pool with fixed number of threads
  - Create the root task (see the binary tree representation for Fibonacci)
  - Add this root task in the **task pool**
    - `pool.invoke`
    - Blocking operation and doesn't return until all tasks are terminated
  - A free thread from thread pool will execute this task and recursively create new tasks that will in turn be added to the task pool

Using RecursiveTask<T> to Return Value

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveTask<Integer> {
    int n;
    public Fibonacci(int _n) { n=_n; }

    public Integer compute() {
        if(n<2) return n;

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```

- RecursiveTask<T> is better suited in scenarios where there is a need to return results from each task (same return type for all tasks)
- Very minimal changes required to our Fibonacci program to use this feature

Performance of Our Parallel Fibonacci



Total Threads in ForkJoinPool

- Increasing the thread pool size decreases the execution time
  - 4 core processor

Too Many Tasks Hamper Performance

```
import java.util.concurrent.*;

public class Fibonacci extends RecursiveTask<Integer> {
    int n;
    static int threshold = 10;
    public Fibonacci(int _n) { n=_n; }
    private int sequential(int n) {
        if(n<2) return n;
        else return sequential(n-1) + sequential(n-2);
    }
    public Integer compute() {
        if(n<threshold) return sequential(n);

        Fibonacci left = new Fibonacci(this.n-1);
        Fibonacci right = new Fibonacci(this.n-2);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Fibonacci task = new Fibonacci(40);
        int result = pool.invoke(task);
    }
}
```

- Although, tasks are lightweight than threads, too many tasks can also hamper the performance
- Use some cut off in your application to stop creation of tasks beyond certain threshold
    - When computation become too small, stop creation of any new task
- Fibonacci on left even with a single thread will run significantly faster than the Fibonacci shown on slide-16

Thread Pool Shutdown

```
import java.util.concurrent.*;

public class Search extends RecursiveAction<...> {
    ......
    public void compute() {
        if(this.searchItemIsFound()) {
            pool.shutdownNow();
        }

        Search left = new Search(...);
        Search right = new Search(...);
        left.fork();
        return right.compute() + left.join();
    }
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(2);
        Search task = new Search(..., pool);
        try {
            pool.invoke(task);
        }
        catch(CancellationException e) {
            System.out.println("Goal is found, pool aborted");
        }
    }
}
```

- For some type of parallel applications (e.g., searching element in a huge array) you would like to stop creating tasks once the goal is found
  - **Speculative parallelism**
- *public void shutdownNow()*
  - Stops everything, i.e., creation of new tasks, all running tasks and previously submitted tasks
  - Throws an unchecked exception CancellationException upon cancellation

Measures of parallel performance

- Speedup = $T_{serial}/T_{parallel}$
- Parallel efficiency = $T_{serial}/(pT_{parallel})$

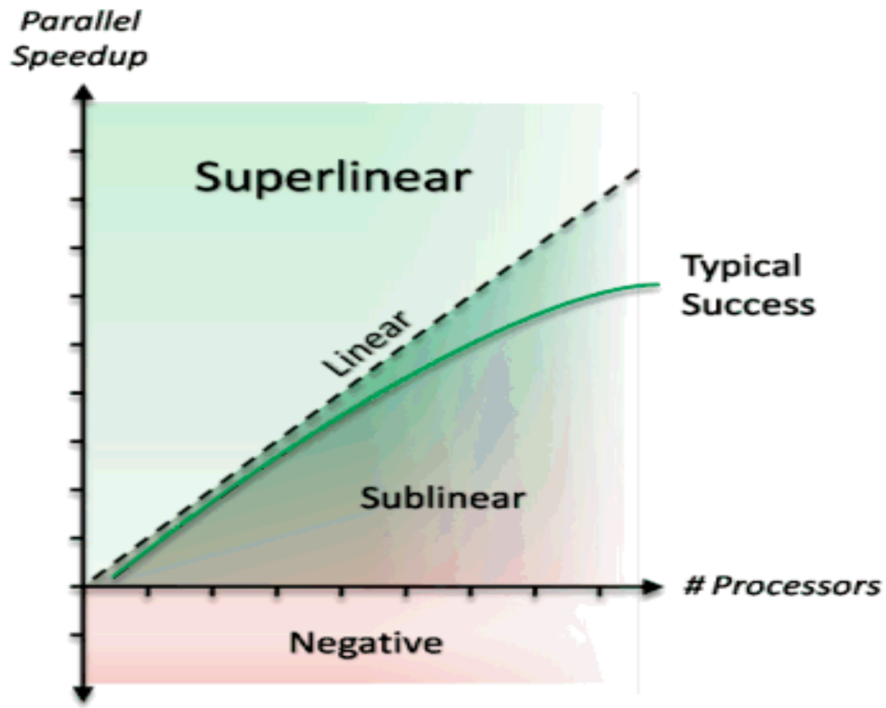
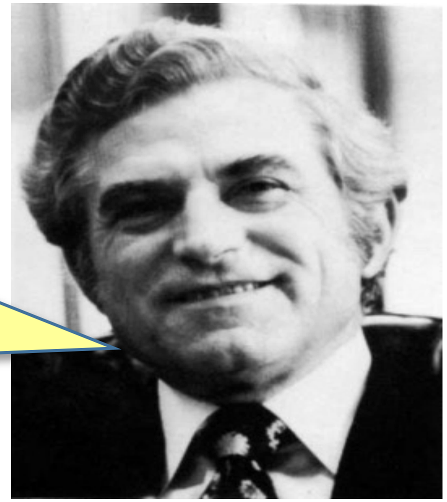


Fig. source: http://www.drdobbs.com/cpp/going-superlinear/206100542

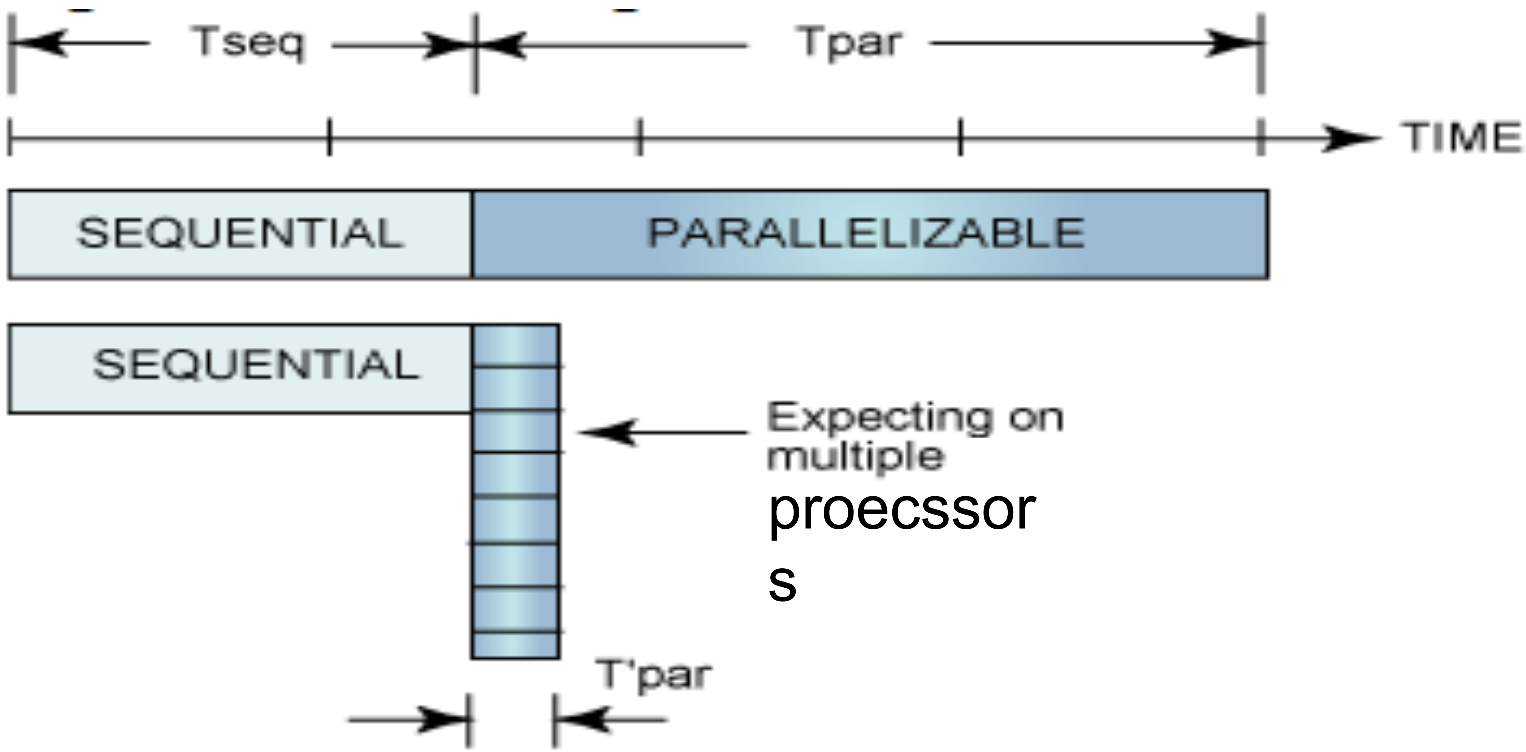
Amdahl's Law

If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on.



Gene M. Amdahl

Amdahl’s Law



T_Parallel = Tseq + T’par

With infinite processors, T’par ~ 0 (theoretically) => T_Parallel = Tseq

Next Lecture

- Mutual exclusion