COMP 322: Fundamentals of Parallel Programming

Lecture 14: Recap of HJ constructs, Point-to-Point Synchronization, Pipeline Parallelism, Introduction to Phasers

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Announcements

- Combined lecture quiz for this week (Week 5) and next week (Week 6) will be assigned on Thursday, Feb 14, and due by Sunday, Feb 17
- No labs or lab quizzes next week (Week 6)
- Homework 3 is due by 11:55pm on Friday, February 22, 2013
- Take-home midterm (Exam 1) will be given after lecture on Wednesday, February 20, 2013
  - Closed-book, closed computer
  - 2-hour duration
  - Will need to be returned to Sherry Nassar (Duncan Hall 3137) by 4pm on Friday, February 22, 2013
Worksheet #13: Forall Loops and Barriers

For the example below, will reordering the five async statements change the meaning of the program? If so, show two orderings that exhibit different behaviors. If not, explain why not. (You can use the space below this slide for your answer.)

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. }

No, reordering consecutive async’s will never change the meaning of the program, whether or not the async’s have await clauses.
Outline of Today’s Lecture

- Recap of HJ constructs studied so far
- Point-to-Point Synchronization, Pipeline Parallelism
- Introduction to Phasers

Acknowledgments

- COMP 322 Module 1 handout, Sections 12.1, 12.2
  —Source for seismic imaging example
  —Source for figures related to pipeline parallelism
Recap of HJ constructs studied so far

- Basic language summary can be found here:
  - [https://wiki.rice.edu/confluence/display/PARPROG/HJLanguageSummary](https://wiki.rice.edu/confluence/display/PARPROG/HJLanguageSummary)
  - Additional documentation in preparation

- Task creation constructs
  - async $Stmt$
  - async<$T>$ { $Stmt$ ; return ... ; }
  - forasync (point[i,j] : ...) $Stmt$
  - forall (point[i,j] : ...) $Stmt$

- Loop constructs
  - point
  - region
  - for (point[i,j] : ...) $Stmt$
Recap of HJ constructs studied so far (contd)

- Synchronization constructs
  - finish
  - f.get()
  - finish accumulators
  - next
  - async await

- Efficiency constructs
  - Converting async to async seq
  - Loop chunking with GetChunk()
  - Converting futures to data-driven futures

- Abstract metrics
  - perf.doWork(n)
Outline of Today’s Lecture

• Recap of HJ constructs studied so far
• **Point-to-Point Synchronization, Pipeline Parallelism**
• Introduction to Phasers

**Acknowledgments**

• COMP 322 Module 1 handout, Sections 12.1, 12.2
• Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. Cherri Pancake.
  —Source for seismic imaging example
  —Source for figures related to pipeline parallelism
Point-to-Point Synchronization: Example 1

1. `finish { // Expanded finish-forasync version of forall`
2. `forasync (point[i] : [1:m]) {`
3. `doPhase1(i);`
4. `// Iteration i waits for i-1 and i+1 to complete Phase 1`
5. `doPhase2(i);`
6. `}`
7. `}

- Need synchronization where iteration i only waits for iterations i-1 and i+1 to complete their work in `doPhase1()` before it starts `doPhase2(i)`
  - Less constrained than a barrier --- only waits for two preceding iterations
  - More general than async await --- waiting occurs in middle of task

```
\begin{array}{cccccccc}
i=1 & i=2 & i=3 & i=4 & i=5 & i=6 & i=7 & i=8 \\
\end{array}
```

\begin{itemize}
\item \textbf{doPhase1(i)}
\item \textbf{doPhase2(i)}
\end{itemize}
Point-to-point Synchronization: Example 2
Pipeline Parallelism

- Seismic imaging pipeline example with three stages
  1. Simulation generates a sequence of results, one per time step.
  2. Rendering takes simulation results for one time step as input, and generates an image for that time step.
  3. Formatting image as input and outputs it into an animation sequence.

- Even though the processing is sequential for a single time step, pipeline parallelism can be exploited via point-to-point synchronization between neighboring stages
General structure of a One-Dimensional Pipeline

Input sequence
\[ d_9 d_8 d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0 \]

- Assuming that the inputs \( d_0, d_1, \ldots \) arrive sequentially, pipeline parallelism can be exploited by enabling task (stage) \( P_i \) to work on item \( d_{k-i} \) when task (stage) \( P_0 \) is working on item \( d_k \).
Timing Diagram for One-Dimensional Pipeline

- Horizontal axis shows progress of time from left to right, and vertical axis shows which data item is being processed by which pipeline stage at a given time.
Complexity Analysis of One-Dimensional Pipeline

• Assume
  — $n =$ number of items in input sequence
  — $p =$ number of pipeline stages
  — each stage takes 1 unit of time to process a single data item

• WORK = $n \times p$ is the total work for all data items

• CPL = $n + p - 1$ is the critical path length of the pipeline

• Ideal parallelism, PAR = WORK/CPL = $np/(n + p - 1)$

• Boundary cases
  — $p = 1$ $\Rightarrow$ PAR = $n/(n + 1 - 1) = 1$
  — $n = 1$ $\Rightarrow$ PAR = $p/(1 + p - 1) = 1$
  — $n = p$ $\Rightarrow$ PAR = $p/(2 - 1/p) \approx p/2$
  — $n \gg p$ $\Rightarrow$ PAR $\approx p$
Outline of Today’s Lecture

• Recap of HJ constructs studied so far
• Point-to-Point Synchronization, Pipeline Parallelism
• Introduction to Phasers

Acknowledgments

• COMP 322 Module 1 handout, Sections 12.1, 12.2
• Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. Cherri Pancake.
  —Source for seismic imaging example
  —Source for figures related to pipeline parallelism
Barrier vs Point-to-Point Synchronization for One-Dimensional Iterative Averaging Example

Barrier synchronization

Point-to-point synchronization
(Left-right neighbor synchronization)
Phasers: a unified construct for barrier and point-to-point synchronization

• Previous examples motivated the need for point-to-point synchronization
• HJ phasers unify barriers with point-to-point synchronization
• A limited version of phasers was also added to the Java 7 java.util.concurrent.Phase library (with acknowledgment to Rice)

• Phaser properties
  — Barrier and point-to-point synchronization
  — Supports dynamic parallelism i.e., the ability for tasks to drop phaser registrations on termination, and for new tasks to add new phaser registrations.
  — Deadlock freedom
  — Support for phaser accumulators (reductions that can be performed with phasers)
Summary of Phaser Construct

- Phaser allocation
  - `phaser ph = new phaser(mode);`
  - Phaser ph is allocated with registration mode
  - Phaser lifetime is limited to scope of Immediately Enclosing Finish (IEF)

- Registration Modes
  - `phaserMode.SIG, phaserMode.WAIT, phaserMode.SIG_WAIT, phaserMode.SIG_WAIT_SINGLE`
  - NOTE: phaser WAIT is unrelated to Java wait/notify (which we will study later)

- Phaser registration
  - `async phased (ph<sub>1</sub><mode<sub>1</sub>>, ph<sub>2</sub><mode<sub>2</sub>>, ... ) <stmt>`
    - Spawned task is registered with `ph<sub>1</sub>` in `mode<sub>1</sub>`, `ph<sub>2</sub>` in `mode<sub>2</sub>`, ...
    - Child task’s capabilities must be subset of parent’s
    - `async phased <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
1. `finish`
2. `ph = new phaser(); // Default mode is SIG_WAIT`
3. `async phased(ph<phaserMode.SIG>){ //A1 (SIG mode)`
4. `    doA1Phase1(); next;`
5. `    doA1Phase2(); }`
6. `async phased { //A2 (default SIG_WAIT mode from parent)`
7. `    doA2Phase1(); next;`
8. `    doA2Phase2(); }`
9. `async phased { //A3 (default SIG_WAIT mode from parent)`
10. `    doA3Phase1(); next;`
11. `    doA3Phase2(); }`
12. `async phased(ph<phaserMode.WAIT>){ //A4 (WAIT mode)`
13. `    doA4Phase1(); next; doA4Phase2(); }`
14. `}`
Semantics of `next` depends on registration mode:

- **SIG_WAIT**: `next = signal + wait`
- **SIG**: `next = signal` *(Don’t wait for any task)*
- **WAIT**: `next = wait` *(Don’t disturb any task)*

A master task receives all signals and broadcasts a barrier completion.
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.
Listing 46: Example of left-right neighbor synchronization for $m = 3$ case