COMP 322: Fundamentals of Parallel Programming

Lecture 7: Parallel N-Queens algorithm, Finish Accumulators

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

http://comp322.rice.edu

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

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

a. Sequential Recursive without Memoization (chooseRecursiveSeq())

b. Parallel Recursive without Memoization (chooseRecursivePar())

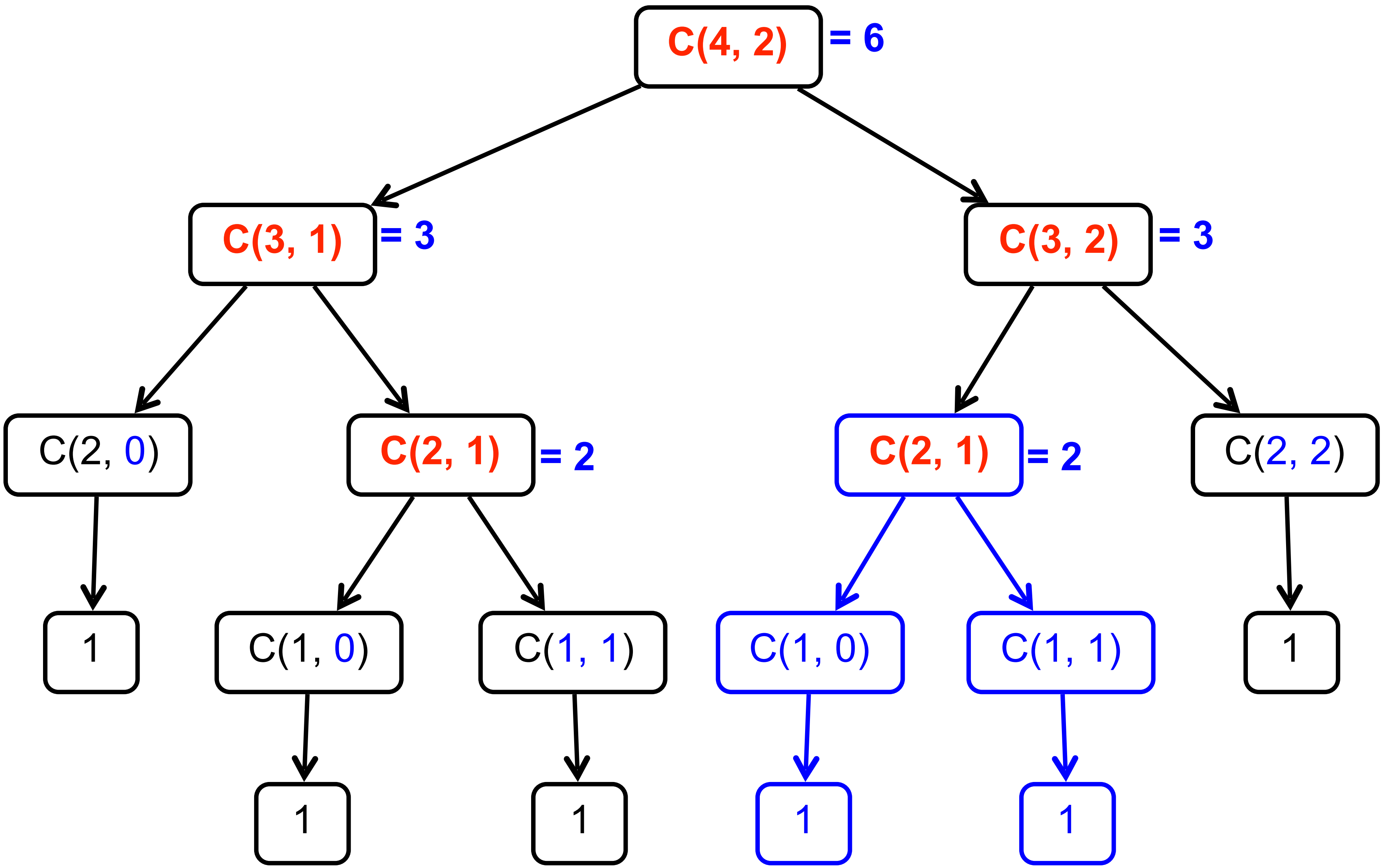
c. Sequential Recursive with Memoization (chooseMemoizedSeq())

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

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



Comparing Async-Finish with Future-Get

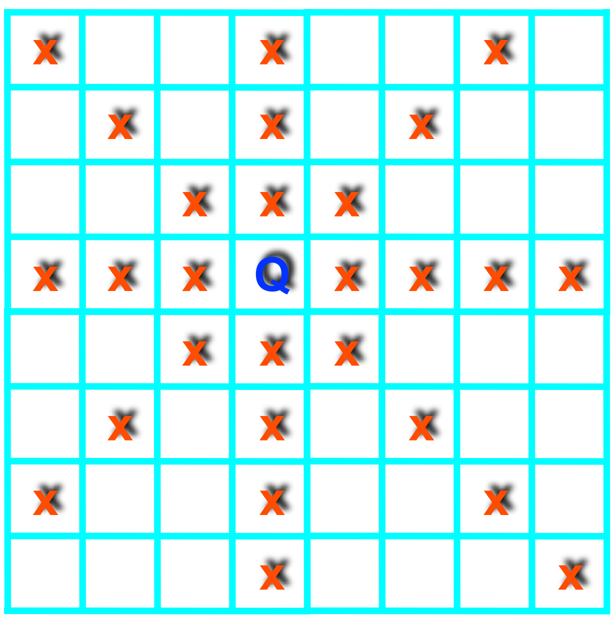
- Similarities:
  - Finish and Get can be used to synchronize and avoid data races
  - Finish waits for both async and future tasks
- Differences:
  - Async supports side effects, Futures have return values
  - Future gets can model a larger set of computation graphs than async-finish
  - Finish can wait for an unbounded set of tasks (determined at runtime)

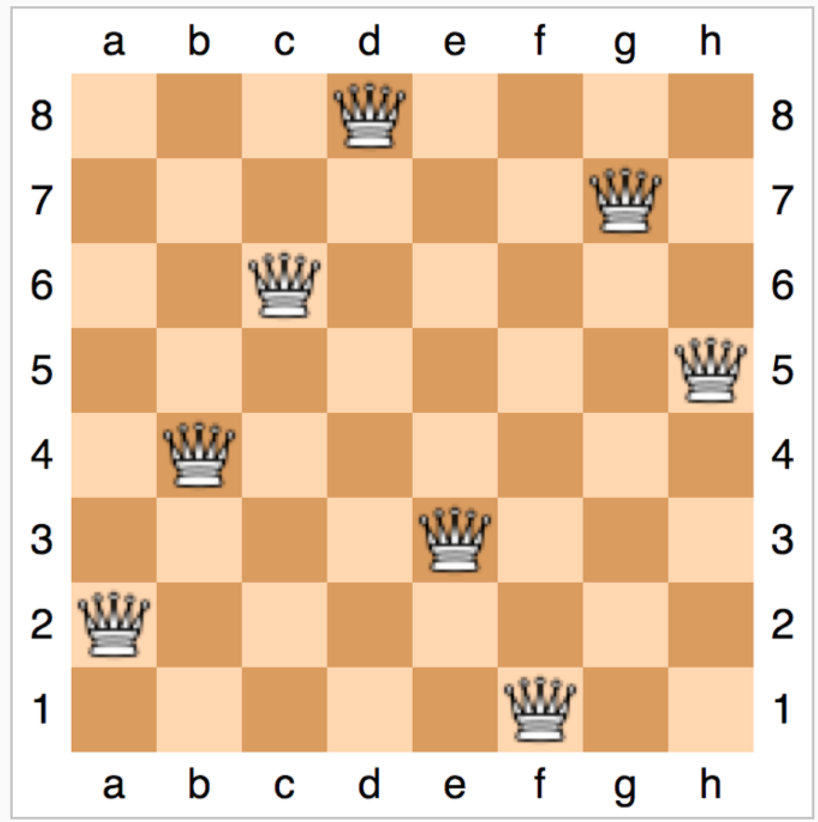
The N-Queens Problem

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.





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 in Decision Trees

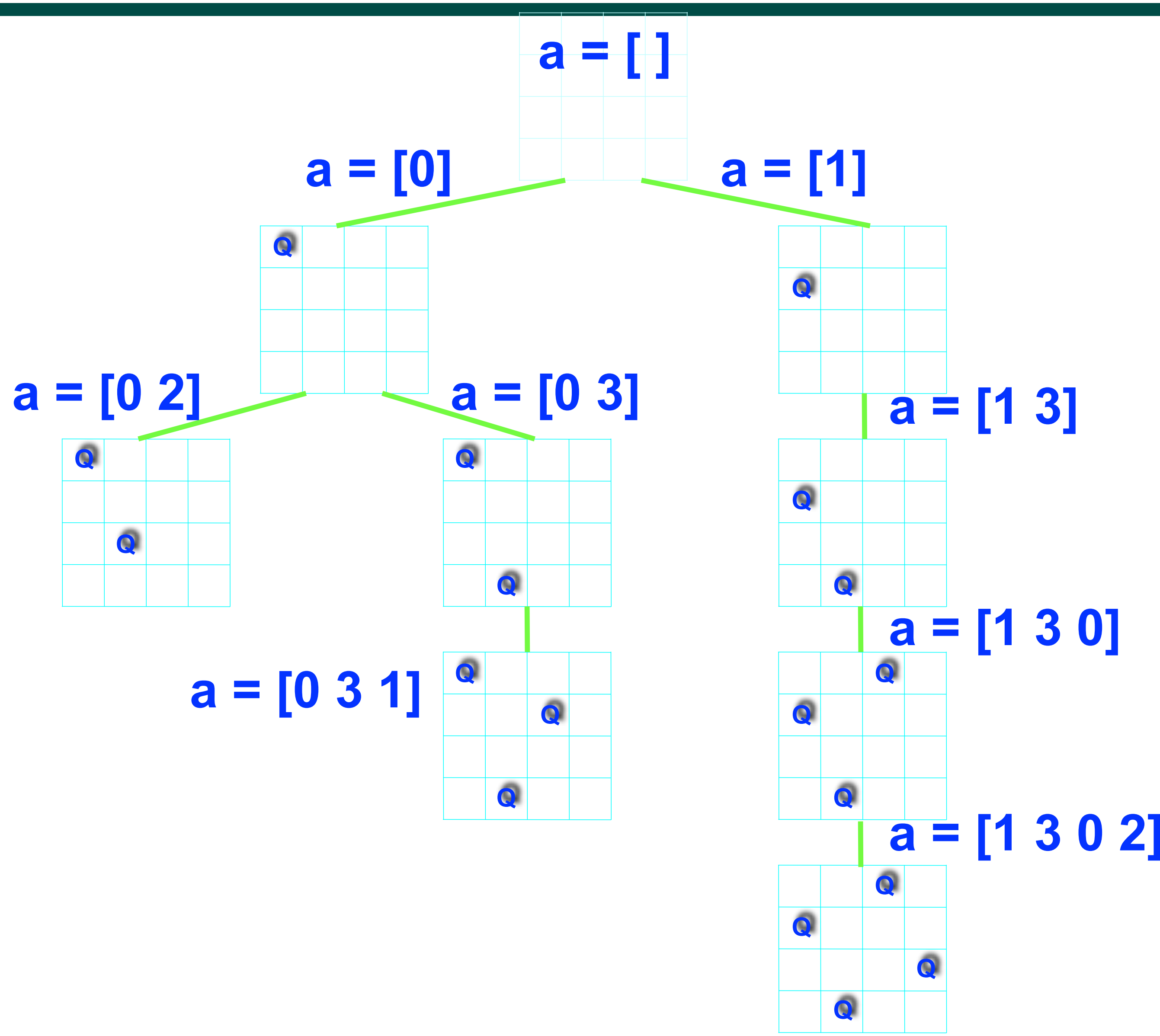
empty board

place 1st queen

place 2nd queen

place 3rd queen

place 4th queen



Sequential solution for NQueens (counting all solutions)

```
count = 0;
size = 8; nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + count);
. . .
void nqueens_kernel(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(b, depth+1);
    } // for
} // nqueens_kernel()
```

How to extend sequential solution to obtain a parallel solution?

```
count = 0;
size = 8; finish nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " + count);
. . .
void nqueens_kernel(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(b, depth+1);
    } // for
} // nqueens_kernel()
```

But there's a data race on count?

Extending Finish Construct with "Finish Accumulators" (Pseudocode)

- Creation

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

  - Operator must be associative and commutative

- 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 finish provides the necessary synchronization

How to extend sequential solution to obtain a parallel solution?

```
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 (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(b, depth+1);
    } // for
} // nqueens_kernel()
```

Error Conditions with Finish Accumulators

1. Non-owner task cannot access accumulator outside registered finish

```
// T1 allocates accumulator a
accumulator a = newFinishAccumulator(...);
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);  }
}
```