COMP 322: Fundamentals of Parallel Programming

Lecture 16: Phasers, Point-to-Point Synchronization

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

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Recap of Multiprocessor Scheduling of a Computation Graph (Lecture 3)

This schedule was obtained by mapping computation graph nodes to processor assuming:
1. Non-preemption (no context switch in the middle of a node)
2. Greedy schedule (a processor is never idle if work is available)

There may be multiple possible schedules with these assumptions

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two possible HJ programs for this Computation Graph (there can be others ...)

There is no significance to the left-to-right ordering of edges in a computation graph, which is why there can be multiple parallel programs for the same computation graph.

// Program Q1
A;
finish {
    async { B; D; }
    async F;
    async { C; E; }
}

// Program Q2
A;
finish {
    async { C; E; }
    async F;
    async { B; D; }
}
Work-first vs. Help-first work-stealing policies (Lec 15)

- When encountering an async
  - Help-first policy
    - Push async on “bottom” of local queue, and execute next statement
  - Work-first policy
    - Push continuation (remainder of task starting with next statement) on “bottom” of local queue, and execute async

- When encountering the end of a finish scope
  - Help-first policy & Work-first policy
    - Store continuation for end-finish
      - Will be resumed by last async to complete in finish scope
    - Pop most recent item from “bottom” of local queue
    - If local queue is empty, steal from “top” of another worker’s queue

- Current HJ-lib runtime only supports help-first policy
Scheduling Program Q1 using a Work-First Work-Stealing Scheduler

1. // Program Q1
2. A; // Executes on P1
3. finish {
4.   // P1 pushes continuation for 9,
5.   // and executes 6
6.   async { B; D; }
7.   // P2 pushes continuation for 11,
8.   // and executes 9
9.   async F;
10.  // P2 executes 11
11.  async { C; E; }
12.  }

<table>
<thead>
<tr>
<th>Start time</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>F</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>9</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Scheduling Program Q1 using a Help-First Work-Stealing Scheduler

1. // Program Q1
2. A; // Executes on P1
3. finish {
4. // P1 pushes 6, which is then
5. // stolen by P2
6. async { B; D; }
7. // P1 pushes 8
8. async F;
9. // P1 pushes 10
10. async { C; E; }
11. }
12. // P1 stores continuation and pops 10
13. // P1 pops 8
Worksheet #15 solution: Work-First vs. Help-First Work-Stealing Policies

For each of the continuations below, label it as “WF” if a work-first worker can switch from one task to another at that point and as “HF” if a help-first worker can switch from one task to another at that point. Some continuations may have both labels.

1. `finish { // F1` **WF**
2. `async A1;` **WF**
3. `finish { // F2` **WF**
4. `async A3;` **WF**
5. `async A4;` **WF, HF**
6. `}` **WF, HF**
7. `S5;` **WF, HF**
8. `}` **WF, HF**
HJ code for One-Dimensional Iterative Averaging with forall-forseq structure and barriers (Recap from Lec 12)

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 executing 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
Question: when can the point-to-point computation graph result in a smaller CPL than the barrier computation graph?
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

```java
1. finish (() -> {
2.     ph = new Phaser(HjPhaserMode.SIG_WAIT); // mode is SIG_WAIT
3.     asyncPhased(ph.inMode(HjPhaserMode.SIG), () -> {
4.         // A1 (SIG mode)
5.         doA1Phase1(); next(); doA1Phase2(); });
6.     asyncPhased(ph.inMode(HjPhaserMode.DEFAULT_MODE), () -> {
7.         // A2 (default SIG_WAIT mode from parent)
8.         doA2Phase1(); next(); doA2Phase2(); });
9.     asyncPhased(ph.inMode(HjPhaserMode.DEFAULT_MODE), () -> {
10.        // A3 (default SIG_WAIT mode from parent)
11.        doA3Phase1(); next(); doA3Phase2(); });
12.    asyncPhased(ph.inMode(HjPhaserMode.WAIT), () -> {
13.        // A4 (WAIT mode)
14.        doA4Phase1(); next(); doA4Phase2(); });
15. });
```
Simple Example with Four Async Tasks and One Phaser

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

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

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

- Synchronization
  - \( \text{next();} \)
  - 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
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.
forall barrier is just an implicit phaser!

1. `forallPhased(iLo, iHi, (i) -> {
2.     s1; next(); s2; next(); {...}
3. });`

is equivalent to

1. `finish(() -> {
2.     // Implicit phaser for forall barrier
3.     final HjPhaser ph = newPhaser(SIG_WAIT);
4.     forseq(iLo, iHi, (i) -> {
5.         asyncPhased(ph.inMode(SIG_WAIT), () -> {
6.             s1; next(); s2; next(); {...}
7.         }); // next statements in async refer to ph
8.     });`
The world according to COMP 322 before Barriers and Phasers

- All the other parallel constructs that we learned focused on task creation and termination
  - async creates a task
    - forasync creates a set of tasks specified by an iteration region
  - finish waits for a set of tasks to terminate
    - forall (like “finish forasync”) creates and waits for a set of tasks specified by an iteration region
  - future get() waits for a specific task to terminate
  - asyncAwait() waits for a set of DataDrivenFuture values before starting

- Motivation for barriers and phasers
  - Deterministic directed synchronization within tasks
  - Separate from synchronization associated with task creation and termination
The world according to COMP 322 after Barriers and Phasers

- **SPMD model:** express iterative synchronization using phasers
  - Implicit phaser in a forall supports barriers as “next” statements
    - Matching of next statements occurs dynamically during program execution
    - Termination signals “dropping” of phaser registration
  - Explicit phasers
    - Can be allocated and transmitted from parent to child tasks
    - Phaser lifetime is restricted to its IEF (Immediately Enclosing Finish) scope of its creation
    - Four registration modes -- SIG, WAIT, SIG_WAIT, SIG_WAIT_SINGLE
    - Signal statement can be used to support “fuzzy” barriers
    - Bounded phasers can limit how far ahead producer gets of consumers

- **Difference between phasers and data-driven tasks (DDTs)**
  - DDTs enforce a single point-to-point synchronization at the start of a task
  - Phasers enforce multiple point-to-point synchronizations within a task