COMP 322: Fundamentals of Parallel Programming

Lecture 7: Finish Accumulators

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

http://comp322.rice.edu

Worksheet #6 solution: Parallelizing Pascal's Triangle with Futures and Memoization

There are four variants of the Binomial Coefficients program provided in four different HJlib methods in the next page:

a. Sequential Recursive without Memoization (chooseRecursiveSeq())
b. Parallel Recursive without Memoization (chooseRecursivePar())
c. Sequential Recursive with Memoization (chooseMemoizedSeq())
d. Parallel Recursive with Memoization (chooseMemoizedPar())

Your task is to analyze the WORK, CPL, and Ideal Parallelism for these four versions, for the input N = 4, and K = 2. Assume that each call to ComputeSum() has COST = 1, and all other operations are free. Complete all entries in the table:

Variant	Work	CPL	Ideal Parallelism
chooseRecursiveSeq	**5**	**5**	**1**
chooseRecursivePar	**5**	**3**	**5/3 = 1.67**
chooseMemoizedSeq	**4**	**4**	**1**
chooseMemoizedPar	**4**	**3**	**4/3 = 1.33**

Worksheet #6 solution: Parallelizing Pascal's Triangle with Futures and Memoization

There are four variants of the Binomial Coefficients program provided in four different HJlib methods in the next page:

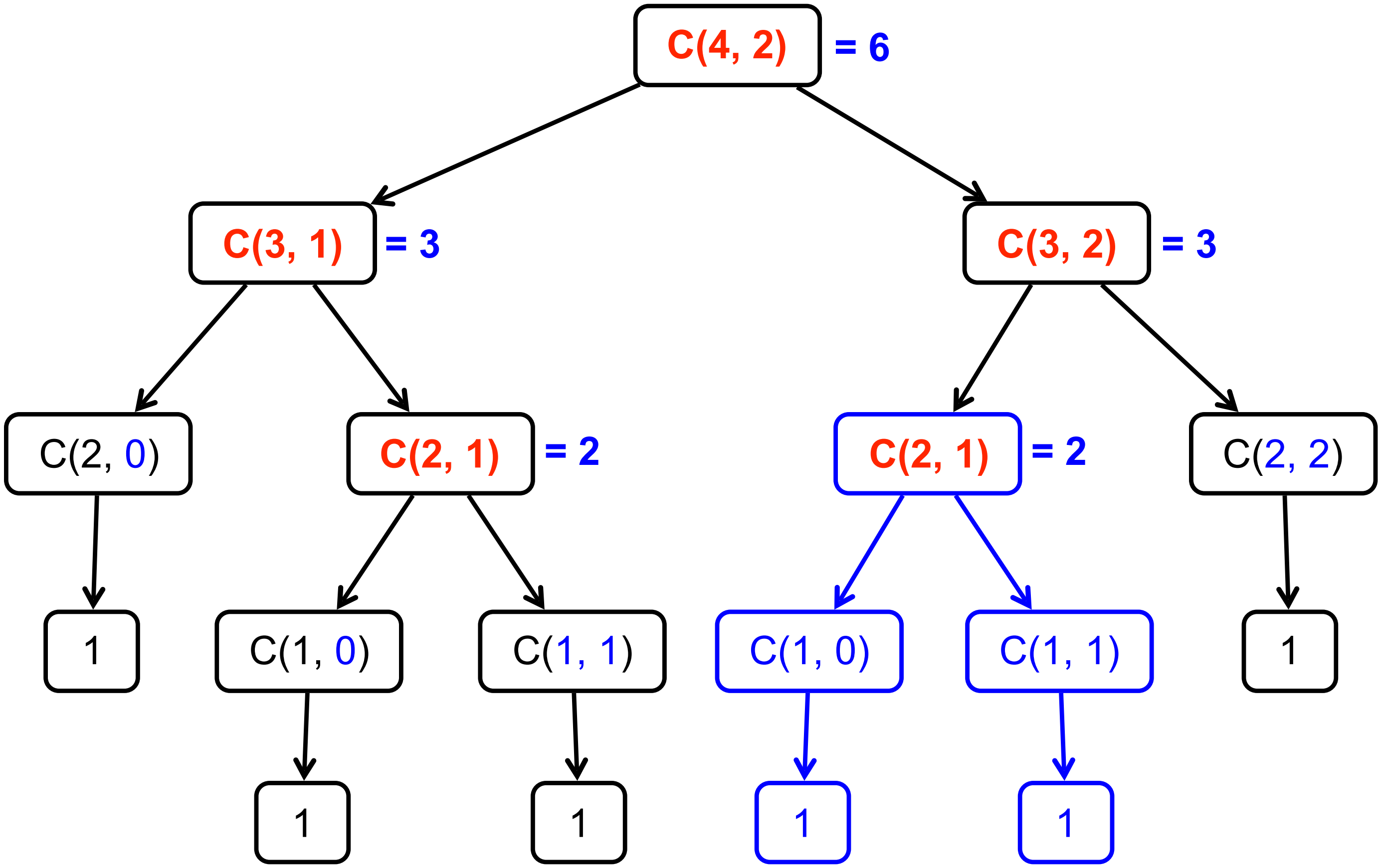
a. Sequential Recursive without Memoization (chooseRecursiveSeq())
b. Parallel Recursive without Memoization (chooseRecursivePar())
c. Sequential Recursive with Memoization (chooseMemoizedSeq())
d. Parallel Recursive with Memoization (chooseMemoizedPar())

Your task is to analyze the WORK, CPL, and Ideal Parallelism for these four versions, for the input N = 4, and K = 2. Assume that each call to ComputeSum() has COST = 1, and all other operations are free. Complete all entries in the table:

Variant	Work	CPL	Ideal Parallelism
chooseRecursiveSeq	5	5	1
chooseRecursivePar	5	3	5/3 = 1.67
chooseMemoizedSeq	4	4	1
chooseMemoizedPar	4	3	4/3 = 1.33

Do you agree with the following statement: "Parallelization of inefficient algorithms often leads to more ideal parallelism than parallelization of efficient algorithms" in the context of this worksheet?

REMINDER: computation structure of C(4,2)
Nodes with calls to ComputeSum() are in red



Extending Finish Construct with "Finish Accumulators" (Pseudocode)

- Creation

```
accumulator ac = newFinishAccumulator(operator, type);
```

- *Operator must be associative and commutative (creating task "owns" accumulator)*
- Registration

```
finish (ac1, ac2, ...) { ... }
```

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

```
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

```
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)

```
// Count all occurrences
int count = 0;
{
 for (int ii = 0; ii <= N - M; ii++) {
   int i = ii;
   // search for match at position i
   for (j = 0; j < M; j++)
     if (text[i+j] != pattern[j]) break;
   if (j == M) count++; // Increment count
  } // for-ii
 }
}
print count; // Output
```

Example: count occurrences of pattern in text (parallel version using finish accumulator)

```
// Count all occurrences
a = new Accumulator(SUM, int)
finish(a) {
 for (int ii = 0; ii <= N - M; ii++) {
  int i = ii;
  async { // search for match at position i
   for (j = 0; j < M; j++)
     if (text[i+j] != pattern[j]) break;
   if (j == M) a.put(1); // Increment count
  } // async
 }
} // finish
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);  }
}
```

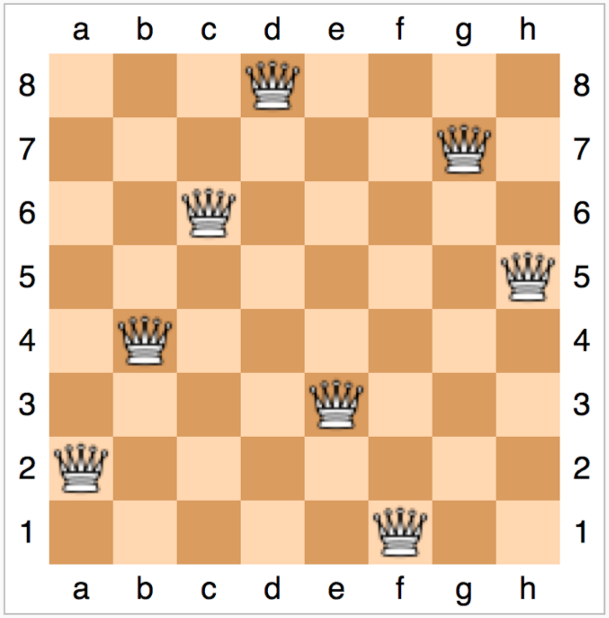
The N-Queens Problem

How can we place n queens on an n×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.

x			x			x	
	x		x		x		
		x	x	x			
x	x	x	Q	x	x	x	x
		x	x	x			
	x		x		x		
x			x			x	
			x				x



One solution to the eight queens puzzle

Backtracking and Decision Tree states

- Idea: Start at the root of the decision tree and move downwards, that is, make a sequence of decisions, until you either reach a solution or you enter a state from where no solution can be reached by any further sequence of decisions.
- In the latter case, backtrack to the parent of the current state and take a different path downwards from there. If all paths from this state have already been explored, backtrack to its parent.
- Continue this procedure until you find a solution (or all solutions), or establish that no solution exists.
- A state in the decision tree can be encoded as an array, a[0..c-1] for c columns, where a[i] = row position of queen in column i.

Backtracking Solution

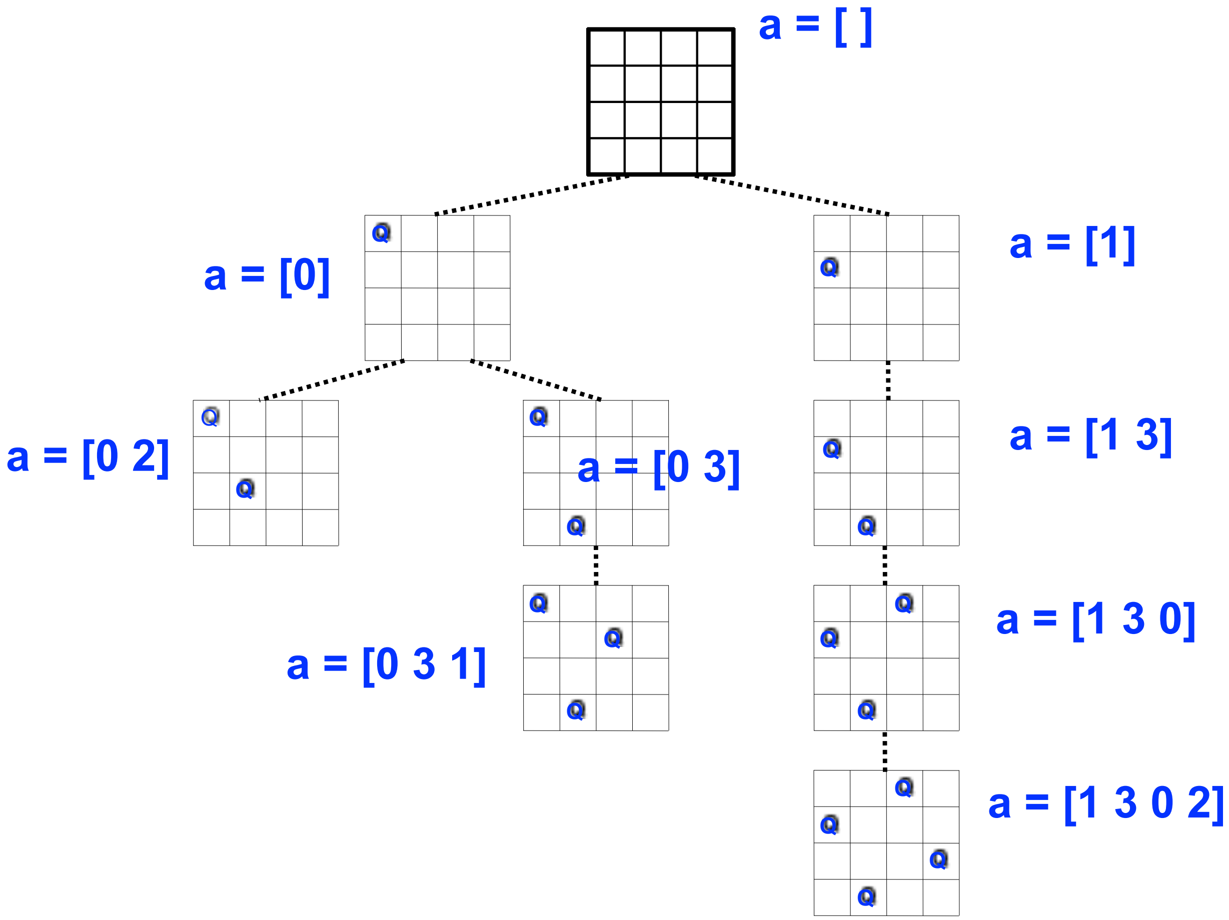
empty board

place 1st queen

place 2nd queen

place 3rd queen

place 4th queen



a = [ ]

a = [0]

a = [1]

a = [0 2]

a = [0 3]

a = [1 3]

a = [0 3 1]

a = [1 3 0]

a = [1 3 0 2]

Sequential solution for NQueens (counting all solutions)

```
count = 0;
size = 8; nqueens_kernel_seq(new int[0], 0);
System.out.println(“No. of solutions = “ + count);
. . .
void nqueens_kernel_seq(int [] a, int depth) {
  if (size == depth) count++;
  else
    /* try each possible position for queen at depth */
    for (int i =  0; i < size; i++) {
      /* 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_seq(b, depth+1);
    } // for
} // nqueens_kernel_seq()
```

How to extend sequential solution to obtain a parallel solution?

```
count = 0;
size = 8; finish nqueens_kernel_par(new int[0], 0);
System.out.println(“No. of solutions = “ + count);
. . .
void nqueens_kernel_par(int [] a, int depth) {
  if (size == depth) count++;
  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
} // nqueens_kernel_par()
```

How to extend sequential solution to obtain a parallel solution?

```
count = 0;
size = 8; finish nqueens_kernel_par(new int[0], 0);
System.out.println(“No. of solutions = “ + count);
. . .
void nqueens_kernel_par(int [] a, int depth) {
  if (size == depth) count++;
  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
} // nqueens_kernel_par()
```

DATA RACE!

How to extend sequential solution to obtain a parallel solution?

```
FinishAccumulator ac = newFinishAccumulator(Operator.SUM, int.class);
size = 8; finish(ac) nqueens_kernel_par(new int[0], 0);
System.out.println(“No. of solutions = “ + ac.get().intValue());
. . .
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

```
FinishAccumulator ac = newFinishAccumulator(Operator.SUM, int.class);
size = 8; finish(ac) nqueens_kernel_par(new int[0], 0);
System.out.println(“No. of solutions = “ + ac.get().intValue());
. . .
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

```
FinishAccumulator ac = newFinishAccumulator(Operator.SUM, int.class);
size = 8; finish(ac) nqueens_kernel_par(new int[0], 0);
System.out.println(“No. of solutions = “ + ac.get().intValue());
. . .
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()
```

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

Efficient Parallelism

```
FinishAccumulator ac = newFinishAccumulator(Operator.SUM, int.class);
size = 8; finish(ac) nqueens_kernel(new int[0], 0);
System.out.println(“No. of solutions = “ + ac.get().intValue());
. . .
void nqueens_kernel(int [] a, int depth) {
  if (depth > size - threshold) {
     nqueens_kernel_seq(a, depth)
  } else {
     nqueens_kernel_par(a, depth)
  }
} // nqueens_kernel()
```

Announcements & Reminders

- IMPORTANT:
  - Watch video & read handout for topic 2.4 for next lecture on Friday, Jan 25th
- HW1 is due by 11:59pm TODAY
- HW2 is out later today
- MIDTERM is on Thursday, February 21st, 4-6:30PM. Room(s) TBA.
- Quiz for Unit 1 (topics 1.1 - 1.5) is due by Friday (Jan 25th) 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.