

Evaluating the `applyAllSplit()` Java Fork-Join Framework Programming Model

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
 - This method uses sublists & “recursive-decomposition” to disperse tasks to worker threads

```
<T> List<T> applyAllSplit  
    (List<T> list,  
     Function<T, T> op,  
     ForkJoinPool fjPool) {  
    class SplitterTask  
    extends RecursiveTask  
        <List<T>> { ... }  
  
    return fjPool  
        .invoke(new  
            SplitterTask(list));  
}
```

Implementing the Method `applyAllSplit()`

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                           ForkJoinPool fjPool) {  
    class SplitterTask extends RecursiveTask<List<T>> { ... }
```

```
    return fjPool  
        .invoke(new SplitterTask(list));  
}
```

Implementing the Method applyAllSplit()

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) {  
    class SplitterTask extends RecursiveTask<List<T>> { ... }
```

These parameters are treated as "effectively final" variables in the anonymous inner class below

```
    return fjPool  
        .invoke(new SplitterTask(list));  
}
```

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) {  
    class SplitterTask extends RecursiveTask<List<T>> { ... }
```

*This task recursively partitions the list
& runs each half in a ForkJoinTask*

```
    return fjPool  
        .invoke(new SplitterTask(list));  
}
```



This use case is an ideal application of the fork-join framework!

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) {  
    class SplitterTask extends RecursiveTask<List<T>> { ... }
```



```
        return fjPool  
            .invoke(new SplitterTask(list));  
    }
```

Invoke a new SplitterTask in the fork-join pool & then wait for & return the results

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {  
    private List<T> mList;
```

*This implementation recursively splits
the list evenly in half & copies data*

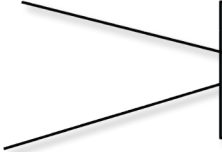
```
    private SplitterTask(List<T> list) {  
        mList = list;  
    }  
    ...  
} ...
```

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
    class SplitterTask extends RecursiveTask<List<T>> {  
        private List<T> mList;
```



*Constructor stores a reference
to a portion of the original list*

```
        private SplitterTask(List<T> list) {  
            mList = list;  
        }
```

```
        ...
```

```
    } ...
```

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {
```

```
    protected List<T> compute() {
```

```
        if (mList.size() <= 1) {
```

```
            List<T> result =  
                new ArrayList<>();
```

```
            for (T t : mList) result.add(op.apply(t));
```

```
            return result;
```

```
        } else {
```

```
            ...
```

```
        } ...
```

*Recursively perform the
computations in parallel
on a subset of the list*

Implementing the Method applyAllSplit()

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {
```

```
    protected List<T> compute() {
```

```
        if (mList.size() <= 1) {
```

```
            List<T> result =  
                new ArrayList<>();
```

*If the list has 1 or 0 elements
create an empty ArrayList*

```
            for (T t : mList) result.add(op.apply(t));
```

```
            return result;
```

```
        } else {
```

```
            ...
```

```
        } ...
```

This if statement handles the base case for the recursion

Implementing the Method applyAllSplit()

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {
```

```
    protected List<T> compute() {
```

```
        if (mList.size() <= 1) {
```

```
            List<T> result =  
                new ArrayList<>();
```

```
            for (T t : mList) result.add(op.apply(t));
```

```
            return result;
```

```
        } else {
```

```
            ...
```

```
        } ...
```

Apply op to the element in this list & add it to the result list

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...  
    class SplitterTask extends RecursiveTask<List<T>> {  
        protected List<T> compute() {  
            if (mList.size() <= 1) {  
                List<T> result =  
                    new ArrayList<>();  
  
                for (T t : mList) result.add(op.apply(t));  
  
                return result;  
            } else {  
                ...  
            } ...  
        }  
    }  
}
```

Return the result list

Implementing the Method applyAllSplit()

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {
```

```
protected List<T> compute() {
```

```
... else {
```

```
    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;
```

```
} ...
```

Determine mid-point of the list



This else statement recursively & evenly partitions the list

Implementing the Method applyAllSplit()

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {
```

```
protected List<T> compute() {
```

```
... else {
```

```
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;
```

```
} ...
```

*Create a new task to handle
left-side of the list & fork it*



This recursive decomposition disperses tasks to (other) worker threads

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
    class SplitterTask extends RecursiveTask<List<T>> {
        protected List<T> compute() {
            ... else {
                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;
            } ...
        }
    }
```



Larger partitions are pushed onto deque before smaller ones, which helps work-stealing

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {
```

```
    protected List<T> compute() {
```

```
        ... else {
```

```
            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;
```

```
        } ...
```

*Update mList to handle the
right-side & compute results*



`compute()` runs in the same task as its "parent" to minimize context switching

Implementing the Method applyAllSplit()

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {  
    protected List<T> compute() {
```

```
        ... else {
```

```
            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;
```

```
        } ...
```

Join w/left-side results



compute() must be called before join()!

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,  
                          ForkJoinPool fjPool) { ...
```

```
class SplitterTask extends RecursiveTask<List<T>> {  
    protected List<T> compute() {
```

```
        ... else {
```

```
            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;
```

```
        } ...
```

*Combine left- & right-
size & return results*



Data may be copied since these the lists are implemented via `ArrayList`

Implementing the Method `applyAllSplit()`

- Apply an 'op' to all list items by recursively splitting via fork-join method calls

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
    class SplitterTask extends RecursiveTask<List<T>> {
        protected List<T> compute() {
            ... else {
                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;
            } ...
        }
    }
```

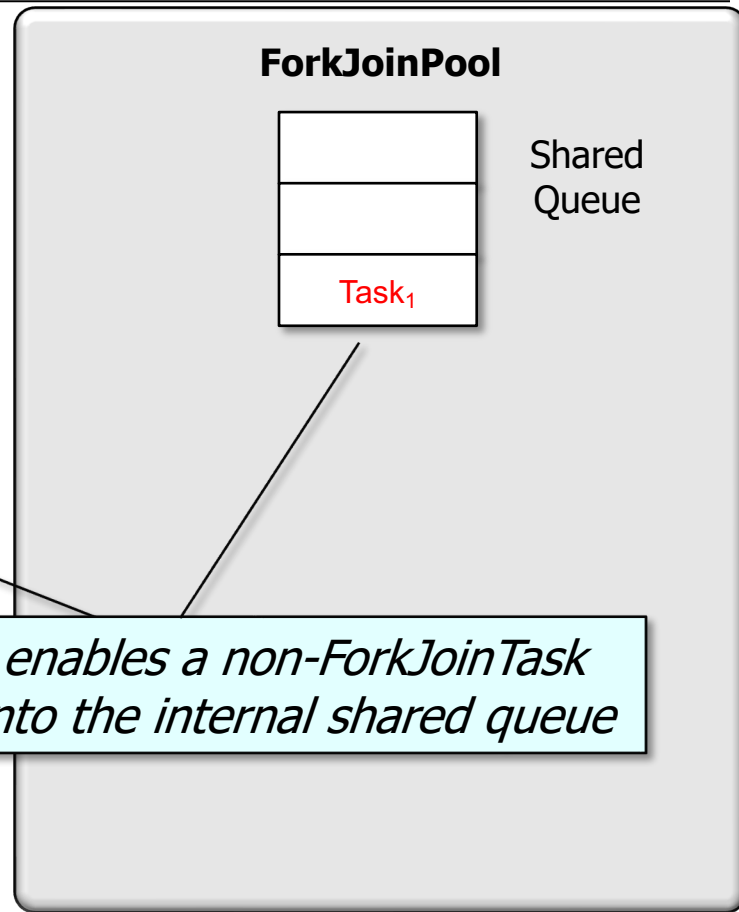
This implementation is harder to program & understand since it's recursive

Visualizing the `applyAllSplit()` Method

Visualizing the applyAllSplit() Method

- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit  
    (List<T> list,  
     Function<T, T> op,  
     ForkJoinPool fjPool) {  
    ...  
    return fjPool  
        .invoke(new SplitterTask  
                (list));  
}
```



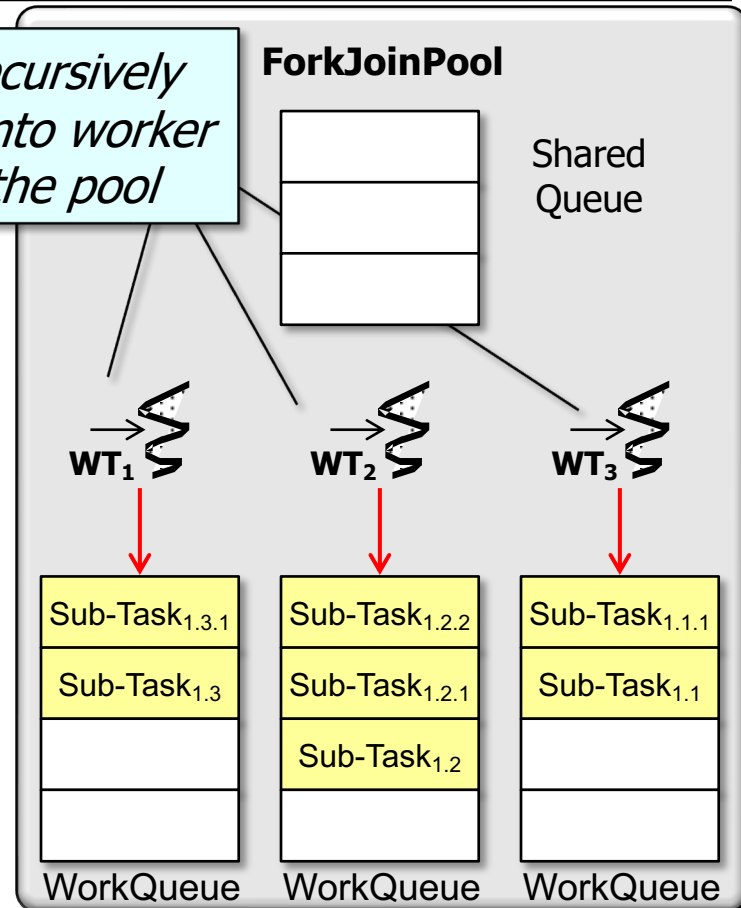
The invoke() method enables a non-ForkJoinTask client to insert a task into the internal shared queue

Visualizing the applyAllSplit() Method

- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit  
    (List<T> list,  
     Function<T, T> op,  
     ForkJoinPool fjPool) { ...  
    class SplitterTask ... {  
        protected List<T> compute() {  
            ... else {  
                ForkJoinTask<List<T>> lt =  
                    new SplitterTask  
                        (mList.subList(0, mid))  
                        .fork();  
                ...  
                List<T> rightResult =  
                    compute(); ...
```

*Sub-tasks recursively
decompose onto worker
threads in the pool*



"Work-stealing" overhead is lower, but copying & method call overhead is higher

Visualizing the applyAllSplit() Method

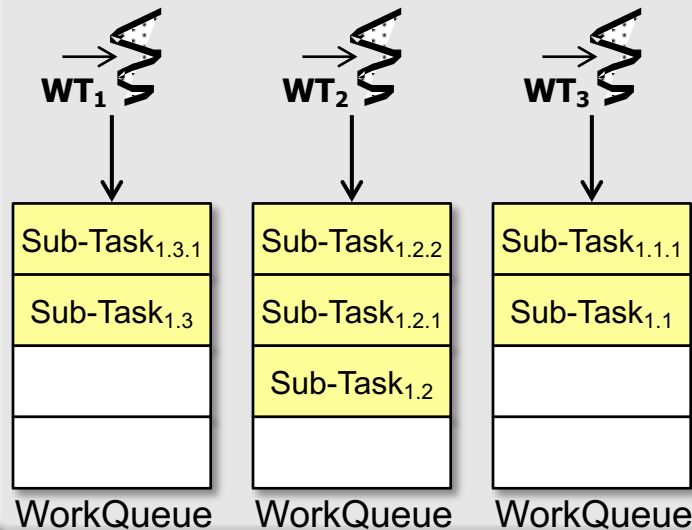
- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit  
    (List<T> list,  
     Function<T, T> op,  
     ForkJoinPool fjPool) { ...  
    class SplitterTask ... {  
        protected List<T> compute() {  
            ... else {  
                ForkJoinTask<List<T>> lt =  
                    new SplitterTask  
                        (mList.subList(0, mid))  
                        .fork();  
                ...  
                List<T> rightResult =  
                    compute(); ...  
            }  
        }  
    }
```

*The fork()'d sub-task
& compute() sub-
task run in parallel*

ForkJoinPool

Shared
Queue



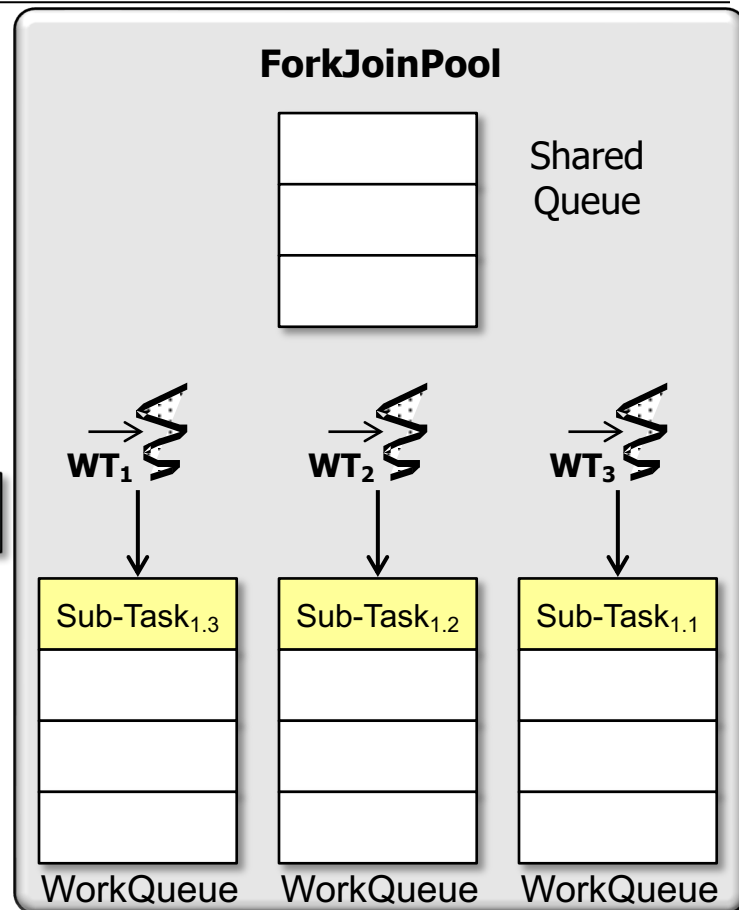
compute() runs in the same task as its "parent" to optimize performance

Visualizing the applyAllSplit() Method

- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit  
    (List<T> list,  
     Function<T, T> op,  
     ForkJoinPool fjPool) { ...  
    class SplitterTask ... {  
        protected List<T> compute() {  
            ... else {  
                ...  
                List<T> leftResult =  
                    lt.join();  
                leftResult  
                    .addAll(rightResult);  
                return leftResult;  
            }  
        }  
    }
```

join() returns a value



There is a “balanced tree” of join() calls, which scales better than applyAllIter()

End of Evaluating the apply AllSplit() Java Fork-Join Framework Programming Model