Lecture 13: Parallelism in Java Streams, Parallel Prefix Sums

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
For the example below, will reordering the five async statements change the meaning of the program (assuming that the semantics of the reader/writer methods depends only on their parameters)? If so, show two orderings that exhibit different behaviors. If not, explain why not.

No, reordering the asyncs doesn’t change the meaning of the program. Regardless of the order, Task 3 will always wait on Task 1. Task 5 will always wait on Task 2. Task 4 will always wait on both Task 1 and 2.

1. `DataDrivenFuture left = new DataDrivenFuture();`
2. `DataDrivenFuture right = new DataDrivenFuture();`
3. `finish {
4.    async await(left) leftReader(left); // Task3
5.    async await(right) rightReader(right); // Task5
6.    async await(left,right)
7.        bothReader(left,right); // Task4
8.    async left.put(leftWriter()); // Task1
9.    async right.put(rightWriter()); // Task2
10. }`
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)
    .mapToInt(a -> a.getCredits()).sum();

3. Complexity of parallelism simplified (for example by replacing stream() by parallelStream())
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, user-written lambda with side effects
- ... but stateless intermediate operations work just fine
  - e.g., filter, map
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 …

```java
// Copy input array A into output array X
X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);

// Update array X with prefix sums
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

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

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
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

Parallel Prefix Sum: Downward Sweep (while returning from recursive calls to scan)
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
Parallel Filter Operation
[Credits: David Walker and Andrew W. Appel (Princeton), Dan Grossman (U. Washington)]

Given an array **input**, produce an array **output** containing only elements such that 
**f(elt)** is true, i.e.,

\[
\text{output} = \text{input}.\text{parallelStream()}.\text{filter(f)}.\text{toArray()}
\]

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

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

**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 \textit{bit-vector} for true elements (can use Java streams)
   \begin{itemize}
   \item \texttt{input} [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
   \item \texttt{bits} [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]
   \end{itemize}

2. Parallel-prefix sum on the bit-vector (not available in Java streams)
   \begin{itemize}
   \item \texttt{bitsum} [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]
   \end{itemize}

3. Parallel map to produce the output (can use Java streams)
   \begin{itemize}
   \item \texttt{output} [17, 11, 13, 19, 24]
   \end{itemize}

\texttt{output = new array of size bitsum[n-1]}
\begin{verbatim}
FORALL (i=0; i < input.length; i++){
  if(bits[i]==1)
    output[bitsum[i]-1] = input[i];
}
\end{verbatim}
Announcements & Reminders

- HW2 is due today by 11:59pm
- Lab #3 due tomorrow by 2pm
- Watch the topic 3.5, 3.6 videos for the next lecture
- Midterm Exam on Thursday, Mar. 11th from 7-9pm in Canvas
Worksheet #13: Parallelism in Java Streams, Parallel Prefix Sums

1. What output will the following Java Streams code print?

   ```java
   Arrays.asList("a1", "a2", "b1", "c2", "c1")
   .parallelStream()
   .filter(s -> s.startsWith("c"))
   .sorted()
   .map(String::toUpperCase)
   .forEach(System.out::println);
   ```

2. Which stream operation in this example could benefit from a parallel prefix sum implementation, and why? (Assume a larger array when answering this question, so that overheads of parallelism are not an issue.)