3.1 Lecture Summary

3 Loop Parallelism

3.1 Parallel Loops

Lecture Summary: In this lecture, we learned different ways of expressing parallel loops. The most general way is to think of each iteration of a parallel loop as an *async* task, with a *finish* construct encompassing all iterations. This approach can support general cases such as parallelization of the following pointer-chasing while loop (in pseudocode):

```
1  finish {
2  for (p = head; p != null ; p = p.next) async compute(p);
3  }
```

However, further efficiencies can be gained by paying attention to *counted-for* loops for which the number of iterations is known on entry to the loop (before the loop executes its first iteration). We then learned the *forall* notation for expressing parallel counted-for loops, such as in the following vector addition statement (in pseudocode):

```
1  forall (i : [0:n-1]) a[i] = b[i] + c[i]
```

We also discussed the fact that Java streams can be an elegant way of specifying parallel loop computations that produce a single output array, e.g., by rewriting the vector addition statement as follows:

```
1  a = IntStream.rangeClosed(0, N-1).parallel().toArray(i -> b[i] + c[i]);
```

In summary, streams are a convenient notation for parallel loops with at most one output array, but the *forall* notation is more convenient for loops that create/update multiple output arrays, as is the case in many scientific computations. For generality, we will use the *forall* notation for parallel loops in the remainder of this module.

Optional Reading:

1. Tutorial on Executing Streams in Parallel