Lecture 6: Finish Accumulators

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu
Worksheet 4 revisited

- Estimate $T(S,P) \sim \frac{\text{WORK}(G,S)}{P} + \text{CPL}(G,S) = \frac{(S-1)}{P} + \log_2(S)$ for the parallel array sum computation shown in slide 4.

- Assume $S = 1024 \implies \log_2(S) = 10$

- Compute for 10, 100, 1000 processors
  
  $T(P) = \frac{1023}{P} + 10$

- The formula $T(S,P) \sim \frac{\text{WORK}(G,S)}{P} + \text{CPL}(G,S) = \frac{(S-1)}{P} + \log_2(S) = \frac{1023}{P} + 10$ is an upper bound (worst case scenario) and obviously too conservative for $P=1$

- If you just use the formula for $P = 1$, you get $\text{speedup} = \frac{1033}{(1023/P + 10)}$

- If you compute $T(S,1)$ precisely, you get $\text{speedup} = \frac{1023}{(1023/P + 10)}$
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 waits for both 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.
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
  
  \[
  \text{accumulator } ac = \text{newFinishAccumulator}(\text{operator}, \text{type});
  \]

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

• Registration
  
  \[
  \text{finish (ac1, ac2, ...)} \{ \ldots \}
  \]

• **Accumulators ac1, ac2, ... are registered with the finish scope**

• Accumulation
  
  \[
  \text{ac.put(data)};
  \]

• **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
  
  \[
  \text{ac.get();}
  \]

• **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)**
No mutation!
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++; // Increment count
10.   } // for-ii
11. }
12.}
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
Error Conditions with Finish Accumulators

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+1,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+1,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. 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+1,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. `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) {
   if (size == depth) ac.put(1);
   else
      /* try each possible position for queen at depth */
      for (int i = 0; i < size; i++) async {
         /* allocate a temporary array and copy array a into it */
         int [] b = new int [depth+1];
         System.arraycopy(a, 0, b, 0, depth);
         b[depth] = i; // Try to place queen in row i of column depth
         if (ok(depth+1, b)) // check if placement is okay
            nqueens_kernel_par(b, depth+1);
      } // for-async
} // 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+1,b)) // check if placement is okay
15.              nqueens_kernel_par(b, depth+1);
16.     } // for-async
17. } // nqueens_kernel_par()
1. FinishAccumulator ac = new FinishAccumulator(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+1,b)) // check if placement is okay
15.                    nqueens_kernel_par(b, depth+1);
16.            } // for-async
17.        } // nqueens_kernel_par()
Efficient Parallelism

1. FinishAccumulator ac = newFinishAccumulator(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()
Announcements & Reminders

• IMPORTANT:
  – Watch video & read handout for topics 2.2 and 2.4 for next lecture on Friday, Jan 26th

• HW1 is due by 11:59pm TODAY

• HW2 is out later today

• MIDTERM is on Thursday, February 22nd, from 4PM to 6:30PM

• Quiz for Unit 1 (topics 1.1 - 1.5) is due by Friday (Jan 26th) on Canvas

• See course web site for all work assignments and due dates

• Use Piazza (public or private posts, as appropriate) for all communications re. COMP 322

• See Office Hours link on course web site for latest office hours schedule.