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


```
<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

```
[1] Starting ForkJoinTest
applyAllIter() steal count = 101
applyAllSplitIndex() steal count = 34
applyAllSplit() steal count = 30
applyAllSplitIndexEx() steal count = 41
[1] Printing 4 results from fastest to slowest
testApplyAllSplit() executed in 9581 msecs
testApplyAllSplitIndex() executed in 9645 msecs
testApplyAllIter() executed in 10448 msecs
testApplyAllSplitIndexEx() executed in 10587 msecs
[1] Finishing ForkJoinTest
```

Tests were conducted on a 3.2 GHz 10-core MacBook Pro laptop with 64 MBs RAM

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

```
[1] Starting ForkJoinTest
applyAllIter() steal count = 101
applyAllSplitIndex() steal count = 34
applyAllSplit() steal count = 30
applyAllSplitIndexEx() steal count = 41
[1] Printing 4 results from fastest to slowest
testApplyAllSplit() executed in 9581 msecs
testApplyAllSplitIndex() executed in 9645 msecs
testApplyAllIter() executed in 10448 msecs
testApplyAllSplitIndexEx() executed in 10587 msecs
[1] Finishing ForkJoinTest
```

Evaluating the Example Applications

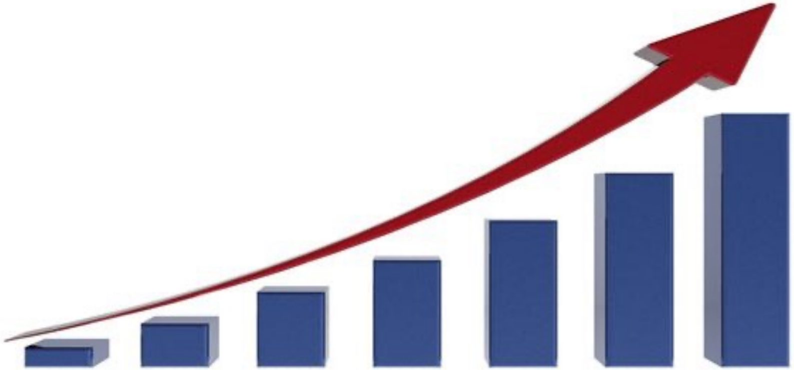
- Each Java fork-join programming model has pros & cons, e.g.
  - Iterative fork()/join() is simple to program/understand
  - Recursive decomposition incurs fewer "steals"

```
[1] Starting ForkJoinTest
applyAllIter() steal count = 101
applyAllSplitIndex() steal count = 34
applyAllSplit() steal count = 30
applyAllSplitIndexEx() steal count = 41
[1] Printing 4 results from fastest to slowest
testApplyAllSplit() executed in 9581 msecs
testApplyAllSplitIndex() executed in 9645 msecs
testApplyAllIter() executed in 10448 msecs
testApplyAllSplitIndexEx() executed in 10587 msecs
[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

```
[1] Starting ForkJoinTest
applyAllIter() steal count = 101
applyAllSplitIndex() steal count = 34
applyAllSplit() steal count = 30
applyAllSplitIndexEx() steal count = 41
[1] Printing 4 results from fastest to slowest
testApplyAllSplit() executed in 9581 msecs
testApplyAllSplitIndex() executed in 9645 msecs
testApplyAllIter() executed in 10448 msecs
testApplyAllSplitIndexEx() executed in 10587 msecs
[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

```
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 is rather idiosyncratic
    - Due to semantics of Java's generics

```
<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 is rather idiosyncratic
    - Due to semantics of Java's generics
    - Changing the API can help!

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

}

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());

}
```

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

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

Convert the list to a parallel stream

}

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());
}
```

Apply op function to each element of the stream

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());

}
```

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

```
applyAllIter() steal count = 101
applyAllSplitIndex() steal count = 34
applyAllSplit() steal count = 30
applyAllSplitIndexEx() steal count = 41
applyParallelStream() steal count = 21
[1] Printing 5 results from fastest to slowest
testApplyAllSplit() executed in 9581 msecs
testParallelStream() executed in 9624 msecs
testApplyAllSplitIndex() executed in 9645 msecs
testApplyAllIter() executed in 10448 msecs
testApplyAllSplitIndexEx() executed in 10587 msecs
[1] Finishing ForkJoinTest
```

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

The parallel stream version performs well & is also *much* easier to program!

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