Lecture 14: Data-Driven Tasks

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
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.

<table>
<thead>
<tr>
<th>How many tasks are created (excluding the main program task)?</th>
<th>Slide 7</th>
<th>Slide 8</th>
<th>Slide 10</th>
<th>Slide 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>m*n</td>
<td>n</td>
<td>m*nc</td>
<td>nc</td>
<td></td>
</tr>
<tr>
<td>Incorrect: n * m</td>
<td>Incorrect: n * nc</td>
<td>Incorrect: n<em>m, m</em>nc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How many barrier operations (calls to next per task) are performed?</th>
<th>Slide 7</th>
<th>Slide 8</th>
<th>Slide 10</th>
<th>Slide 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>m</td>
<td>0</td>
<td>m</td>
<td></td>
</tr>
<tr>
<td>Incorrect: m</td>
<td>Incorrect: m*n</td>
<td>Incorrect: m*nc, nc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Which SPMD version is the most efficient?
HJ code for One-Dimensional Iterative Averaging using nested forseq–forall structure

1. double[] myVal=new double[n+2]; myVal[n+1] = 1;
2. double[] myNew=new double[n+2]; myNew[n+1] = myVal[n+1];
3. forseq(0, m-1, (iter) -> {
4.   // Compute MyNew as function of input array MyVal
5.   forall(1, n, (j) -> {
6.     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7.   }); // forall
8.   // Swap myVal & myNew;
9.   temp=myVal; myVal=myNew; myNew=temp;
10.  // myNew becomes input array for next iteration
11.}); // for
Converting `forseq-forall` version into a `forall-forseq` version with barriers

1. double[] gVal=new double[n+2]; gVal[n+1] = 1;
2. double[] gNew=new double[n+2];
3. forallPhased(1, n, (j) -> { // Create n tasks
   4. // Initialize myVal and myNew as local pointers
   5. double[] myVal = gVal; double[] myNew = gNew;
   6. forseq(0, m-1, (iter) -> {
      7. // Compute MyNew as function of input array MyVal
      8. myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
      9. next(); // Barrier before next iteration of iter loop
     10. // Swap local pointers, myVal and myNew
     11. double[] temp=myVal; myVal=myNew; myNew=temp;
     12. // myNew becomes input array for next iteration
     13. }); // forseq
     14.}); // forall
Example: HJ for One-Dimensional Iterative Averaging with chunking

1. double[] myVal=new double[n+2]; myVal[n+1] = 1;
2. double[] myNew=new double[n+2]; myNew[n+1] = myVal[n+1];
3. int nc = numWorkerThreads();
4. forseq(0, m-1, (iter) -> {
   5.   // Compute MyNew as function of input array MyVal
   6.       forallChunked(1, n, n/nc, (j) -> {
         7.         myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
       8.   }); // forallChunked
   9.   temp=myVal; myVal=myNew; myNew=temp;// Swap myVal & myNew;
10.  // myNew becomes input array for next iteration
11.}); // for
Inner chunked loop goes inside `forseq-iter` loop

1. double[] gVal = new double[n+2]; gVal[n+1] = 1;
2. double[] gNew = new double[n+2];
3. HjRegion1D iterSpace = newRectangularRegion1D(1,n);
4. int nc = numWorkerThreads();
5. forallPhased(0, nc-1, (jj) -> { // Create nc tasks
6.     // Initialize myVal and myNew as local pointers
7.     double[] myVal = gVal; double[] myNew = gNew;
8.     forseq(0, m-1, (iter) -> {
9.         // Compute MyNew as function of input array MyVal
10.        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
11.    }); // forseq
12.    next(); // Barrier before executing next iteration of iter loop
13.    // Swap local pointers, myVal and myNew
14.    double[] temp = myVal; myVal = myNew; myNew = temp;
15.    // myNew becomes input array for next iter
16.    }); // forseq
17. })); // forall
An operator executes when all its input values are present; copies of the result value are distributed to the destination operators.

\[
x = a + b;
y = b \times 7;
z = (x-y) \times (x+y);
\]
Macro-Dataflow Programming

• “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)

Communication via “single-assignment” variables
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
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`
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.

```java
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.
Implementing Future Tasks using DDFs

- **Future version**
  1. `HjFuture<T> f = future(() -> { return g(); });`
  2. S1
  3. async(() -> {
     4. `... = f.get();` // blocks if needed
     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();` // does not block — why?
     6. S2;
     7. S3;
     8. });
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, whereas 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 = new DataDrivenFuture();`
2. `HjDataDrivenFuture right = new DataDrivenFuture();`
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