COMP 322: Fundamentals of Parallel Programming

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

Lecture 14: Point-to-point Synchronization, Pipeline Parallelism, Phasers

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

Announcements

- Homework 4 due by 5pm on Wednesday, Feb 16th
  - We will try and return graded homeworks by Feb 23rd
- Guest lecture on Bitonic Sort by John Mellor-Crummey on Friday, Feb 18th
- Feb 23rd lecture will be a Midterm Review
- No lecture on Friday, Feb 25th since midterm is due that day
  - Midterm will be a 2-hour take-home written exam
    - Closed-book, closed-notes, closed-computer
  - Will be given out at lecture on Wed, Feb 23rd
  - Must be handed in by 5pm on Friday, Feb 25th

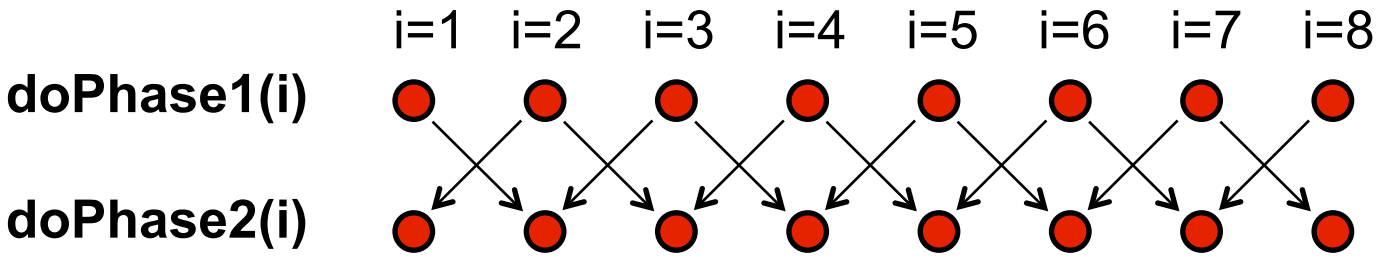
Acknowledgments for Today's Lecture

- [1] "X10: an object-oriented approach to non-uniform computing". Philippe Charles et al. OOPSLA 2005.
- [4] Knowing When to Parallelize: Rules-of-Thumb based on User Experiences. Cherri Pancake.
  - Source for seismic imaging example
- [5] Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. Jun Shirako et al. ICS '08
- [6] Barry Wilkinson and Michael Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition). Prentice-Hall, 2004.
  - Source for figures related to pipeline parallelism
- Handout for Lectures 14 and 15

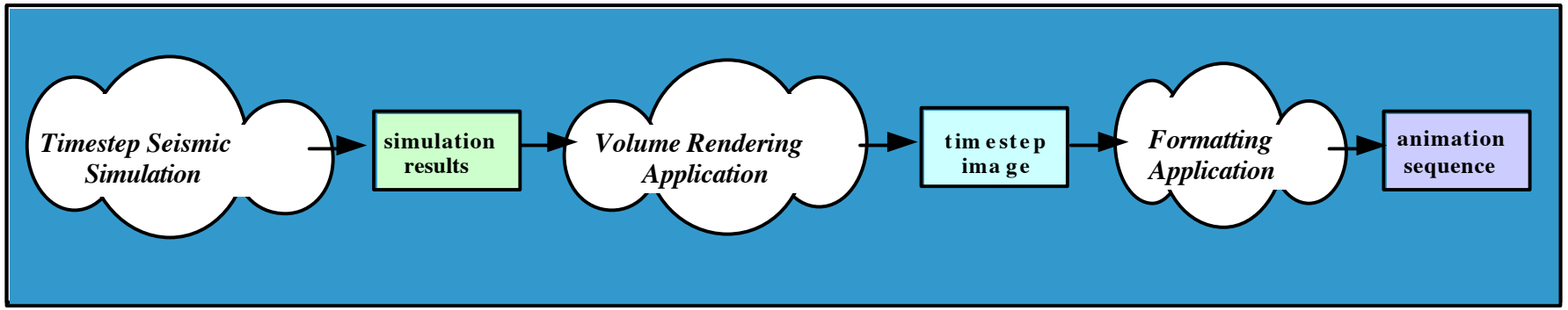
Point-to-Point Synchronization: Example 1

```
1.  finish { // Expanded finish-for-async version of forall
2.    for (point[i] : [1:m])
3.      async {
4.        doPhase1(i);
           // 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)

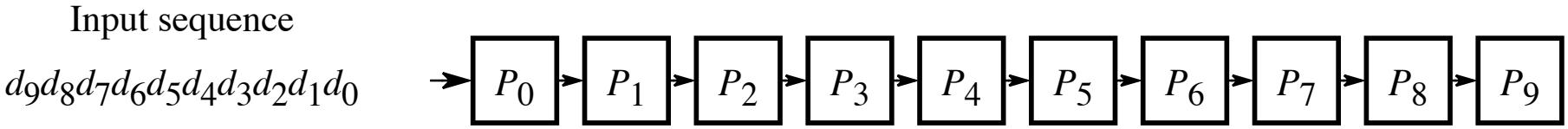


Point-to-point Synchronization: Example 2 Pipeline Parallelism



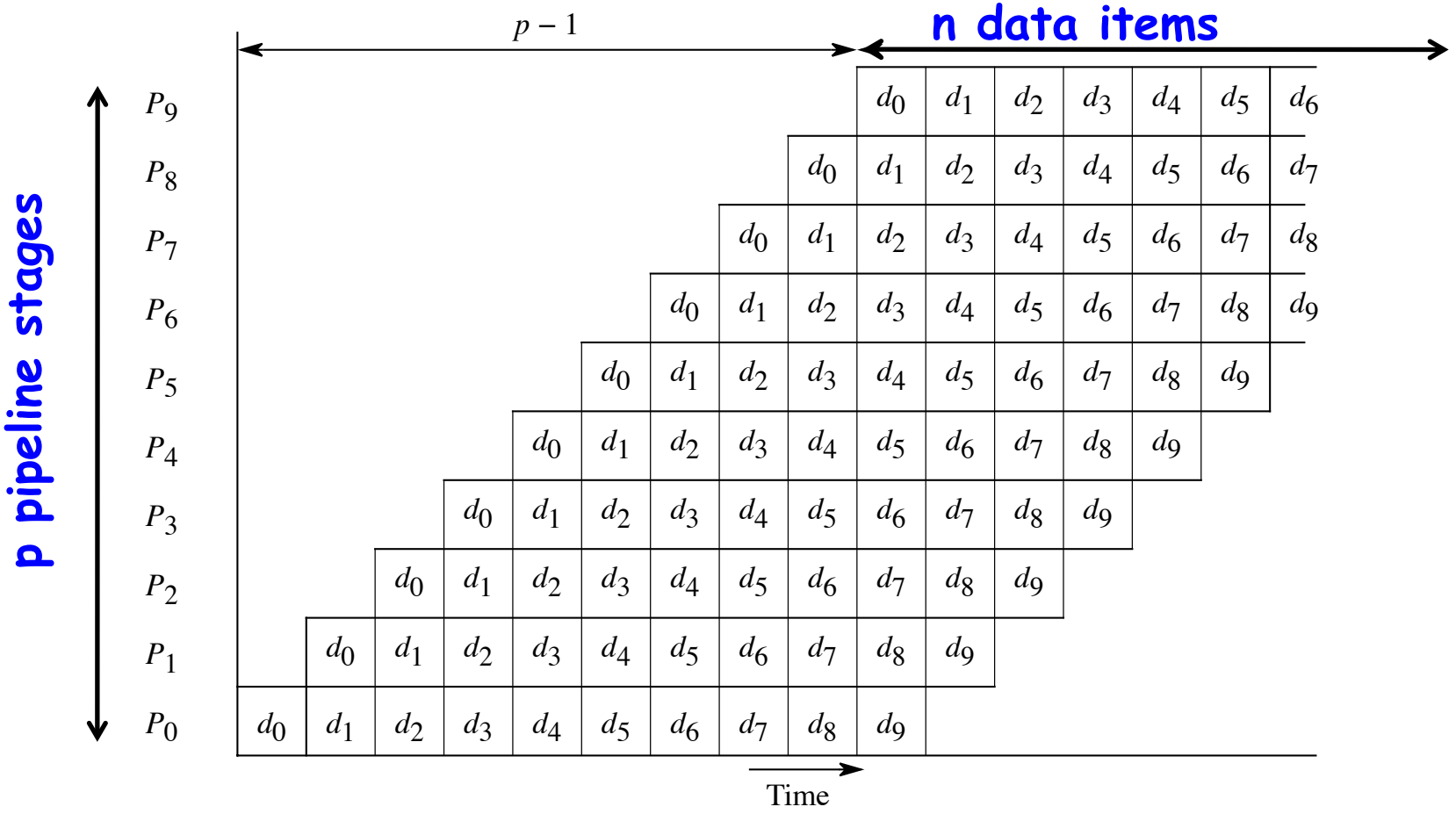
- Seismic imaging pipeline example with three stages
  1. *Simulation* generates a sequence of results, one per time step.
  2. *Rendering* takes simulation results for one time step as input, and generates an image for that time step.
  3. *Formatting* image as input and outputs it into an animation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



- Assuming that the inputs d_0, d_1, . . . 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.

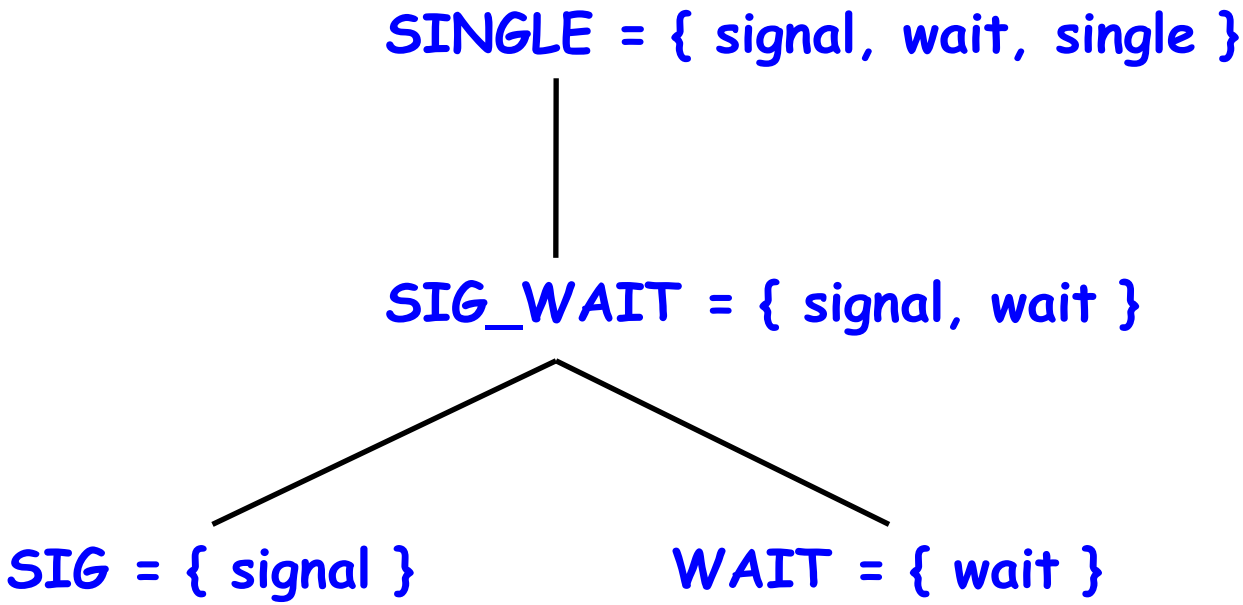
Complexity Analysis of One-Dimensional Pipeline

- 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×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 ➔ PAR = n/(n + 1 – 1) = 1
  - n = 1 ➔ PAR = p/(1 + p – 1)
  - n = p ➔ PAR = p/(2 – 1/p)
  - n ≫ p ➔ PAR approaches p in the limit

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

- Previous examples motivated the need for point-to-point synchronization
- HJ phasers were derived from the clock construct in X10, with extensions added for point-to-point synchronization
- A limited version of phasers was also added to the Java 7 java.util.concurrent.Phaser library [3]
- Phaser capabilities
  - Unifies point-to-point and barrier synchronization
  - Supports dynamic parallelism i.e., the ability for tasks to drop phaser registrations and for new tasks to add new phaser registrations.
  - Deadlock freedom
  - Support for phaser accumulators (reductions that can be performed with phasers)
  - Support for streaming parallelism

Capability Hierarchy



Phaser Operations in Habanero Java

- 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*
  - *SIG*
  - *WAIT*
  - *SIG_WAIT*
  - *SINGLE*
- Phaser registration
  - async phased (ph_1<$mode_1$>, ph_2<$mode_2$>, …) <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*
    - 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

A Simple Example

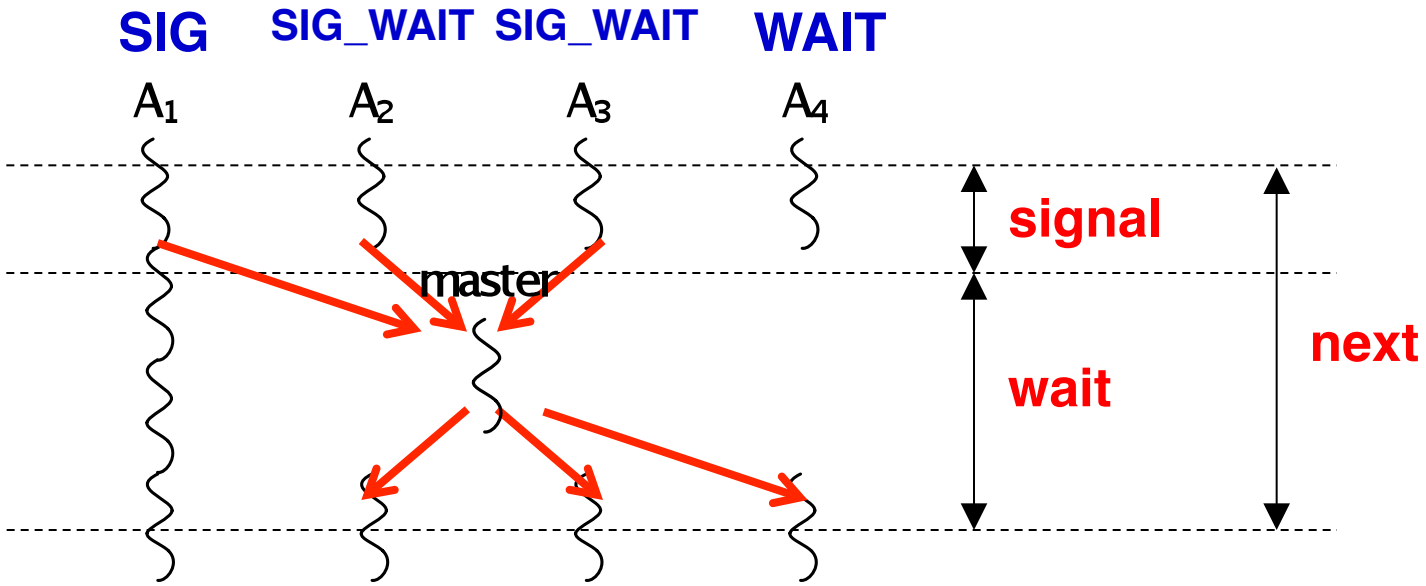
```
1  finish {
2    ph = new phaser(); // Default mode is SIG_WAIT
3    async phased(ph<SIG>){doA1Phase1(); next; doA1Phase2();} //A1 (SIG mode)
4    async phased{doA2Phase1(); next; doA2Phase2();} //A2 (SIG_WAIT mode)
5    async phased{doA3Phase1(); next; doA3Phase2();} //A3 (SIG_WAIT mode)
6    async phased(ph<WAIT>){doA4Phase1(); next; doA4Phase2();} //A4 (WAIT mode)
7  }
```

Listing 2: Simple example with four **async** tasks and one phaser

next operation

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

Left-Right Neighbor Synchronization Example for m=3

```
1  finish {
2    phaser ph1 = new phaser(); // Default mode is SIG_WAIT
3    phaser ph2 = new phaser(); // Default mode is SIG_WAIT
4    phaser ph3 = new phaser(); // Default mode is SIG_WAIT
5    async phased(ph1<SIG>, ph2<WAIT>) { // i = 1
6      doPhase1(1);
7      next; // Signals ph1, and waits on ph2
8      doPhase2(1);
9    }
10   async phased(ph2<SIG>, ph1<WAIT>, ph3<WAIT>) { // i = 2
11     doPhase1(2);
12     next; // Signals ph2, and waits on ph1 and ph3
13     doPhase2(2);
14   }
15   async phased(ph3<SIG>, ph2<WAIT>) { // i = 3
16     doPhase1(3);
17     next; // Signals ph3, and waits on ph2
18     doPhase2(3);
19   }
20 }
```

Listing 3: Extension of example in Listing 1 with three phasers for $m = 3$