COMP 322: Fundamentals of Parallel Programming

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

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/

Worksheet #12 Solution: Iterative Averaging Revisited

Answer the questions in the table below for the versions of the Iterative Averaging code shown in slides 5, 7, 8, 10. Write in your answers as functions of m, n, and nc.

	Slide 5	Slide 7	Slide 8	Slide 10
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 10 is the most efficient because it only creates nc tasks. (Task creation is more expensive than a barrier operation.)

HJ code for One-Dimensional Iterative Averaging using nested `forseq-forall` structure (Slide 5, Lecture 13)

```
double[] myVal=new double[n+2]; myVal[n+1] = 1;
double[] myNew=new double[n+2]; myNew[n+1] = myVal[n+1];
forseq(0, m-1, (iter) -> {
  // Compute MyNew as function of input array MyVal
  forall(1, n, (j) -> { // Create n tasks
     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
  }); // forall
  // Swap myVal & myNew;
   temp=myVal; myVal=myNew; myNew=temp;
   // myNew becomes input array for next iteration
}); // for
```

Example: HJ code for One-Dimensional Iterative Averaging with forseq-forall structure w/ chunking (Slide 7, Lecture 13)

```
double[] myVal=new double[n+2]; myVal[n+1] = 1;
double[] myNew=new double[n+2]; myNew[n+1] = myVal[n+1];
int nc = numWorkerThreads();
 forseq(0, m-1, (iter) -> {
   // Compute MyNew as function of input array MyVal
   forallChunked(1, n, n/nc, (j) -> { // Create nc tasks
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
   }); // forallChunked
  temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
  // myNew becomes input array for next iteration
 }); // for
```

Converting forseq-forall version into a forall-forseq version with barriers (Slide 8, Lecture 13)

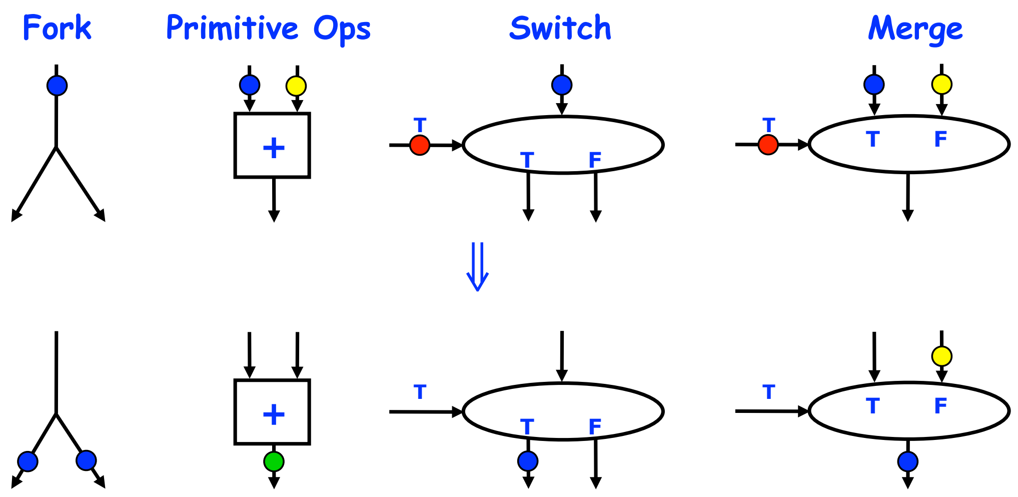
```
double[] gVal=new double[n+2]; gVal[n+1] = 1;
double[] gNew=new double[n+2];
forallPhased(1, n, (j) -> { // Create n tasks
  // Initialize myVal and myNew as local pointers
  double[] myVal = gVal; double[] myNew = gNew;
  forseq(0, m-1, (iter) -> {
    // Compute MyNew as function of input array MyVal
    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    next(); // Barrier before executing next iteration of iter loop
    // Swap local pointers, myVal and myNew
    double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iteration
  }); // forseq
}); // forall
```

Converting forall-forseq-next version into a forall-forseq-forseq-next version with grouping (Slide 10, Lecture 13)

```
double[] gVal=new double[n+2]; gVal[n+1] = 1;
double[] gNew=new double[n+2];
HjRegion1D iterSpace = newRectangularRegion1D(1,n);
int nc = numWorkerThreads();
forallPhased(0, nc-1, (jj) -> { // Create nc tasks
  // Initialize myVal and myNew as local pointers
  double[] myVal = gVal; double[] myNew = gNew;
  forseq(0, m-1, (iter) -> {
    forseq(myGroup(jj,iterSpace,nc), (j) -> {
      // Compute MyNew as function of input array MyVal
      myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
    }); // forseq
    next(); // Barrier before executing next iteration of iter loop
    // Swap local pointers, myVal and myNew
    double[] temp=myVal; myVal=myNew; myNew=temp;
    // myNew becomes input array for next iter
  }); // forseq
}); // forall
```

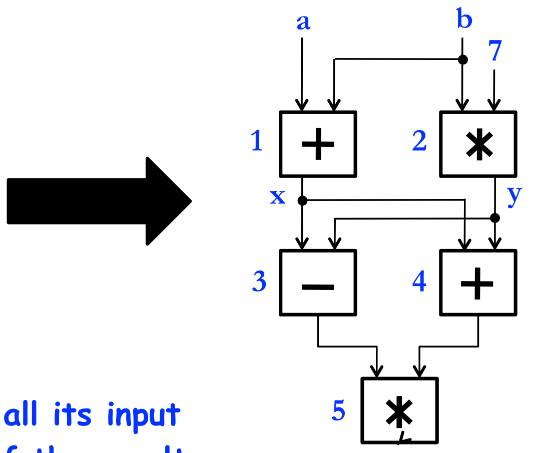
Dataflow Computing

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



Example instruction sequence and its dataflow graph

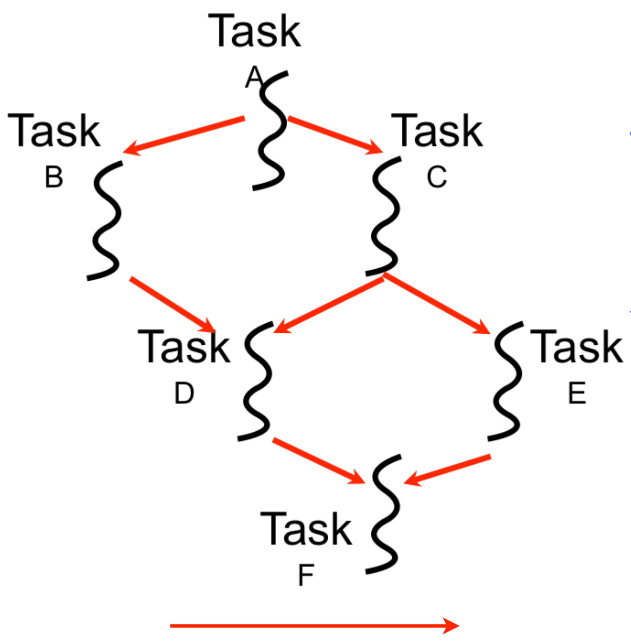
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



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" 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)

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

- Allocate an instance of a <u>data-driven-future</u> 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

Implementing Future Tasks using DDFs

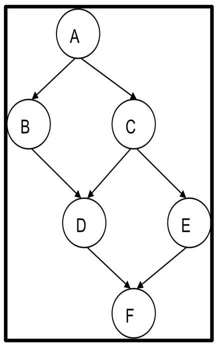
- Future version

```
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;
});
```

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();
  async(() -> { ... ; ddfA.put(…); }); // Task A
  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
}); // 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

- <u>Case 1:</u> 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)

- <u>Case 2:</u> 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)

```
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