COMP 322: Fundamentals of Parallel Programming

Lecture 15: Data-Driven Tasks

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Worksheet #14 Solution: Iterative Averaging Revisited

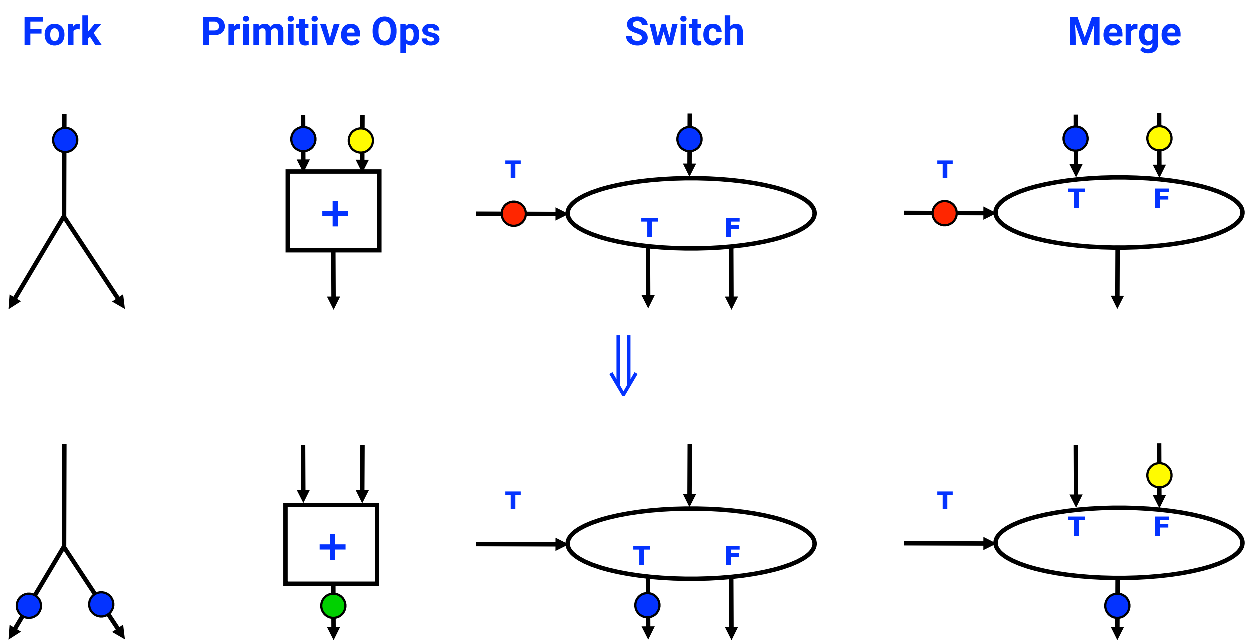
Answer the questions in the table below for the versions of the Iterative Averaging code shown in slides 7, 8, 10, 12. Write in your answers as functions of m, n, and nc.

	Slide 7	Slide 8	Slide 10	Slide 12
How many tasks are created (excluding the main program task)?	m*n	n Incorrect: n * m	m*nc Incorrect: n * nc	nc Incorrect: n*m, m*nc
How many barrier operations (calls to next per task) are performed?	0 Incorrect: m	m Incorrect: m*n	0 Incorrect: m	m Incorrect: m*nc, nc

The SPMD version on slide 12 is the most efficient because it only creates nc tasks (assuming task creation is more expensive than a barrier operation.)

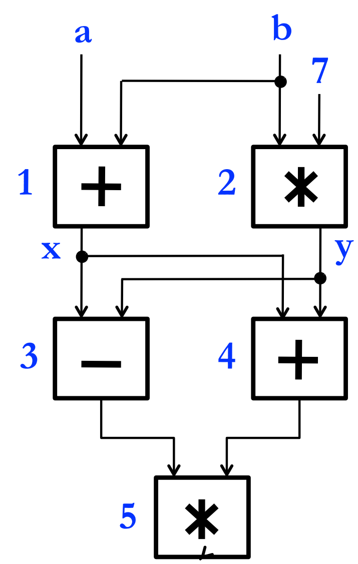
Dataflow Computing

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



Example instruction sequence and its dataflow graph

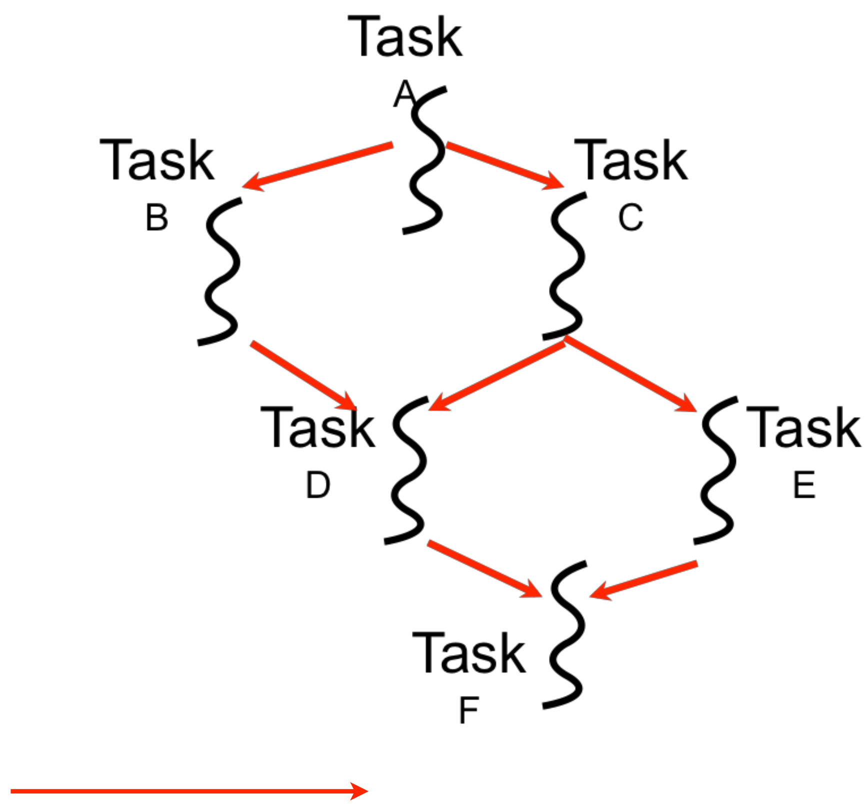
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



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

No separate branch instructions

Macro-Dataflow Programming



Communication via “single-assignment” variables

- “Macro-dataflow” = expansion of the 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 (similar to futures)
  - 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)

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

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1, and can only be assigned once via put() operations
- HjDataDrivenFuture extends the HjFuture interface

```
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

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

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

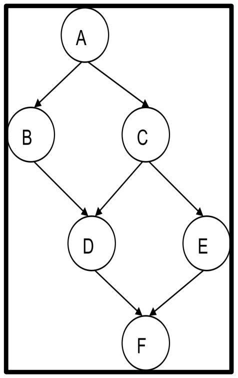
- Create a new data-driven-task to start executing Stmt after all of ddfA, ddfB, … become available (i.e., after task becomes "enabled")
- Await clause can be used to implement "nodes" and "edges" in a computation graph

```
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

Converting previous Future example to Data-Driven Futures and AsyncAwait Tasks

```
finish(() -> {
  HjDataDrivenFuture<Void> ddfA = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfB = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfC = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfD = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfE = newDataDrivenFuture();
  asyncAwait(ddfA, () -> { ... ;  ddfB.put(…); }); // Task B
  asyncAwait(ddfA, () -> { ... ;  ddfC.put(…); }); // Task C
  asyncAwait(ddfB, ddfC, ()->{ ... ; ddfD.put(…); }); // Task D
  asyncAwait(ddfC, () -> { ... ;  ddfE.put(…); }); // Task E
  asyncAwait(ddfD, ddfE, () -> { ... }); // Task F
  // Note that creating a “producer” task after its “consumer”
  // task is permitted with DDFs & DDTs, but not with futures
  async(() -> { ... ; ddfA.put(…); }); // Task A
}); // 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
  - Future task can only write to a single future object, whereas 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 DDTs can be created before the producer DDTs
- DDTs and DDFs can be implemented more efficiently than futures
  - An "asyncAwait" statement does not block the worker, unlike a future.get()

Two Exception (error) cases for DDFs that cannot occur with 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, and a put on that DDF hasn't happened yet, then an exception is thrown because DDF's do not support blocking gets
  - Futures support blocking gets

What is Deadlock?

- A parallel program execution contains a deadlock if some task's execution remains incomplete due to it being *blocked indefinitely* awaiting some condition
- Example of a program with a deadlocking execution

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
   async await ( left ) right.put(rightBuilder()); // Task1
   async await ( right ) left.put(leftBuilder()); // Task2
}
```

- In this case, Task1 and Task2 are in a deadlock cycle.
- HJ-Lib has a deadlock detection debug option, which can be enabled as follows:
  - System.setProperty(HjSystemProperty.trackDeadlocks.propertyKey(), "true");
  - Throws an edu.rice.hj.runtime.util.DeadlockException when deadlock detected

Deadlock cannot happen when using only finish-async, futures or barriers!

Implementing Future Tasks using DDFs

- Future version

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

- DDF version

```
HjDataDrivenFuture<T> f = newDataDrivenFuture();
async(() -> { f.put(g()) });
S1
asyncAwait(f, () -> {
  ... = f.get(); // does not block -- why?
  S2;
  S3;
});
```