COMP 322: Fundamentals of Parallel Programming

Lecture 14: Data-Driven Tasks and Data-Driven Futures

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

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

Dataflow Computing

- Original idea: replace machine instructions by a small set of dataflow operators

Fork

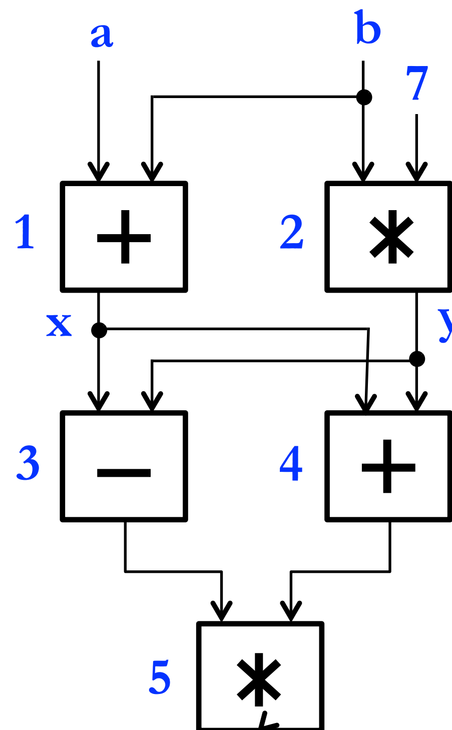
Primitive Ops

Switch

Merge

Example instruction sequence and its dataflow graph

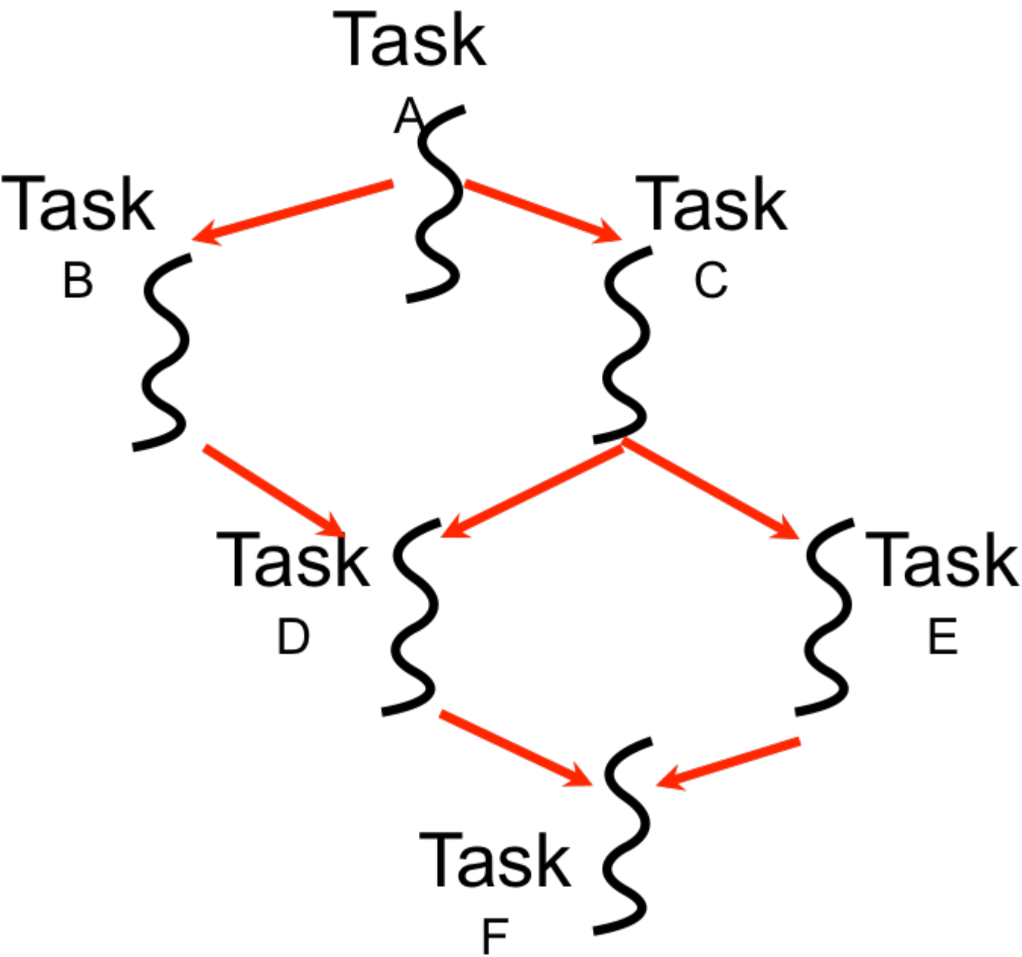
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



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

No separate control flow

Macro-Dataflow Programming



Communication via "single-assignment" variables

- "Macro-dataflow" = extension of dataflow model from instruction-level to task-level operations
- General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
    - Static dataflow ==> graph fixed when program execution starts
    - Dynamic dataflow ==> graph can grow dynamically
- Semantic guarantees: race-freedom, determinism
    - Deadlocks are possible due to unavailable inputs (but they are deterministic)

Extending HJ Futures for Macro-Dataflow: Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)

```
HjDataDrivenFuture<T1> ddfA = newDataDrivenFuture();
```

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1
- Used to implement "edges" in a computation graph

```
asyncAwait(ddfA, ddfB, …, () -> Stmt);
```

- Create a new data-driven-task to start executing Stmt after all of ddfA, ddfB, … become available (i.e., after task becomes "enabled")
- Used to implement "nodes" in a computation graph

```
ddfA.put(V) ;
```

- Store object V (of type T1) in ddfA, thereby making ddfA available
- Single-assignment rule: at most one put is permitted on a given DDF

ddfA.get()

- Return value (of type T1) stored in ddfA
- Throws an exception if put() has not been performed
  - Should be performed by async's that contain ddfA in their await clause, or if there's some other synchronization to guarantee that the put() was performed

Implementing Future Tasks using DDFs

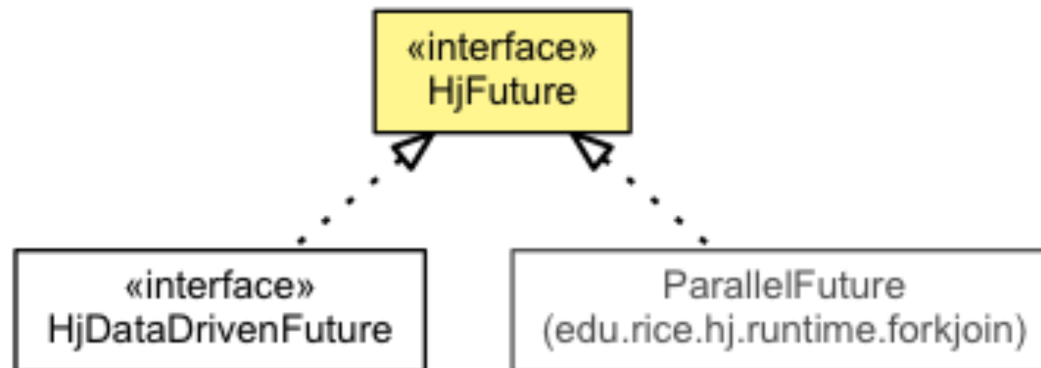
- **Future version**

```
1. final HjFuture<T> f = future(() -> { return g(); });
2. S1
3. async(() -> {
4.   ... = f.get();
5.   S2;
6.   S3;
7. });
```

- **DDF version**

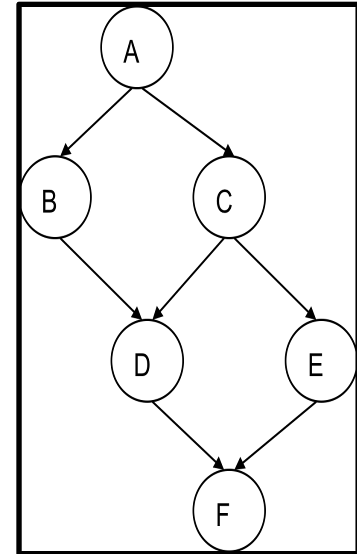
```
1. HjDataDrivenFuture<T> f = newDataDrivenFuture();
2. async(() -> { f.put(g()) });
3. S1
4. asyncAwait(f, () -> {
5.   ... = f.get();
6.   S2;
7.   S3;
8. });
```

HjFutures and HjDataDrivenFuture



- future.get()
  - Returns the value wrapped in the future.
- future.resolved()
  - Returns whether the future has been resolved, i.e. the value has been computed.
    - WARNING: use of resolved() can introduce nondeterminism

Use of DDFs with dummy objects (like future<Void>)




```
1. finish(() -> {
2.   HjDataDrivenFuture<Void> ddfA = newDataDrivenFuture();
3.   HjDataDrivenFuture<Void> ddfB = newDataDrivenFuture();
4.   HjDataDrivenFuture<Void> ddfC = newDataDrivenFuture();
5.   HjDataDrivenFuture<Void> ddfD = newDataDrivenFuture();
6.   HjDataDrivenFuture<Void> ddfE = newDataDrivenFuture();
7.   async(() -> { ... ; ddfA.put(null); }); // Task A
8.   asyncAwait(ddfA, () -> { ... ;  ddfB.put(null); }); // Task B
9.   asyncAwait(ddfA, () -> { ... ;  ddfC.put(null); }); // Task C
10.  asyncAwait(ddfB, ddfC, ()->{ ... ; ddfD.put(null); }); // Task D
11.  asyncAwait(ddfC, () -> { ... ;  ddfE.put(null); }); // Task E
12.  asyncAwait(ddfD, ddfE, () -> { ... }); // Task F
13. }); // finish
```

Differences between Futures and DDFs/ DDTs

- Consumer task blocks on get() for each future that it reads, whereas async-await does not start execution till all DDFs are available
- Future tasks cannot deadlock, but it is possible for a DDT to block indefinitely ("deadlock") if one of its input DDFs never becomes available
- DDTs and DDFs are more general than futures
  - Producer task can only write to a single future object, where as a DDT 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 DDT
  - Consumer tasks can be created before the producer tasks
- DDTs and DDFs can be more implemented more efficiently than futures
  - An "asyncAwait" statement does not block the worker, unlike a future.get()

Two Exception (error) cases for DDFs that do not occur in futures

- 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
  - There can be at most one value provided for a future object (since it comes from the producer task's return statement)

- 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
  - Futures support blocking gets

Deadlock example with DDTs

```
1. HjDataDrivenFuture left = newDataDrivenFuture();
2. HjDataDrivenFuture right = newDataDrivenFuture();
3. finish(() -> {
4.   asyncAwait(left, () -> {
5.       right.put(rightWriter()); });
6.   asyncAwait(right, () -> {
7.       left.put(leftWriter()); });
8. });
```

- HJ-Lib has deadlock detection mode
- Enabled using:
  – System.setProperty(HjSystemProperty.trackDeadlocks.propertyKey(), “true");
  – Reports an edu.rice.hj.runtime.util.DeadlockException when deadlock detected