The Java Fork-Join Pool: Evaluating the Example Applications

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

- Apply the fork-join framework in practice
- Examine the `applyAllIter()` method
- Examine the `applyAllSplit()` method
- Examine the `applyAllSplitIndex()` method
- Evaluate the example applications of the Fork-Join Pool 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

[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

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”

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

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
    • Especially for generics
    • Changing the API can help!

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

...
End of the Java Fork-Join Pool: Evaluating the Example Applications