Comparing & Contrasting All the Java Fork-Join Framework Programming Models

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Evaluate different fork-join framework programming models in practice
• Evaluate the applyAllIter() method
• Evaluate the applyAllSplit() method
• Evaluate the applyAllSplitIndex() method
• Compare & contrast all the programming models for the Java Fork-Join framework
Evaluating the
eXample Applications
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand

```java
<T> List<T> applyAllIter(
    List<T> list,
    Function<T, T> op,
    ForkJoinPool fjPool) {
...
    for (T t : list)
        forks.add(
            (new RecursiveTask<T>() {
                protected T compute() {
                    return op.apply(t);
                }
            }).fork());

    for (ForkJoinTask<T> task : forks)
        results.add(task.join());
...
```
Evaluating the Example Applications

• Each Java fork-join programming model has pros & cons, e.g.
  • Iterative fork()/join() is simple to program/understand
  • but it incurs more work-stealing

Tests were conducted on a 2.6 GHz 6-core Lenovo laptop with 64 MBs RAM

<table>
<thead>
<tr>
<th>Method</th>
<th>Steal Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>applyAllIter()</td>
<td>31</td>
</tr>
<tr>
<td>applyAllSplitIndex()</td>
<td>16</td>
</tr>
<tr>
<td>applyAllSplit()</td>
<td>21</td>
</tr>
<tr>
<td>applyAllSplitIndexEx()</td>
<td>21</td>
</tr>
<tr>
<td>testApplyAllSplitIndexEx()</td>
<td>4575 ms</td>
</tr>
<tr>
<td>testApplyAllSplitIndex()</td>
<td>5145 ms</td>
</tr>
<tr>
<td>testApplyAllSplit()</td>
<td>5172 ms</td>
</tr>
<tr>
<td>testApplyAllIter()</td>
<td>5599 ms</td>
</tr>
</tbody>
</table>

[1] Printing 4 results from fastest to slowest
Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21

[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest

Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
    - but it incurs more work-stealing
    - which lowers performance
Evaluating the Example Applications

• Each Java fork-join programming model has pros & cons, e.g.
  • Iterative fork()/join() is simple to program/understand
  • Recursive decomposition incurs fewer “steals”

Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21

[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms

[1] Finishing ForkJoinTest
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
  - Recursive decomposition incurs fewer “steals”
    - which improves performance

Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest

There are also other factors (e.g., less data copying) that improve performance
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
  - Recursive decomposition incurs fewer “steals”
    - which improves performance
  - but is more complicated to program

```java
class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
        ...
        int mid = mList.size() / 2;
        ForkJoinTask<List<T>> lt =
            new SplitterTask(mList.subList(0, mid)).fork();
        mList = mList
            .subList(mid, mList.size());
        List<T> rightResult = compute();
        List<T> leftResult = lt.join();
        leftResult.addAll(rightResult);
        return leftResult;
    }
} ...
Evaluating the Example Applications

• Each Java fork-join programming model has pros & cons, e.g.
  • Iterative fork()/join() is simple to program/understand
  • Recursive decomposition incurs fewer “steals”
    • which improves performance
    • but is more complicated to program
  • & also does more “work” wrt method calls, etc.
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
  - Recursive decomposition incurs fewer “steals”
  - RecursiveAction’s overhead is lower than RecursiveTask’s

Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21

[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms

[1] Finishing ForkJoinTest
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
  - Recursive decomposition incurs fewer “steals”
  - RecursiveAction’s overhead is lower than RecursiveTask’s
  - But RecursiveAction is also more idiosyncratic

```java
<T> List<T> applyAllSplitIndex(
    List<T> list,
    Function<T, T> op,
    ForkJoinPool fjPool) {
    T[] results = (T[]) Array.newInstance(
        list.get(0).getClass(),
        list.size());
    ...
```
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
  - Recursive decomposition incurs fewer “steals”
  - RecursiveAction’s overhead is lower than RecursiveTask’s
    - But RecursiveAction is also more idiosyncratic
    - Especially for generics

```java
<T> List<T> applyAllSplitIndex
(List<T> list,
 Function<T, T> op,
 ForkJoinPool fjPool) {
    T[] results = (T[]) Array.newInstance
        (list.get(0).getClass(),
         list.size());
...
Evaluating the Example Applications

- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
  - Recursive decomposition incurs fewer “steals”
  - RecursiveAction’s overhead is lower than RecursiveTask’s
    - But RecursiveAction is also more idiosyncratic
      - Especially for generics
    - Changing the API can help!

```java
<T> List<T> applyAllSplitIndexEx
  (List<T> list,
   Function<T, T> op,
   ForkJoinPool fjPool,
   T[] results) {
  ...
```

TIME
for
Change
Evaluating the Example Applications

Ironically, the most concise solution involves the use of parallel streams.

```java
<T> List<T> applyParallelStream(List<T> list, Function<T, T> op) {
    return list
        .parallelStream()
        .map(op)
        .collect(toList());
}

See earlier lessons on the “Java Parallel Streams Framework”
Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams

```
<T> List<T> applyParallelStream
    (List<T> list,
     Function<T, T> op) {
    return list
        .parallelStream()
        .map(op)
        .collect(toList());
}
```

The params & return value are similar

However, the parallel streams framework uses the common fork-join pool
Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams

```java
<T> List<T> applyParallelStream
  (List<T> list,
   Function<T, T> op) {
    return list
      .parallelStream()
      .map(op)
      .collect(toList());
  }
```
Evaluating the Example Applications

• Ironically, the most concise solution involves the use of parallel streams

```
<T> List<T> applyParallelStream(List<T> list,
    Function<T, T> op) {
    return list
        .parallelStream()
        .map(op)
        .collect(toList());
}
```
• Ironically, the most concise solution involves the use of parallel streams

```java
<T> List<T> applyParallelStream(List<T> list,
    Function<T, T> op) {
    return list
        .parallelStream()
        .map(op)
        .collect(toList());
}
```

*Convert the transformed stream back into a list & return it to the caller*
Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams

```java
<T> List<T> applyParallelStream(List<T> list, Function<T, T> op) {
    return list
        .parallelStream()
        .map(op)
        .collect(toList());
```

- applyAllIter() steal count = 31
- applyAllSplitIndex() steal count = 18
- applyAllSplit() steal count = 22
- applyAllSplitIndexEx() steal count = 22
- applyParallelStream() steal count = 22

[1] Printing 5 results from fastest to slowest

- testApplyAllSplitIndexEx() executed in 3883 msecs
- testApplyAllSplitIndex() executed in 3886 msecs
- testApplyAllSplit() executed in 3980 msecs
- testParallelStream() executed in 3987 msecs
- testApplyAllIter() executed in 5402 msecs

The parallel stream version performs relatively well & is much easier to program!
End of Comparing & Contrasting All the Java Fork-Join Framework Programming Models