COMP 322: Fundamentals of Parallel Programming

Lecture 12: Finish Accumulators, Forall Statements & Barriers

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Goals for Today’s Lecture

• Finish Accumulators
• Forall statements and barriers
Summing Values from Multiple Async’s using AtomicInteger (Recap from Lecture 6)

1. import java.util.concurrent.atomic.AtomicInteger;
2. // Example 1: compute sum from async tasks in a loop
3. AtomicInteger a1 = new AtomicInteger();
4.   finish while(...)
5.     async { ...; elem = ...; a1.addAndGet(elem); ...; }
6. int sum1 = a1.get(); // returns sum from while loop
7. // Example 2: compute sum in a recursive method
8. AtomicInteger a2 = new AtomicInteger();
9. void visit(...)
10. { ...; elem = ...; a2.addAndGet(elem);
11.   async visit(...); ...;
12. }
13. ... finish visit(...); ...
14. int sum2 = a2.get(); // returns sum from while loop
From Atomic Variables to Accumulators

- Atomic variables are overkill if you just want the final sum
  - Semantic issues: programs that read the return value of methods like `addAndGet()` are nondeterministic in general
  - Performance issues: atomic variable can be a sequential bottleneck
- Instead, need a construct that just returns the final sum without revealing intermediate values
- Historically, this pattern has been captured by “accumulators” in computer instructions and programming languages
  - Load value from memory/register into accumulator
  - Add values into accumulator
  - Load value from accumulator into memory/register
Finish Accumulators in HJ

- **Creation**

  ```java
  accumulator ac = accumulator.factory.accumulator(operator, type);
  - operator can be Operator.SUM, Operator.PROD, Operator.MIN, or Operator.MAX
  - type can be int.class or double.class
  - extensions to support generic types, and user-defined operators and types are in progress
  ```

- **Accumulation**

  ```java
  ac.put(data);
  - data must be of type java.lang.Number, int, or double
  ```

- **Retrieval**

  ```java
  Number n = ac.get();
  - get() can only be performed outside finish scope that ac is registered with
  - result from get() must be deterministic if HJ program does not use atomic or isolated constructs and is data-race-free
  ```
Replacing AtomicInteger by Finish Accumulators in Examples 1 & 2

1. // Example 1: compute sum from async tasks in a loop
2. accumulator acl = accumulator.factory.accumulator
3. (accumulator.Operator.SUM, int.class);
4. finish (acl) // permits ac.put() by async tasks in finish
5. while(...) async { ...; elem = ...; acl.put(elem); ...; }
6. int sum1 = acl.get().intValue(); // returns sum from while loop
7. // Example 2: compute sum in a recursive method
8. accumulator acl = accumulator.factory.accumulator
9. (accumulator.Operator.SUM, int.class);
10. finish (acl) visit(ac2, ...);
11. int sum2 = ac2.get().intValue(); // returns sum from visit()
12. ...
13. void visit(accumulator ac2, ...)
14. { ...; elem = ...; ac2.put(elem);
15. async visit(...); ...;
16. }
Access Rules for Finish Accumulators

• Accumulator put() and get() operations can be performed by
  — Task that created the accumulator (owner)
  — Any async task in a finish scope that is registered on accumulator e.g., “finish (ac)”
  — If a get() operation is performed by a non-owner task inside a finish scope, the value returned is the value on entry to the finish scope
Example with Multiple Finish Accumulators

1. // T1 allocates accumulator a and b
2. accumulator a = accumulator.factory.accumulator(SUM, int.class);
3. accumulator b = accumulator.factory.accumulator(MIN, double.class);
4. // T1 can invoke put()/get() on a and b any time
5. a.put(1); // adds 1 to accumulator a
6. Number v1 = a.get(); // Returns 1
7. // T1 creates a finish scope registered on a and b
8. finish (a, b) {
9.    // Any task can invoke put() within the finish
10.   b.put(2.5); // min operation with accumulator b
11.   finish { // Inner finish inherits registrations for a & b
12.      async a.put(2);
13.      b.put(1.5);
14.   }
15.    // Unlikely case: if a task invokes get() within the finish,
16.    // the value returned value is that on entry to the finish
17.   Number v2 = a.get(); // Returns 1
18. }
19. // T1 obtains overall sum and min values after end-finish
20. Number v3 = a.get(); // Returns 1 + 2 = 3
21. Number v4 = b.get(); // Returns min(2.5,1.5) = 1.5
Error Conditions with Finish Accumulators

1. Non-owner task cannot access accumulators outside registered finish
   // T1 allocates accumulator a
   accumulator a = accumulator.factory.accumulator(...);
   async { // T2 cannot access a
     a.put(1); Number v1 = a.get();
   }

2. Non-owner task cannot register accumulators with a finish
   // T1 allocates accumulator a
   accumulator a = accumulator.factory.accumulator(...);
   async {
     // T2 cannot register a with finish
     finish (a) { async a.put(1);  }
   }
Solution Counting Pattern using Finish Accumulators (NQueens revisited)

1. static accumulator a;
2. ...
3. a = accumulator.factory.accumulator(SUM, int.class);
4. finish(a) nqueens_kernel(new int[0], 0);
5. System.out.println("No. of solutions = " + a.get().intValue())
6. ...
7. void nqueens_kernel(int [] a, int depth) {
8. if (size == depth) a.put(1);
9. else
10. /* try each possible position for queen at depth */
11. for (int i = 0; i < size; i++) async {
12. /* allocate a temporary array and copy array a into it */
13. int [] b = new int [depth+1];
14. System.arraycopy(a, 0, b, 0, depth);
15. b[depth] = i;
16. if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
17. } // for-async
18. } // nqueens_kernel()
Current Implementation of Finish Accumulators in HJ

- **Work-sharing runtime ("eager" accumulation)**
  - Each finish accumulator is implemented using java.util.concurrent atomic objects
  - Finish accumulators support operations not supported by AtomicInteger
    - Operator.PROD, Operator.MIN, Operator.MAX
    - Implementations of these operations is analogous to that of AtomicInteger operations

- **Work-stealing runtime ("lazy" accumulation)**
  - Create an array of accumulators, one per worker
    - accumulator must be allocated with extra “true” parameter
      - e.g., accumulator.factory.accumulator(SUM, int.class, true);
  - Each task updates the accumulator for its worker
  - At end-finish of registrations scope, the array is reduced to a single value
Atomic Variables vs. Accumulators

Atomic variables

• Pros:
  – simple construct that can be used anywhere in HJ code
  – supports nondeterminism e.g., work-sharing example in Lecture 6

• Cons:
  – can be a sequential bottleneck with large number of simultaneous parallel accesses
  – supports nondeterminism

Finish accumulators

• Pros:
  – integration with finish structure guarantees determinism and reduces errors
  – supports more reduction operations (max, min, product) than AtomicInteger
  – lazy implementation with work-stealing schedulers is more scalable than AtomicInteger operations

• Con:
  – does not support nondeterminism
Goals for Today’s Lecture

- Finish Accumulators
- Forall statements and barriers
HJ’s forall statement = finish + forall + barriers

Goal 1 (minor): replace common finish–forall sync idiom by forall

e.g., replace

\[
\begin{align*}
&\text{finish forallsync (point } [I,J] : [0:N-1,0:N-1]) \\
&\text{for (point}[K] : [0:N-1]) \\
&C[I][J] += A[I][K] \times B[K][J];
\end{align*}
\]

by

\[
\begin{align*}
&\text{forall (point } [I,J] : [0:N-1,0:N-1]) \\
&\text{for (point}[K] : [0:N-1]) \\
&C[I][J] += A[I][K] \times B[K][J];
\end{align*}
\]

Goal 2 (major): Also support “barrier” synchronization
Hello-Goodbye Forall Example

AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
    int r = rank.getAndIncrement();
    System.out.println("Hello from task ranked " + r);
    System.out.println("Goodbye from task ranked " + r);
}

• Sample output for m = 4
  Hello from task ranked 0
  Hello from task ranked 1
  Goodbye from task ranked 0
  Hello from task ranked 2
  Goodbye from task ranked 2
  Goodbye from task ranked 1
  Hello from task ranked 3
  Goodbye from task ranked 3
Hello-Goodbye Forall Example (contd)

```
AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
    int r = rank.getAndIncrement();
    System.out.println("Hello from task ranked "+ r);
    System.out.println("Goodbye from task ranked "+ r);
}
```

- Question: how can we transform this code so as to ensure that all tasks say hello before any tasks say goodbye?

- Approach 1: Replace the forall loop by two forall loops, one for the hello’s and one for the goodbye’s
  — Need to communicate local r values from one forall to the next

- Approach 2: insert a “barrier” between the hello’s and goodbye’s
  — “next” statement in HJ’s forall loops
Barrier Synchronization: HJ’s “next” statement

```java
AtomicInteger rank = new AtomicInteger();
forall (point[i] : [0:m-1]) {
    int r = rank.getAndIncrement();
    System.out.println("Hello from task ranked " + r);
    next; // Acts as barrier between phases 0 and 1
    System.out.println("Goodbye from task ranked " + r);
}
```

- `next` ➔ each forall iteration suspends at `next` until all iterations arrive (complete previous phase), after which the phase can be advanced
  - If a forall iteration terminates before executing “next”, then the other iterations do not wait for it
  - Scope of synchronization is the closest enclosing forall statement
  - Special case of “phaser” construct (will be covered in following lectures)
Impact of barrier on scheduling forall iterations

Modeling a next operation in the computation graph
Observation 1: Scope of synchronization for “next” is closest enclosing forall statement

```java
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 previous example

- 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.

\[
\begin{array}{cccccccc}
  i=0 & i=1 & i=2 & i=3 & i=4 & i=5 & i=6 & i=7 \\
  (0,0) & (1,0) & (2,0) & (3,0) & (4,0) & (5,0) & (6,0) & (7,0) \\
  \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} \\
  (1,1) & (2,1) & (3,1) & (4,1) & (5,1) & (6,1) & (7,1) \\
  \text{end} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} \\
  (2,2) & (3,2) & (4,2) & (5,2) & (6,2) & (7,2) \\
  \text{end} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} \\
  (3,3) & (4,3) & (5,3) & (6,3) & (7,3) \\
  \text{end} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} \\
  (4,4) & (5,4) & (6,4) & (7,4) \\
  \text{end} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} \\
  (5,5) & (6,5) & (7,5) \\
  \text{end} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} \\
  (6,6) & (7,6) \\
  \text{end} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} & \text{next} \\
  (7,7) \\
  \text{end} & \text{next}
\end{array}
\]

- $i=0...7$ 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 (barrier matching problem)

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

- 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 $\texttt{oddPhase1}()$
**Next-with-Single Statement**

The `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**:
  - `A_1` to `next-start`
  - `A_2` to `next-start`
  - `A_3` to `next-start`
  - `A_4` to `next-start`

- **Wait edges**:
  - `next-start` to `single-statement`
  - `single-statement` to `next-end`
  - `next-end` to `A_1`
  - `next-end` to `A_2`
  - `next-end` to `A_3`
  - `next-end` to `A_4`
Use of next-with-single to print a log message between Hello and Goodbye phases (Listing 6)

1. rank.count = 0; // rank object contains an int field, count
2. \texttt{forall (point[i] : [0:m-1])}{
3. \hspace{1em} // Start of Hello phase
4. \hspace{1em} int r;
5. \hspace{1em} \texttt{isolated}{r = rank.count++;}
6. \hspace{1em} System.out.println("Hello from task ranked " + r);
7. \hspace{1em} \texttt{next} { // single statement
8. \hspace{2em} System.out.println("LOG: Between Hello & Goodbye Phases");
9. \hspace{2em} }
10. \hspace{1em} } // forall
11. \hspace{1em} // Start of Goodbye phase
12. \hspace{1em} System.out.println("Goodbye from task ranked " + r);
13. \hspace{1em} }
14. // forall