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):

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

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):

```
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:

```
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

3.2 Lecture Summary

---

3 Loop Parallelism

3.2 Parallel Matrix Multiplication

In this lecture, we reminded ourselves of the formula for multiplying two $n \times n$ matrices, a and b, to obtain a product matrix, c, of size $n \times n$:

$c[i][j] = \sum_{k=0}^{n-1} a[i][k] * b[k][j]$

This formula can be easily translated to a simple sequential algorithm for matrix multiplication as follows (with pseudocode for counted-for loops):

```
for(i : [0:n-1]) {
  for(j : [0:n-1]) { c[i][j] = 0;
    for(k : [0:n-1]) {
      c[i][j] = c[i][j] + a[i][k]*b[k][j]
    }
  }
}
```

The interesting question now is: which of the for-i, for-j and for-k loops can be converted to forall loops, i.e., can be executed in parallel? Upon a close inspection, we can see that it is safe to convert for-i and for-j into *forall* loops, but for-k must remain a sequential loop to avoid data races. There are some trickier ways to also exploit parallelism in the for-k loop, but they rely on the observation that summation is algebraically associative even though it is computationally non-associative.

Optional Reading:

1. Wikipedia article on Matrix multiplication algorithm

3.3 Lecture Summary

---

3 Loop Parallelism

3.3 Barriers in Parallel Loops

Lecture Summary: In this lecture, we learned the *barrier* construct through a simple example that began with the following *forall* parallel loop (in pseudocode):

```
forall (i : [0:n-1]) {
    myId = lookup(i); // convert int to a string
    print HELLO, myId;
    print BYE, myId;
}
```

We discussed the fact that the HELLO's and BYE's from different *forall* iterations may be interleaved in the printed output, e.g., some HELLO's may follow some BYE's. Then, we showed how inserting a barrier between the two print statements could ensure that all HELLO's would be printed before any BYE's.

Thus, barriers extend a parallel loop by dividing its execution into a sequence of *phases*. While it may be possible to write a separate *forall* loop for each phase, it is both more convenient and more efficient to instead insert barriers in a single *forall* loop, e.g., we would need to create an intermediate data structure to communicate the myId values from one *forall* to another *forall* if we split the above *forall* into two (using the notation *next*) loops. Barriers are a fundamental construct for parallel loops that are used in a majority of real-world parallel applications.

3.4 Lecture Summary

---

3 Loop Parallelism

3.4 One-Dimensional Iterative Averaging

Lecture Summary: In this lecture, we discussed a simple *stencil* computation to solve the recurrence, Xi= (Xi−1 + Xi+1)/2 with boundary conditions, X0 = 0 and Xn= 1. Though we can easily derive an analytical solution for this example, (Xi= i/n), most stencil codes in practice do not have known analytical solutions and rely on computation to obtain a solution.

The Jacobi method for solving such equations typically utilizes two arrays, oldX[] and newX[]. A naive approach to parallelizing this method would result in the following pseudocode:

```
for (iter: [0:nsteps-1]) {
  forall (i: [1:n-1]) {
    newX[i] = (oldX[i-1] + oldX[i+1]) / 2;
  }
  swap pointers newX and oldX;
}
```

Though easy to understand, this approach creates $nsteps \times (n - 1)$ tasks, which is too many. Barriers can help reduce the number of tasks created as follows:

```
forall ( i: [1:n-1]) {
  localNewX = newX; localOldX = oldX;
  for (iter: [0:nsteps-1]) {
    localNewX[i] = (localOldX[i-1] + localOldX[i+1]) / 2;
```

```
    NEXT; // Barrier
    swap pointers localNewX and localOldX;
  }
}
```

In this case, only ($n - 1$) tasks are created, and there are *nsteps* barrier (*next*) operations, each of which involves all ($n - 1$) tasks. This is a significant improvement since creating tasks is usually more expensive than performing barrier operations.

Optional Reading:

1. Wikipedia article on Stencil codes

3.5 Lecture Summary

---

3 Loop Parallelism

3.5 Iteration Grouping: Chunking of Parallel Loops

Lecture Summary: In this lecture, we revisited the vector addition example:

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

We observed that this approach creates *n* tasks, one per *forall* iteration, which is wasteful when (as is common in practice) *n* is much larger than the number of available processor cores.

To address this problem, we learned a common tactic used in practice that is referred to as *loop chunking* or *iteration grouping*, and focuses on reducing the number of tasks created to be closer to the number of processor cores, so as to reduce the overhead of parallel execution:

With iteration grouping/chunking, the parallel vector addition example above can be rewritten as follows:

```
forall (g:[0:ng-1])
  for (i : mygroup(g, ng, [0:n-1])) a[i] = b[i] + c[i]
```

Note that we have reduced the degree of parallelism from *n* to the number of groups, *ng*, which now equals the number of iterations/tasks in the *forall* construct.

There are two well known approaches for iteration grouping: *block*and *cyclic*. The former approach (*block*) maps consecutive iterations to the same group, whereas the latter approach (*cyclic*) maps iterations in the same congruence class (mod *ng*) to the same group. With these concepts, you should now have a better understanding of how to execute *forall* loops in practice with lower overhead.

For convenience, the PCDP library provides helper methods, *forallChunked()* and *forall2dChunked()*, that automatically create one-dimensional or two-dimensional parallel loops, and also perform a block-

style iteration grouping/chunking on those parallel loops. An example of using the *forall2dChunked()* API for a two-dimensional parallel loop (as in the matrix multiply example) can be seen in the following Java code sketch:

```
forall2dChunked(0, N - 1, 0, N - 1, (i, j) -> {
        . . . // Statements for parallel iteration (i,j)

    });
```

Mini Project 3: Parallelizing Matrix-Matrix Multiply Using Loop Parallelism

---

miniproject_3.zip

Project Goals & Outcomes

The aim of this project is intended to test your ability to exploit parallelism in loops. The provided code includes a sequential program for multiplying two matrices. Your task is to write a correct parallel version of the program that runs faster than the sequential version.

In lectures this week, you learned about expressing parallelism across loop iterations. Sequential loops are common targets for parallelization for two reasons. First, because loop bodies are often repeated many times, the majority of execution time of sequential programs is often spent inside of one or more loops. Second, because loops express the same computation repeated across many points in the loop iteration space, parallelism arises naturally as long as those iterations are independent of each other. Of course, it is not always legal to parallelize a loop as data races may result.

More specifically, this week covered two ways of expressing loop parallelism in Java: 1) PCDP's forall methods, and 2) parallel streams in standard Java. In this mini-project, you will get hands-on experience with PCDP's forall methods by parallelizing matrix-matrix multiply. Before getting started with this mini-project, it is recommended you review the first demo video in Week 3 in which Professor Sarkar walks through an example similar to this project.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the provided project files using the gray button labeled miniproject_3.zip at the top of this description, you should see the project source code at:

miniproject_3/src/main/java/edu/coursera/parallel/MatrixMultiply.java

and the project tests at

miniproject_3/src/test/java/edu/coursera/parallel/MatrixMultiplyTest.java

Project Instructions

Your modifications should be done entirely inside of MatrixMultiply.java. You may not make any changes to the signatures of any public or protected methods inside of MatrixMultiply, or remove any of them. However, you are free to add any new methods you like. Any changes you make to MatrixMultiplyTest.java will be ignored in the final grading process. As in past mini-projects in this course, the provided code is a fully functioning Maven project but you are not required to use Maven if you prefer since we have also provided the necessary JARs to use an alternate approach to build and execute your code

Your main goals for this assignment are as follows (note that MatrixMultiply.java also contains helpful TODOs):

Modify the MatrixMultiply.parMatrixMultiply method to implement matrix multiply in parallel using PCDP's forall or forallChunked methods. This will closely follow the demo by Prof. Sarkar in the first demo video of Week 3. There is one TODO in MatrixMultiply.parMatrixMultiply to help indicate the changes to be made. A parallel implementation of MatrixMultiply.parMatrixMultiply should result in near-linear speedup (i.e. the speedup achieved should be close to the number of cores in your machine).

Project Evaluation

Your assignment submission should consist of only the MatrixMultiply.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera’s auto-grading system using 2 and 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the “Score” column of the “My submission” tab based on the following rubric:

- 20% – correctness and performance of the parallel matrix multiply implementation on a 512x512 matrix using 2 cores
- 30% – correctness and performance of the parallel matrix multiply implementation on a 512x512 matrix using 4 cores
- 20% – correctness and performance of the parallel matrix multiply implementation on a 768x768 matrix using 2 cores
- 30% – correctness and performance of the parallel matrix multiply implementation on a 768x768 matrix using 4 cores