The Java Fork-Join Pool: Implementing applyAllSplitIndex()

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

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

   class SplitterTask
   extends RecursiveAction
   { ... }

   fjPool.invoke(new SplitterTask
     (0, list.size()));

   return Arrays.asList(results);
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex22

Implementing the Method applyAllSplitIndex()

Implementing the Method applyAllSplitIndex()

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

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


  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

See LiveLessons/blob/master/Java8/ex22/src/utils/ForkJoinUtils.java

Implementing the Method applyAllSplitIndex()

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

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

  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

Create a new array to hold the results (yes, it's ugly…)

See stackoverflow.com/a/18137953

Implementing the Method applyAllSplitIndex()

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

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

  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

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

Implementing the Method applyAllSplitIndex()

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

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

  class SplitterTask extends RecursiveAction { ... }

  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

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

Implementing the Method applyAllSplitIndex()

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

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

  class SplitterTask extends RecursiveAction { ... }

  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

Create & return a list from the array of results

These conversions to & from list → array → list incur overhead

Implementing the Method applyAllSplitIndex()

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

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

An alternative—more elegant & more efficient—approach passes the results array as a parameter

```
  class SplitterTask extends RecursiveAction {
    /* same implementation as follows next.. */
  }

  fjPool.invoke(new SplitterTask(0, list.size()));
}
```

This approach is cleaner & faster since it has no conversion overhead

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    private int mLo;
    private int mHi;

    private SplitterTask(int lo, int hi) {
        mLo = lo;
        mHi = hi;
    }
    ...

  } ...
```

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

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    private int mLo;
    private int mHi;

    private SplitterTask(int lo, int hi) {
        mLo = lo;
        mHi = hi;
    }
    ...

  } ...
```

It uses indices to avoid the overhead of copy sub-lists

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Recursively perform the computations in parallel

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Find mid-point in current range

This code recursively & evenly partitions the list

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Apply op if there's just one element

This if statement handles the base case for the recursion

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                                Function<T, T> op,
                                ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

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

This implementation uses recursive decomposition to disperse tasks to worker threads

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Compute the right-hand side in parallel with left-hand side

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

Implementing the Method applyAllSplitIndex()

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

```
<T> List<T> applyAllSplitIndex(List<T> list,
                                Function<T, T> op,
                                ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Join with left-hand side (this is a synchronization point)

See stackoverflow.com/questions/4800503/memory-visibility-in-fork-join

Implementing the Method applyAllSplitIndex()

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

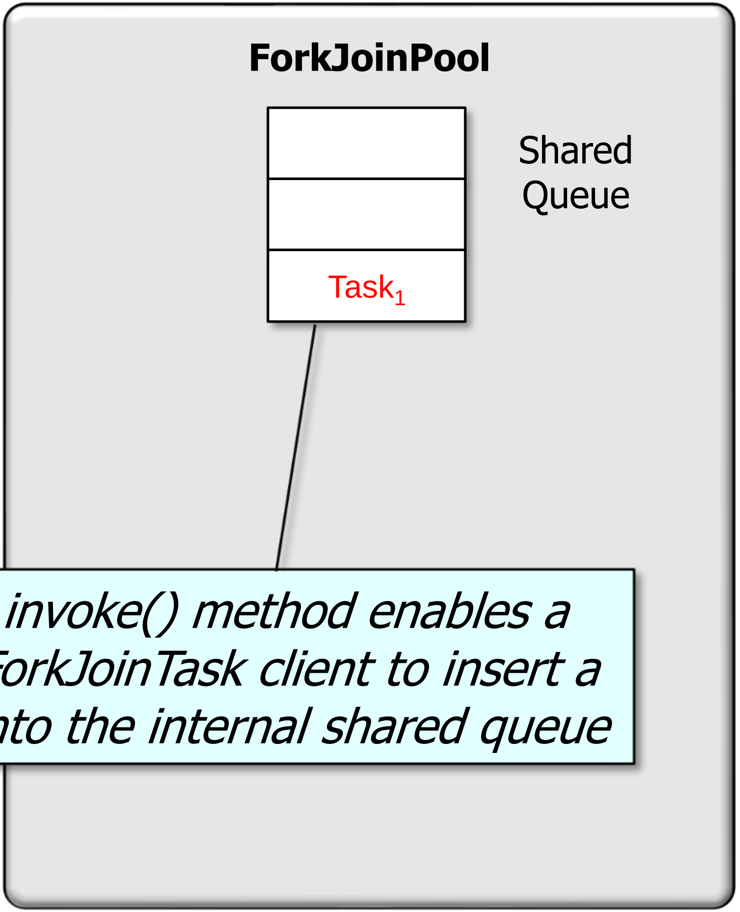
```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

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

Implementing the Method applyAllSplitIndex()

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  fjPool
    .invoke(new SplitterTask
              (0,
               list.size()));
  ...
```



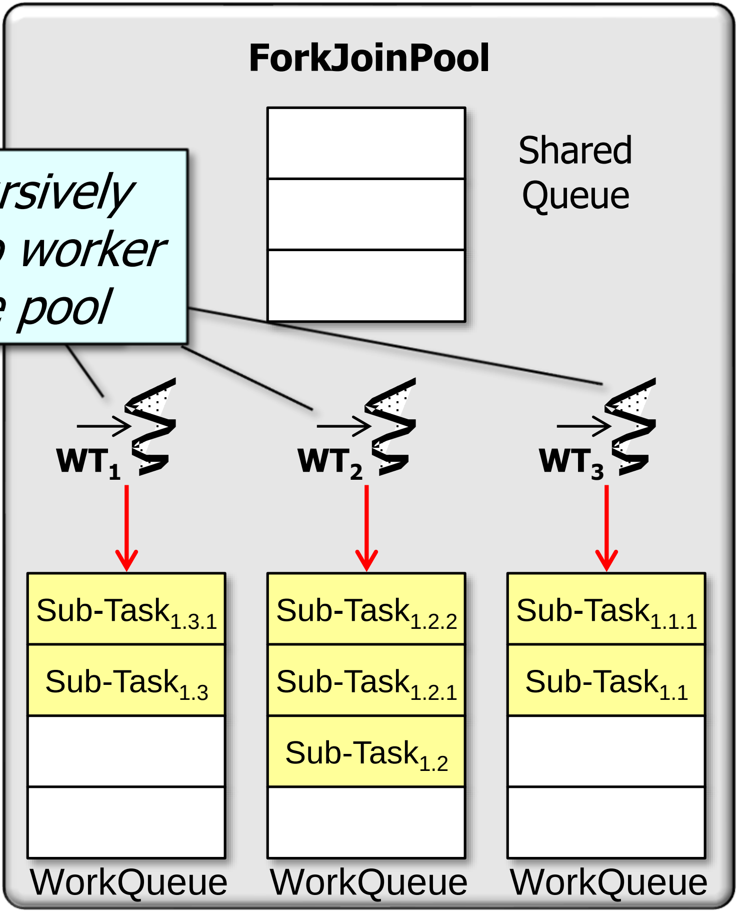
The invoke() method enables a non-ForkJoinTask client to insert a task into the internal shared queue

Implementing the Method applyAllSplitIndex()

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
          (List<T> list,
           Function<T, T>
           ForkJoinPool fj
  class SplitterTask ... {
    protected void compute() {
      ... else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo,
                           mLo = mid)
          .fork();
        compute();
        lt.join();
      }
```

Sub-tasks recursively decompose onto worker threads in the pool



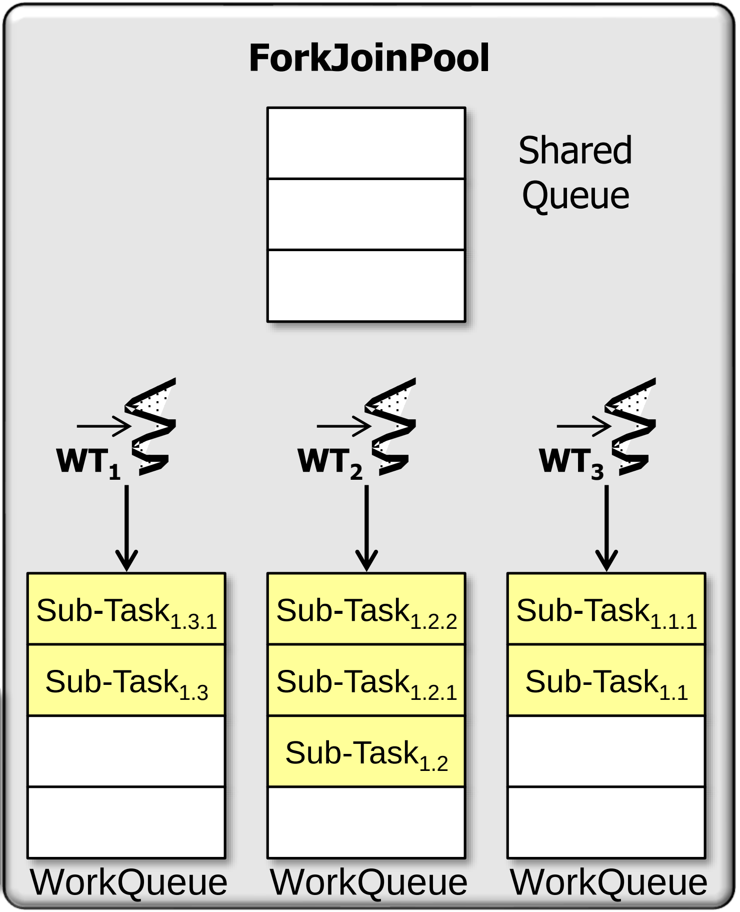
"Work-stealing" & copying overhead is low, but method call overhead is high

Implementing the Method applyAllSplitIndex()

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) { ...
  class SplitterTask ... {
    protected void compute() {
      ... else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo,
                           mLo = mid)
          .fork();
        compute();
        lt.join();
      }
```

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

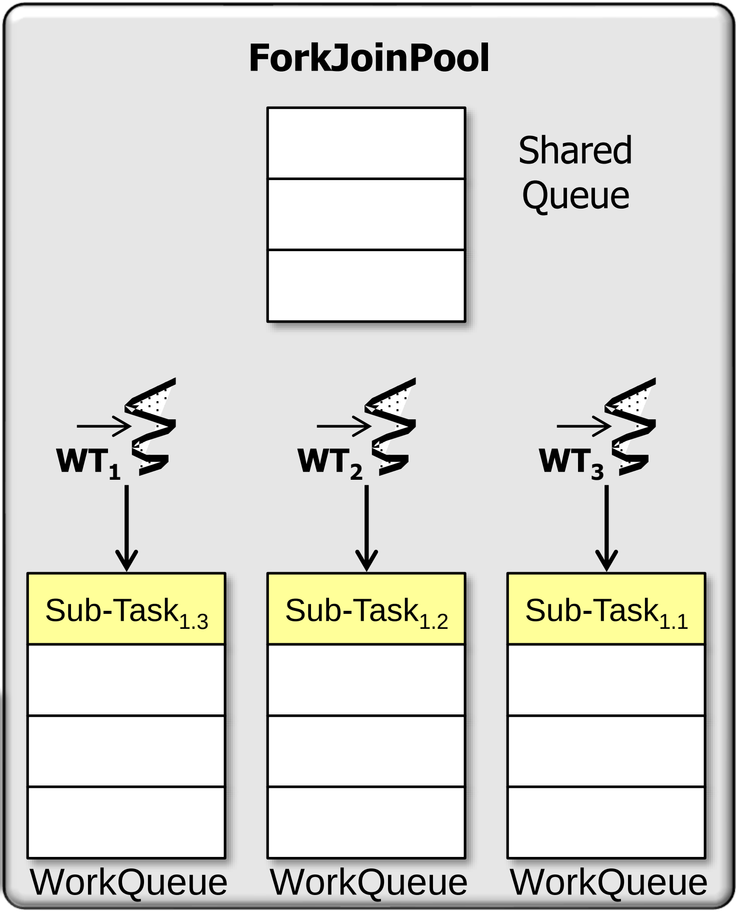


Implementing the Method applyAllSplitIndex()

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
             (List<T> list,
              Function<T, T> op,
              ForkJoinPool fjPool) { ...
  class SplitterTask ... {
    protected void compute() {
      ... else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo,
                           mLo = mid)
            .fork();
        compute();
        lt.join();
      }
```

join() returns no value & just serves as a synchronization point



There is a "balanced tree" of join() calls

End of the Java Fork-Join Pool: Implementing applyAllSplitIndex()