Algorithmics

Parallel algorithms

Vicente García Díaz – garciavicente@uniovi.es
University of Oviedo, 2016

Table of contents

Basic concepts

Concurrent computing

- The concurrent computation focuses on performing two or more tasks on one or more CPUs
- It is the opposite to sequential computation
- Very common to perform tasks "simultaneously“
  - When we are waiting for the user to enter a value, we may be saving others in a database
  - Another example are the tasks that are performed in the background
- Actually they do not have to do the tasks at the same time because they may be using only one CPU
  - It gives the feeling of parallelism but sentences are executed one after the other in sequence
- There are used Java objects of type `Thread` to work with threads

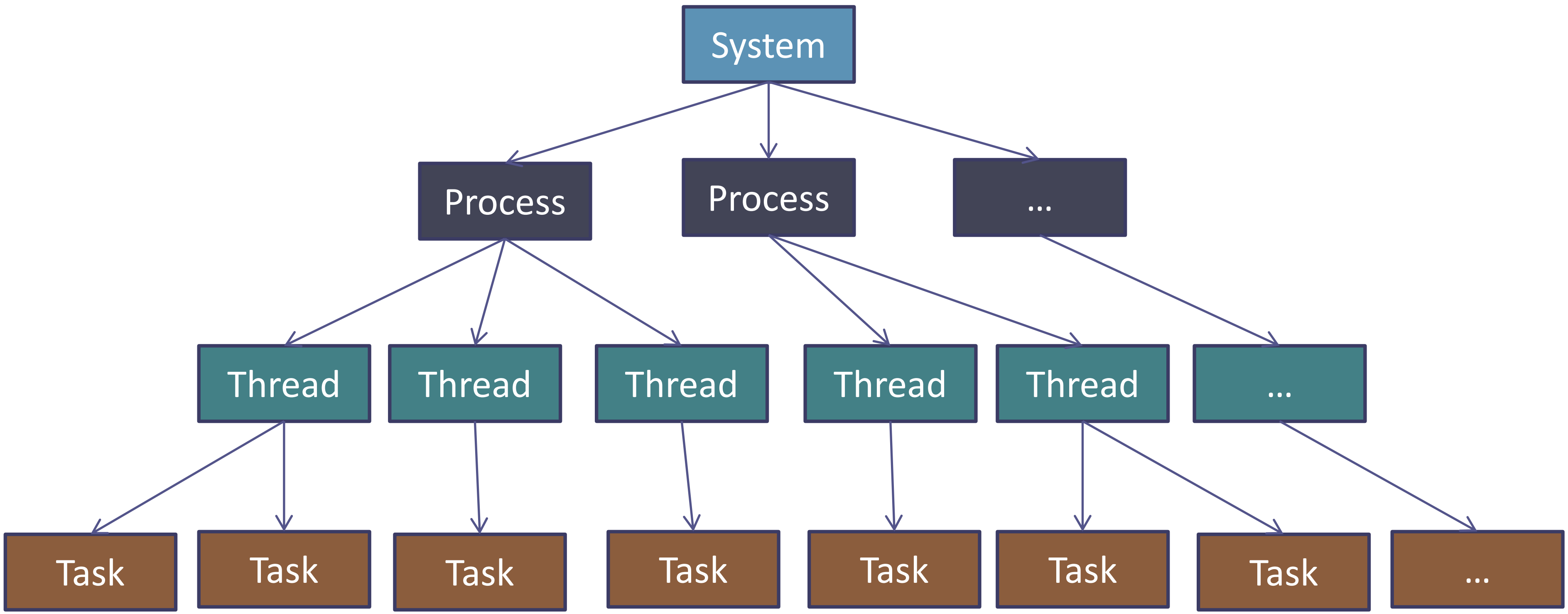
Mechanism to create concurrence (I)

- Process
  - An operating-system abstraction that allows one computer system to support different units of execution
- Thread
  - Belongs to exactly one process and is a block that is executed independently of others. Threads can share resources in the process
- Task
  - A compontent or small unit of execution

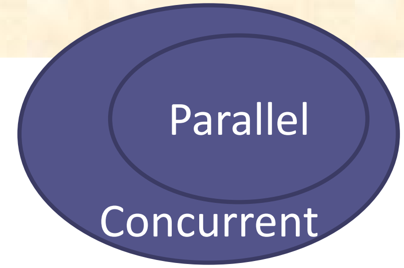
- The operating system is responsible for executing the threads concurrently on one or more processors (concurrently or in parallel)

- You have to synchronize the threads so that threads do not block each other and do not try to simultaneously access certain system resources

Mechanism to create concurrence (II)



Parallel computing



- It is intrinsically concurrent (multithreaded)
- Used to perform simultaneous tasks using more than one CPU
- Can greatly speed up various operations
  - P.e., sorting, searching, transformations, …
- It is definitely true for any task that can be broken down into smaller components (divide and conquer)

Fork/Join Framework

Concept

- It is available since Java 7
  - `java.util.concurrent`
- The aim is to utilize the full power of parallel processing to improve application performance
- General idea:

```
if (work is small enough)
   Do the work directly
else{
   Break up the work into smaller components
   Perform the resolution of components in parallel
   Wait for results
}
```

Advantages

1. It facilitates the creation and use of multiple threads
2. It makes use of multiple processors in parallel automatically

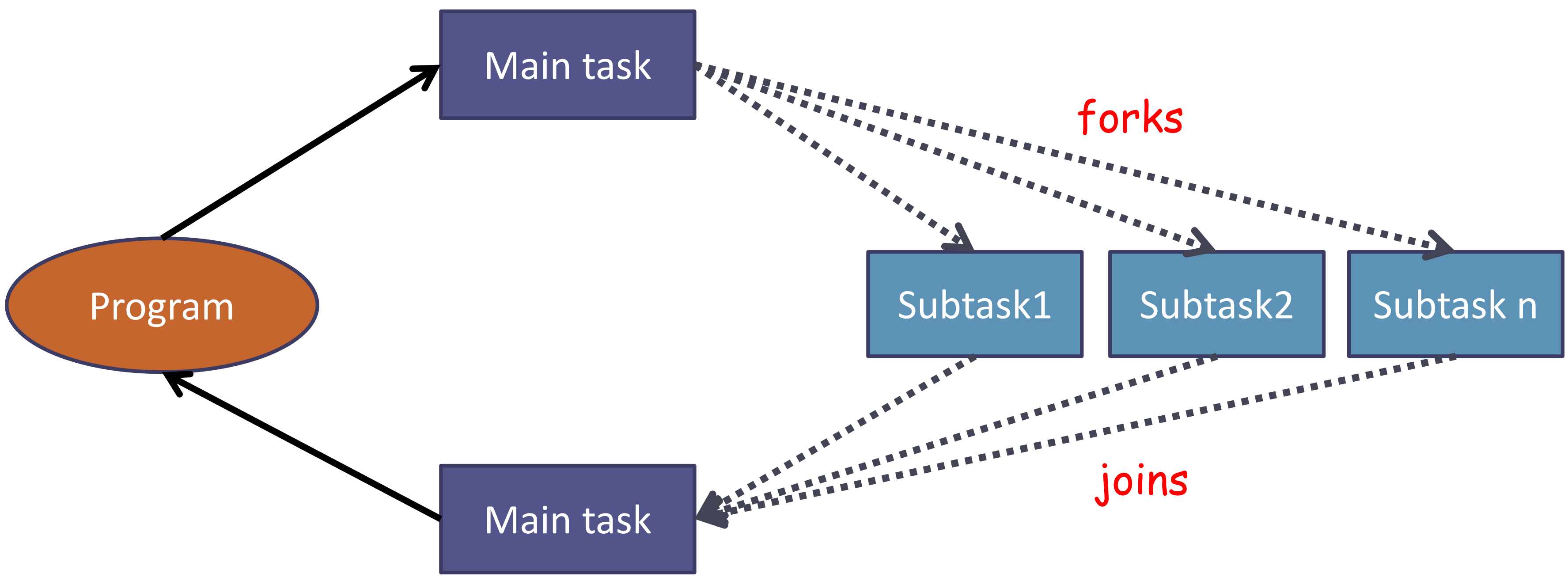
Pseudocode

```
public <T> solve(Problem problem){
        if (problem.size < SEQUENTIAL_THRESHOLD)
                return solveSequentially(problem);
        else {
                <T> left, right;
                INVOKE-IN-PARALLEL {
                        left = solve(extractLeftHalf(problem));
                        right = solve(extractRightHalf(problem));
                }
                return combine(left, right);
        }
}
```

Approach (I)

1. Partition into subproblems
   - Divide a large problem into smaller ones
2. Create subtasks
   - Design a solution for each of the subproblems independently (through a thread)
3. Fork subtasks
   - Indicate that you want to start solving subproblems (the threads are sent to a pool of threads)
   - The pool size depends on the number of CPUs and other considerations
4. Join subtasks
   - Wait for processing and solving each of the subproblems (usually all)
5. Compose solution
   - Compose the solution from the obtained partial solutions

Approach (II)


Main task
forks
Program
Subtask1
Subtask2
Subtask n
joins
Main task

Main components

- `ForkJoinTask<V>`
  - An abstract class that defines a task
- `ForkJoinPool`
  - A class that manages the execution of `ForkJoinTasks`
- `RecursiveAction`
  - A subclass of `ForkJoinTask<V>` for tasks that do not return values
- `RecursiveTask<V>`
  - A subclass of `ForkJoinTask<V>` for tasks that return values

`ForkJoinTask<V>` (I)

- It is an abstract class that defines a class that can be managed by `ForkJoinPool`
- `V` specifies the result type of the task
- `ForkJoinTask` raises the level of abstraction of `Thread`
  - Task VS Thread of execution
- `ForkJoinTasks` are executed by threads managed by a thread pool of type `ForkJoinPool`
- `ForkJoinTasks` are much more efficient than threads

`ForkJoinTask<V>` (II)

- `final ForkJoinTask<V> fork()`
  - Submits the invoking task for aynchronous execution

- `final V join()`
  - Waits until the task on which it is called terminates

- `final V invoke()`
  - Combines the `fork` and `join` operations into a single call

- `static void invokeAll(ForkJoinTask<?> task1, ForkJoinTask<?> task2)`

- `static void invokeAll(ForkJoinTask<?>… taskList)`

`RecursiveAction`

- It is a subclass of `ForkJoinTask`
- Encapsulates a task that does not return values (`void`)
- `protected abstract void compute()`
- It is used to implement a recursive, **divide-and-conquer** strategy

`RecursiveTask<V>`

- It is a subclass of `ForkJoinTask`
- Encapsulates a task that returns a result
- `protected abstract V compute()`
- It is used to implement a recursive, **divide-and-conquer** strategy

`ForkJoinPool` (I)

- It is responsible for launching and managing the execution of `ForkJoinTasks`
- Two very typical constructors
  - `ForkJoinPool()`
    - Supports a level of parallelism equal to the number of processors available in the system
  - `ForkJoinPool(int pLevel)`
    - Lets you specify the level of parallelism
- Level of parallelism
  - Number of threads that can be executed concurrently / in parallel

`ForkJoinPool` (II)

- The level of parallelism does not limit the number of tasks that can be managed by the pool
- …although the number of tasks that can execute simultaneously cannot exceed the number of processors
- …and the level of parallelism is a target, not a garantee
- `<T> invoke(ForkJoinTask<T> task)`
  - The calling code waits until the return of the method
- `void execute(ForkJoinTask<?> task)`
  - The calling code continues execution asynchronously

Divide and Conquer strategy

- The idea is to make recursive calls to classes that extend `RecursiveTask` or `RecursiveAction`
- Reducing the size of the problem to a size that can be managed sequentially faster than by creating another division
- The key is to find a good **threshold** to stop the decomposing of the problem
- We must find a balance between what it costs to break the problem and what it costs to solve it
- It **cannot** be based on the **number of processors** because it changes in different computers and other applications may be making use of them

Problem

76	13	97	90	48	76	94	59	42	5

5776	169	9409	8100	2304	5776	8836	3481	1764	25

Fork/Join Framework

`RecursiveAction`. Obtaining the square of the values of an array

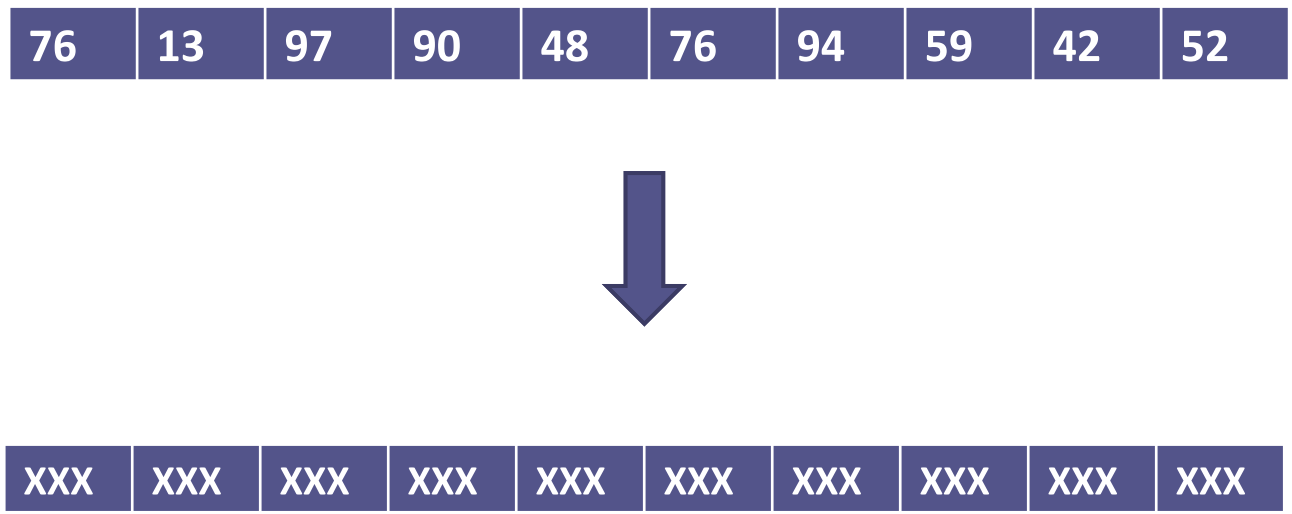
```
package com.vgd.algorithmsparallel;

import java.util.concurrent.*;

public class RecursiveActionSquare extends RecursiveAction {
    private static final long serialVersionUID = 1L;

    int[] data; //Array with numbers (data)
    int start, end; //Deterines what part of data to process.
    //In real word code, its optimal value can be determined by experimentation
    final int seqThreshold = 100; //Arbitrary set at 100

    RecursiveActionSquare(int[] data, int start, int end) {
        this.data = data;
        this.start = start;
        this.end = end;
    }

    @Override
    protected void compute() {
        //If number of elements is below the sequential threshold, then process sequentially
        if((end - start) < seqThreshold) {
          for(int i = start; i < end; i++) {
             data[i] = data[i]*data[i]; //Transform each element into its square
          }
        }
        else { //Continue to break the data into smaller components
          int middle = (start + end) / 2; //Find the midpoint
          //Invoke new subtasks
          invokeAll(new RecursiveActionSquare(data, start, middle),
                    new RecursiveActionSquare(data, middle, end));
        }
    }
}
```

Fork/Join Framework

`RecursiveAction`. Obtaining the square of the values of an array

```
package com.vgd.algorithmsparallel;

import java.util.Random;
import java.util.concurrent.*;

public class RecursiveActionSquareTest {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(); //Task pool
        Random rnd = new Random(); //Random numbers
        int[] data = new int[1000]; //Numbers to work with

        for(int i = 0; i < data.length; i++) //Some values
          data[i] = rnd.nextInt(100);

        System.out.println("The original sequence:");
        for(int i=0; i < data.length; i++)
          System.out.print(data[i] + " ");
        System.out.println("\n");

        RecursiveActionSquare task = new RecursiveActionSquare(data, 0, data.length);
        pool.invoke(task); //Start the main ForkJoinTask

        System.out.println("The transformed sequence:");
        for(int i=0; i < data.length; i++)
            System.out.print(data[i] + " ");
        System.out.println();
    }
}
```

Problem


76 13 97 90 48 76 94 59 42 52
XXX XXX XXX XXX XXX XXX XXX XXX XXX XXX

Fork/Join Framework

`RecursiveAction`. Comparison of different thresholds and CPUs

```
public class RecursiveActionComparison extends RecursiveAction {
    private static final long serialVersionUID = 1L;

    int[] data; //Array with numbers (data)
    int start, end; //Deterines what part of data to process.
    //In real word code, its optimal value can be determined by experimentation
    int threshold = 100; //Arbitrary set at 1000

    RecursiveActionComparison(int[] data, int start, int end, int threshold) {
        this.data = data;
        this.start = start;
        this.end = end;
        this.threshold = threshold;
    }

    @Override
    protected void compute() {
        //If number of elements is below the sequential threshold, then process sequentially
        if((end - start) < threshold) {
          //Time consuming task so the effects of concurrent execution are more obvervable
          for(int i = start; i < end; i++) {
             data[i] = (int)(Math.cbrt(data[i]));
          }
        }
        else { //Continue to break the data into smaller components
          int middle = (start + end) / 2; //Find the midpoint
          //Invoke new subtasks
          invokeAll(new RecursiveActionComparison(data, start, middle, threshold),
                    new RecursiveActionComparison(data, middle, end, threshold));
        }
    }
}
```

Fork/Join Framework

`RecursiveAction`. Comparison of different thresholds and CPUs

```java
public class RecursiveActionComparisonTest {
    public static void main(String[] args) {
        int level; //Level of parallelism
        int threshold;

        if(args.length !=  2) {
          System.out.println("Usage: RecursiveActionComparisonTest threshold parallism");
          return;
        }

        level = Integer.parseInt(args[0]);
        threshold = Integer.parseInt(args[1]);

        ForkJoinPool pool = new ForkJoinPool(level); //Task pool with the level
        Random rnd = new Random(); //Random numbers
        int[] data = new int[10000000]; //Numbers to work with

        for(int i = 0; i < data.length; i++) //Some values
          data[i] = rnd.nextInt(100);

        RecursiveActionComparison task =
                new RecursiveActionComparison(data, 0, data.length, threshold);

        long t1 = System.currentTimeMillis(); //to measure the time
        pool.invoke(task); //Start the main ForkJoinTask
        long t2 = System.currentTimeMillis();

        System.out.println("Level of parallelism: " + level);
        System.out.println("Sequential threshold: " + threshold);
        System.out.println("Elapsed time: " + (t2-t1) + " ms");
        System.out.println();
    }
}
```

Get information about the parallelism

- You can know the current level of parallelism and the number of processors available on the computer by calling two simple methods:

```
public class ParallelismInfo {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(5); //Task pool

        System.out.println("Level of parallelism: " +
        pool.getParallelism());

        System.out.println("Available processors: " +
        Runtime.getRuntime().availableProcessors());
    }
}
```

Problem

76	**13**	**97**	**90**	**48**	**76**	**94**	**59**	**42**	**52**

647

Fork/Join Framework

`RecursiveTask<V>`. Sum the elements of an array

```
public class RecursiveTaskSum extends RecursiveTask<Double> {
    private static final long serialVersionUID = 1L;

    double[] data; //Array with numbers (data)
    int start, end; //Deterines what part of data to process.
    int threshold = 10000; //Arbitrary set

    RecursiveTaskSum(double[] data, int start, int end) {
        this.data = data;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Double compute() {
        double sum = 0;
        if((end - start) < threshold) {
          for(int i = start; i < end; i++) {
             sum += data[i];
          }
        }
        else { //Continue to break the data into smaller components
          int middle = (start + end) / 2; //Find the midpoint
          //Invoke new subtasks
          RecursiveTaskSum subTaskA = new RecursiveTaskSum(data, start, middle);
          RecursiveTaskSum subTaskB = new RecursiveTaskSum(data, middle, end);
          subTaskA.fork(); //Start each subtask by forking
          //Wait for the subtasks to return, and aggregate the results
          sum = subTaskB.compute() + subTaskA.join();
          //sum = subTaskA.invoke() + subTaskB.invoke();
        }
        return sum;
    }
}
```

Fork/Join Framework

RecursiveTask<V>. Sum the elements of an array

```
public class RecursiveTaskSumTest {
    public static void main(String[] args) {
        ForkJoinPool pool = new ForkJoinPool(); //Task pool with the level
        double[] data = new double[9999999]; //Numbers to work with

        //Initialize nums with values that alternate between positive and negative.
        for(int i=0; i < data.length; i++)
          data[i] = (double)(((i%2) == 0) ? i : -i) ;


        RecursiveTaskSum task =
                new RecursiveTaskSum(data, 0, data.length);

        long t1 = System.currentTimeMillis(); //to measure the time
        double result = pool.invoke(task); //Start the main ForkJoinTask
        long t2 = System.currentTimeMillis();

        System.out.println("Elapsed time: " + (t2-t1) + " ms");
        System.out.println("Result: " + result);
    }
}
```

Alternatives for obtaining task results

```java
Temp subTaskA = new Temp(data, start, middle);
Temp subTaskB = new Temp(data, middle, end);
subTaskA.fork();
subTaskB.fork();
sum = subTaskA.join() + subTaskB.join();
```

```java
Temp subTaskA = new Temp(data, start, middle);
Temp subTaskB = new Temp(data, middle, end);
sum = subTaskA.invoke() + subTaskB.invoke();
```

```java
Temp subTaskA = new Temp(data, start, middle);
Temp subTaskB = new Temp(data, middle, end);
subTaskA.fork();
sum = subTaskA.join() + subTaskB.compute();
```

```java
Temp subTaskA = new Temp(data, start, middle);
Temp subTaskB = new Temp(data, middle, end);
subTaskA.fork();
sum = subTaskA.join() + subTaskB.invoke();
```

```java
Temp subTaskA = new Temp(data, start, middle);
Temp subTaskB = new Temp(data, middle, end);
subTaskA.fork();
subTaskB.fork();
sum = subTaskB.join() + subTaskA.join();
```

```java
Temp subTaskA = new Temp(data, start, middle);
Temp subTaskB = new Temp(data, middle, end);
subTaskA.fork();
sum = subTaskB.compute() + subTaskA.join();
```

```java
Temp subTaskA = new Temp(data, start, middle);
Temp subTaskB = new Temp(data, middle, end);
subTaskA.fork();
sum = subTaskB.invoke() + subTaskA.join();
```

Other methods of `ForkJoinTask`

- To cancel a task
  - `boolean cancel(boolean interruptOK)`
- To know whether a task has been canceled
  - `boolean isCancelled()`
- To determine the state of completeness of a task
  - `final boolean isCompletedNormally()`
  - `final boolean isCompletedAbnormally()`
  - `final boolean isCompleted()`

Problem

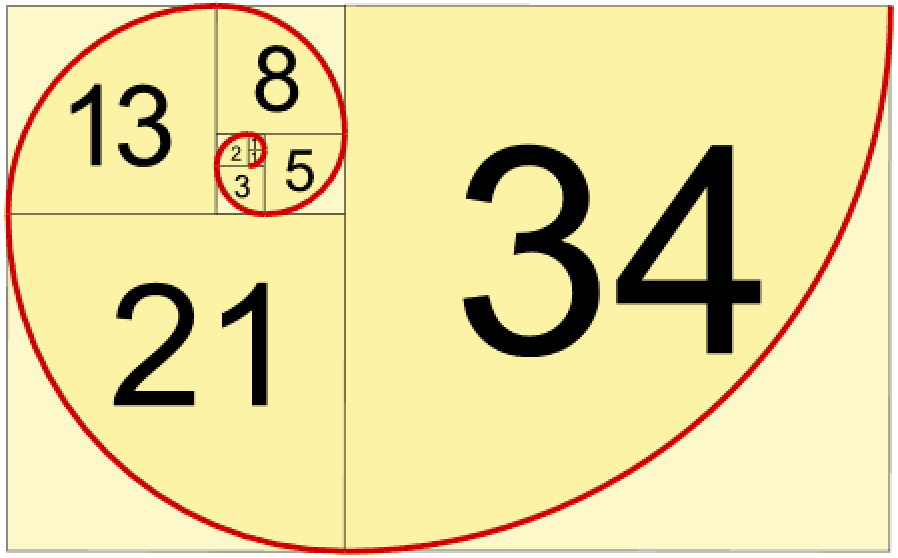


Image: http://www.mathsisfun.com/numbers/fibonacci-sequence.html

Classic Fibonacci Algorithm

```
public class FibonacciAlgorithm {
    public int n;

    public FibonacciAlgorithm(int n){
        this.n = n;
    }

    public long solve(){
        return fibonacci(this.n);
    }

    private long fibonacci(int n) {
        if (n <= 1)
            return n;
        else
            return fibonacci(n-1) + fibonacci(n-2);
    }

}
```

```
public class FibonacciAlgorithmTest {

    public static void main(String[] args) throws Exception {
        int n = 10;
        FibonacciAlgorithm problem = new FibonacciAlgorithm(n);

        long t1 = System.currentTimeMillis();
        long result = problem.solve();
        long t2 = System.currentTimeMillis();

        System.out.println("Fibonacci problem: " + n);
        System.out.println("Result: " + result);
        System.out.println("Elapsed time: " + (t2-t1) + " ms");
    }
}
```

Algorithm with `RecursiveTask<V>`

```java
public class FibonacciTaskTest {
    public static void main(String[] args) throws Exception {
        int n = 10;
        FibonacciAlgorithm problem = new FibonacciAlgorithm(n);
        FibonacciTask task = new FibonacciTask(problem);
        ForkJoinPool pool = new ForkJoinPool();

        long t1 = System.currentTimeMillis();
        long result = pool.invoke(task);
        long t2 = System.currentTimeMillis();

        System.out.println("Fibonacci problem: " + n);
        System.out.println("Result: " + result);
        System.out.println("Elapsed time: " + (t2-t1) + " ms");
    }
}
```

```java
public class FibonacciTask extends RecursiveTask<Long> {
    private static final long serialVersionUID = 1L;
    private static final int THRESHOLD = 9;
    private FibonacciAlgorithm problem;
    public long result;

    public FibonacciTask(FibonacciAlgorithm problem) {
        this.problem = problem;
    }

    @Override
    public Long compute() {
        if (problem.n < THRESHOLD) {
            result = problem.solve();
        }
        else {
            FibonacciTask subTask1 = new
                    FibonacciTask(new FibonacciAlgorithm(problem.n-1));
            FibonacciTask subTask2 = new
                    FibonacciTask(new FibonacciAlgorithm(problem.n-2));
            subTask1.fork();
            result = subTask2.compute() + subTask1.join();
        }
        return result;
    }
}
```

Problem

- File processing tasks are favorable to be performed in parallel
- The number of files and the information contained in them can be very high
- For example:
  - Version management systems
  - Systems that process data
  - Searchers
  - …

Fork/Join Framework

Processing files concurrently

```
class FileProcessingTask extends RecursiveAction {
    private static final long serialVersionUID = 1L;
    private static final int THRESHOLD = 5;
    List<File> javaFiles = null;
    String dirPath;

    public FileProcessingTask(String dirPath, List<File> javaFiles) {
        this.dirPath = dirPath;
        this.javaFiles = javaFiles;
    }

    @Override
    protected void compute() {
        if (javaFiles == null) { //First time to start processing files
            javaFiles = new ArrayList<File>();
            File sourceDir = new File(dirPath);
            if (sourceDir.isDirectory()) {
                for (File file : sourceDir.listFiles()){
                    javaFiles.add(file);
                }
            }
        }
        if (javaFiles.size() <= THRESHOLD) {
            processFiles(javaFiles);
        }
        else {
            int center = javaFiles.size() / 2;
            List<File> part1 = javaFiles.subList(0, center);
            List<File> part2 = javaFiles.subList(center, javaFiles.size());
            invokeAll(new FileProcessingTask(dirPath, part1),
                    new FileProcessingTask(dirPath, part2));
        }
    }
```

```
    protected void processFiles(List<File> filesToProcess) {
        for (File file : filesToProcess){
            System.out.println(Thread.currentThread().getName()
                    + " " + file.getName());
        }
    }
}
```

Fork/Join Framework

Processing files concurrently

```
public class FileProcessingTasksTest {
    public static void main(String[] args) {
        FileProcessingTask problem = new FileProcessingTask("D:\\test\\source", null);
        ForkJoinPool pool = new ForkJoinPool();

        long t1 = System.currentTimeMillis();
        pool.invoke(problem);
        long t2 = System.currentTimeMillis();

        System.out.println("Elapsed time: " + (t2-t1) + " ms");;
    }
}
```

Problem

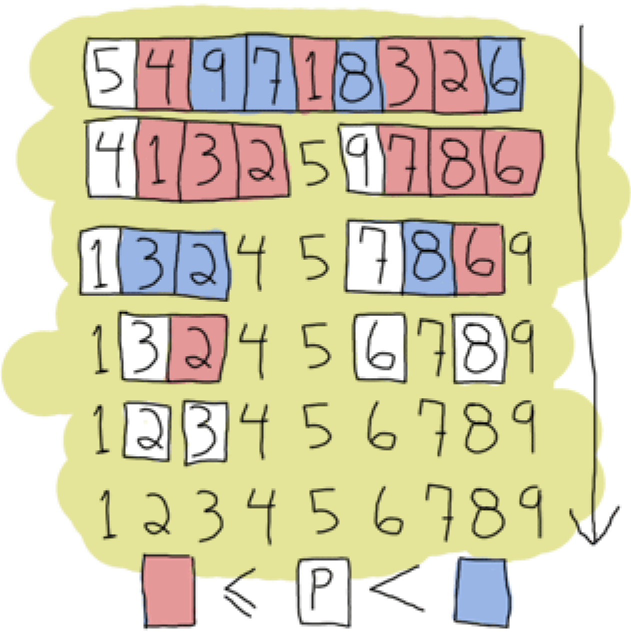
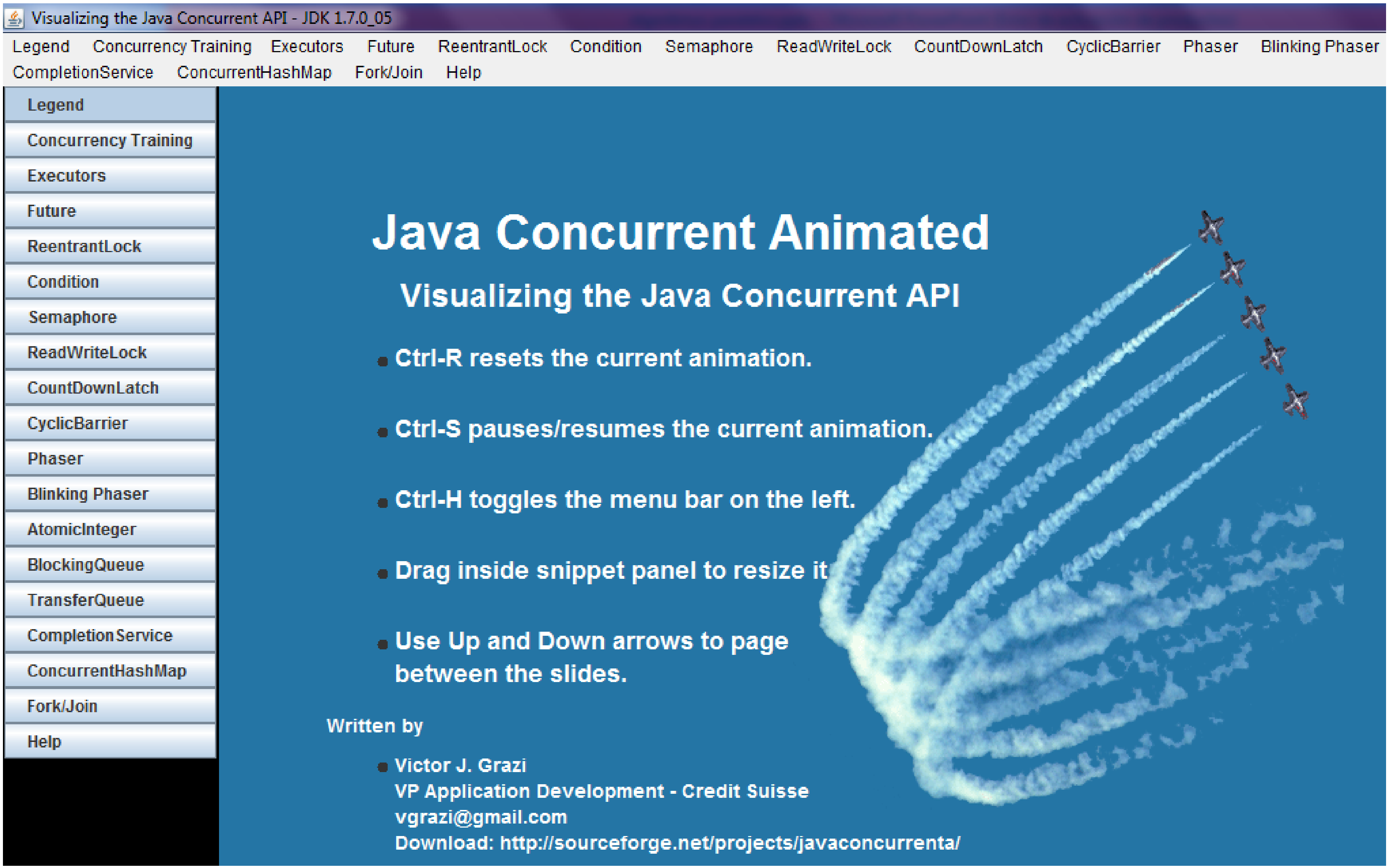


Image: *http://franzejr.wordpress.com/2011/08/28/a-quick-analysis-about-quicksort*

Fork/Join Framework



Bibliography

VICTOR J. GRAZI ; (2002) ***Java Concurrent Animated***. Application http://sourceforge.net/projects/javaconcurrenta/