CS131: Programming Languages

Lun Liu

05.26.17

Parallelism in Java

- Explicit threading
- `Stream`
- `fork-join` framework

java.util.Stream

- Support functional style operations on streams of elements
- Stream can be obtained in a number of ways
  - From a Collection via the stream() and parallelStream() methods;
  - From an array via Arrays.stream(Object[]);
  - …

```
int sum = Arrays.stream(a).reduce(0, (i1,i2) ->
i1+i2);
```

- Facilitate parallel execution by reframing the computation as a pipeline of aggregate operations
  - `.parallel(), .parallelStream()`

Lambda Expressions in Java

- Available in Java 8
- Shorthand for anonymous functions
    - (parameters) -> {body}

```
words.sort((a, b) -> a.length() – b.length())
```

`fork-join` Framework

- Distributes tasks to worker threads in a thread pool
- An implementation of the `ExecutorService` interface

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces invoke the two pieces and wait for the results
```

`fork-join` Framework

- **`ForkJoinPool`**: An instance of this class is used to run all your fork-join tasks in the whole program.
- **`RecursiveTask<V>:`** You run a subclass of this in a pool and have it return a result
- **`RecursiveAction`**: just like RecursiveTask except it does not return a result
- **`ForkJoinTask<V>:`** superclass of RecursiveTask<V> and RecursiveAction. fork and join are methods defined in this class. You won't use this class directly, but it is the class with most of the useful javadoc documentation, in case you want to learn about additional methods.

http://homes.cs.washington.edu/~djg/teachingMaterials/spac/grossmanSPAC_forkJoinFramework.html

fork-join Framework

```
class Sum extends RecursiveTask<Long> {
  static final int SEQUENTIAL_THRESHOLD = 5000;
  int low; int high; int[] array;
  Sum(int[] arr, int lo, int hi) {
    array = arr;
    low = lo;
    high = hi;
  }
  protected Long compute(){…}
  static long sumArray(int[] array) {
    return ForkJoinPool.commonPool().invoke(new
Sum(array,0,array.length));
  }
}
```

fork-join Framework

```
class Sum extends RecursiveTask<Long> {
  static final int SEQUENTIAL_THRESHOLD = 5000;
  int low; int high; int[] array;
  Sum(int[] arr, int lo, int hi) {
    array = arr;
    low = lo;
    high = hi;
  }
  protected Long compute(){…}
  static long sumArray(int[] array) {
    return ForkJoinPool.commonPool().invoke(new
Sum(array,0,array.length));
  }
}
```

fork-join Framework

```
class Sum extends RecursiveTask<Long> {
  static final int SEQUENTIAL_THRESHOLD = 5000;
  int low; int high; int[] array;
  Sum(int[] arr, int lo, int hi) {
    array = arr;
    low = lo;
    high = hi;
  }
  protected Long compute(){…}
  static long sumArray(int[] array) {
    return ForkJoinPool.commonPool().invoke(new
Sum(array,0,array.length));
  }
}
```

fork-join Framework

```
class Sum extends RecursiveTask<Long> {
  static final int SEQUENTIAL_THRESHOLD = 5000;
  int low; int high; int[] array;
  Sum(int[] arr, int lo, int hi) {
    array = arr;
    low = lo;
    high = hi;
  }
  protected Long compute(){…}
  static long sumArray(int[] array) {
    return ForkJoinPool.commonPool().invoke(new
Sum(array,0,array.length));
  }
}
```

```
protected Long compute() {
  if(high - low <= SEQUENTIAL_THRESHOLD) {
    long sum = 0;
    for(int i=low; i < high; ++i)
      sum += array[i];
    return sum;
  } else {
    int mid = low + (high - low) / 2;
    Sum left = new Sum(array, low, mid);
    Sum right = new Sum(array, mid, high);
    left.fork();
    long rightAns = right.compute();
    long leftAns = left.join();
    return leftAns + rightAns;
  }
}
```

fork-join Framework

```
  protected Long compute() {
    if(high - low <= SEQUENTIAL_THRESHOLD) {
      long sum = 0;
      for(int i=low; i < high; ++i)
        sum += array[i];
      return sum;
    } else {
      int mid = low + (high - low) / 2;
      Sum left = new Sum(array, low, mid);
      Sum right = new Sum(array, mid, high);
      left.fork(); //create a new task, invoke
compute() to calculate Sum for left part
      long rightAns = right.compute();
      long leftAns = left.join();
      return leftAns + rightAns;
    }
  }
```

fork-join Framework

```
  protected Long compute() {
    if(high - low <= SEQUENTIAL_THRESHOLD) {
      long sum = 0;
      for(int i=low; i < high; ++i)
        sum += array[i];
      return sum;
    } else {
      int mid = low + (high - low) / 2;
      Sum left = new Sum(array, low, mid);
      Sum right = new Sum(array, mid, high);
      left.fork();
      long rightAns = right.compute(); //Sum of
ritht is calculated by current Thread
      long leftAns = left.join();
      return leftAns + rightAns;
    }
  }
```

fork-join Framework

```
  protected Long compute() {
    if(high - low <= SEQUENTIAL_THRESHOLD) {
      long sum = 0;
      for(int i=low; i < high; ++i)
        sum += array[i];
      return sum;
    } else {
      int mid = low + (high - low) / 2;
      Sum left = new Sum(array, low, mid);
      Sum right = new Sum(array, mid, high);
      left.fork();
      long rightAns = right.compute();
      long leftAns = left.join();  //wait (block
until getting Sum of left from other thread)
      return leftAns + rightAns;
    }
  }
```

fork-join Framework

```
protected Long compute() {
  if(high - low <= SEQUENTIAL_THRESHOLD) {
    long sum = 0;
    for(int i=low; i < high; ++i)
      sum += array[i];
    return sum;
  } else {
    int mid = low + (high - low) / 2;
    Sum left = new Sum(array, low, mid);
    Sum right = new Sum(array, mid, high);
    left.fork();
    long rightAns = right.compute();
    long leftAns = left.join();
    return leftAns + rightAns; //merge results
  }
}
```

fork-join Framework

```
protected Long compute() {
  if(high - low <= SEQUENTIAL_THRESHOLD) {
    long sum = 0;
    for(int i=low; i < high; ++i)
      sum += array[i];
    return sum;
  } else {
    int mid = low + (high - low) / 2;
    Sum left = new Sum(array, low, mid);
    Sum right = new Sum(array, mid, high);
    left.fork();
    long rightAns = right.compute();
    long leftAns = left.join();
    return leftAns + rightAns;
  }
}
```

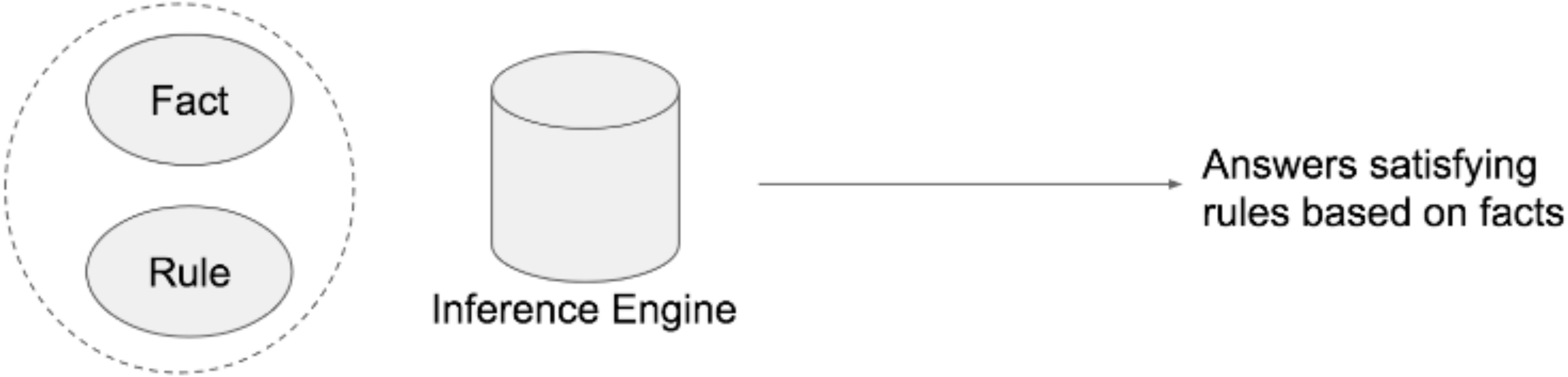
```
if (my portion of the work is
small enough)
  do the work directly
else
  split my work into two pieces
  left.fork()
  right.compute()
  left.join()
  merge
```

Declarative Programming

- “What” instead of “How”

Prolog

- Specify logical constraints, the language searches for a solution to those constraints



Prolog: Facts and Queries

```
cat(tom).   /* tom is a cat */
chase(tom, jerry) /* tom chases jerry */

?- cat(tom)   /*is tom a cat */
yes
?- chase(tom, jerry)  /* does tom chase jerry */
yes
```

Prolog: Variables and Unification

- Variables are distinguished by starting with a **capital letter**
- The process of matching items with variables is known as **unification**

```
loves(john,mary).
loves(fred,hobbies).

?- loves(john,Who).  /* Who does john love? */
Who=mary             /* yes , Who gets bound to mary */
yes                  /* and the query succeeds*/
?- loves(arnold,Who) /* does arnold love anybody */
no                   /* no, arnold doesn't match john or fred */
?- loves(fred,Who).  /* Who does fred love */
Who = hobbies        /* Note the to Prolog Who is just the name of a variable, it */
yes                  /* semantic connotations are not picked up, hence Who unifies
with hobbies */
```

Prolog: Rules

```
animal(X):-cat(X).       /* a cat is an animal */
cat(tom).

?- animal (tom).    /* is tom an animal */
yes                 /* inference rules reduce animal(X)
                    to  cat(X), and cat(tom) is in
                    knowledge base */
```

Prolog: Rules

- AND

```
p(X):-a(X), b(X). /* if a(X) and b(X), then p(X) */
```

- OR

```
parent(X, Y):-father(X, Y).
parent(X, Y):-mother(X, Y).
```

- Rules can be recursive

```
ancestor(X, Y):-parent(X, Y).
ancestor(X, Y):-parent(Z, Y), ancestor(X, Z).
```

Prolog: Lists

- `[e1, e2, e3…]`
- `[H|T]`
- `head(H, H|_).`
- `second(_, Snd|_, Snd).`

Backup

Java First-Class Functions

- Classes as first-class functions
- Anonymous classes as first-class functions
- Lambdas

Explicit Threading in Java

- Two ways to create thread in Java
  - Implement runnable interface (`java.lang.Runnable`)
  - Extend the Thread class (`java.lang.Thread`)