COMP 322: Fundamentals of Parallel Programming

Lecture 17: Pipeline Parallelism, Signal Statement, Fuzzy Barriers

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Complete the phased clause below to implement the left-right neighbor synchronization shown above.

1. \textbf{finish} () -> {
2. \hspace{1cm} \textbf{final \ HjPhaser[]} \ ph =
   \hspace{1cm} \hspace{1cm} new \ HjPhaser[m+2]; // array of phaser objects
3. \hspace{1cm} \textbf{forseq}(0, \ m+1, \ (i) \to \ { \ ph[i] = \textbf{newPhaser}(SIG\_WAIT) });
4. \hspace{1cm} \textbf{forseq}(1, \ m, \ (i) \to \ {
5. \hspace{1cm} \hspace{1cm} \textbf{asyncPhased}(
6. \hspace{1cm} \hspace{1cm} \hspace{1cm} \ph[i-1].\text{inMode}(WAIT),
7. \hspace{1cm} \hspace{1cm} \hspace{1cm} \ph[i].\text{inMode}(SIG),
8. \hspace{1cm} \hspace{1cm} \hspace{1cm} \ph[i+1].\text{inMode}(WAIT), \ () \to \ { \ doPhase1(i); \next(); \doPhase2(i); }); // asyncPhased
9. \hspace{1cm} }); // forseq
10. }); // finish
Left-Right Neighbor Synchronization (restricted to m=3)

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)
Computation Graph for m=3 example (with async-finish nodes and edges)
Medical imaging pipeline

- New reconstruction methods
  - decrease radiation exposure (CT)
  - number of samples (MR)
- 3D/4D image analysis pipeline
  - Denoising
  - Registration
  - Segmentation
- Analysis
  - Real-time quantitative cancer assessment applications
- Potential:
  - order-of-magnitude performance improvement
  - power efficiency improvements
  - real-time clinical applications and simulations using patient imaging data

Slide credit: NSF Expeditions Center for Domain-Specific Computing (UCLA, Rice, OSU, UCSB)
Medical imaging pipeline with three stages

1. Denoising stage generates a sequence of results, one per image.
2. Registration stage’s input is Denoising stage’s output.
3. Segmentation stage’s input is Registration stage’s output.

Even though the processing is sequential for a single image, *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

- \( \text{WORK} = n \times p \) is the total work for all data items

- \( \text{CPL} = n + p - 1 \) is the critical path length of the pipeline

- Ideal parallelism, \( \text{PAR} = \frac{\text{WORK}}{\text{CPL}} = \frac{np}{n + p - 1} \)

- Boundary cases
  - \( p = 1 \) \( \Rightarrow \) \( \text{PAR} = \frac{n}{n + 1 - 1} = 1 \)
  - \( n = 1 \) \( \Rightarrow \) \( \text{PAR} = \frac{p}{1 + p - 1} = 1 \)
  - \( n = p \) \( \Rightarrow \) \( \text{PAR} = \frac{p}{2 - 1/p} \approx \frac{p}{2} \)
  - \( n \gg p \) \( \Rightarrow \) \( \text{PAR} \approx p \)
Producer-Consumer pattern with phasers
(used for implementing pipeline parallelism)

1. asyncPhased(ph.inMode(SIG), () -> {
2.     for (int i = 0; i < rounds; i++) {
3.         buffer.insert(...);
4.         // producer can go ahead as they are in SIG mode
5.         next();
6.     }
7. });
8. }
9. asyncPhased(ph.inMode(WAIT), () -> {
10.    for (int i = 0; i < rounds; i++) {
11.        next();
12.        buffer.remove(...);
13.    }
14. });
Signal statement & Fuzzy barriers

- When a task T performs a **signal** operation, it notifies all the phasers it is registered on that it has completed all the work expected by other tasks (“shared” work) in the current phase.

- Later, when T performs a **next** operation, the next degenerates to a wait since a signal has already been performed in the current phase.

- The execution of “local work” between **signal** and **next** is overlapped with the phase transition (referred to as a “split-phase barrier” or “fuzzy barrier”)

```
1. forall (point[i] : [0:1]) {
2.   A(i); // Phase 0
3.   if (i==0) { signal; B(i); }
4.   next; // Barrier
5.   C(i); // Phase 1
6.   if (i==1) { D(i); }
7. }
```
Another Example of a Split-Phase Barrier using the Signal Statement

```java
1. finish(() -> {
2.   final HjPhaser ph = newPhaser(SIG_WAIT);
3.   asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
4.     a = ... ; // Shared work in phase 0
5.     signal(); // Signal completion of a's computation
6.     b = ... ; // Local work in phase 0
7.     next(); // Barrier -- wait for T2 to compute x
8.     b = f(b,x); // Use x computed by T2 in phase 0
9.   });
10.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
11.    x = ... ; // Shared work in phase 0
12.    signal(); // Signal completion of x's computation
13.    y = ... ; // Local work in phase 0
14.    next(); // Barrier -- wait for T1 to compute a
15.    y = f(y,a); // Use a computed by T1 in phase 0
16.   });
17.}); // finish
```
Computation Graph for Split-Phase Barrier Example (without async-finish nodes and edges)
Full Computation Graph for Split-Phase Barrier Example

2

4 → 5-signal → 6

ph.next
-start(0→1)

7-wait

14-wait → 15

12-signal → 13

17-drop → 17-end-finish

8

spawn

continue

signal

wait

join
Announcements

• Take-home midterm exam (Exam 1) will be given after lecture on Wednesday, February 25, 2015
  — Closed-book, closed computer, written exam that can be taken in any 2-hour duration during that period
  — Will need to be returned to Bel Martinez (Duncan Hall 3122) by 4pm on Friday, February 27, 2015
    — Exam can also be picked up from Bel Martinez starting 2pm on Feb 25th if you’re unable to attend lecture.
  — No lecture on Friday, Feb 27th

• Homework 3 is due by by 5:00pm on Friday, March 13, 2015
  — Programming assignment is more challenging than in previous homeworks --- start early!
Scope of Midterm Exam

• Midterm exam will cover material from Lectures 1 - 18
  — Lecture 19 (Feb 25th) will be a Midterm review

• Excerpts from midterm exam instructions
  — “closed-book, closed-notes, closed-computer”
  — “Record start time when you open the exam, and end time when you finish. The total duration must be at most 2 hours.”
  — “Since this is a written exam and not a programming assignment, syntactic errors in program text will not be penalized (e.g., missing semicolons, incorrect spelling of keywords, etc) so long as the meaning of your solution is unambiguous.”
  — “If you believe there is any ambiguity or inconsistency in a question, you should state the ambiguity or inconsistency that you see, as well as any assumptions that you make to resolve it.”