coursera

4.5 Lecture Summary

4 Dataflow Synchronization and Pipelining

4.5 Data Flow Parallelism

Lecture Summary: Thus far, we have studied computation graphs as structures that are derived from parallel programs. In this lecture, we studied a dual approach advocated in the *data flow parallelism* model, which is to specify parallel programs as computation graphs. The simple data flow graph studied in the lecture consisted of five nodes and four edges: $A \rightarrow C, A \rightarrow D, B \rightarrow D, B \rightarrow E$. While futures can be used to generate such a computation graph, e.g., by including calls to A.get() and B.get() in task D, the computation graph edges are implicit in the get() calls when using futures. Instead, we introduced the asyncAwait notation to specify a task along with an explicit set of preconditions (events that the task must wait for before it can start execution). With this approach, the program can be generated directly from the computation graph as follows:

```
async( () -> {/* Task A */; A.put(); } ); // Complete task and trigger event A
async( () -> {/* Task B */; B.put(); } ); // Complete task and trigger event B
asyncAwait(A, () -> {/* Task C */} );     // Only execute task after event A is
  triggered
asyncAwait(A, B, () -> {/* Task D */} );    // Only execute task after events A,
  B are triggered
asyncAwait(B, () -> {/* Task E */} );     // Only execute task after event B is
  triggered
```

Interestingly, the order of the above statements is not significant. Just as a graph can be defined by enumerating its edges in any order, the above data flow program can be rewritten as follows, without changing its meaning:

```
asyncAwait(A, () -> {/* Task C */} );     // Only execute task after event A is
  triggered
asyncAwait(A, B, () -> {/* Task D */} );    // Only execute task after events A,
  B are triggered
asyncAwait(B, () -> {/* Task E */} );     // Only execute task after event B is
  triggered
async( () -> {/* Task A */; A.put(); } ); // Complete task and trigger event A
async( () -> {/* Task B */; B.put(); } ); // Complete task and trigger event B
```

Finally, we observed that the power and elegance of data flow parallel programming is accompanied by the possibility of a lack of progress that can be viewed as a form of "deadlock" if the program omits a put() call for signalling an event.