Lecture 6: Finish Accumulators

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Worksheet #5: Computation Graphs for Async-Finish and Future Constructs

1) Can you write pseudocode with `async-finish` constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

No. Finish cannot be used to ensure that D only waits for B and C, while E waits only for C.

2) Can you write pseudocode with `future async-get` constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

Yes, see program sketch with dummy return values.
Worksheet #5 solution (contd)

1. HjFuture<String> A = future(() -> {
    return "A";
});
2. HjFuture<String> B = future(() -> {
    A.get();
    return "B";
});
3. HjFuture<String> C = future(() -> {
    A.get();
    return "C";
});
4. HjFuture<String> D = future(() -> {
    // Order of B.get() & C.get() doesn’t matter
    B.get();
    C.get();
    return "D";
});
5. HjFuture<String> E = future(() -> {
    C.get();
    return "E";
});
6. HjFuture<String> F = future(() -> {
    D.get();
    E.get();
    return "F";
});
7. F.get();
Comparing Async-Finish with Future-Get

- **Similarities:**
  - Finish and Get can be used to synchronize and avoid data races
  - Finish waits for both async and future tasks

- **Differences:**
  - Async supports side effects, Futures have return values
  - Future gets can model a larger set of computation graphs than async-finish
  - Finish can wait for an unbounded set of tasks (determined at runtime)
Extending Finish Construct with “Finish Accumulators” (Pseudocode)

- Creation
  
  ```java
  accumulator ac = newFinishAccumulator(operator, type);
  ```

- Operator must be **associative and commutative** (creating task “owns” accumulator)

- Registration
  
  ```java
  finish (ac1, ac2, ...) {
  ...
  }
  ```

- Accumulators `ac1`, `ac2`, ... are registered with the finish scope

- Accumulation
  
  ```java
  ac.put(data);
  ```

- Can be performed in parallel by any statement in finish scope that registers `ac`. Note that a put contributes to the accumulator, but does not overwrite it.

- Retrieval
  
  ```java
  ac.get();
  ```

- Returns initial value if called before end-finish, or final value after end-finish

- `get()` is nonblocking because no synchronization is needed (finish provides the necessary synchronization)
Example: count occurrences of pattern in text (sequential version)

1. // Count all occurrences
2. int count = 0;
3. {
4.   for (int ii = 0; ii <= N - M; ii++) {
5.     int i = ii;
6.     // search for match at position i
7.     for (j = 0; j < M; j++)
8.       if (text[i+j] != pattern[j]) break;
9.     if (j == M) count++;
10.    } // for-ii
11. }
12. } // for-ii
13. print count; // Output
Example: count occurrences of pattern in text (parallel version using finish accumulator)

1. // Count all occurrences
2. a = new Accumulator(SUM, int)
3. finish(a) {
4. for (int ii = 0; ii <= N - M; ii++) {
5. int i = ii;
6. async { // search for match at position i
7. for (j = 0; j < M; j++)
8. if (text[i+j] != pattern[j]) break;
9. if (j == M) a.put(1); // Increment count
10. } // async
11. }
12. } // finish
13.print a.get(); // Output
1. Non-owner task cannot access accumulator outside registered finish

// T1 allocates accumulator a
accumulator a = newFinishAccumulator(...);
a.put(1); // T1 can access a
async { // T2 cannot access a
    a.put(1); Number v1 = a.get();
}

2. Non-owner task cannot register accumulator with a finish

// T1 allocates accumulator a
accumulator a = newFinishAccumulator(...);
async {
    // T2 cannot register a with finish
    finish (a) { async a.put(1); }
}
The N-Queens Problem

How can we place $n$ queens on an $n \times n$ chessboard so that no two queens can capture each other?

A queen can move any number of squares horizontally, vertically, and diagonally.

Here, the possible target squares of the queen Q are marked with an x.
Backtracking Solution

empty board

place 1\textsuperscript{st} queen

place 2\textsuperscript{nd} queen

place 3\textsuperscript{rd} queen

place 4\textsuperscript{th} queen
Sequential solution for NQueens (counting all solutions)

1. count = 0;
2. size = 8; nqueens_kernel_seq(new int[0], 0);
3. System.out.println(“No. of solutions = “ + count);
4. ...
5. void nqueens_kernel_seq(int [] a, int depth) {
6. if (size == depth) count++;
7. else
8. /* try each possible position for queen at depth */
9. for (int i = 0; i < size; i++) {
10. /* allocate a temporary array and copy array a into it */
11. int [] b = new int [depth+1];
12. System.arraycopy(a, 0, b, 0, depth);
13. b[depth] = i; // Try to place queen in row i of column depth
14. if (ok(depth,b)) // check if placement is okay
15. nqueens_kernel_seq(b, depth+1);
16. } // for
17. } // nqueens_kernel_seq()
How to extend sequential solution to obtain a parallel solution?

1. `count = 0;`
2. `size = 8; finish nqueens_kernel_par(new int[0], 0);`
3. `System.out.println("No. of solutions = " + count);`
4. ...
5. `void nqueens_kernel_par(int [] a, int depth) {`
6.   if (size == depth) `count++;`
7.   else
8.     /* try each possible position for queen at depth */
9.   for (int i = 0; i < size; i++) `async {
10.      /* allocate a temporary array and copy array a into it */
11.      int [] b = new int [depth+1];
12.      System.arraycopy(a, 0, b, 0, depth);
13.      b[depth] = i; // Try to place queen in row i of column depth
14.      if (ok(depth,b)) // check if placement is okay
15.         nqueens_kernel_par(b, depth+1);
16.    } // for
17. } // nqueens_kernel_par()`
How to extend sequential solution to obtain a parallel solution?

1. \texttt{FinishAccumulator ac = newFinishAccumulator(Operator.SUM, int.class);} 
2. \texttt{size = 8; finish(ac) nqueens_kernel_par(new int[0], 0);} 
3. \texttt{System.out.println("No. of solutions = " + ac.get().intValue());} 
4. . . . 
5. \texttt{void nqueens_kernel_par(int [] a, int depth) {} 
6. \hspace{1em} if (size == depth) ac.put(1);} 
7. \hspace{1em} else 
8. \hspace{2em} /* try each possible position for queen at depth */ 
9. \hspace{2em} for (int i = 0; i < size; i++) \texttt{async} 
10. \hspace{3em} /* allocate a temporary array and copy array a into it */ 
11. \hspace{3em} int [] b = new int [depth+1]; 
12. \hspace{3em} System.arraycopy(a, 0, b, 0, depth); 
13. \hspace{3em} b[depth] = i; // Try to place queen in row i of column depth 
14. \hspace{3em} if (ok(depth,b)) // check if placement is okay 
15. \hspace{4em} nqueens_kernel_par(b, depth+1); 
16. \hspace{2em} } // for-async 
17. } // nqueens_kernel_par()}
Efficient Parallelism

1. FinishAccumulator ac = newFinishAccumulator(Operator.SUM, int.class);
2. size = 8; finish(ac) nqueens_kernel_par(new int[0], 0);
3. System.out.println(“No. of solutions = “ + ac.get().intValue());
4. ...
5. void nqueens_kernel_par(int [] a, int depth) {
6.     if (size == depth) ac.put(1);
7.     else
8.         /* try each possible position for queen at depth */
9.         for (int i = 0; i < size; i++) async {
10.             /* allocate a temporary array and copy array a into it */
11.                 int [] b = new int [depth+1];
12.                 System.arraycopy(a, 0, b, 0, depth);
13.                 b[depth] = i; // Try to place queen in row i of column depth
14.                 if (ok(depth,b)) // check if placement is okay
15.                     nqueens_kernel_par(b, depth+1);
16.             } // for-async
17. } // nqueens_kernel_par()

When depth is close to size, the async tasks get too small
Efficient Parallelism

1. FinishAccumulator ac = new FinishAccumulator(Operator.SUM, int.class);
2. size = 8; finish(ac) nqueens_kernel(new int[0], 0);
3. System.out.println("No. of solutions = " + ac.get().intValue());
4. ...
5. void nqueens_kernel(int [] a, int depth) {
6.   if (depth > size - threshold) {
7.     nqueens_kernel_seq(a, depth)
8.   } else {
9.     nqueens_kernel_par(a, depth)
10. }
11. } // nqueens_kernel()