Lecture 14: Iterative Averaging Revisited, SPMD pattern

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/
1. What output will the following Java Streams code print?
   
   C1 or C2
   C2 or C1

   The sorted() operation is a no-op if the printing is performed in parallel!

2. Which stream operation in this example could benefit from a parallel prefix sum implementation, and why?

   The filter operation since parallel prefix can be used to compute the indices in the output array.

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

Given an array `input`, produce an array `output` containing only elements such that `f(elt)` is true, i.e., `output = input.parallelStream.filter(f).toArray`

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

`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
Recap: 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];
   }
   ```
One-Dimensional Iterative Averaging Revisited

- Initialize a one-dimensional array of \((n+2)\) double’s with boundary conditions, \(myVal[0] = 0\) and \(myVal[n+1] = 1\).
- In each iteration, each interior element \(myVal[i]\) in \(1..n\) is replaced by the average of its left and right neighbors.
  - Two separate arrays are used in each iteration, one for old values and the other for the new values.
- After a sufficient number of iterations, we expect each element of the array to converge to \(myVal[i] = (myVal[i-1]+myVal[i+1])/2\), for all \(i\) in \(1..n\).

Illustration of an intermediate step for \(n = 8\) (source: Figure 6.19 in Lin-Snyder book)
Iterative Averaging structure is akin to that of a Finite Difference solution to the One-Dimensional Heat Equation

- In the 1D heat equation, $u(j,t)$ represents the temperature at position $j$ in a 1D object at $n^{th}$ time step (different use of “n” from our code).

- Recall that for a “properly-behaved” function $f$, and a sufficiently small $h$, we can approximate the first derivative of $f$ as follows:
  \[ f'(a) \approx \frac{f(a + h) - f(a)}{h} \]

- We can use this principle in both space and time to compute
  - $u(j,n+1) = f( u(j-1,n), u(j,n), u(j+1,n) )$

  for multiple time steps $n$ until we reach a convergence with an acceptably small error between time steps.

- These algorithms are also referred to as “stencil codes”


- See also: [https://en.wikipedia.org/wiki/Heat_equation](https://en.wikipedia.org/wiki/Heat_equation)
HJ code for One-Dimensional Iterative Averaging using nested forseq-forall structure (Lecture 11)

1. double[] myVal=new double[n+2]; myVal[n+1] = 1;
2. double[] myNew=new double[n+2]; myNew[n+1] = myVal[n+1];
3. forseq(0, m-1, (iter) -> {
   // Compute MyNew as function of input array MyVal
   forall(1, n, (j) -> { // Create n tasks
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
   }); // forall
   // Swap myVal & myNew;
   temp=myVal; myVal=myNew; myNew=temp;
   // myNew becomes input array for next iteration
}); // for
Converting forseq-forall version (Slide 7) into a forall-forseq version with barriers

1. double[] gVal=new double[n+2]; gVal[n+1] = 1;
2. double[] gNew=new double[n+2];
3. forallPhased(1, n, (j) -> {
   // Create n tasks
   // Initialize myVal and myNew as local pointers
   double[] myVal = gVal; double[] myNew = gNew;
   forseq(0, m-1, (iter) -> {
      // Compute MyNew as function of input array MyVal
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
      next(); // Barrier before next iteration of iter loop
      // Swap local pointers, myVal and myNew
      double[] temp=myVal; myVal=myNew; myNew=temp;
      // myNew becomes input array for next iteration
   }); // forseq
   }); // forall
General Approach for Iteration Grouping (Loop Chunking)

Without chunking:

```java
for (iter : [0:iterations-1]) {
    forall (j : [1:n])
        myNew[j] = (myVal[j-1] + myVal[j+1])/2;
    Swap myNew and myVal
}
```

With chunking (replace “forall” by “forall-for”):

```java
for (iter : [0:iterations-1]) {
    forall (g : [0:ng-1])
        forall (j : myGroup(g,[1:n],ng)
            myNew[j] = (myVal[j-1] + myVal[j+1])/2;
        Swap myNew and myVal
}
```
Example: HJ code for One-Dimensional Iterative Averaging with Slide 7 code w/ chunking (Lecture 11)

1. double[] myVal = new double[n+2]; myVal[n+1] = 1;
2. double[] myNew = new double[n+2]; myNew[n+1] = myVal[n+1];
3. int nc = numWorkerThreads();
4. forseq(0, m-1, (iter) -> {
   // Compute MyNew as function of input array MyVal
5.   forallChunked(1, n, n/nc, (j) -> {
         // Create nc tasks
6.       myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7.     }); // forallChunked
9.   temp = myVal; myVal = myNew; myNew = temp; // Swap myVal & myNew;
10.  // myNew becomes input array for next iteration
11. }); // for
Barrier-based solution:

// Note that iter-loop is inserted between forall-g and for-j loops
forall (g : [0:ng-1])
  for (iter : [0:iterations-1]) {
    for(j : myGroup(g, [1:n], ng))
      myNew[j] = (myVal[j-1] + myVal[j+1])/2;
      next; // Barrier
    Swap myNew and myVal
  } // for iter

Also referred to as a “single program multiple data” (SPMD) pattern
Chunking the forall loop in Slide 8 — inner chunked loop goes inside forseq-iter loop

1. double[] gVal = new double[n+2]; gVal[n+1] = 1;
2. double[] gNew = new double[n+2];
3. HjRegion1D iterSpace = new RectangularRegion1D(1,n);
4. int nc = numWorkerThreads();
5. forallPhased(0, nc-1, (jj) -> {
   // Create nc tasks
   // Initialize myVal and myNew as local pointers
   double[] myVal = gVal; double[] myNew = gNew;
   forseq(0, m-1, (iter) -> {
      forseq(myGroup(jj, iterSpace, nc), (j) -> {
         // Compute MyNew as function of input array MyVal
         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
      }); // forseq
      next(); // Barrier before executing next iteration of iter loop
   }); // forseq
   // Swap local pointers, myVal and myNew
   double[] temp = myVal; myVal = myNew; myNew = temp;
   // myNew becomes input array for next iter
}); // forall
Single Program Multiple Data (SPMD)

Basic idea

- Run the same code (program) on P workers
- Use the “rank” --- an ID ranging from 0 to (P-1) --- to determine what data is processed by which worker
  - Hence, “single-program” and “multiple-data”
  - Rank is equivalent to index in a top-level “forall (point[i] : [0:P-1])” loop
- Lower-level programming model than dynamic async/finish parallelism
  - Programmer’s code is essentially at the worker level (each forall iteration is like a worker), and work distribution is managed by programmer by using barriers and other synchronization constructs
  - Harder to program but can be more efficient for restricted classes of applications (e.g. for OneDimAveraging, but not for NQueens)
- Convenient for hardware platforms that are not amenable to efficient dynamic task parallelism
  - General-Purpose Graphics Processing Unit (GPGPU) accelerators
  - Distributed-memory parallel machines
Slide 12 viewed as exemplar of SPMD pattern

1. `double[] gVal = new double[n+2]; gVal[n+1] = 1;`
2. `double[] gNew = new double[n+2];`
3. `HjRegion1D iterSpace = newRectangularRegion1D(1,m);`
4. `int nc = numWorkerThreads();`
5. `forallPhased(1, nc, (jj) -> { // Create nc tasks`

6.   // Initialize myVal and myNew as local pointers
7.   double[] myVal = gVal; double[] myNew = gNew;
8.   forseq(0, m-1, (iter) -> {
9.       forseq(myGroup(jj, iterSpace, nc), (j) -> {
10.          // Compute MyNew as function of input array MyVal
11.          myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
12.       }); // forseq
13.   next(); // Barrier before executing next iteration of iter loop
14.   // Swap local pointers, myVal and myNew
15.   double[] temp=myVal; myVal=myNew; myNew=temp;
16.   // myNew becomes input array for next iter
17. }); // forseq

Instead of async-finish, this SPMD version creates one task per worker, uses myGroup() to distribute work, and use barriers to synchronize workers.