COMP 322: Fundamentals of Parallel Programming

Lecture 15: Data-Driven Tasks, Point-to-Point Synchronization with Phasers

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

http://comp322.rice.edu

Worksheet #14 Solution: Iterative Averaging Revisited

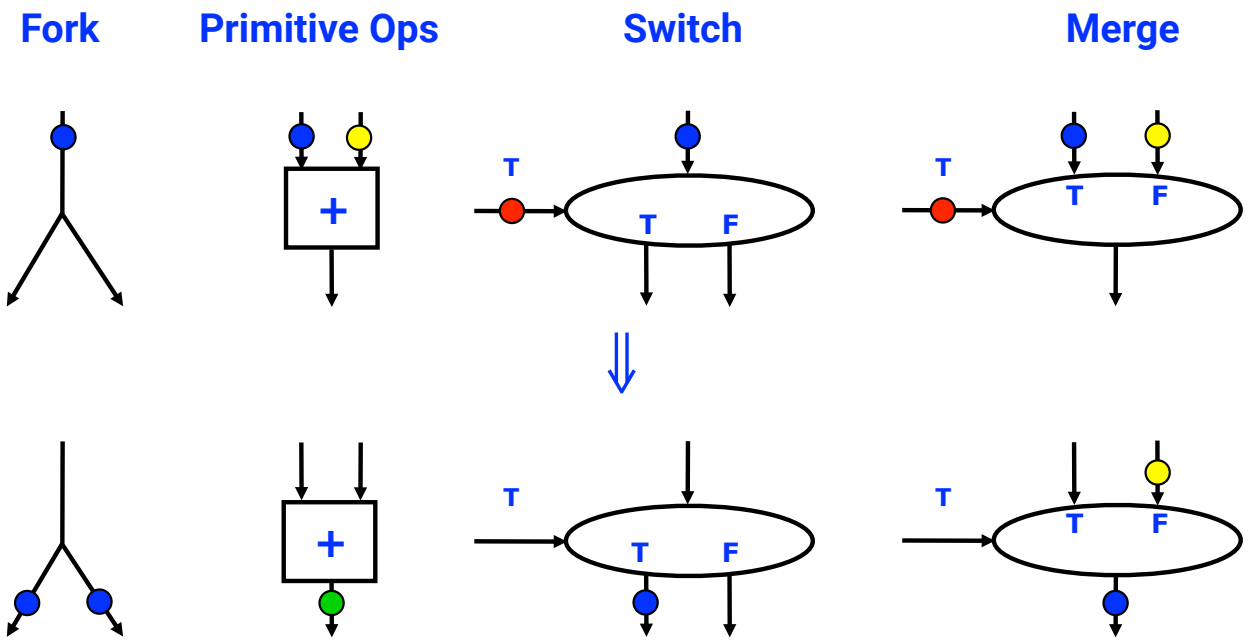
Answer the questions in the table below for the versions of the Iterative Averaging code shown in slides 7, 8, 10, 12. Write in your answers as functions of m, n, and nc.

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

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

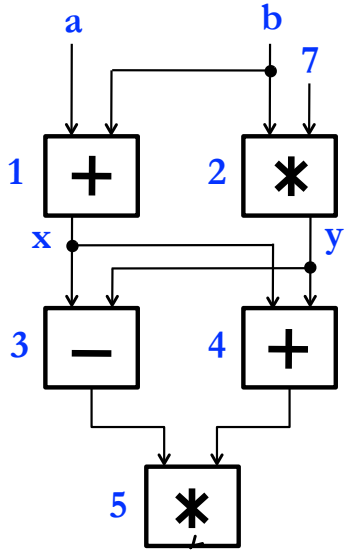
Dataflow Computing

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




Example instruction sequence and its dataflow graph

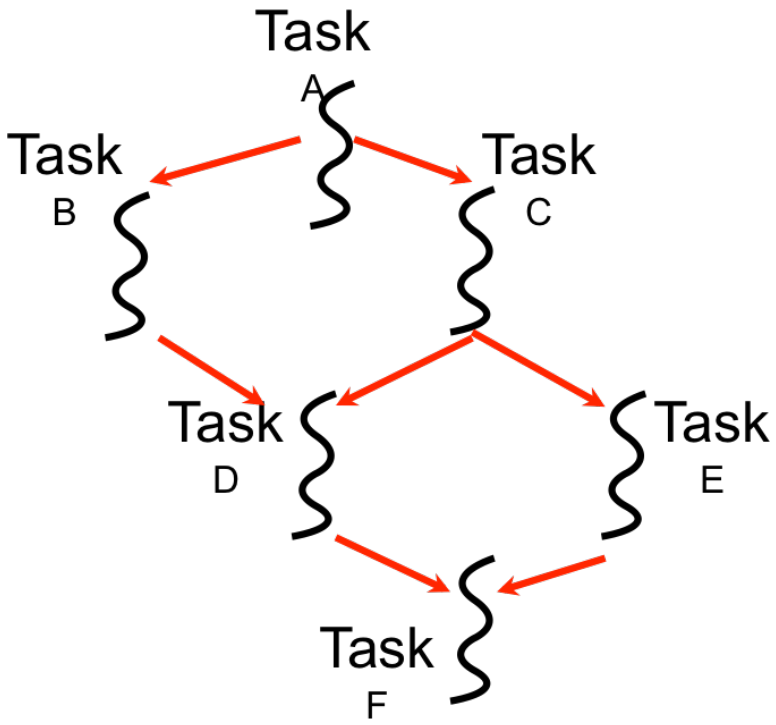
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



An operator executes when all its input values are present; 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" = 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 (like 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)

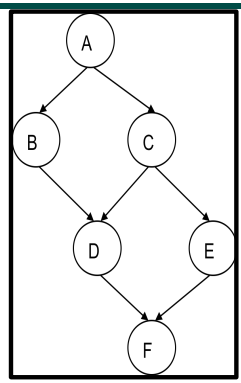
```
asyncAwait(ddfA, ddfB, …, () -> Stmt);
```

- 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




```
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.   asyncAwait(ddfA, () -> { ... ;  ddfB.put(…); }); // Task B
8.   asyncAwait(ddfA, () -> { ... ;  ddfC.put(…); }); // Task C
9.   asyncAwait(ddfB, ddfC, ()->{ ... ; ddfD.put(…); }); // Task D
10.  asyncAwait(ddfC, () -> { ... ;  ddfE.put(…); }); // Task E
11.  asyncAwait(ddfD, ddfE, () -> { ... }); // Task F
12.  // Note that creating a “producer” task after its “consumer”
13.  // task is permitted with DDFs & DDTs, but not with futures
14.  async(() -> { ... ; ddfA.put(…); }); // Task A
15. }); // 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, 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 tasks
- 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, then an exception is thrown because DDF's do not support blocking gets
  - Futures support blocking gets

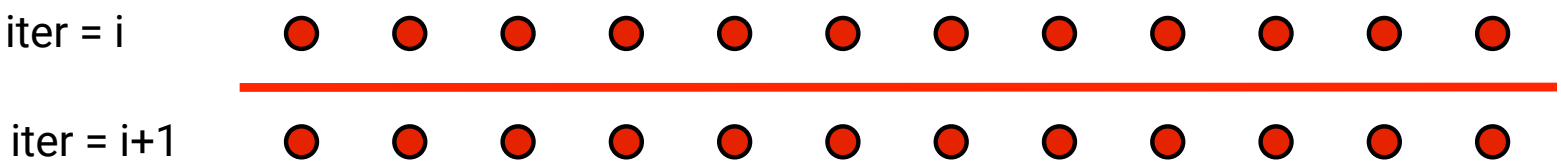
Deadlock example with DDTs (cannot be reproduced with futures)

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

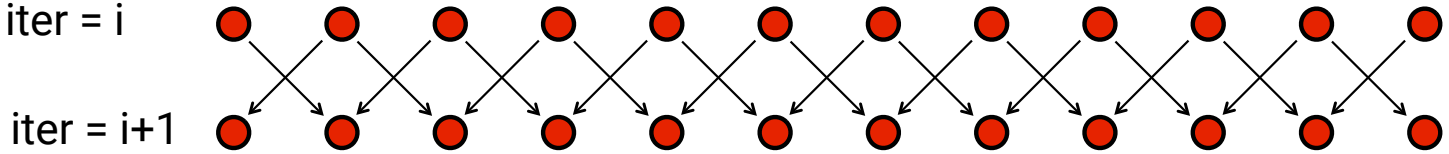
```
1. HjDataDrivenFuture left = newDataDrivenFuture();
2. HjDataDrivenFuture right = newDataDrivenFuture();
3. finish(() -> {
4.   asyncAwait(left, () -> {
5.      right.put(rightWriter()); });
6.   asyncAwait(right, () -> {
7.      left.put(leftWriter()); });
8. });
```

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

Barrier vs Point-to-Point Synchronization in One-Dimensional Iterative Averaging Example



Barrier synchronization



Point-to-point synchronization

Question: when can the point-to-point computation graph result in a smaller CPL than the barrier computation graph?
Answer: when there is variability in the node execution times.

Phasers: a unified construct for barrier and point-to-point synchronization

- HJ phasers unify barriers with point-to-point synchronization
  - Inspiration for java.util.concurrent.Phaser
- Previous example motivated the need for "point-to-point" synchronization
  - With barriers, phase i of a task waits for *all* tasks associated with the same barrier to complete phase i-1
  - With phasers, phase i of a task can select a subset of tasks to wait for
- Phaser properties
  - Support for barrier and point-to-point synchronization
  - Support for dynamic parallelism --- the ability for tasks to drop phaser registrations on termination (end), and for new tasks to add phaser registrations (async phased)
  - A task may be registered on multiple phasers in different modes

Simple Example with Four Async Tasks and One Phaser

```
1.  finish (() -> {
2.    ph = newPhaser(SIG_WAIT); // mode is SIG_WAIT
3.    asyncPhased(ph.inMode(SIG), () -> {
4.      // A1 (SIG mode)
5.      doA1Phase1(); next(); doA1Phase2(); });
6.    asyncPhased(ph.inMode(SIG_WAIT), () -> {
7.      // A2 (SIG_WAIT mode)
8.      doA2Phase1(); next(); doA2Phase2(); });
9.    asyncPhased(ph.inMode(HjPhaserMode.SIG_WAIT), () -> {
10.     // A3 (SIG_WAIT mode)
11.     doA3Phase1(); next(); doA3Phase2(); });
12.   asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
13.     // A4 (WAIT mode)
14.     doA4Phase1(); next(); doA4Phase2(); });
15. });
```

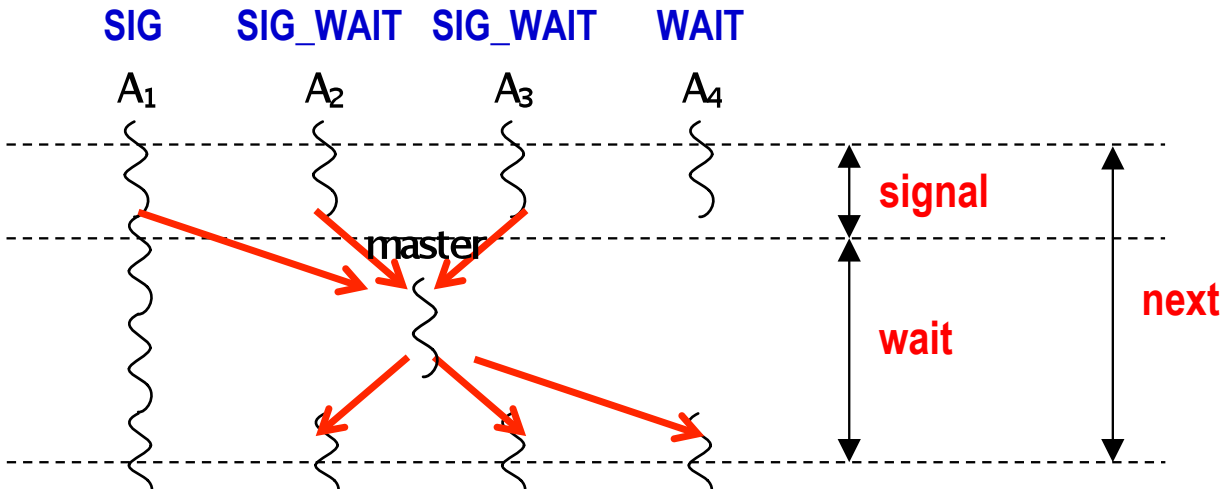

Computation Graph Schema Simple Example with Four Async Tasks and One Phaser

Semantics of next depends on registration mode

SIG_WAIT: **next = signal + wait**

SIG: **next = signal**

WAIT: **next = wait**

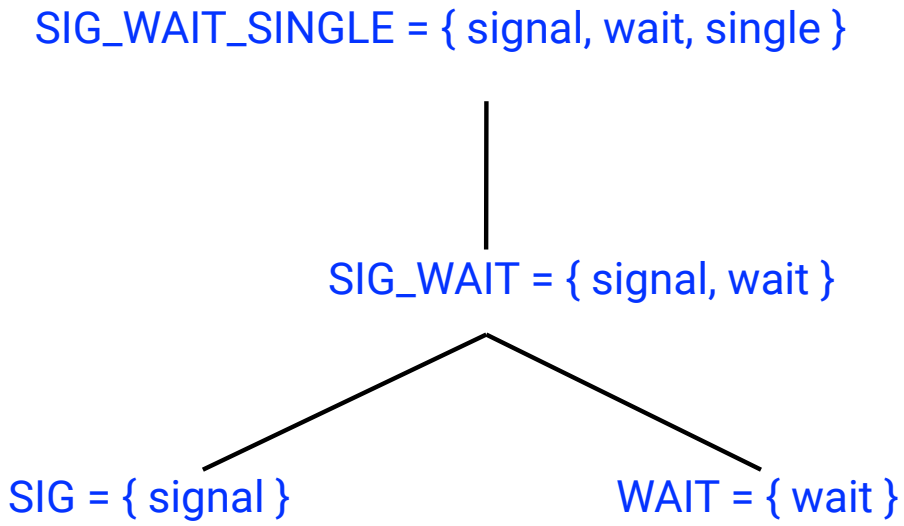


Summary of Phaser Construct

- Phaser allocation
  - HjPhaser ph = newPhaser(mode);
    - Phaser ph is allocated with registration mode
    - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)
- Registration Modes
  - HjPhaserMode.SIG, HjPhaserMode.WAIT, HjPhaserMode.SIG_WAIT, HjPhaserMode.SIG_WAIT_SINGLE
    - NOTE: phaser WAIT is unrelated to Java wait/notify (which we will study later)
- Phaser registration
  - asyncPhased (ph_1.inMode(<$mode_1$>), ph_2.inMode(<$mode_2$>), … () -> <stmt>)
    - Spawned task is registered with ph_1 in $mode_1$, ph_2 in $mode_2$, …
    - Child task's capabilities must be subset of parent's
    - asyncPhased <stmt> propagates all of parent's phaser registrations to child
- Synchronization
  - next();
    - Advance each phaser that current task is registered on to its next phase
    - Semantics depends on registration mode
    - Barrier is a special case of phaser, which is why next is used for both

Capability Hierarchy

- A task can be registered in one of four modes with respect to a phaser: SIG_WAIT_SINGLE, SIG_WAIT, SIG, or WAIT. The mode defines the set of capabilities — signal, wait, single — that the task has with respect to the phaser. The subset relationship defines a natural hierarchy of the registration modes. A task can drop (but not add) capabilities after initialization.



Left-Right Neighbor Synchronization (with m=3 tasks)

```
1.finish(() -> { // Task-0
2.    final HjPhaser ph1 = newPhaser(SIG_WAIT);
3.    final HjPhaser ph2 = newPhaser(SIG_WAIT);
4.    final HjPhaser ph3 = newPhaser(SIG_WAIT);
5.    asyncPhased(ph1.inMode(SIG),ph2.inMode(WAIT),
6.      () -> { doPhase1(1);
7.        next(); // signals ph1, waits on ph2
8.        doPhase2(1);
9.    }); // Task T1
10.   asyncPhased(ph2.inMode(SIG),ph1.inMode(WAIT),ph3.inMode(WAIT),
11.     () -> { doPhase1(2);
12.       next(); // signals ph2, waits on ph3
13.       doPhase2(2);
14.    }); // Task T2
15.    asyncPhased(ph3.inMode(SIG),ph2.inMode(WAIT),
16.      () -> { doPhase1(3);
17.        next(); // signals ph3, waits on ph2
18.        doPhase2(3);
19.    }); // Task T3
20.}); // finish
```


Computation Graph for m=3 example
(without async-finish nodes and edges)