COMP 322: Fundamentals of Parallel Programming

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

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

http://comp322.rice.edu/
Dataflow Computing

- Original idea: replace machine instructions by a small set of dataflow operators
Example instruction sequence and its dataflow graph

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

- “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)

```java
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)

\[ \text{asyncAwait}(\text{ddfA}, \text{ddfB}, \ldots, () \to \text{Stmt}); \]

- Create a new \textit{data-driven-task} to start executing \texttt{Stmt} after all of \texttt{ddfA}, \texttt{ddfB}, \ldots become available (i.e., after task becomes “enabled”)

- Await clause can be used to implement “nodes” and “edges” in a computation graph

\texttt{ddfA.get()}

- Return value (of type \texttt{T1}) stored in \texttt{ddfA}

- Thrown an exception if \texttt{put()} has not been performed
  
  \textit{Should be performed by async’s that contain \texttt{ddfA} in their await clause, or if there’s some other synchronization to guarantee that the \texttt{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” task is permitted with DDFs & DDTs, but not with futures`
13. `async(() -> { ... ; ddfA.put(...); }); // Task A`
14. `}); // 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, whereas the choice of output DDF can be deferred to any point with a DDT.
  - Consumer tasks can be created before the producer tasks in DDTs.

- 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 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
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");`
  - Reports an `edu.rice.hj.runtime.util.DeadlockException` when deadlock detected.
Deadlock example with DDTs (cannot be reproduced with futures)

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

- 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
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.Phase
- 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(); });`
Semantics of `next` depends on registration mode:

- SIG_WAIT: next = signal + wait
- SIG: next = signal
- WAIT: next = wait

A master thread (worker) gathers all signals and broadcasts a barrier completion.
Summary of Phaser Construct

- Phaser allocation
  - \texttt{HjPhaser ph = newPhaser(mode);}
  - Phaser \( ph \) is allocated with registration mode
  - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)

- Registration Modes
  - \texttt{HjPhaserMode.SIG}, \texttt{HjPhaserMode.WAIT}, \texttt{HjPhaserMode.SIG\_WAIT}, \texttt{HjPhaserMode.SIG\_WAIT\_SINGLE}
  - \textbf{NOTE:} phaser WAIT is unrelated to Java wait/notify (which we will study later)

- Phaser registration
  - \texttt{asyncPhased(ph\_1.inMode(<mode\_1>), ph\_2.inMode(<mode\_2>), … () -> <stmt> )}
  - Spawned task is registered with \( ph\_1 \) in \( \texttt{mode\_1} \), \( ph\_2 \) in \( \texttt{mode\_2} \), …
  - Child task’s capabilities must be \textit{subset} of parent’s
  - \texttt{asyncPhased <stmt>} propagates all of parent’s phaser registrations to child

- Synchronization
  - \texttt{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 \texttt{next} is used for both
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.

\[
\begin{align*}
\text{SIG\_WAIT\_SINGLE} &= \{ \text{signal, wait, single} \} \\
\text{SIG\_WAIT} &= \{ \text{signal, wait} \} \\
\text{SIG} &= \{ \text{signal} \} \\
\text{WAIT} &= \{ \text{wait} \}
\end{align*}
\]
Left-Right Neighbor Synchronization
(with m=3 tasks)

```java
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)
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>Question</th>
<th>Slide 7</th>
<th>Slide 8</th>
<th>Slide 10</th>
<th>Slide 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many tasks are created (excluding the main program task)?</td>
<td>$m*n$</td>
<td>$n$</td>
<td>$m*nc$</td>
<td>$nc$</td>
</tr>
<tr>
<td>Incorrect: $n*m$</td>
<td>Incorrect: $n*m$</td>
<td>Incorrect: $n*nc$</td>
<td>Incorrect: $n<em>m$, $m</em>nc$</td>
<td></td>
</tr>
<tr>
<td>How many barrier operations (calls to next per task) are performed?</td>
<td>0</td>
<td>$m$</td>
<td>0</td>
<td>$m$</td>
</tr>
<tr>
<td>Incorrect: $m$</td>
<td>Incorrect: $m*n$</td>
<td>Incorrect: $m$</td>
<td>Incorrect: $m*nc$, $nc$</td>
<td></td>
</tr>
</tbody>
</table>
For the example below, will reordering the five async statements change the meaning of the program (assuming that the semantics of the reader/writer methods depends only on their parameters)? If so, show two orderings that exhibit different behaviors. If not, explain why not.

1. `DataDrivenFuture left = new DataDrivenFuture();`
2. `DataDrivenFuture right = new DataDrivenFuture();`
3. `finish {
   4.   async await(left) leftReader(left); // Task3
   5.   async await(right) rightReader(right); // Task5
   6.   async await(left,right)
   7.      bothReader(left,right); // Task4
   8.   async left.put(leftWriter()); // Task1
   9.   async right.put(rightWriter()); // Task2
   10. }`
Worksheet #15b: Left-Right Neighbor Synchronization using Phasers (turn page over for worksheet 15a)

Name: ___________________  Netid: ___________________

Worksheet #15b: Left-Right Neighbor Synchronization using Phasers (turn page over for worksheet 15a)

Complete the phased clause below to implement the point-to-point synchronization shown above for m tasks (generalization of slide 11)

1. \texttt{finish} () \to \{
2. \hspace{1em} \texttt{final HjPhaser[]} ph =
3. \hspace{2em} \texttt{new HjPhaser[m+2];} // array of phaser objects
4. \hspace{1em} \texttt{forseq}(0, m+1, (i) \to \{ \texttt{ph[i]} = \texttt{newPhaser(SIG\_WAIT)} \});
5. \hspace{1em} \texttt{forseq}(1, m, (i) \to \{
6. \hspace{2em} \texttt{asyncPhased(}
7. \hspace{3em} \texttt{ph[i-1].inMode(.....),}
8. \hspace{3em} \texttt{ph[i].inMode(.....),}
9. \hspace{3em} \texttt{ph[i+1].inMode(.....),} \to \{
10. \hspace{4em} \texttt{doPhase1(i);}
11. \hspace{4em} \texttt{next();}
12. \hspace{4em} \texttt{doPhase2(i); \}}; // asyncPhased
13. \hspace{4em} \}}; // forseq
14. \hspace{1em} \}}; // finish