1.1 Lecture Summary

---

1 Task-level Parallelism

1.1 Task Creation and Termination (Async, Finish)

Lecture Summary: In this lecture, we learned the concepts of *task creation* and *task termination* in parallel programs, using array-sum as an illustrative example. We learned the *async* notation for task creation: "*async ⟨stmt1⟩*", causes the parent task (*i.e.*, the task executing the async statement) to create a new child task to execute the body of the *async, ⟨stmt1⟩, asynchronously* (*i.e.*, before, after, or in parallel) with the remainder of the parent task. We also learned the *finish* notation for task termination: "*finish ⟨stmt2⟩*" causes the parent task to execute *⟨stmt2⟩*, and then wait until *⟨stmt2⟩* and all *async* tasks created within *⟨stmt2⟩* have completed. Async and finish constructs may be arbitrarily nested.

The example studied in the lecture can be abstracted by the following pseudocode:

```
finish {
  async S1; // asynchronously compute sum of the lower half of the array
  S2;       // compute sum of the upper half of the array in parallel with S1
}
S3; // combine the two partial sums after both S1 and S2 have finished
```

While *async* and *finish* notations are useful algorithmic/pseudocode notations, we also provide you access to a high-level open-source Java-8 library called PCDP (for Parallel, Concurrent, and Distributed Programming), for which the source code is available at https://github.com/habanero-rice/pcdp. PCDP contains APIs (application programming interfaces) that directly support *async* and *finish* constructs so that you can use them in real code as well. In the next lecture, you will learn how to to implement the *async* and *finish* functionality using Java's standard Fork/Join (FJ) framework.

Optional Reading:

1. Wikipedia article on Asynchronous method invocation

1.2 Lecture Summary

---

1 Task-level Parallelism

1.2 Creating Tasks in Java's Fork/Join Framework

Lecture Summary: In this lecture, we learned how to implement the *async* and *finish* functionality using Java's standard Fork/Join (FJ) framework. In this framework, a task can be specified in the compute() method of a user-defined class that extends the standard RecursiveAction class in the FJ framework. In our Array Sum example, we created class ASum with fields A for the input array, LO and HI for the subrange for which the sum is to be computed, and SUM for the result for that subrange. For an instance of this user-defined class (e.g., L in the lecture), we learned that the method call, L.fork(), creates a new task that executes L's compute() method. This implements the functionality of the *async* construct that we learned earlier. The call to L.join() then waits until the computation created by L.fork() has completed. Note that join() is a lower-level primitive than *finish* because join() waits for a specific task, whereas *finish* implicitly waits for all tasks created in its scope. To implement the *finish* construct using join() operations, you have to be sure to call join() on every task created in the finish scope.

A sketch of the Java code for the ASum class is as follows:

```
private static class ASum extends RecursiveAction {
  int[] A; // input array
  int LO, HI; // subrange
  int SUM; // return value
  . . .
  @Override
  protected void compute() {
    SUM = 0;
    for (int i = LO; i <= HI; i++) SUM += A[i];
  } // compute()
}
```

FJ tasks are executed in a ForkJoinPool, which is a pool of Java threads. This pool supports the invokeAll() method that combines both the fork and join operations by executing a set of tasks in parallel, and waiting for their completion. For example, ForkJoinTask.invokeAll(left,right) implicitly performs fork() operations on left and right, followed by join() operations on both objects.

Optional Reading:

1. Tutorial on Java's Fork/Join framework
2. Documentation on Java's RecursiveAction class

1.3 Lecture Summary

---

1 Task-level Parallelism

1.3 Computation Graphs, Work, Span, Ideal Parallelism

Lecture Summary: In this lecture, we learned about Computation Graphs (CGs), which model the execution of a parallel program as a partially ordered set. Specifically, a CG consists of:

- A set of *vertices* or *nodes*, in which each node represents a *step* consisting of an arbitrary sequential computation.
- A set of *directed edges* that represent ordering constraints among steps.

For *fork–join* programs, it is useful to partition the edges into three cases:

1. *Continue* edges that capture sequencing of steps within a task.

2. *Fork* edges that connect a fork operation to the first step of child tasks.

3. *Join* edges that connect the last step of a task to all *join* operations on that task.

CGs can be used to define *data races*, an important class of bugs in parallel programs. We say that a data race occurs on location *L* in a computation graph, *G*, if there exist steps *S*1 and *S*2 in *G* such that there is no path of directed edges from *S*1 to *S*2 or from *S*2 to *S*1 in *G*, and both *S*1 and *S*2 read or write *L* (with at least one of the accesses being a write, since two parallel reads do not pose a problem).

CGs can also be used to reason about the *ideal parallelism* of a parallel program as follows:

- Define *WORK(G)* to be the sum of the execution times of all nodes in CG *G*,
- Define *SPAN(G)* to be the length of a longest path in *G*, when adding up the execution times of all nodes in the path. The longest paths are known as *critical paths*, so *SPAN*also represents the *critical path length* (CPL) of *G*.

Given the above definitions of *WORK* and *SPAN*, we define the *ideal parallelism* of Computation Graph *G* as the ratio, *WORK(G)/SPAN(G)*. The ideal parallelism is an upper limit on the speedup factor that can be obtained from parallel execution of nodes in computation graph *G*. Note that ideal parallelism is only a function of the parallel program, and does not depend on the actual parallelism available in a physical computer.

Optional Reading:

1. Wikipedia article on Analysis of parallel algorithms

1.4 Lecture Summary

---

1 Task-level Parallelism

1.4 Multiprocessor Scheduling, Parallel Speedup

Lecture Summary: In this lecture, we studied the possible executions of a Computation Graph (CG) on an idealized parallel machine with *P* processors. It is idealized because all processors are assumed to

be identical, and the execution time of a node is assumed to be fixed, regardless of which processor it executes on. A *legalschedule* is one that obeys the dependence constraints in the CG, such that for every directed edge (*A, B*), the schedule guarantees that step *B* is only scheduled after step *A* completes. Unless otherwise specified, we will restrict our attention in this course to schedules that have no *unforced idleness*, i.e., schedules in which a processor is not permitted to be idle if a CG node is available to be scheduled on it. Such schedules are also referred to as "greedy" schedules.

We defined T_P as the execution time of a CG on P processors, and observed that

$T_\infty \leq T_P \leq T_1$

We also saw examples for which there could be different values of T_P for different schedules of the same CG on P processors.

We then defined the parallel speedup for a given schedule of a CG on P processors as $Speedup(P) = T_1/T_P$, and observed that $Speedup(P)$ must be ≤ the number of processors P , and also ≤ the ideal parallelism, WORK/SPAN.

1.5 Lecture Summary

---

1 Task-level Parallelism

1.5 Amdahl's Law

Lecture Summary: In this lecture, we studied a simple observation made by Gene Amdahl in 1967: if $q \leq 1$ is the fraction of *WORK* in a parallel program that must be executed *sequentially*, then the best speedup that can be obtained for that program for any number of processors, P , is $Speedup(P) \leq 1/q$.

This observation follows directly from a lower bound on parallel execution time that you are familiar with, namely $T_P \geq SPAN(G)$. If fraction q of $WORK(G)$ is sequential, it must be the case that $SPAN(G) \geq q \times WORK(G)$. Therefore, $Speedup(P) = T_1/T_P$ must be $\leq WORK(G)/(q \times WORK(G)) = 1/q$ since $T_1 = WORK(G)$ for greedy schedulers.

Amdahl's Law reminds us to watch out for sequential bottlenecks both when designing parallel algorithms and when implementing programs on real machines. As an example, if $q = 10\%$, then Amdahl's Law reminds us that the best possible speedup must be ≤ 10 (which equals $1/q$), regardless of the number of processors available.

Optional Reading:

1. Wikipedia article on Amdahl's law.

Mini Project 1: Reciprocal-Array-Sum using the Java Fork/Join Framework

---

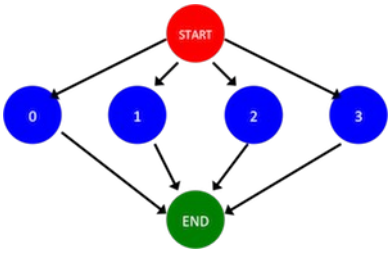
miniproject_1.zip

Project Goals and Outcomes

You have learned how to express parallel algorithms by creating asynchronous tasks (async) , and waiting on collections of tasks (finish). In this mini-project, we will use the Java Fork Join framework to write a parallel implementation of the Reciprocal Array Sum computation. Before starting on this mini-project, it is recommended you review the second demo video in Module 1 in which Professor Sarkar walks through an example similar to this project.

Computing the reciprocal array sum of a vector or array involves adding the reciprocals of all elements of the array. The reciprocal of a value v is simply 1/v. The pseudocode below illustrates how you might sequentially compute the reciprocal sum of an array A:

It should be clear that the computation of a reciprocal in each iteration of the above for-loop is independent of the reciprocal in any other iteration, and so this problem can be easily parallelized. In particular, the parallel computation graph of this problem can be visualized as follows for a 4-element array:



where the red circle indicates the start of the parallel program, the green node indicates the end, and the blue nodes each represent an iteration of the for loop.

The goal of this mini--project is to use the Java Fork Join framework to parallelize the provided sequential implementation of Reciprocal Array Sum.

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 project files using the gray button labeled miniproject_1.zip at the top of this description, you should see the project source code file in

miniproject_1/src/main/java/edu/coursera/parallel/ReciprocalArraySum.java

and the project tests in

miniproject_1/src/test/java/edu/coursera/parallel/ReciprocalArraySumTest.java

Project Instructions

Your modifications should be made entirely inside of ReciprocalArraySum.java. You should not change the signatures of any public or protected methods inside of ReciprocalArraySum, but you can edit the method bodies and add any new methods that you choose. We will use our copy of ReciprocalArraySumTest.java in the final grading process, so do not change that file or any other file except ReciprocalArraySum.java.

Your main goals for this assignment are as follows:

1. Modify the ReciprocalArraySum.parArraySum() method to implement the reciprocal-array-sum computation in parallel using the Java Fork Join framework by partitioning the input array in half and computing the reciprocal array sum on the first and second half in parallel, before combining the results. There are TODOs in the source file to guide you, and you are free to refer to the lecture and demonstration videos.

Note that Goal #2 below is a generalization of this goal, so If you are already confident in your understanding of how to partition work among multiple Fork Join tasks (not just two), you are free to skip ahead to Goal #2 below and then implement parArraySum() for Goal #1 by calling the parManyTaskArraySum() method that you will implement for Goal #2 by requesting only two tasks. Otherwise, it is recommended that you work on Goal #1 first, and then proceed to Goal #2.

Note that to complete this and the following goal you will want to explicitly create a ForkJoinPool inside of parArraySum() and parManyTaskArraySum() to run your tasks inside. For example, creating a ForkJoinPool with 2 threads requires the following code:

2. Modify the ReciprocalArraySum.parManyTaskArraySum method to implement the reciprocal-array-sum computation in parallel using Java's Fork Join framework again, but using a given number of tasks (not just two). Note that the getChunkStartInclusive and getChunkEndExclusive utility methods are provided for your convenience to help with calculating the region of the input array a certain task should process.

Project Evaluation

Your assignment submission should consist of only the ReciprocalArraySum.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:

- 10% - performance of the two-task parallel version on two cores processing 1,000,000 elements
- 20% – performance of the two-task parallel version on two cores processing 100,000,000 elements
- 20% – performance of the many-task parallel version on two cores processing 1,000,000 elements
- 20% – performance of the many-task parallel version on two cores processing 100,000,000 elements
- 30% – performance of the many-task parallel version on four cores processing 100,000,000 elements