COMP 322: Fundamentals of Parallel Programming

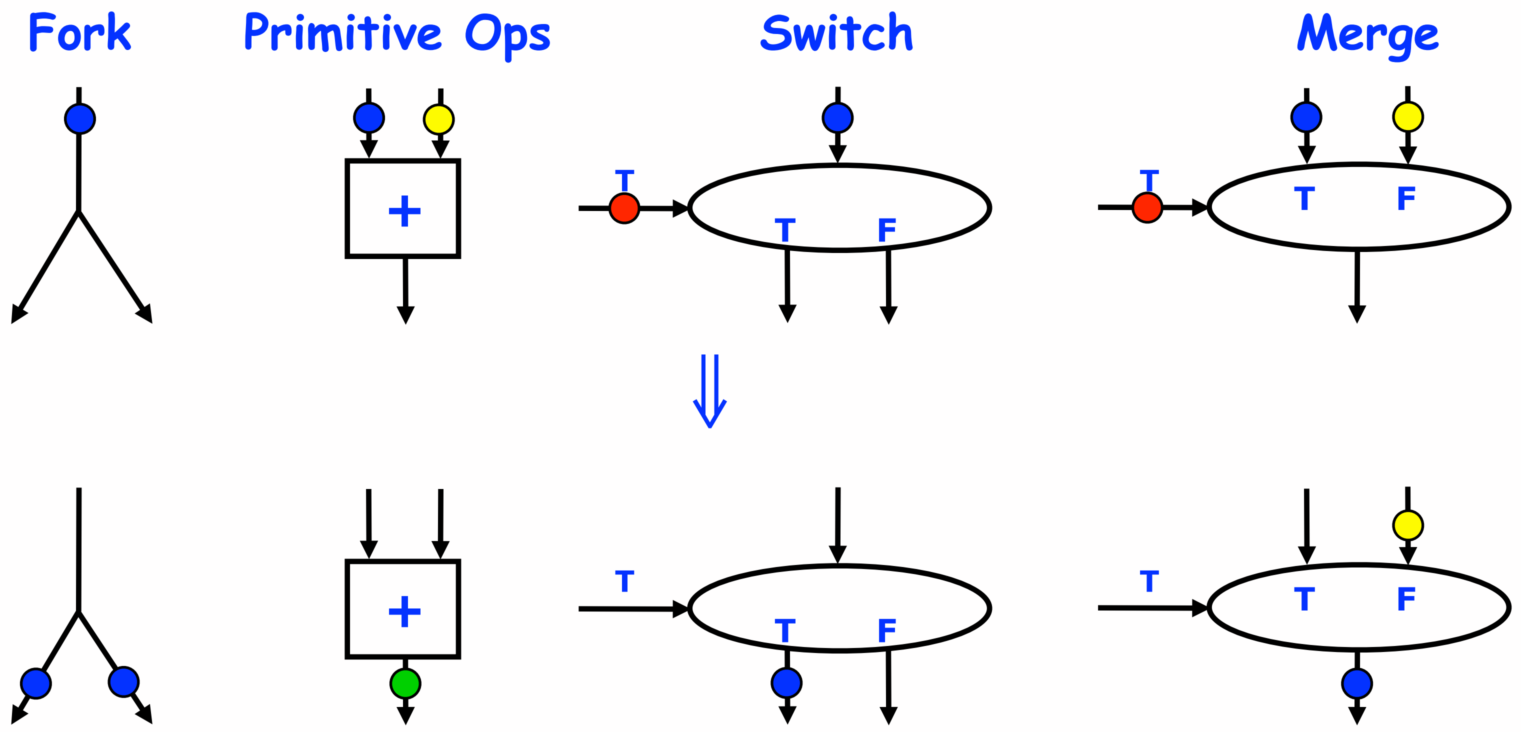
Lecture 14: Data-Driven Tasks and Data-Driven Futures

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

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

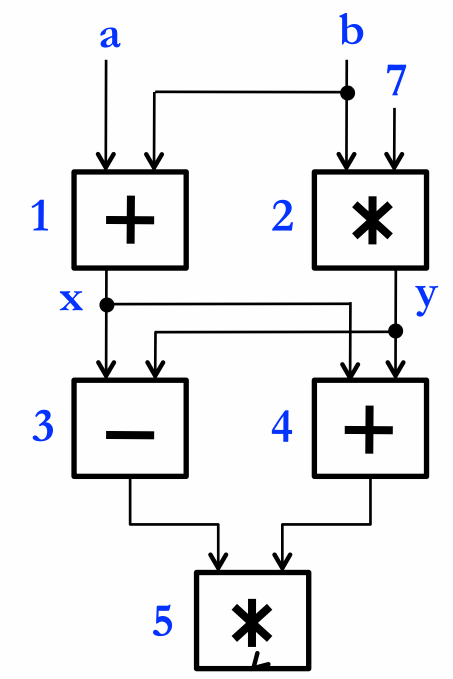
Dataflow Computing

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



Example instruction sequence and its dataflow graph

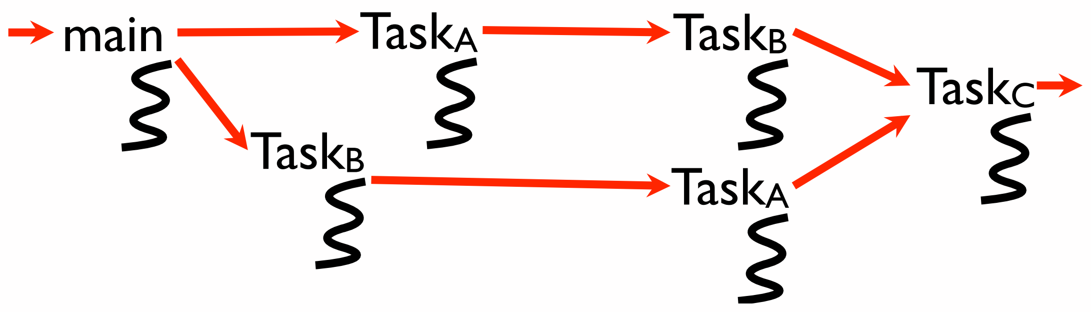
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



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 variable

- "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
- HjDataDrivenFuture interface extends the HjFuture interface

```
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")

```
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

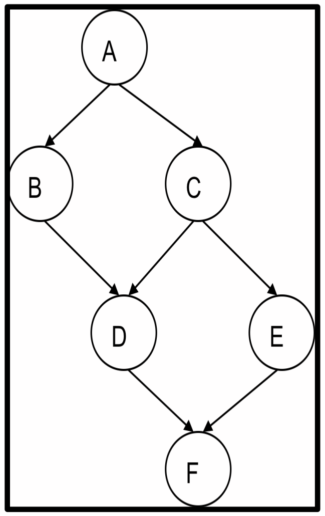
- **Future version**

```
final HjFuture<T> f = future(() -> { return g(); });
S1
async(() -> {
  ... = f.get();
  S2;
  S3;
});
```

- **DDF version**

```
HjDataDrivenFuture<T> f = newDataDrivenFuture();
async(() -> { f.put(g()) });
S1
asyncAwait(f, () -> {
  ... = f.get();
  S2;
  S3;
});
```

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



```
finish(() -> {
  HjDataDrivenFuture<Void> ddfA = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfB = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfC = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfD = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfE = newDataDrivenFuture();
  async(() -> { ... ; ddfA.put(null); }); // Task A
  asyncAwait(ddfA, () -> { ... ;  ddfB.put(null); }); // Task B
  asyncAwait(ddfA, () -> { ... ;  ddfC.put(null); }); // Task C
  asyncAwait(ddfB, ddfC, ()->{ ... ; ddfD.put(null); }); // Task D
  asyncAwait(ddfC, () -> { ... ;  ddfE.put(null); }); // Task E
  asyncAwait(ddfD, ddfE, () -> { ... }); // Task F
}); // 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
- DDTs and DDFs can be more implemented more efficiently than futures
  - An "asyncAwait" statement does not block the worker, unlike a future.get()
  - You will never see the following message with "asyncAwait"
    - "ERROR: Maximum number of hj threads per place reached"

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

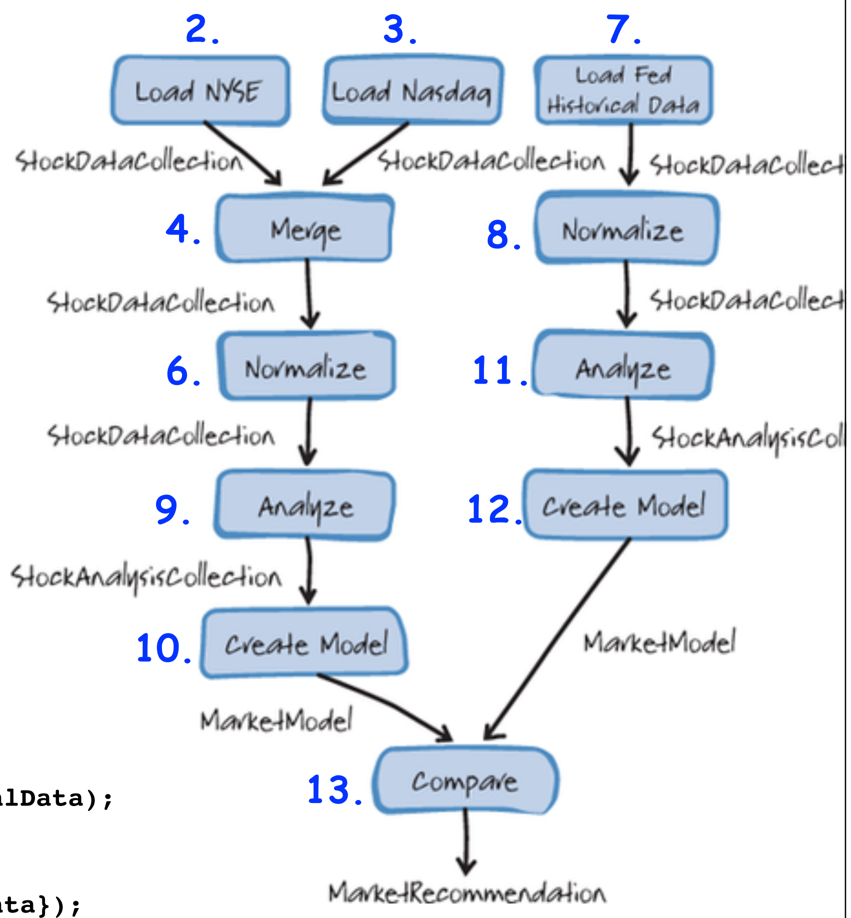
Deadlock example with DDTs

```
HjDataDrivenFuture left = newDataDrivenFuture();
HjDataDrivenFuture right = newDataDrivenFuture();
finish(() -> {
  asyncAwait(left, () -> {
      right.put(rightWriter()); });
  asyncAwait(right, () -> {
      left.put(leftWriter()); });
});
```

- 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

“Adatum Dashboard” Example: Sequential Version

```
1.public MarketRecommendation DoAnalysisSequential() {
2.  StockDataCollection nyseData = LoadNyseData();
3.  StockDataCollection nasdaqData = LoadNasdaqData();
4.  StockDataCollection mergedMarketData =
5.    MergeMarketData(new[]{nyseData, nasdaqData});
6.  StockDataCollection normalizedMarketData =
        NormalizeData(mergedMarketData);
7.  StockDataCollection fedHistoricalData =
        LoadFedHistoricalData();
8.  StockDataCollection normalizedHistoricalData =
        NormalizeData(fedHistoricalData);
9.  StockAnalysisCollection analyzedStockData =
        AnalyzeData(normalizedMarketData);
10. MarketModel modeledMarketData = RunModel(analyzedStockData);
11. StockAnalysisCollection analyzedHistoricalData =
        AnalyzeData(normalizedHistoricalData);
12. MarketModel modeledHistoricalData = RunModel(analyzedHistoricalData);
13. MarketRecommendation recommendation =
14.    CompareModels(new[] {modeledMarketData, modeledHistoricalData});
15.  return recommendation;
16.}
```



Source: http://programming4.us/enterprise/3004.aspx

“Adatum Dashboard” Example (Pseudocode): Parallel Version using DDTs and DDFs

```
public MarketRecommendation DoAnalysisParallelDDT() {
  async nyseData.put(LoadNyseData());
  async nasdaqData.put(LoadNasdaqData());
  async await(nyseData, nasdaqData)
    mergedMarketData.put(MergeMarketData(new[]{nyseData.get(), nasdaqData.get()}));
  async await(mergedMarketData) normalizedMarketData.put(NormalizeData(mergedMarketData.get()));
  async fedHistoricalData.put(LoadFedHistoricalData());
  async await(fedHistoricalData) normalizedHistoricalData.put(NormalizeData(fedHistoricalData.get()));
  async await(normalizedMarketData) analyzedStockData.put(AnalyzeData(normalizedMarketData.get()));
  async await(analyzedStockData) modeledMarketData.put(RunModel(analyzedStockData.get()));
  async await(normalizedHistoricalData) analyzedHistoricalData.put(AnalyzeData(normalizedHistoricalData.get()));
  async await(analyzedHistoricalData) modeledHistoricalData.put(RunModel(analyzedHistoricalData.get()));
  MarketRecommendation recommendation =
     CompareModels(new[] {modeledMarketData.get(), modeledHistoricalData.get()});
  return recommendation;
}
```

Note that the put, await, and get clauses follow directly from the data flow structure of the program!

Worksheet 14: Data Driven Futures

Name: ______________________ Netid: ______________________

For the example below, will reordering the five async statements change the meaning of the program? If so, show two orderings that exhibit different behaviors. If not, explain why not. (You can use the space below this slide for your answer.)

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await(left) leftReader(left); // Task3
  async await(right) rightReader(right); // Task5
  async await(left,right)
        bothReader(left,right); // Task4
  async left.put(leftWriter()); // Task1
  async right.put(rightWriter());// Task2
}
```