CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar
Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/
Recap of Lecture 5

- Design Patterns for Parallel Programming
- Forasync and Forall loops
Worksheet #5 (to be done in pairs):
Use of seq clause in Quicksort() program

Insert seq clauses on the right to ensure that an async is only created for calls to quicksort with >= 10,000 elements

1. static void quicksort(int[] A, int M, int N) {
2.     if (M < N) { // sort A[M...N]
3.         // partition() selects a pivot element in A[M...N]
5.         point p = partition(A, M, N);
6.         int I=p.get(0); int J=p.get(1);
7.         async seq(J-M+1 < 10000) quicksort(A, M, J);
8.         async seq(N-I+1 < 10000) quicksort(A, I, N);
9.     }
10. } //quicksort
11. . . .
12. finish quicksort(A, 0, A.length-1);
One-Dimensional Iterative Averaging Example

- Initialize a one-dimensional array of (n+2) double's with boundary conditions, myVal[0] = 0 and myVal[n+1] = 1.

- In each iteration, each interior element myVal[i] in 1..n is replaced by the average of its left and right neighbors.
  - Two separate arrays are used in each iteration, one for old values and the other for the new values.

- After a sufficient number of iterations, we expect each element of the array to converge to myVal[i] = i/(n+1)
  - In this case, myVal[i] = (myVal[i-1]+myVal[i+1])/2, for all i in 1..n

Illustration of an intermediate step for n = 8 (source: Figure 6.19 in Lin-Snyder book)
HJ code for One-Dimensional Iterative Averaging with barriers (forall-for-next structure)

1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2]; gVal[n+1]=1; gNew[n+1]=1;
2. forall (point [j] : [1:n]) {
3.    double[] myVal = gVal; double[] myNew = gNew; // Local copy of myVal/myNew pointers
4.    for (point [iter] : [0:numIters-1]) {
5.        // Compute MyNew as function of input array MyVal
6.        myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
7.        next; // Barrier before executing next iteration of iter loop
8.        // Swap myVal and myNew (each forall iteration swaps
9.        // its pointers in local vars)
10.       double[] temp=myVal; myVal=myNew; myNew=temp;
11.       // myNew becomes input array for next iter
12.    } // for
13. } // forall

- Overhead issue --- this version creates n async tasks, but performs numIters barrier operations on n tasks
  — Good trade-off since barrier operations have lower overhead than task creation
Outline of Today’s Lecture

• Three observations related to Forall Barriers
• Point-to-point Synchronization and Phasers
• Phasers and Forall Loops, Single statement, Phaser Accumulators
• Signal statement and split-phase barriers
Observation 1: Scope of synchronization for “next” is closest enclosing forall statement

forall (point [i] : [0:m-1]) {
    System.out.println("Starting forall iteration " + i);
    next; // Acts as barrier for forall-i
forall (point [j] : [0:n-1]) {
    System.out.println("Hello from task (" + i + "," + j + ")");
    next; // Acts as barrier for forall-j
    System.out.println("Goodbye from task (" + i + "," + j + ")");
} // forall-j
next; // Acts as barrier for forall-i
System.out.println("Ending forall iteration " + i);
} // forall-i
Observation 2: If a forall iteration terminates before "next", then other iterations do not wait for it

1. forall (point[i] : [0:m-1]) {
2.     for (point[j] : [0:i]) {
3.         // Forall iteration i is executing phase j
4.         System.out.println("(" + i + "," + j + ")");
5.         next;
6.     }
7. }

- Outer forall-i loop has m iterations, 0…m-1
- Inner sequential j loop has i+1 iterations, 0…i
- Line 4 prints (task,phase) = (i, j) before performing a next operation.
- Iteration i = 0 of the forall-i loop prints (0, 0), performs a next, and then terminates. Iteration i = 1 of the forall-i loop prints (1,0), performs a next, prints (1,1), performs a next, and then terminates. And so on.
Illustration of Observation 2

- Iteration $i=0$ of the forall-$i$ loop prints (0, 0) in Phase 0, performs a next, and then ends Phase 1 by terminating.

- Iteration $i=1$ of the forall-$i$ loop prints (1, 0) in Phase 0, performs a next, prints (1, 1) in Phase 1, performs a next, and then ends Phase 2 by terminating.

- And so on until iteration $i=8$ ends an empty Phase 8 by terminating.

$i=0 \ldots 7$ are forall iterations

$(i,j) =$ println output

next = barrier operation

end = termination of a forall iteration
Observation 3: Different forall iterations may perform “next” at different program points

1. `forall (point[i] : [0:m-1])` {
2.     `if (i % 2 == 1)` { // i is odd
3.         `oddPhase0(i);`
4.         `next;`
5.         `oddPhase1(i);`
6.     } else { // i is even
7.         `evenPhase0(i);`
8.         `next;`
9.         `evenPhase1(i);`
10. } // if-else
11. } // forall

- Barrier operation synchronizes odd-numbered iterations at line 4 with even-numbered iterations in line 8
- `next` statement may even be in a method such as `oddPhase1()`
Outline of Today’s Lecture

- Three observations related to Forall Barriers
- **Point-to-point Synchronization and Phasers**
- Phasers and Forall Loops, Single statement, Phaser Accumulators
- Signal statement and split-phase barriers
Barrier vs Point-to-Point Synchronization for One-Dimensional Iterative Averaging Example

**Barrier synchronization**

iter = i

iter = i + 1

**Point-to-point synchronization**

(Left-right neighbor synchronization)

iter = i

iter = i + 1
Phasers: a unified construct for barrier and point-to-point synchronization

- Previous example 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 has no relationship to Java wait/notify

- 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
Simple Example with Four Async Tasks and One Phaser

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. `}`
Simple Example with Four Async Tasks and One Phaser (contd)

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.

- SIG_WAIT_SINGLE = \{ signal, wait, single \}
- SIG_WAIT = \{ signal, wait \}
- SIG = \{ signal \}
- WAIT = \{ wait \}

1. SIG_WAIT_SINGLE is a superset of SIG_WAIT.
2. SIG is a subset of SIG_WAIT.
3. WAIT is a subset of SIG_WAIT.

The hierarchy can be visualized as follows:

```
SIG_WAIT_SINGLE
   /          /
SIG_WAIT    SIG
         /     /   
      SIG    WAIT
```
Left-Right Neighbor Synchronization Example for m=3

```java
finish {
    phaser ph1 = new phaser(); // Default mode is SIG_WAIT
    phaser ph2 = new phaser(); // Default mode is SIG_WAIT
    phaser ph3 = new phaser(); // Default mode is SIG_WAIT
    async phased(ph1<SIG>, ph2<WAIT>) { // i = 1
        doPhase1(1);
        next; // Signals ph1, and waits on ph2
        doPhase2(1);
    }
    async phased(ph2<SIG>, ph1<WAIT>, ph3<WAIT>) { // i = 2
        doPhase1(2);
        next; // Signals ph2, and waits on ph1 and ph3
        doPhase2(2);
    }
    async phased(ph3<SIG>, ph2<WAIT>) { // i = 3
        doPhase1(3);
        next; // Signals ph3, and waits on ph2
        doPhase2(3);
    }
}
```
Computation Graph for m=3 example (see Module 1 handout for details)

Let's try another phaser example in Worksheet 7!
One-Dimensional Iterative Averaging with Point-to-Point Synchronization

1. `double[] gVal = new double[n+2]; double[] gNew = new double[n+2];`
2. `gVal[n+1] = 1; gNew[n+1] = 1;`
3. `phaser ph = new phaser[n+2];`
4. `finish { // phasers must be allocated in finish scope`
5. `forall(point [i]:[0:n+1]) ph[i] = new phaser();`
7. `double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers`
8. `for (point [iter] : [0:numIters-1]) {`
9. `myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;`
10. `next; // Point-to-point synchronization`
11. `// Swap myVal and myNew`
12. `double[] temp=myVal; myVal=myNew; myNew=temp;`
13. `// myNew becomes input array for next iter`
14. } // for-iter`
15. `}` // forasync-j
16. `} // finish`
Outline of Today’s Lecture

- Three observations related to Forall Barriers
- Point-to-point Synchronization and Phasers
- Phasers and Forall Loops, Single statement, Phaser Accumulators
- Signal statement and split-phase barriers
forall barrier is just an implicit phaser

1. forall (point[i,j] : [iLo:iHi,jLo:jHi])
2. <body>

is equivalent to

3. finish {
4.     // Implicit phaser
5.     phaser ph = new phaser(phaserMode.SIG_WAIT_SINGLE);
6.     for(point[i,j] : [iLo:iHi,jLo:jHi])
7.         async phased(phaserMode.SIG_WAIT_SINGLE)
8.         <body>  // next statements refer to ph
9.     }

Next-with-Single Statement (for SIG_WAIT_SINGLE registration mode)

next <single-stmt> is a barrier in which single-stmt is performed exactly once after all tasks have completed the previous phase and before any task begins its next phase.

Modeling next-with-single in the Computation Graph

- **signal edges**
  - A1 → next-start
  - A2 → next-start
  - A3 → next-start
  - A4 → next-start

- **single-statement**
  - next-start → single-statement

- **wait edges**
  - single-statement → A1
  - single-statement → A2
  - single-statement → A3
  - single-statement → A4
Use of next-with-single to print a log message between Hello and Goodbye phases

1. `forall (point[i] : [0:m−1])` {
2.     // Start of Hello phase
3.     System.out.println("Hello from task " + i);
4.     next single {
5.         System.out.println("LOG: Between Hello & Goodbye Phases")
6.     }
7.     // Start of Goodbye phase
8.     System.out.println("Goodbye from task " + i);
9. } // forall
Accumulator motivation: Adding a max reduction to One-Dimensional Iterative Averaging with barriers

1. double[] gVal=new double[n+2]; double[] gNew=new double[n+2];
2. gVal[n+1]=1; gNew[n+1]=1;
3. forall (point [j] : [1:n]) {
4.   double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers
5.   for (point [iter] : [0:numIters-1]) {
6.     // Compute MyNew as function of input array MyVal
7.     myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
8.     // Compute normalized diff of element j w.r.t. converged value, j/(n+1)
9.     double nDiff = Math.abs(myNew[j]-myVal[j])/((double)j/(double)(n+1));
10.    // QUESTION: how to compute max(nDiff) for all elements in this phase??
11.    next; // Barrier before executing next iteration of iter loop
12.    // Swap myVal and myNew (each forall iteration swaps
13.    // its pointers in local vars)
14.    double[] temp=myVal; myVal=myNew; myNew=temp;
15.    // myNew becomes input array for next iter
16.   } // for
17. } // forall
Phaser Accumulators

- Phaser accumulators can accumulate values within a single phase e.g., between two “next” operations
- HJ provides different implementations for the same accumulator semantics
  - **Eager**: Concurrent atomic accumulation by multiple tasks
    - Optional delay function to reduce bus congestion in atomic updates
  - **Dynamic-lazy**: Sequential accumulation at synchronization point
  - **Fixed-Lazy**: Lightweight implementation of dynamic-lazy (limited dynamic parallelism)
- NOTE: phasers and phaser accumulators are currently only supported by HJ’s work-sharing runtime (w/ or w/o the fork-join variant, -fj), but not HJ’s work-stealing runtime system
Operations on Phaser Accumulators

• Creation

    accumulator ac = accumulator.factory.accumulator(op, type, phaser);

    - operator can be Operator.SUM, Operator.PROD, Operator.MIN, or Operator.MAX (as in finish accumulators)

    * Support for custom operators is in progress

    - type can be int.class or double.class (as in finish accumulators)

    - an extra “true” parameter results in lazy accumulation as in finish accumulators e.g.,

        accumulator.factory.accumulator(op, type, phaser, true)

• Accumulation

    ac.put(data);

    - data must be of type java.lang.Number, int, or double

    - Provides data for accumulation in current phase (can only be performed by a task registered on the phaser)

• Retrieval

    Number n = ac.get();

    - get() returns value from previous phase (can only be performed by a task registered on the phaser)

    - get() is non-blocking because the synchronization is handled by “next”

    - result from get() will be deterministic if HJ program does not use atomic or isolated constructs and is data-race-free (ignoring nondeterminism due to non-commutativity of arithmetic operations, e.g., underflow, overflow, rounding)
Example Usage of Phaser Accumulator API

1. `finish` {
2.   phaser ph = new phaser();
3.   accumulator a = accumulator.factory.accumulator(accumulator.SUM, int.class, ph);
4.   accumulator b = accumulator.factory.accumulator(accumulator.MIN, double.class, ph);
5.   for (int i = 0; i < n; i++) {
6.     async phased(ph<phaserMode.SIG_WAIT>) {
7.       int iv = 2*i + j;
8.       double dv = -1.5*i + j;
9.       a.put(iv);
10.      b.put(dv);
11.     } // async
12.   } // for
13. } // finish
14. int sum = a.get().intValue;
15. double min = b.get().doubleValue;
16. ...
17. } // for
18. } // for
19. }

Allocation: Specify operator and type

put: Send a value to accumulator

get: Return accumulator result from previous phase
```java
1. double gblMin = Double.MAX_VALUE;  double threshold = ...;
2. SearchSpace gss = new SearchSpace(...);  // Whole search space
3. finish {
4.   phaser ph = new phaser();
5.   accumulator a = accumulator.factory.accumulator(accumulator.MIN,
6.   double.class, ph);
7.   calcMin(ph, gss, a);
8. }
9. ...
10. void calcMin(phaser ph, SearchSpace mySs, accumulator a) {
11.   while (gblMin > threshold) {
12.     if (mySs.tooLarge()) {
13.       SearchSpace childSs = split(mySs);
14.       async phased { calcMin(ph, childSs, a); }
15.     }
16.     double localMin = findMin(mySs);
17.     a.put(localMin);
18.     next;
19.     gblMin = a.get().doubleValue();
20.     // update search spaces ...
21.   } // while
22. } // calcMin
```
Execution of previous HJ program

Root

async

localMin

a.put()

next

async

localMin

a.put()

next

async

localMin

a.put()

next

1st iteration of while loop

Reduction

globMin = a.get().doubleValue()

update

update

update

2nd iteration

Reduction

globMin = a.get().doubleValue()
Outline of Today’s Lecture

• Three observations related to Forall Barriers
• Point-to-point Synchronization and Phasers
• Phasers and Forall Loops, Single statement, Phaser Accumulators
• Signal statement and split-phase barriers
Signal statement

- 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 in the current phase ("shared" work).
  - Since signal is a non-blocking operation, an early execution of signal cannot create a deadlock.

- 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 performed during phase transition
  - Referred to as a "split-phase barrier" or "fuzzy barrier"
Example of Split-Phase Barrier

```
finish {
    phaser ph = new phaser(phaserMode.SIG_WAIT);
    async phased { // Task T1
        a = ... ; // Shared work in phase 0
        signal; // Signal completion of a’s computation
        b = ... ; // Local work in phase 0
        next; // Barrier — wait for T2 to compute x
        b = f(b,x); // Use x computed by T2 in phase 0
    }
    async phased { // Task T2
        x = ... ; // Shared work in phase 0
        signal; // Signal completion of x’s computation
        y = ... ; // Local work in phase 0
        next; // Barrier — wait for T1 to compute a
        y = f(y,a); // Use a computed by T1 in phase 0
    }
} // finish
```
Computation Graph for Split-Phase Barrier Example (without async and finish nodes and edges)

4 \rightarrow 5\text{-signal} \rightarrow 6 \rightarrow 7\text{-wait} \rightarrow 8

\text{spawn} \rightarrow \text{continue} \rightarrow \text{signal} \rightarrow \text{wait} \rightarrow \text{join}

\text{ph.next-start}(0 \rightarrow 1) \rightarrow \text{ph.next-end}(0 \rightarrow 1)
Full Computation Graph for Split-Phase Barrier Example

2

4 → 5-signal → 6

ph.next
- start(0 → 1)

11 → 12-signal → 13

14-wait → 15

20-drop → 20-end-finish

8

spawn

continue

signal

wait

join
SPMD Execution Model

• SPMD: Single Program Multiple Data
• Run the same program on P processing elements (PEs)
• Use the “rank” ... an ID ranging from 0 to (P-1) ... to determine what computation is performed on what data by a given PE
• Different PEs can follow different paths through the same code, unlike Single Instruction Multiple Data (SIMD) pattern
• Convenient pattern for hardware platforms that are not amenable to efficient forms of dynamic task parallelism
  — General-Purpose Graphics Processing Units (GPGPUs)
  — Distributed-memory parallel machines
• Key design decisions --- what data and computation should be replicated or partitioned across PEs?
Typical SPMD Program Phases

- **Initialize**
  - Establish localized data structure and communication channels

- **Obtain a unique identifier**
  - Each thread acquires a unique identifier, typically range from 0 to N=1, where N is the number of threads.
  - Both OpenMP and CUDA have built-in support for this.

- **Distribute Data**
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads

- **Run the core computation**
  - More details in next slide...

- **Finalize**
  - Reconcile global data structure, prepare for the next major iteration
SPMD Example #1

- Assign a chunk of iterations to each thread
  - The last thread also finishes up the remainder iterations

```c
num_steps = 1000000;

i_start = my_id * (num_steps / num_threads);
i_end = i_start + (num_steps / num_threads);
if (my_id == (num_threads - 1)) i_end = num_steps;

for (i = i_start; i < i_end; i++) {
    ....
}
```

Reconciliation of results across threads if necessary.
SPMD Example #2: Iterative Averaging Example with one Async per Processor

1. `double[] gVal=new double[n+2]; double[] gNew=new double[n+2];`
2. `gVal[n+1] = 1; // Boundary condition`
3. `int Cj = Runtime.getNumOfWorkers();`
4. `forall (point [jj]:[0:Cj-1]) { // SPMD computation`
5. `double[] myVal = gVal; double[] myNew = gNew; // Local copy`
6. `for (point [iter] : [0:numIters-1]) {
7.    // Compute MyNew as function of input array MyVal
8.       for (point [j]:getChunk([1:n],[Cj],[jj]))
9.          myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
10.     next; // Barrier before executing next iteration of iter loop
11.    // Swap myVal and myNew (replicated computation)
12.       double[] temp=myVal; myVal=myNew; myNew=temp;
13.    // myNew becomes input array for next iter
14. } // for
15.} // forall
1. double[] gVal = new double[n+2]; double[] gNew = new double[n+2];
2. gVal[n+1] = 1; gNew[n+1] = 1; // boundary conditions
3. int Cj = Runtime.getNumOfWorkers(); // number of chunks
4. finish {
5.   phaser ph = new phaser[Cj+2];
6.   for(point [i]:[0:Cj+1]) ph[i] = new phaser();
7.   forasync(point [jj]:[0:Cj-1]) phased(ph[jj+1]<SIG>, ph[jj]<WAIT>, ph[jj+2]<WAIT>) {
8.     double[] myVal = gVal; double[] myNew = gNew; // Local copy of pointers
9.     for (point [iter] : [0:numIters-1]) {
10.    region r = getChunk([1:n],Cj,jj); int lo = r.rank(0).low(); int hi = r.rank(0).high();
11.    myNew[lo] = (myVal[lo-1] + myVal[lo+1])/2.0;
12.    myNew[hi] = (myVal[hi-1] + myVal[hi+1])/2.0;
13.    signal; // done with shared work -- signal ph[jj+1]
14.   for (point [j]: [lo+1:hi-1]) // Iterate within chunk
15.    myNew[j] = (myVal[j-1] + myVal[j+1])/2.0;
16.   next; // done with local work --- wait on ph[jj] and ph[jj+2]
17.   // Swap myVal and myNew
18.   double[] temp=myVal; myVal=myNew; myNew=temp;
19.   // myNew becomes input array for next iter
20. } // for
21. } // finish
Worksheet #7 (to be done in pairs):
Left-Right Neighbor Synchronization using Phasers

Name 1: ___________________     doPhase1(i)
Name 2: ___________________     doPhase2(i)

Complete the phased clause below to implement the left-right neighbor synchronization shown above

1. finish {
2.     phaser[] ph = new phaser[m+2];
3.     for(point [i]:[0:m+1]) ph[i] = new phaser();
4.     for(point [i] : [1:m])
5.         async phased(________________________________________) {
6.             doPhase1(i);
7.             next;
8.             doPhase2(i);
9.         }
10.}