The Java Fork-Join Pool Framework (Part 4)

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

- Understand how the Java fork-join framework processes tasks in parallel
- Recognize the structure & functionality of the fork-join framework
- Know how the fork-join framework is implemented internally
- Recognize the key methods in the ForkJoinPool class & related classes
- Apply the fork-join framework in practice
  - Implement operations on BigFractions using several different fork-join pool programming models

<<Java Class>>

BigFraction

(default package)

mNumerator: BigInteger
mDenominator: BigInteger

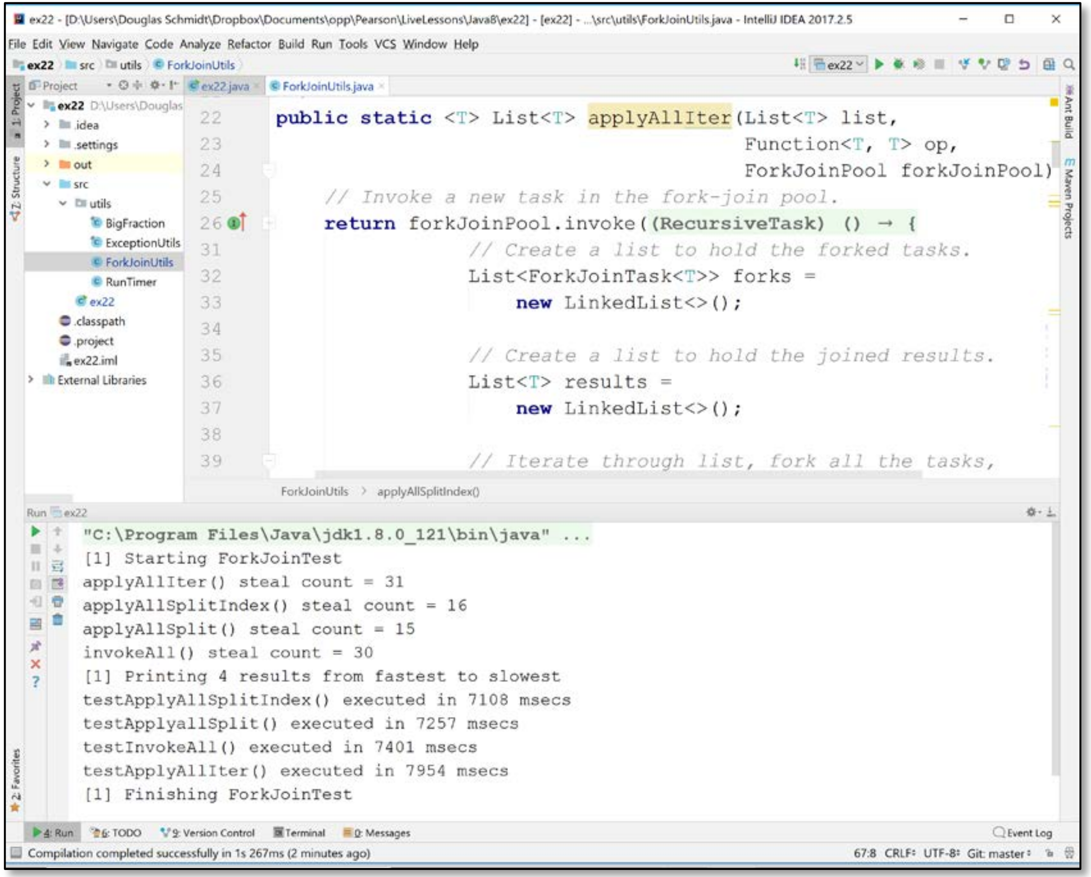
valueOf(Number):BigFraction
valueOf(Number,Number):BigFraction
valueOf(String):BigFraction
valueOf(Number,Number,boolean):BigFraction
reduce(BigFraction):BigFraction
getNumerator():BigInteger
getDenominator():BigInteger
add(Number):BigFraction
subtract(Number):BigFraction
multiply(Number):BigFraction
divide(Number):BigFraction
gcd(Number):BigFraction
toMixedString():String

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

Applying the Java Fork-Join Pool Framework

Applying the Java Fork-Join Pool Framework

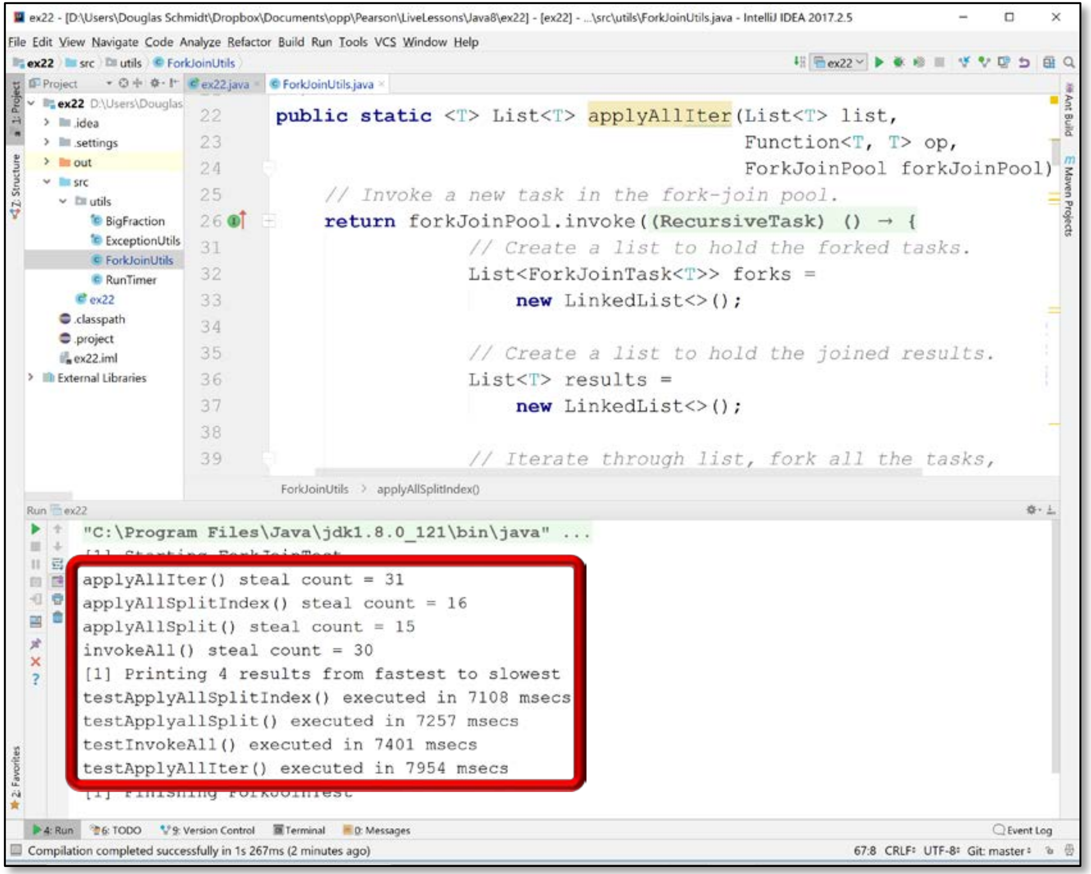
- Reduce & multiply big fractions using several different models of programming the Java fork-join pool framework



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

Applying the Java Fork-Join Pool Framework

- Reduce & multiply big fractions using several different models of programming the Java fork-join pool framework
    - These model have different performance pros & cons



e.g., some incur more "stealing", copy more data, make more method calls, etc.

Applying the Java Fork-Join Pool Framework

- Reduce & multiply big fractions using the Java fork-join pool framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

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

Applying the Java Fork-Join Pool Framework

- Reduce & multiply big fractions using the Java fork-join pool framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

Generate random BigFractions up to sMAX_FRACTIONS

Applying the Java Fork-Join Pool Framework

- Reduce & multiply big fractions using the Java fork-join pool framework

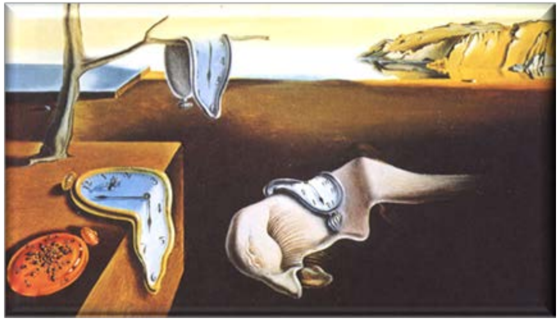
```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

A function that reduces & multiplies a big fraction

Applying the Java Fork-Join Pool Framework

- Reduce & multiply big fractions using the Java fork-join pool framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```



This function takes a surprisingly long time to run!

Applying the Java Fork-Join Pool Framework

- Reduce & multiply big fractions using the Java fork-join pool framework

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

Run various fork-join tests

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() utility methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }

void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }

void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Applying the Java Fork-Join Pool Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() utility methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }

void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }

void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Each utility method uses a different means of applying the fork-join framework

Applying the Java Fork-Join Pool Framework

- Test the applyAllIter(), applyAllSplit(), & applyAllSplitIndex() utility methods

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }

void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }

void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Each call to a utility method gets its own fork-join pool instance

Implementing the applyAllIter() Method

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) {
  return fjPool
    .invoke(new RecursiveTask<List<T>>() {
             protected List<T> compute() {
               ...
             }
           });
}
```

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

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) {
  return fjPool
    .invoke(new RecursiveTask<List<T>>() {
              protected List<T> compute() {
                ...
              }
            });
}
```

Create an anonymous RecursiveTask & invoke it on the fork-join pool

Code is verbose due to lack of functional interface (& thus can't use lambdas)..

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                  protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Implements the main fork-join task

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                  protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Lists that hold the forked tasks & the results

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                 protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Iterate through input list, fork all the tasks, & add them to the forks list

This implementation relies on “work-stealing” to disperse tasks to worker threads

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                  protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Join all results of forked tasks & add to results list

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                 protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Return the results list

Implementing the applyAllIter() Method

- Apply op to all items in the list by calling fork-join methods iteratively

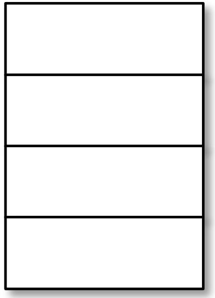
```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                  protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

This implementation is very simple to program & understand since it's iterative

Implementing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
         (List<T> list,
          Function<T, T> op,
          ForkJoinPool fjPool) {
  fjPool.invoke
    (new RecursiveTask<List<T>>() {
       protected List<T> compute() {
         ...
       }
  }
  ...
```



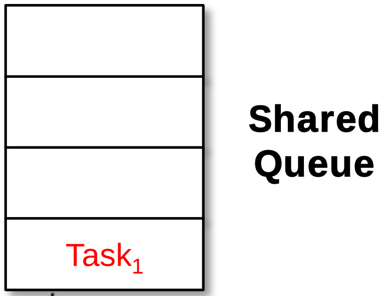
Shared Queue

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

Implementing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
            (List<T> list,
             Function<T, T> op,
             ForkJoinPool fjPool) {
  fjPool.invoke
    (new RecursiveTask<List<T>>() {
       protected List<T> compute() {
         ...
       }
  }
  ...
```



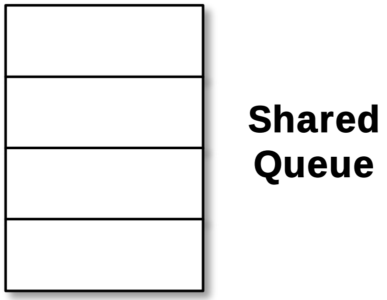
The invoke() method enables a non-ForkJoinTask client to insert a task into the internal shared queue

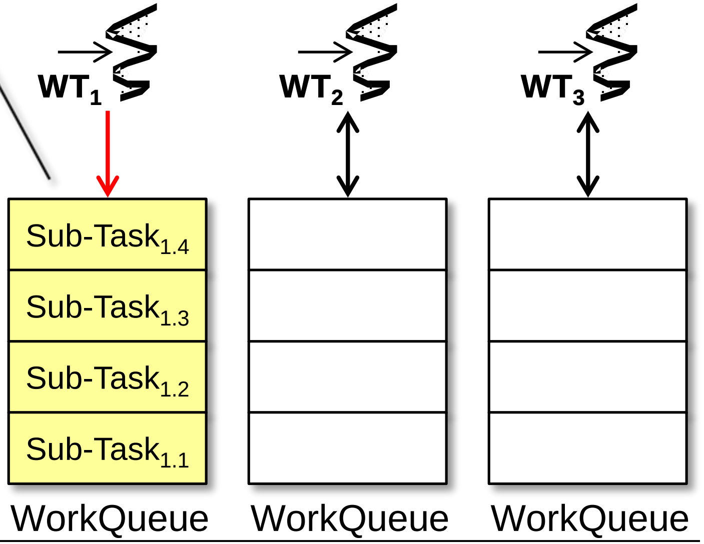
Implementing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op
           ForkJoinPool fjPo
  ...
  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());
  ...
```

Worker thread WT_1 creates n new sub-tasks that run an op on each list element

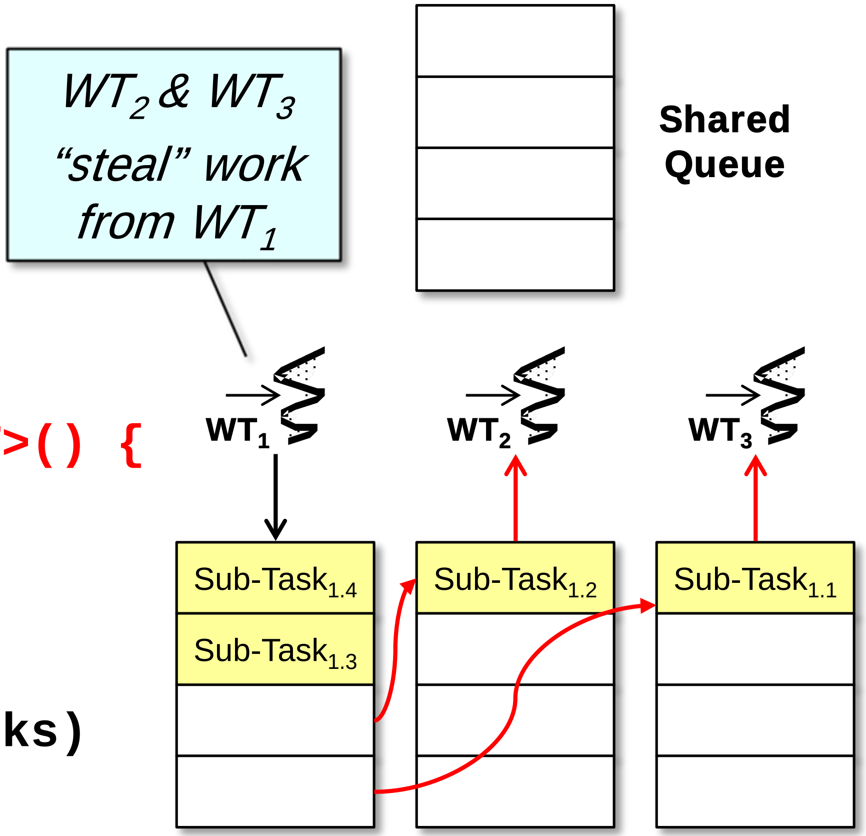




Implementing the applyAllIter() Method

- Visualizing applyAllIter()

```
<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());
  ...
```

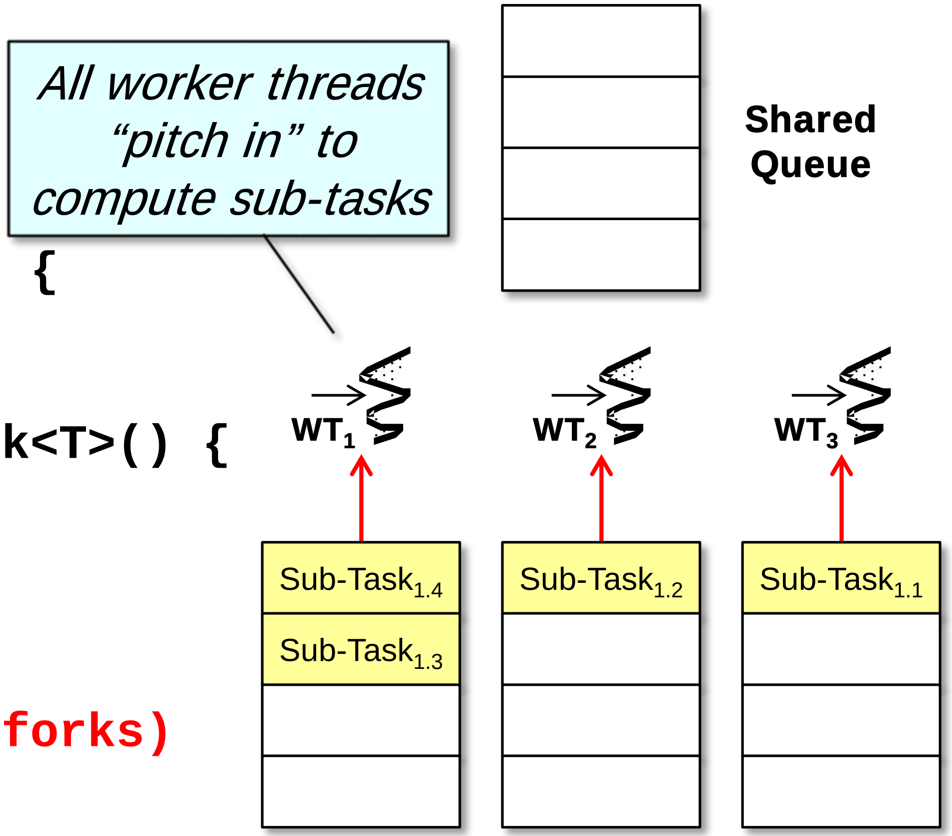


"Work-stealing" is high, but copying & method calls are low

Implementing the applyAllIter() Method

- Visualizing applyAllIter()

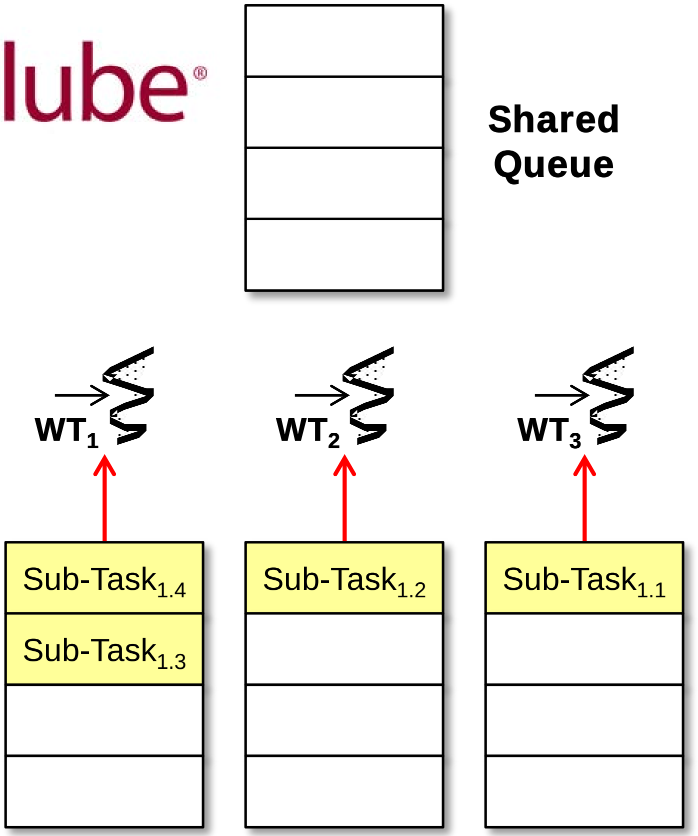
```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  for (T t : list)
    forks.add(new RecursiveTask<T>() {
      protected T compute()
      { return op.apply(t); }
    }.fork());

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



Implementing the applyAllIter() Method

- Visualizing applyAllIter()

```
<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());
  ...
```



"Collaborative Jiffy Lube" model of processing!

Implementing the Method applyAllSplitter()

Implementing the Method applyAllSplitter()

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

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




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

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

Implementing the Method applyAllSplitter()

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

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

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

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

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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 applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(List<T> list, Function<T, T> op,
                             ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    private List<T> mList;


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

  } ...
```

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

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(List<T> list, Function<T, T> op,
                             ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    private List<T> mList;



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

  } ...
```

Stores a reference to a portion of the original list

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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 {
        ...
    } ...
```

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

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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 results

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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 applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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 implementation uses recursive decomposition to disperse tasks to worker threads

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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

This implementation uses recursive decomposition to disperse tasks to worker threads

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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 with left-side results

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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

Implementing the Method applyAllSplitter()

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

```
<T> List<T> applyAllSplitter(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

Implementing the Method applyAllSplitter()

- Visualizing applyAllSplitter()

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

Shared Queue

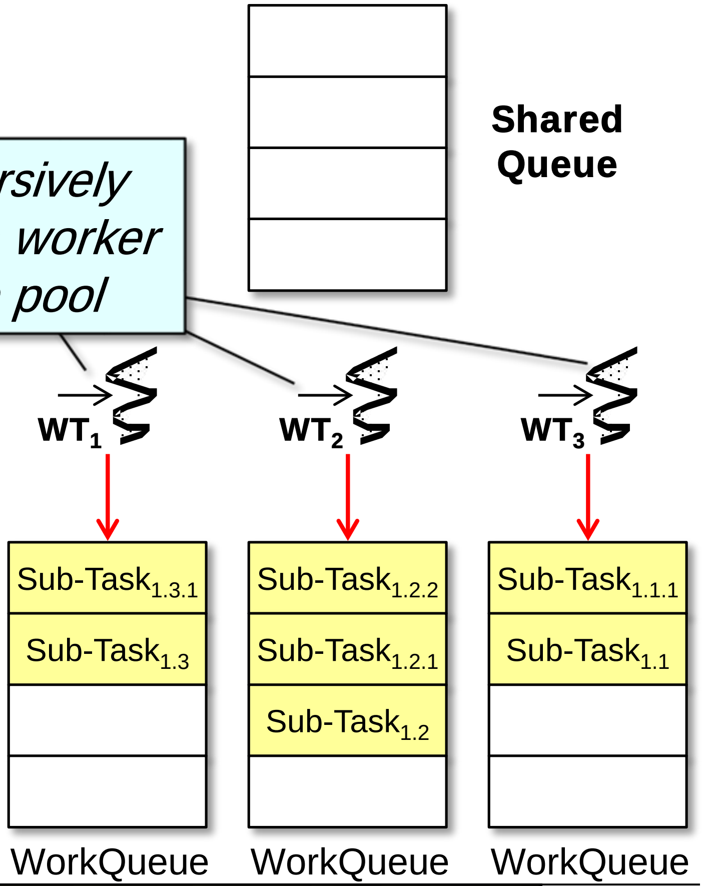
The invoke() method enables a non-ForkJoinTask client to insert a task into the internal shared queue

Implementing the Method applyAllSplitter()

- Visualizing applyAllSplitter()

```
<T> List<T> applyAllSplitter
          (List<T> list,
           Function<T, T> o
           ForkJoinPool fjP
 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



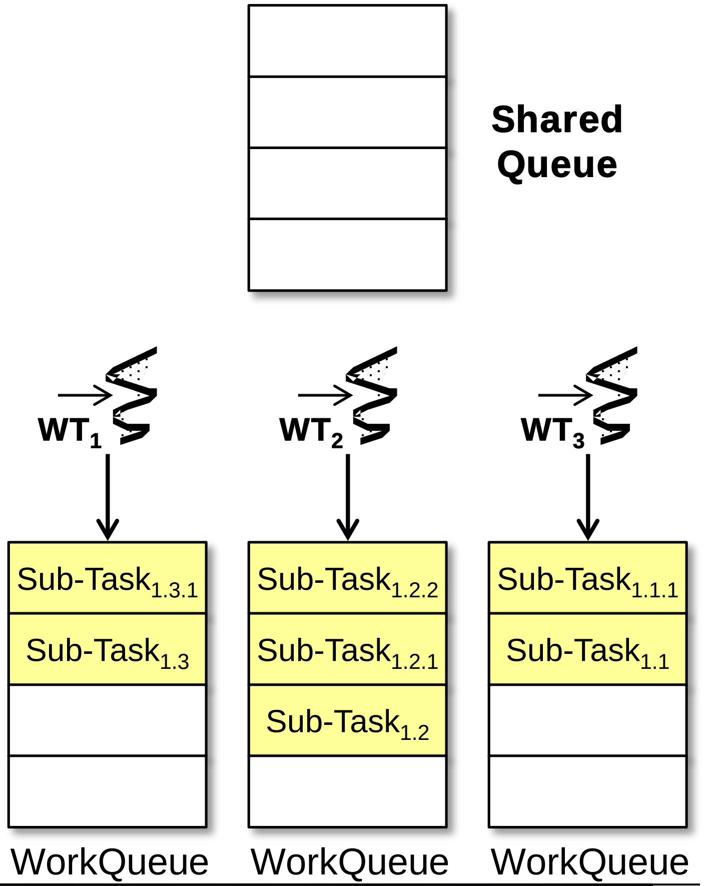
“Work-stealing” is lower, but copying & method calls are higher

Implementing the Method applyAllSplitter()

- Visualizing applyAllSplitter()

```
<T> List<T> applyAllSplitter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) { ...
  class SplitterTask ... {
                      ompute() {

                      <List<T>> lt =
         new SplitterTask
           (mList.subList(0, mid))
         .fork();
       ...
       List<T> rightResult =
         compute(); ...
```

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



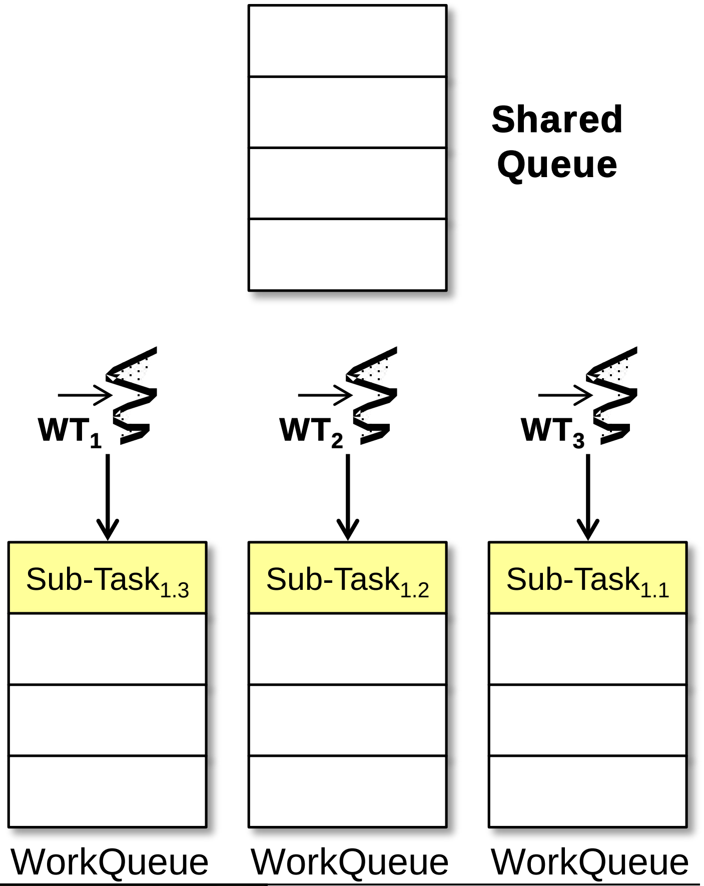
compute() runs in the same task as its "parent" to optimize performance

Implementing the Method applyAllSplitter()

- Visualizing applyAllSplitter()

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

join() returns a value



There is a “balanced tree” of join() calls

Implementing the Method applyAllSplitterIndex()

Implementing the Method applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

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

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

```
  class SplitterTask extends RecursiveAction { ... }


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

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

Implementing the Method applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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

Implementing the Method applyAllSplitterIndex()

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

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

  class SplitterTask extends RecursiveAction { ... }


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

An alternative—more streamlined—approach is to pass the results array as a parameter

Implementing the Method applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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

Implementing the Method applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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

Implementing the Method applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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

Implementing the Method applyAllSplitterIndex()

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

```
<T> List<T> applyAllSplitterIndex(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 applyAllSplitterIndex()

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

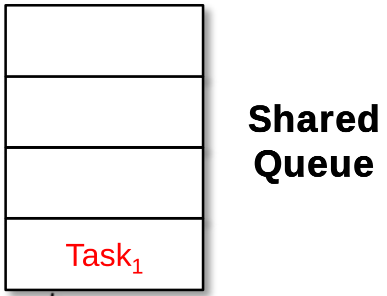
```
<T> List<T> applyAllSplitterIndex(List<T> list,
                                  Function<T, T> op,
                                  ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        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 applyAllSplitterIndex()

- Visualizing applyAllSplitterIndex()

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



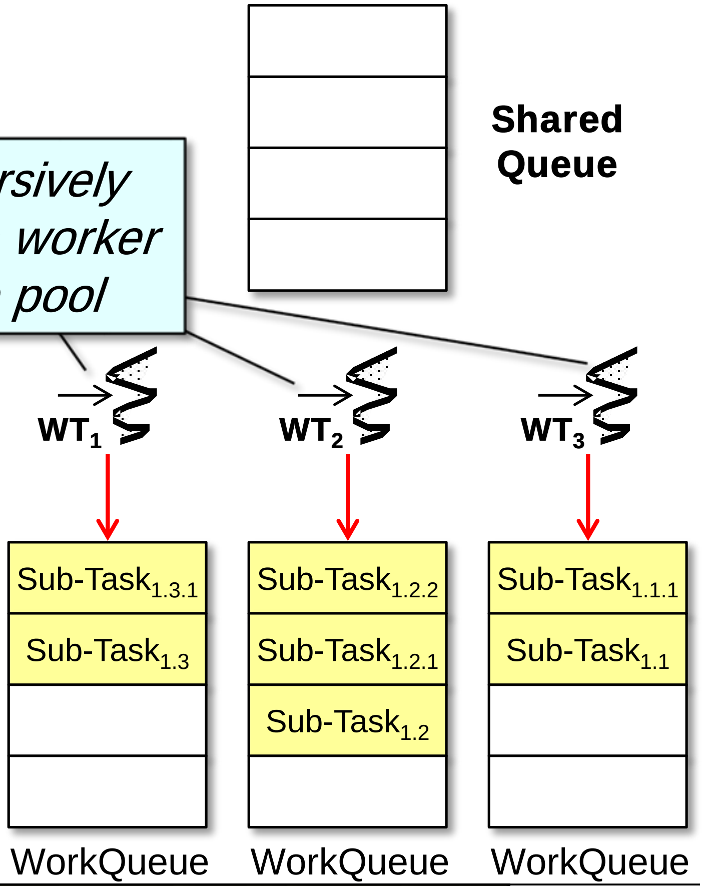
The invoke() method enables a non-ForkJoinTask client to insert a task into the internal shared queue

Implementing the Method applyAllSplitterIndex()

- Visualizing applyAllSplitterIndex()

```
<T> List<T> applyAllSplitterIndex
          (List<T> list,
           Function<T, T> o
           ForkJoinPool fjP
  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



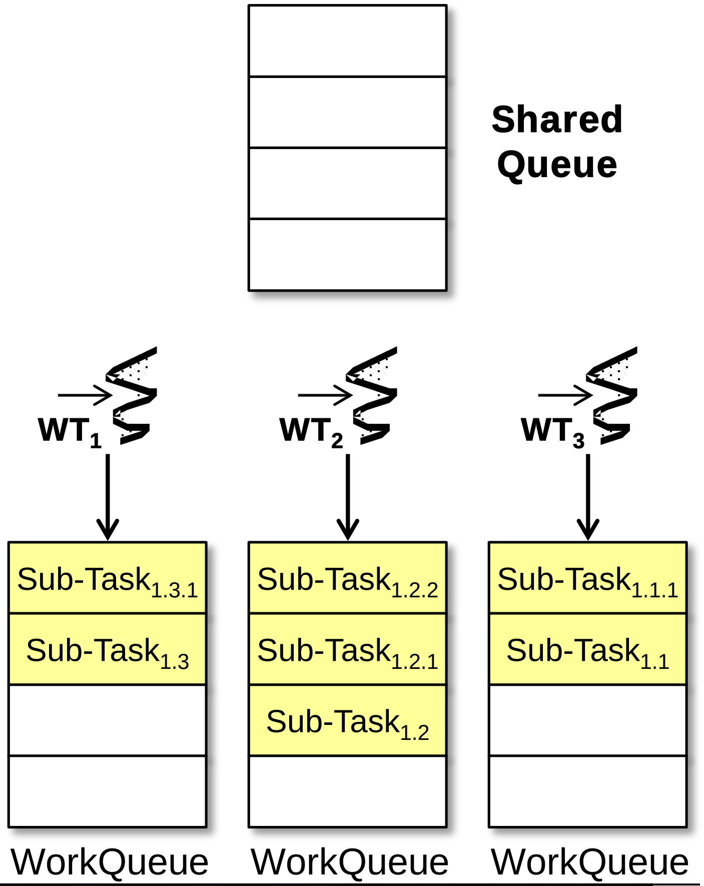
“Work-stealing” & copying are low, but method calls are high

Implementing the Method applyAllSplitterIndex()

- Visualizing applyAllSplitterIndex()

```
<T> List<T> applyAllSplitterIndex
          (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



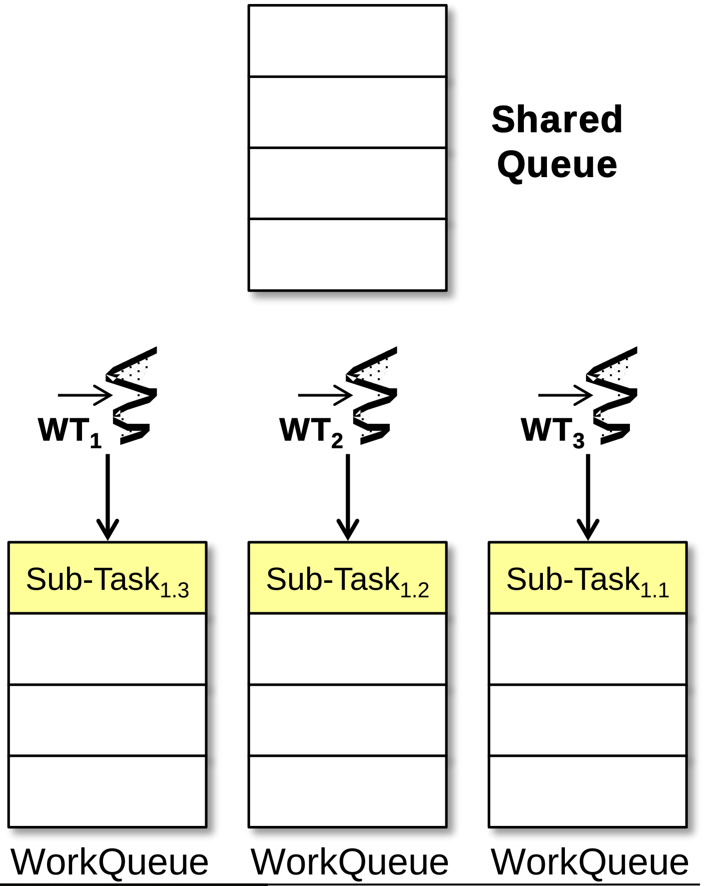
compute() runs in the same task as its "parent" to optimize performance

Implementing the Method applyAllSplitterIndex()

- Visualizing applyAllSplitterIndex()

```
<T> List<T> applyAllSplitterIndex
         (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

Wrapping Up

Wrapping Up

- Each Java fork-join programming model has pros & cons

Wrapping Up

- 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());
  ...
```

Wrapping Up

- 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

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java" ...
[1] Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 15
invokeAll() steal count = 30
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndex() executed in 7108 msecs
testApplyallSplit() executed in 7257 msecs
testInvokeAll() executed in 7401 msecs
testApplyAllIter() executed in 7954 msecs
[1] Finishing ForkJoinTest
```

Wrapping Up

- 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

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java" ...
[1] Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 15
invokeAll() steal count = 30
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndex() executed in 7108 msecs
testApplyallSplit() executed in 7257 msecs
testInvokeAll() executed in 7401 msecs
testApplyAllIter() executed in 7954 msecs
[1] Finishing ForkJoinTest
```

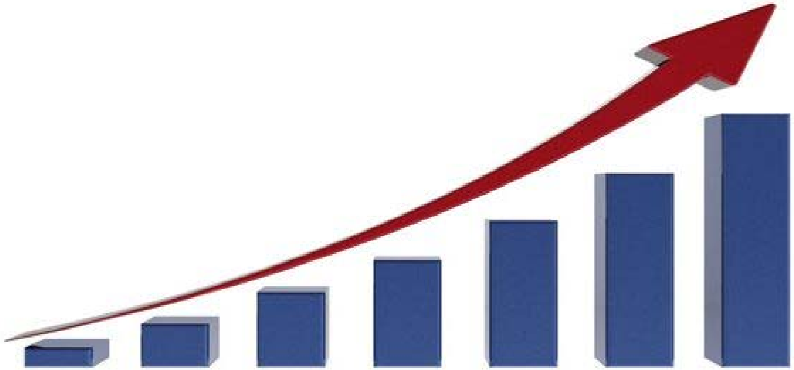
Wrapping Up

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

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java" ...
[1] Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 15
invokeAll() steal count = 30
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndex() executed in 7108 msecs
testApplyallSplit() executed in 7257 msecs
testInvokeAll() executed in 7401 msecs
testApplyAllIter() executed in 7954 msecs
[1] Finishing ForkJoinTest
```

Wrapping Up

- 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



```
"C:\Program Files\Java\jdk1.8.0_121\bin\java" ...
[1] Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 15
invokeAll() steal count = 30
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndex() executed in 7108 msecs
testApplyallSplit() executed in 7257 msecs
testInvokeAll() executed in 7401 msecs
testApplyAllIter() executed in 7954 msecs
[1] Finishing ForkJoinTest
```

Wrapping Up

- 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;
} ...
```

Wrapping Up

- 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.

Wrapping Up

- 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

```
"C:\Program Files\Java\jdk1.8.0_121\bin\java" ...
[1] Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 15
invokeAll() steal count = 30
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndex() executed in 7108 msecs
testApplyallSplit() executed in 7257 msecs
testInvokeAll() executed in 7401 msecs
testApplyAllIter() executed in 7954 msecs
[1] Finishing ForkJoinTest
```

Wrapping Up

- 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

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

Wrapping Up

- 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

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

End of the Java Fork-Join Pool Framework (Part 4)