COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 25: Dataflow Programming and Data-Driven Futures

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments for Today’s Lecture

- Lecture 24 handout
- Slides from Prof. Guang Gao, U. Delaware — Topic-III-2-dataflow.pptx
Announcements

- HW5 submission deadline is 5pm TODAY
Dataflow Computing

- Basic idea: replace machine instructions by a small set of dataflow operators
Figure 1: Example instruction sequence and its dataflow graph

An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators.

No separate control flow
Extending Futures with Dataflow Principles: HJ Data-Driven Futures (DDFs)

ddfA = new DataDrivenFuture()

• Allocate an instance of a DDF object (container)

async await(ddfA, ddfB, ...) <Stmt>

• Create a new async task to start executing Stmt after all of ddfA, ddfB, ... become available

• Task is said to be enabled when ddfA, ddfB, ... become available

ddfA.put(V)

• Store object V in ddfA, thereby making ddfA available

• Single-assignment rule: at most one put is permitted on a given DDF

ddfA.get()

• Return value stored in ddfA

• Can only be performed by async’s that contain ddfA in their await clause (no blocking is necessary)
Figure 2: Example Habanero Java code fragment with Data-Driven Futures

```java
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
    async left.put(leftBuilder()); // Task1
    async right.put(rightBuilder()); // Task2
    async await (left) leftReader(left); // Task3
    async await (right) rightReader(right); // Task4
    async await (left, right)
        bothReader(left, right); // Task5
}
```
// Assume that left and right are fields in this object

finish {
    async left = put(leftBuilder()); // Task1
    async right = put(rightBuilder()); // Task2
}

finish {
    async leftReader(left); // Task3
    async rightReader(right); // Task4
    async bothReader(left, right); // Task5
}
Two Exception cases for DDFs

• Case 1: If two put's are attempted on the same DDF, an exception is thrown because of the violation of the single-assignment rule

• Case 2: If a get is attempted by a task on a DDF that was not in the task's await list, then an exception is thrown because DDF's do not support blocking gets.
Differences between Futures and DDFs

- Consumer task blocks on `get()` for each future that it reads, where as async-await does not start execution till all DDFs are available
- Producer task can only write to a single future object, where as a DDF task can write to multiple DDF objects
- The choice of which future object to write to is tied to a future task at creation time, where as the choice of output DDF can be deferred to any point with a DDF task
- Future tasks cannot deadlock, but it is possible for a DDF task to never be enabled, if one of its input DDFs never becomes available. This can be viewed as a special case of deadlock.
  - This deadlock case is resolved by ensuring that each finish construct moves past the end-finish when all enabled async tasks in its scope have terminated, thereby ignoring any remaining non-enabled async tasks.
Implementing Future Tasks using DDFs

- **Future version**
  ```java
  final future<int> f = async<int> { return g(); };
  ...
  int local = f.get();
  ```

- **DDF version**
  ```java
  DataDrivenFuture f = new DataDrivenFuture();
  async { f.put(g()); };
  ...
  async await (f) { int local = f.get(); };
  ```
Listing 1: use of DDFs with empty objects

```java
finish {
    DataDrivenFuture ddfA = new DataDrivenFuture();
    DataDrivenFuture ddfB = new DataDrivenFuture();
    DataDrivenFuture ddfC = new DataDrivenFuture();
    DataDrivenFuture ddfD = new DataDrivenFuture();
    DataDrivenFuture ddfE = new DataDrivenFuture();
    async { ... ; ddfA.put("" ); } // Task A
    async await(ddfA) { ... ; ddfB.put("" ); } // Task B
    async await(ddfA) { ... ; ddfC.put("" ); } // Task C
    async await(ddfB, ddfC) { ... ; ddfD.put("" ); } // Task D
    async await(ddfC) { ... ; ddfE.put("" ); } // Task E
    async await(ddfD, ddfE) { ... } // Task F
} // finish
```

Diagram:
```
A ---- B ---- D
      |      |
      |      v
      C ---- E
            |
            v
            F
```
Using Future Tasks to generate Computation Graph CG3 from Homework 2

// NOTE: return statement is optional when return type is void

final future<void> A = async<void>{ . . . ; return;}

final future<void> B = async<void>
{ A.get(); . . . ; return;}

final future<void> C = async<void>
{ A.get(); . . . ; return;}

final future<void> D = async<void>
{ B.get(); C.get(); . . . ; return;}

final future<void> E = async<void>
{ C.get(); . . . ; return;}

final future<void> F = async<void>
{ D.get(); E.get(); . . . ; return;}

Computation Graph CG3