Lecture 22: Parallelism in Java Streams, Parallel Prefix Sums

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/
Q: Compute the WORK and CPL metrics for this program. Indicate if your answer depends on the execution order of isolated constructs.

Answer: WORK = 25, CPL = 9. These metrics do not depend on the execution order of isolated constructs.
How Java Streams addressed pre-Java-8 limitations of Java Collections

1. Iteration had to be performed explicitly using for/foreach loop, e.g.,

   // Iterate through students (collection of Student objects)
   for (Student s in students) System.out.println(s);

   ⇒ Simplified using Streams as follows
   students.stream().forEach(s -> System.out.println(s));

2. Overhead of creating intermediate collections

   List<Student> activeStudents = new ArrayList<Student>();
   for (Student s in students)
     if (s.getStatus() == Student.ACTIVE) activeStudents.add(s);
   for (Student a in activeStudents) totalCredits += a.getCredits();

   ⇒ Simplified using Streams as follows
   totalCredits = students.stream().filter(s -> s.getStatus() == Student.ACTIVE)
                 .map(a -> a.getCredits()).sum();

3. Complexity of parallelism simplified (for example) by replacing stream() by parallelStream()
Java 8 Streams Cheat Sheet

Definitions
- A stream is a pipeline of functions that can be evaluated.
- Streams can transform data.
- A stream is not a data structure.
- Streams cannot mutate data.

Intermediate operations
- Always return streams.
- Lazily executed.

Common examples include:

<table>
<thead>
<tr>
<th>Function</th>
<th>Preserves count</th>
<th>Preserves type</th>
<th>Preserves order</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>filter</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>distinct</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sorted</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>peek</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Stream examples
Get the unique surnames in uppercase of the first 15 book authors that are 50 years or old over.

```java
library.stream()
    .map(book -> book.getAuthor())
    .filter(author -> author.getAge() >= 50)
    .map(String::toUpperCase)
    .distinct()
    .limit(15)
    .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```java
library.stream()
    .map(Book::getAuthor)
    .filter(a -> a.getGender() == Gender.FEMALE)
    .map(Author::getAge)
    .filter(age -> age < 25)
    .reduce(0, Integer::sum):
```

Terminal operations
- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

<table>
<thead>
<tr>
<th>Function</th>
<th>Output</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>concrete type</td>
<td>to cumulate elements</td>
</tr>
<tr>
<td>collect</td>
<td>list, map or set</td>
<td>to group elements</td>
</tr>
<tr>
<td>forEach</td>
<td>side effect</td>
<td>to perform a side effect on elements</td>
</tr>
</tbody>
</table>

Parallel streams
Parallel streams use the common ForkJoinPool for threading.

```java
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations
Grouping:

```
library.stream().collect(
    groupingBy(Book::getGenre));
```

Stream ranges:
```
IntStream.range(0, 20)...
```

Infinite streams:
```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:
```
IntStream.range(1, 10).max();
```

FlatMap:
```
twitterList.stream()
    .map(member -> member.getFollowers())
    .flatMap(followers -> followers.stream())
    .collect(toList());
```

Pitfalls
- Don’t update shared mutable variables i.e.
  ```java
  List<Book> myList = new ArrayList<>();
  library.stream().forEach((e -> myList.add(e));
  ```
- Avoid blocking operations when using parallel streams.

Source: http://zeroturnaround.com/rebellabs/java-8-streams-cheat-sheet/
Parallelism in processing Java Streams

• Parallelism can be introduced at a stream source ...
  — e.g., library.parallelStream()...

• ... or as an intermediate operation
  — e.g., library.stream().sorted().parallel()...

• Stateful intermediate operations should be avoided on parallel streams ...
  — e.g., distinct, sorted, use-written lambda with side effects

• ... but stateless intermediate operations work just fine
  — e.g., filter, map

• Parallelism is usually more efficient on unordered streams ...
  — e.g., stream created from unordered source (HashSet), or from .unordered() intermediate operation

• ... and with unordered collectors
  — e.g., ConcurrentHashMap
Beyond Sum/Reduce Operations — Prefix Sum (Scan) Problem Statement

Given input array $A$, compute output array $X$ as follows

$$X[i] = \sum_{0 \leq j \leq i} A[j]$$

- The above is an **inclusive** prefix sum since $X[i]$ includes $A[i]$
- For an **exclusive** prefix sum, perform the summation for $0 \leq j < i$
- It is easy to see that inclusive prefix sums can be computed sequentially in $O(n)$ time ...

// Copy input array $A$ into output array $X$

```java
X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);
```

// Update array $X$ with prefix sums

```java
for (int i=1 ; i < X.length ; i++ ) X[i] += X[i-1];
```

- ... and so can exclusive prefix sums
An Inefficient Parallel Algorithm for Exclusive Prefix Sums

1. `forall(0, x.length-1, (i) -> {
   2.      // computeSum() adds A[0..i-1]
   3.      x[i] = computeSum(A, 0, i-1);
   4. })`

Observations:

- Critical path length, `CPL = O(log n)`
- Total number of operations, `WORK = O(n^2)`
- With `P = O(n)` processors, the best execution time that you can achieve is `T_P = max(CPL, WORK/P) = O(n)`, which is no better than sequential!
How can we do better?

Assume that input array $A = [3, 1, 2, 0, 4, 1, 1, 3]$

Define $\text{scan}(A) =$ exclusive prefix sums of $A = [0, 3, 4, 6, 6, 10, 11, 12]$

Hint:

- Compute $B$ by adding pairwise elements in $A$ to get $B = [4, 2, 5, 4]$
- Assume that we can recursively compute $\text{scan}(B) = [0, 4, 6, 11]$
- How can we use $A$ and $\text{scan}(B)$ to get $\text{scan}(A)$?
Another way of looking at the parallel algorithm

Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.


Approach:

- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an “upward sweep” to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a “downward sweep” to compute prefix sums while reusing partial sum terms stored in upward sweep
Parallel Prefix Sum: Upward Sweep (while calling scan recursively)

Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

1. Receive values from left and right children
2. Compute left+right and store in box
3. Send left+right value to parent

Input array, A: 3 1 2 0 4 1 1 3
Parallel Prefix Sum: Downward Sweep (while returning from recursive calls to scan)

1. Receive value from parent (root receives 0)
2. Send parent’s value to LEFT child (prefix sum for elements to left of left child’s subtree)
3. Send parent’s value + left child’s box value to RIGHT child (prefix sum for elements to left of right child’s subtree)
4. Add A[i] to get inclusive prefix sum
Summary of Parallel Prefix Sum Algorithm

• Critical path length, CPL = $O(\log n)$
• Total number of add operations, WORK = $O(n)$
• Optimal algorithm for $P = O(n/\log n)$ processors
  — Adding more processors does not help
• Parallel Prefix Sum has several applications that go beyond computing the sum of array elements
  • Parallel Prefix Sum can be used for any operation that is associative (need not be commutative)
    — In contrast, finish accumulators required the operator to be both associative and commutative
Given an array input, produce an array output containing only elements such that \( f(\text{elt}) \) is true, i.e., output =
input.parallelStream.filter(f).toArray

Example: input \([17, 4, 6, 8, 11, 5, 13, 19, 0, 24]\)

\[ f: \text{is elt > 10} \]

output \([17, 11, 13, 19, 24]\)

Parallelizable?

—Finding elements for the output is easy

—But getting them in the right place seems hard
Parallel prefix to the rescue

1. Parallel map to compute a **bit-vector** for true elements (can use Java streams)
   
   **input**  \[17, 4, 6, 8, 11, 5, 13, 19, 0, 24\]
   **bits**  \[1, 0, 0, 0, 1, 0, 1, 1, 0, 1\]

2. Parallel-prefix sum on the bit-vector (not available in Java streams)
   
   **bitsum**  \[1, 1, 1, 1, 2, 2, 3, 4, 4, 5\]

3. Parallel map to produce the output (can use Java streams)
   
   **output**  \[17, 11, 13, 19, 24\]

   ```java
   output = new array of size bitsum[n-1]
   FORALL (i=0; i < input.length; i++){
     if(bits[i]==1)
       output[bitsum[i]-1] = input[i];
   }
   ```
## Parallelizing Quicksort

(Remember Homework 1?)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Pick a pivot element</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>2.</td>
<td>Partition all the data into:</td>
<td>$O(n)$</td>
</tr>
<tr>
<td></td>
<td>A. The elements less than the pivot</td>
<td></td>
</tr>
<tr>
<td></td>
<td>B. The pivot</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C. The elements greater than the pivot</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Recursively sort A and C</td>
<td>$2T(n/2)$</td>
</tr>
</tbody>
</table>

### Best / Expected Case Work

- **Step 1**: Pick a pivot element. $O(1)$
- **Step 2**: Partition all the data into:
  - **A**: The elements less than the pivot $O(n)$
  - **B**: The pivot
  - **C**: The elements greater than the pivot
- **Step 3**: Recursively sort A and C $2T(n/2)$

**Simple approach:**
- Work: unchanged at $O(n \log n)$
- Span: now $\text{CPL}(n) = O(n) + \text{CPL}(n/2) = O(n)$
- So parallelism (i.e., work / span) is $O(\log n)$

**Sophisticated approach:**
- Work: unchanged at $O(n \log n)$
- Span: now $\text{CPL}(n) = O(\log n) + \text{CPL}(n/2) = O(\log^2 n)$
- So average parallelism (i.e., work / span) is $O(n / \log n)$
Example

- Step 1: pick pivot as median of three

- Steps 2: implement partition step as two filter/pack operations that store result in a second array

- Step 3: Two recursive sorts in parallel