Parallel Programming Practice

Fork-Join Framework

Susanne Cech Previtali
Thomas Gross

Last update: 2009-11-05, 09:39
Today

Nested classes in Java
Parallel decomposition
Fork-Join framework
Nested classes in Java
Overview

Nested class

- A class defined within another class

Usage

- Logical grouping of classes
  - If a class is useful to only one other class
  - “Helper” classes
- Increased encapsulation
  - Classes A, B: B must access private members of A
- More readable, maintainable code
  - Code placed closer to where it is used
Categorization

Set-oriented view of classes in Java

- Top-level classes
- Nested classes
  - Static nested classes
  - Inner classes
  - Local classes
  - Anonymous classes
Categorization

Top-level classes

Nested classes

Static nested classes

Inner classes
Local classes
Anonymous classes
Static nested classes

Member of the outer class
Packaging convenience
  ▶ Behavior like a top-level class

```java
public class Outer {
  private String name;

  static class StaticNested {
    private int count;
    public void set(Outer o) {
      count = o.name.length();
    }
  }
}
```
Static nested classes

Member of the outer class
Packaging convenience
  ▸ Behavior like a top-level class

```java
public class Outer {
  private String name;
  static class StaticNested {
    private int count;
    public void set(Outer o) {
      count = o.name.length();
    }
  }
}
```

```java
Outer o = new Outer();
Outer.StaticNested s = new Outer.StaticNested();
s.set(o);
```
Categorization

Top-level classes

Nested classes

Static nested classes

Inner classes

Local classes

Anonymous classes
Inner classes

Member of the outer class

- Cannot define static members
- Object exists *within* instance of outer class ⇒ like instance members

```java
public class Outer {
  private String name;
  class Inner {
    private int count;
    public void set() {
      count = name.length();
    }
  }
}
```

```java
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.set();
```
Inner classes

Member of the outer class

- Cannot define static members
- Object exists *within* instance of outer class ⇒ like instance members

```java
public class Outer {
  private String name;
  class Inner {
    private int count;
    public void set() {
      count = name.length();
    }
  }
}
```

```java
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.set();
```

access to members of the outer class
public class DataStructure {
    private int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    public void printEven() {
        MyIterator it = this.new MyIterator();
        while (it.hasNext()) { System.out.print(it.getNext() + " "); }
    }
}

public class DataStructure {
    private int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    public void printEven() {
        MyIterator it = this.new MyIterator();
        while (it.hasNext()) { System.out.print(it.getNext() + " "); }
    }
    private class MyIterator {
        private int next = 0;
        public boolean hasNext() {
            return (next <= array.length - 1);
        }
        public int getNext() {
            int retValue = array[next];
            next += 2;
            return retValue;
        }
    }
}
public class DataStructure {
    private int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    public void printEven() {
        MyIterator it = this.new MyIterator();
        while (it.hasNext()) { System.out.print(it.getNext() + " "); }
    }
    private class MyIterator {
        private int next = 0;
        public boolean hasNext() {
            return (next <= array.length - 1);
        }
        public int getNext() {
            int retValue = array[next];
            next += 2;
            return retValue;
        }
    }
}
public class DataStructure {
    private int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    public void printEven() {
        MyIterator it = this.new MyIterator();
        while (it.hasNext()) {
            System.out.print(it.getNext() + " ");
        }
    }

    private class MyIterator {
        private int next = 0;
        public boolean hasNext() {
            return (next <= array.length - 1);
        }
        public int getNext() {
            int retValue = array[next];
            next += 2;
            return retValue;
        }
    }
    private static final int MAX = 10;
}
Categorization

Top-level classes

Nested classes

Static nested classes

Inner classes

Local classes

Anonymous classes
Local classes

Named class

- Implicitly final

Scope local to a block

```java
public class MethodLocal1 {
    public void execute() {
        class MyRunnable implements Runnable {
            public void run() {
                System.out.println("Working a lot....");
            }
        }
        new Thread(new MyRunnable()).start();
    }
}
```
Categorization

Top-level classes

Nested classes

Static nested classes

Inner classes

Local classes

Anonymous classes
Anonymous classes

Unnamed
Local to a method or a field

```java
public class FieldLocal {
    private Runnable r = new Runnable() {
        public void run() {
            System.out.println("Working a lot....");
        }
    };
    public void execute() {
        new Thread(this.r).start();
    }
}
```
Anonymous classes

Unnamed

Local to a method or a field

```java
public class MethodLocal2 {
    public void execute() {
        Runnable r = new Runnable() {
            public void run() {
                System.out.println("Working a lot....");
            }
        };
        new Thread(r).start();
    }
}
```
Anonymous classes

Unnamed
Local to a method or a field

```java
public class MethodLocal3 {
    public void execute() {
        new Thread(new Runnable() {
            public void run() {
                System.out.println("Working a lot....");
            }
        }).start();
    }
}
```
Categorization

- Top-level classes
- Nested classes
  - Static nested classes
  - Inner classes
  - Local classes
  - Anonymous classes
Categorization

Top-level classes

Nested classes

Static nested classes

- Inner classes
- Local classes
- Anonymous classes

on its own

relative to an object
Categorization

Top-level classes

Nested classes

Static nested classes

Inner classes

Local classes

Anonymous classes

on its own

relative to an object
Categorization

- Top-level classes
- Nested classes
  - Static nested classes
    - Inner classes
    - Local classes
    - Anonymous classes
  - on its own
  - relative to an object
  - name
  - local to method or field
Compiling nested classes

// Static nested classes and inner classes
$ javac OuterClass.java
OuterClass$StaticNestedClass.class OuterClass.class

// Local classes
$ javac MethodLocal1.java
MethodLocal1$1MyRunnable.class MethodLocal1.class

// Anonymous classes
$ javac MethodLocal3.java
MethodLocal3$1.class MethodLocal3.class
Disassembling

class MethodLocal$1 extends java.lang.Object implements java.lang.Runnable{
final MethodLocal this$0;

MethodLocal$1(MethodLocal);
  Code:
  0:  aload_0
  1:  aload_1
  2:  putfield #1; //Field this$0:LMethodLocal;
  5:  aload_0
  6:  invokespecial #2; //Method java/lang/Object."<init>":(V
  9:  return

public void run();
  Code:
  0:  getstatic #3; //Field java/lang/System.out:Ljava/io/PrintStream;
  3:  ldc #4; //String Working ....
      // ...
}
Local variables of enclosing method

Local classes have access to local variables of method

- Local variables must be declared final

```java
public class MethodLocal4 {
    public void execute() {
        final int i = 3;
        final Integer x = new Integer(42);
        Runnable r = new Runnable() {
            public void run() {
                System.out.println("Working ...." + i + " " + x);
            }
        }; 
        new Thread(r).start();
    }
}
```
Disassembling for local variables

Compiled from "MethodLocal4.java"
class MethodLocal4$1 extends java.lang.Object implements java.lang.Runnable{
final java.lang.Integer val$x;
final MethodLocal4 this$0;
MethodLocal4$1(MethodLocal4, java.lang.Integer);
    Code:
      0:  aload_0
      1:  aload_1
      2:  putfield   #1; //Field this$0:LMethodLocal4;
      5:  aload_0
      6:  aload_2
      7:  putfield   #2; //Field val$x:Ljava/lang/Integer;
     10:  aload_0
     11:  invokespecial  #3; //Method java/lang/Object."<init>":()V
     14:  return

public void run();
    Code: // removed...
Disassembly for local variables

```java
// continued

public void run();

Code:

0:  getstatic  #4; //Field java/lang/System.out:Ljava/io/PrintStream;
3:  new #5;  //class java/lang/StringBuilder
6:   dup
7:  invokespecial  #6; //Method java/lang/StringBuilder."<init>"():V
10: ldc #7;  //String Working ...3
12: invokevirtual  #8; //Method java/lang/StringBuilder.append:(.....
15: aload_0
16: getfield  #2; //Field val$x:Ljava/lang/Integer;
19: invokevirtual  #9; //Method java/lang/StringBuilder.append:(.....
22: invokevirtual  #10; //Method java/lang/StringBuilder.toString():....
25: invokevirtual  #11; //Method java/io/PrintStream.println:(....
28: return
```

Parallel decomposition
Problem decomposition

Goal: Map a problem to multiple threads

Two principle approaches

- Task partitioning
  - Focus on computation
- Data partitioning
  - Focus on data
Ways to exploit parallelism

Task decomposition

- Each thread works on a subset of the tasks

Data decomposition

- Each thread works on a subset of the data
- Single program, multiple data
Task decomposition

Task parallelism

1. Divide tasks among processors
2. Decide which data elements are going to be accessed (read and/or written) by which processors

Example
- Event handler for GUI
Example: Functional decomposition
Example: Functional decomposition
Domain decomposition

Data parallelism

1. Divide data elements among processors
2. Assign tasks to each processor
Example: Data decomposition

Search for maximum in array
Example: Data decomposition

Search for maximum in array
Example: Data decomposition

Search for maximum in array

![Diagram showing data decomposition across CPUs]
Example: Data decomposition

Search for maximum in array

<table>
<thead>
<tr>
<th>CPU 0</th>
<th>CPU 1</th>
<th>CPU 2</th>
<th>CPU 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU 3 contains the maximum value.
Task versus data partitioning

Java 1.0-1.4: Thread objects
Java 5: Course-grained parallelism
Java 7: Fine-grained parallelism
Java 1.0-1.4: Thread objects

Hardware
- No (or limited) parallel hardware

Constructs
- Thread, synchronized, volatile
- Broken memory model

Programming
- Asynchronous tasks
- Error-prone
Java 5: Concurrent components

Hardware
  ‣ Multi-cores

Components
  ‣ Executor framework
  ‣ Course-grained concurrency
  ‣ Task decomposition
    ‣ Asynchronous tasks
      ‣ \#task \approx \#cores

Discussion
  ‣ Does not scale to many-cores
JSR-166: Fine-grained parallelism

Hardware
  ‣ Multi- and manycores

Components
  ‣ Fork-join framework
  ‣ Divide and conquer algorithms
  ‣ Task and data decomposition
Fork-join framework
Introduction

Remember divide-and-conquer algorithm design?

Examples

- Sorting and searching
- Data structures
- Matrix algorithms
- Image processing

Parallelize easily if recursive tasks are independent, either

- Operate on different data sets
- Solve different subproblems
- No communication needed
Fork-join decompositions

Parallel version of divide-and-conquer
Generic divide-and-conquer algorithm

// Pseudo code
Result solve(Problem problem) {
    if (problem.size < SEQ_THRESHOLD) {
        return solveSequentially(problem);
    } else {
        Result left, right;
        INVOKE_IN_PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
Generic divide-and-conquer algorithm

// Pseudo code
Result solve(Problem problem) {
    if (problem.size < SEQ_THRESHOLD) {
        return solveSequentially(problem);
    } else {
        Result left, right;
        INVOKE_IN_PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}

sequential solution would be faster

divide in subproblems
solve recursively
Generic divide-and-conquer algorithm

// Pseudo code
Result solve(Problem problem) {
    if (problem.size < SEQ_THRESHOLD) {
        return solveSequentially(problem);
    } else {
        Result left, right;
        INVOKE_IN_PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}

cost function: coordinate parallel tasks
sequential solution would be faster
divide in subproblems solve recursively
Task activity diagram

fork: start subtasks
join: wait for completion
Task granularity and structure

Maximizing parallelism
Minimizing overhead
Minimizing contention
Maximizing locality

Note
  ▶ Every solution is a compromise!
Task granularity and structure

Maximizing parallelism

- The smaller the tasks, the more opportunity for parallelism
- Using more fine-grained tasks
  - Keeps more CPUs busy
  - Improves load balancing, locality, scalability
  - Decreases time that CPUs must wait for one another

Minimizing overhead

Minimizing contention

Maximizing locality
Task granularity and structure

Maximizing parallelism
Minimizing overhead
  ▸ Task and thread creation versus sequential objects
  ▸ Memory consumption, garbage collection
Minimizing contention
Maximizing locality
Task granularity and structure

Maximizing parallelism
Minimizing overhead
Minimizing contention
  ▸ Not much speed-up if
    ▸ Frequent communication
    ▸ Block waiting for other threads/resources
  ▸ Minimize shared resources, global variables, locks
Maximizing locality
Task granularity and structure

Maximizing parallelism
Minimizing overhead
Minimizing contention
Maximizing locality
  ‣ Memory access patterns \(\rightarrow\) caches

Note again:
  ‣ Every solution is a compromise
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size;
    public SelectMaxProblem(int[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
        this.size = end - start;
    }
    // to be continued...
}

// to be continued...
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size;

    public SelectMaxProblem(int[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
        this.size = end - start;
    }

    // to be continued...
}

parameters of the problem established in constructor
Select max element from array

```java
public class SelectMaxProblem {
    private final int[] numbers;
    private final int start;
    private final int end;
    public final int size;

    public SelectMaxProblem(int[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
        this.size = end - start;
    }

    // to be continued...
}
```

Parameters of the problem established in constructor.

Constructor

Copy array references disjoint data subsets, read-only.
public class SelectMaxProblem {
    // continued

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<end; i++) {
            int n = numbers[i];
            if (n > max) max = n;
        }
        return max;
    }

    public SelectMaxProblem divide(int subStart, int subEnd) {
        return new SelectMaxProblem(numbers,
                                      start + subStart,
                                      start + subEnd);
    }
}
public class SelectMaxProblem {
    // continued

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<end; i++) {
            int n = numbers[i];
            if (n > max) max = n;
        }
        return max;
    }

    public SelectMaxProblem divide(int subStart, int subEnd) {
        return new SelectMaxProblem(numbers,
                                      start + subStart,
                                      start + subEnd);
    }
}
Select max element from array

```java
public class SelectMaxProblem {
    // continued

    public int solveSequentially() {
        int max = Integer.MIN_VALUE;
        for (int i=start; i<end; i++) {
            int n = numbers[i];
            if (n > max) max = n;
        }
        return max;
    }

    public SelectMaxProblem divide(int subStart, int subEnd) {
        return new SelectMaxProblem(numbers,
                                     start + subStart,
                                     start + subEnd);
    }
}
```
Example FJ framework

```java
public class MaxWithFJ extends RecursiveAction {
  private final int threshold;
  private final SelectMaxProblem p; // problem
  public int result;
  public MaxWithFJ(SelectMaxProblem problem, int threshold) { .. }
  protected void compute() {
    if (p.size < threshold)
      result = p.solveSequentially();
    else {
      int mid = p.size / 2;
      MaxWithFJ left = new MaxWithFJ(p.divide(0, mid), threshold);
      MaxWithFJ right = new MaxWithFJ(p.divide(mid, p.size), threshold);
      invokeAll(left, right);
      result = Math.max(left.result, right.result);
    }
  }
}
```
Solving the problem

```java
public static void main(String[] args) {
    int size = 500000;
    int[] array = new int[size];
    for (int i = 0; i < size; i++) {
        array[i] = i;
    }
    SelectMaxProblem problem = new SelectMaxProblem(array, 0, size);
    int threshold = 100;
    int nThreads = 4;
    MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
    ForkJoinPool fjPool = new ForkJoinPool(nThreads);
    fjPool.invoke(mfj);
    int result = mfj.result;
}
```
Solving the problem

```java
public static void main(String[] args) {
    int size = 500000;
    int[] array = new int[size];
    for (int i = 0; i < size; i++) {
        array[i] = i;
    }
    SelectMaxProblem problem = new SelectMaxProblem(array, 0, size);
    int threshold = 100;
    int nThreads = 4;
    MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
    ForkJoinPool fjPool = new ForkJoinPool(nThreads);
    fjPool.invoke(mfj);
    int result = mfj.result;
}
```

implements Executor, ExecutionService optimized for fork-join tasks
Solving the problem

```java
public static void main(String[] args) {
    int size = 500000;
    int[] array = new int[size];
    for (int i = 0; i < size; i++) {
        array[i] = i;
    }
    SelectMaxProblem problem = new SelectMaxProblem(array, 0, size);
    int threshold = 100; // avoid ridiculously low and high
    int nThreads = 4; // use Runtime.availableProcessors();
    MaxWithFJ mfj = new MaxWithFJ(problem, threshold);
    ForkJoinPool fjPool = new ForkJoinPool(nThreads); // ForkJoinPool()
    fjPool.invoke(mfj);
    int result = mfj.result;
}
```

implements Executor, ExecutionService optimized for fork-join tasks
Execution sequence (threshold=2)

10, 3, 7, 99, 2, 12, 1, 15, 42, 4
Execution sequence (threshold=2)
Execution sequence (threshold=2)
Execution sequence (threshold=2)
Execution sequence (threshold=2)
Execution sequence (threshold=2)
Execution sequence (threshold=2)
Execution sequence (threshold=2)

```
10 3 7 99 2 12 1 15 42 4
10 3 7 99 2
12 1 15 42 4
10 3
7 99 2
12 1
15 42 4
7 99 2
10
12
15 42 4
15 42 4
99
99
99
42
42
```
Execution sequence (threshold=2)
Implementation considerations of FJ

Thread objects

- Fork operation: `Thread.start()`
- Join operation: `Thread.join()`
- Expensive thread creation, number of threads

Executor thread pools

- Tasks wait for other tasks to complete ⇒ high contention
- Designed for independent, maybe blocking, coarse-grained tasks

Ideal solution minimizes

- Context switch overhead between worker threads
- Contention for task queue ⇒ avoids a common task queue
Thread scheduling

deque

head
tail

push task
pop task

thread pool
Thread scheduling

ddeque

head

tail

thread pool

pop task

push task

pop task
Thread scheduling

thread pool

deque

head

tail

push task

pop task
Thread scheduling

deque

head

tail

thread pool

push task

push task

pop task
Thread scheduling

deque

thread pool

tail

head

push task

pop task
Thread scheduling

deque

thread pool

tail

head

push task

pop task
Thread scheduling
Thread scheduling

deque

head

tail

thread pool

push task

pop task
Thread scheduling

deque

thread pool

tail

head

push task

pop task
Thread scheduling

deque

thread pool

head

tail

push task

pop task
Thread scheduling

deque

thread pool

tail

head

push task

pop task
Thread scheduling

deque

thread pool

tail

head

take task

push task

pop task
Thread scheduling

Double ended queues ("deck")
- LIFO for owner
- FIFO for other

Work stealing
- When no local tasks to run
- Steal task from other thread

On join operation
- Process other tasks
Advantages of work-stealing

Reduces contention

- Steal from opposite side of the deque as owner

Fits divide-and-conquer

- Generate large tasks early
- Older stolen task $\Rightarrow$ large work unit $\Rightarrow$ work decomposition
- $\#\text{pop} > \#\text{take}$
Advantages of work-stealing

Reduces contention

- Steal from opposite side of the deque as owner

Fits divide-and-conquer

- Generate large tasks early
- Older stolen task $\Rightarrow$ large work unit $\Rightarrow$ work decomposition
- #pop > #take
ForkJoinTask<V>

Lightweight form of Future<V> because of restrictions
Intended use as computational tasks
  ▸ Calculating pure functions (no side effects)
  ▸ Operating on purely isolated objects
Restrictions
  ▸ Avoid synchronized methods and blocks
  ▸ Minimize other blocking (except join)
    ▸ No blocking IO
    ▸ Access independent variables
Begin execution when submitted to a ForkJoinPool
  ▸ Once started, it will start other subtasks
ForkJoinTask\(<V>\): Coordination mechanisms

fork()

- Arrange for asynchronous execution

Variants

- invoke()
  - Semantically: fork(); join();
  - Attempts to begin execution in current thread

- invokeAll()
  - Most common form: Fork a set of tasks and join them all

join()

- Do not proceed until the task’s result has been computed

Variants

- Future.get()
  - Interruptible/timed waits

- helpJoin()
  - Actively execute other threads while waiting for completion
ForkJoinTask<V>: Queries

Execution status of tasks

- `isDone()`
- `isCompletedNormally()`
- `isCancelled()`
ForkJoinTask<V>: Usage

Use one of the subclasses
- RecursiveAction<V> = resultless
- RecursiveTask<V> = result-bearing

Declare fields
- Comprise parameters
- Established in constructor

Override compute()
- Use control/coordination methods
ForkJoinPool\(<V>\): Overview

Extends

- AbstractExecutorService

Implements

- Executor, ExecutorService

Difference to other ExecutorServices

- Employs work-stealing
ForkJoinPool\(<V>\): Queries

Status checking to help in tuning and debugging

- `getStealCount()`
  - Estimated number of stolen tasks
- `getActiveThreadCount()`
  - Estimated number of thread currently stealing or executing
- `getQueuedSubmissionCount()`
  - Estimated number of tasks submitted but not yet executed
- `getRunningThreadCount()`
  - Estimated number of threads that are not blocked
Howto use JSR166

Goto

› http://gee.cs.oswego.edu/dll/concurrency-interest/index.html

Use

› JSR166 maintenance updates

Compile with jar file included in classpath

› export CLASSPATH=$CLASSPATH:<path to jar file>/jsr166.jar

Execute

› java -Xbootclasspath/p:<path to jar file>/jsr166.jar Main