COMP 322: Fundamentals of Parallel Programming

Lecture 6: Parallel N-Queens algorithm, Finish Accumulators

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

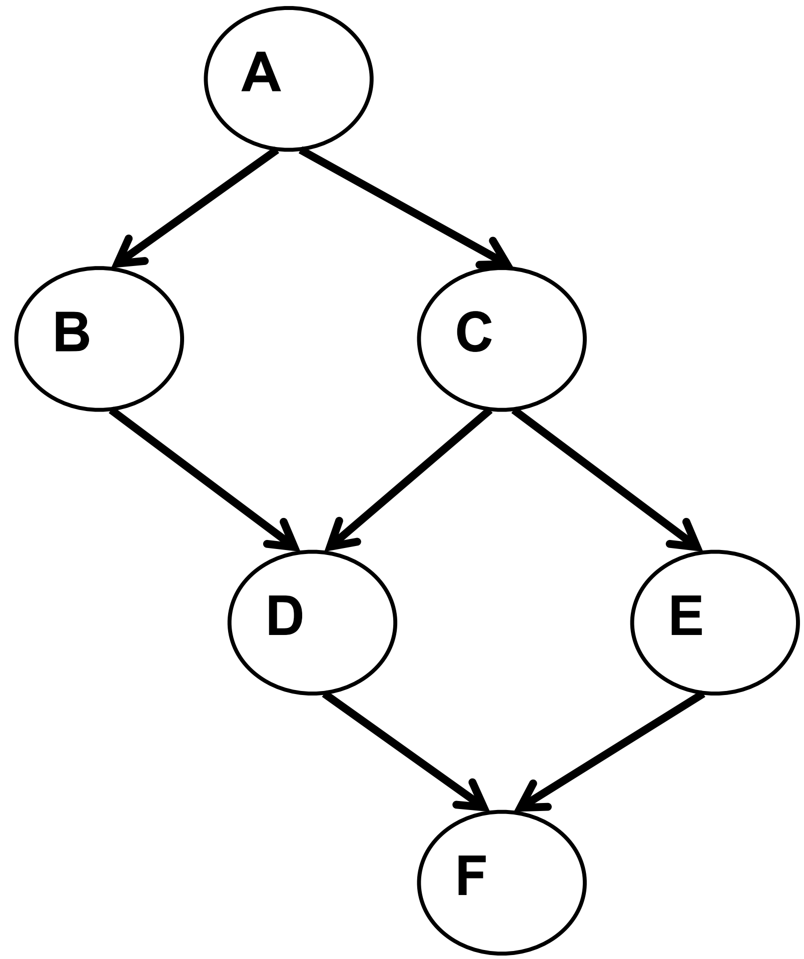
Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Worksheet #5 solution: Computation Graphs for Async-Finish and Future Constructs

1) Can you write an HJ program with <u>async-finish</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right?
No

2) Can you write an HJ program with <u>future async-get</u> 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 in next slide with void futures. A dummy return value can also be used.



Worksheet #5 solution (contd)

```
final HjFuture<Void> A =
    future(() -> { return null; });
final HjFuture<Void> B =
    future(() -> { A.get(); return null; });
final HjFuture<Void> C =
    future(() -> { A.get(); return null;});
final HjFuture<Void> D =
    future(() -> { B.get(); C.get(); return null; });
final HjFuture<Void> E =
    future(() -> {C.get(); return null; });
final HjFuture<Void> F =
    future(() -> { D.get(); E.get(); return null; });
F.get();
```

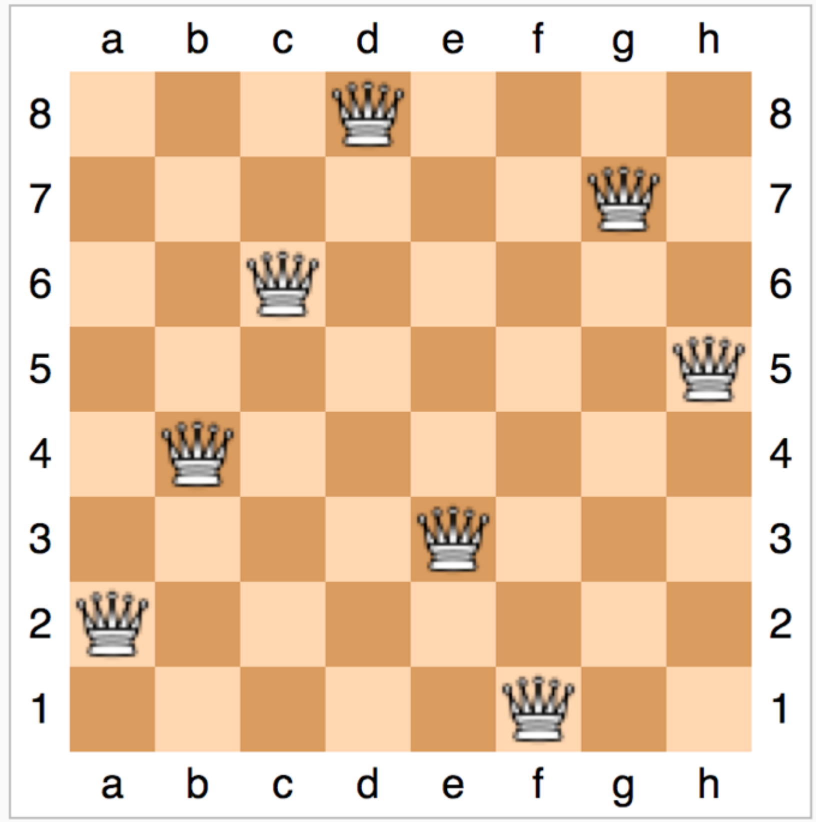
The N-Queens Problem

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

Decision Trees

- In any solution of the n-queens problem, there must be exactly one queen in each column of the board.
- Otherwise, the two queens in the same column could capture each other.
- Therefore, we can describe the solution of this problem as a sequence of n decisions:
- Decision 1: Place a queen in the first column.
- Decision 2: Place a queen in the second column.
- .
  .
  .

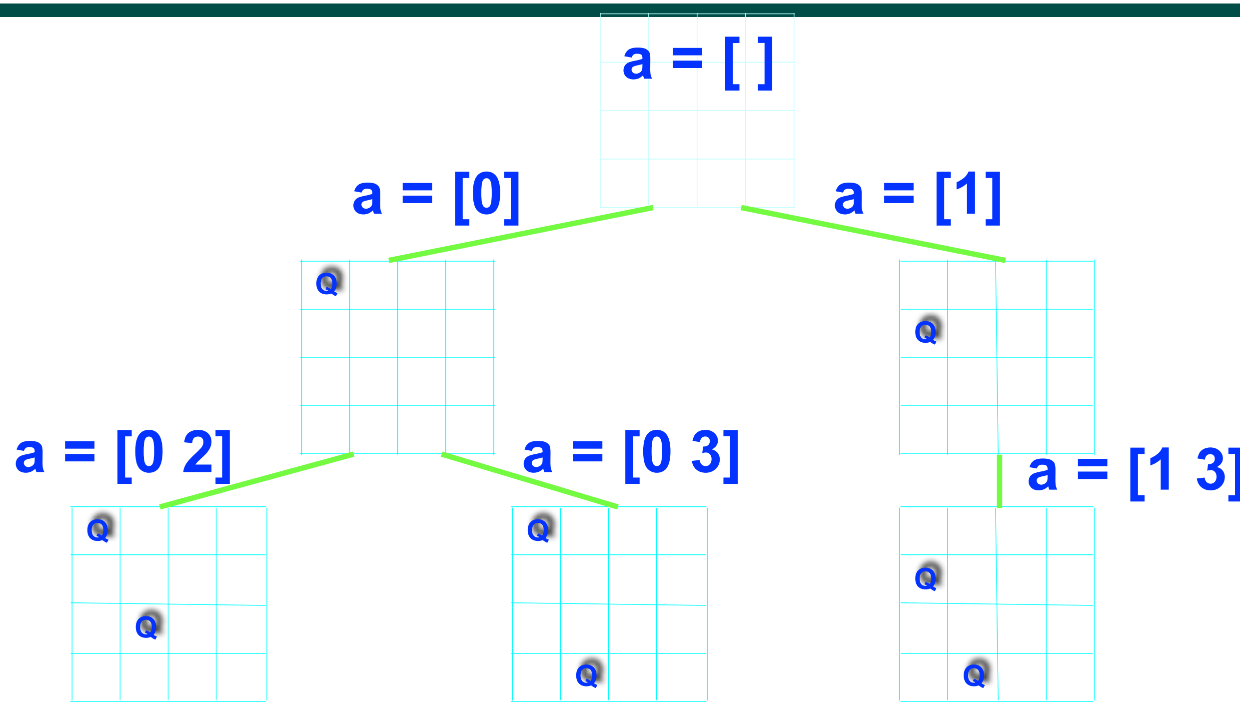
  Decision n: Place a queen in the n-th column.
- Since there are multiple choices for each decision, we get a "decision tree"

Decision Tree

empty board

place 1st queen

place 2nd queen



Backtracking in Decision Trees

- There are problems that require us to perform an exhaustive search of all possible sequences of decisions in order to find the solution.
- We can solve such problems by constructing the complete decision tree and then find a path from its root to a leaf that corresponds to a solution of the problem
- In many cases, the efficiency of this procedure can be dramatically increased by a technique called backtracking (depth-first search, without constructing the tree explicitly).

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

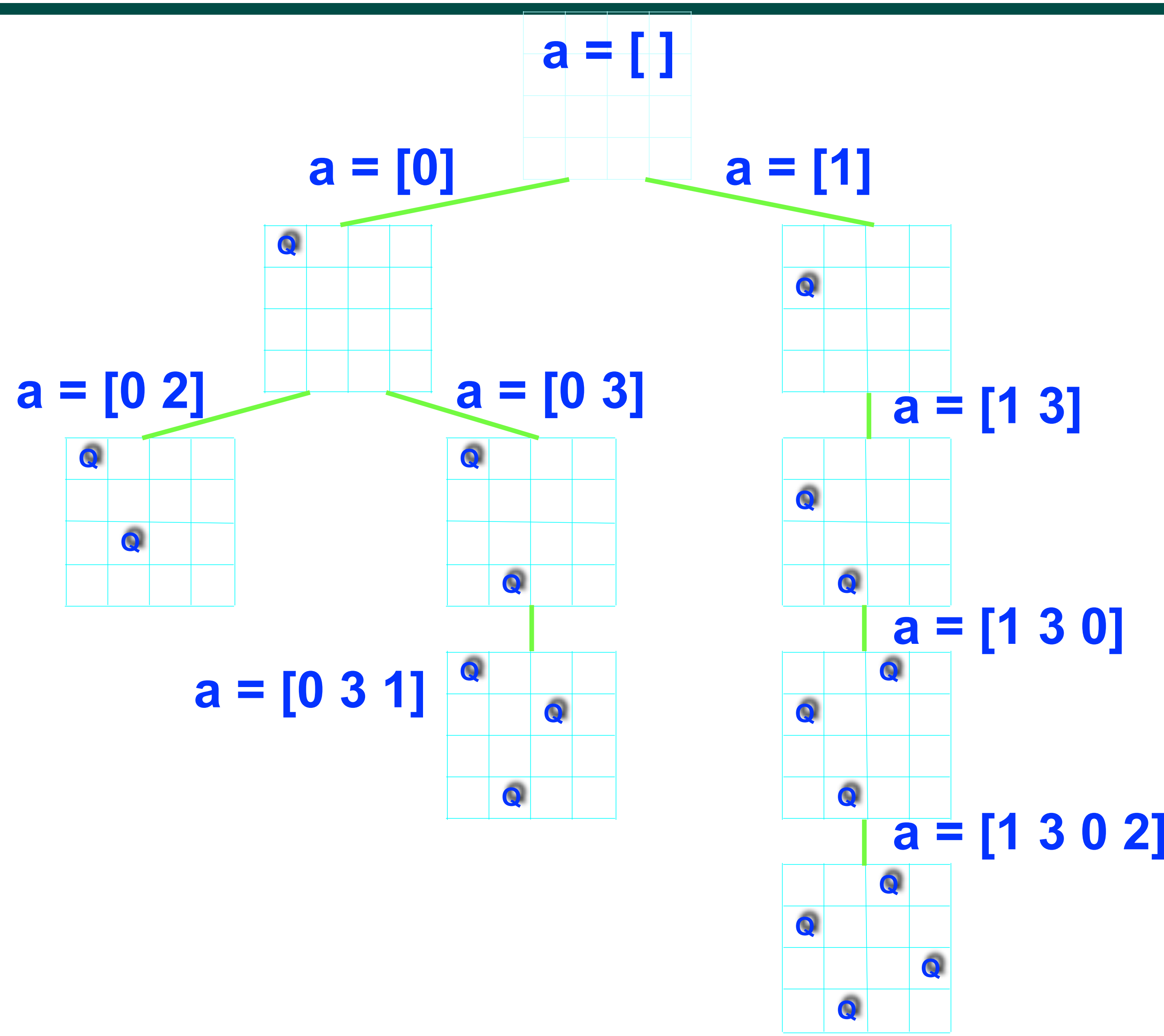
empty board

place 1st queen

place 2nd queen

place 3rd queen

place 4th queen



Sequential solution for NQueens (counting all solutions)

```
static int count;
. . .
count = 0;
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;
      if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
    } // for-async
} // nqueens_kernel()
```

Parallel Solution to NQueens Problem?

```
static accumulator count;
. . .
count = 0;
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;
       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
     } // for-async
} // nqueens_kernel()
```

Parallel NQueens Example (contd)

```
// Challenge: how to count number of solutions found?

finish nqueens_kernel(new int[0], 0);
System.out.println("No. of solutions = " ...);
. . .
void nqueens_kernel(int [] a, int depth) {
  if (size == depth) // Solution found: how to 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;
       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
     } // for-async
} // nqueens_kernel()
```

Finish Accumulators (Pseudocode)

- Creation

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

- Registration

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

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

- Accumulation

```
ac.put(data);
```

  - *can be performed by any statement in finish scope that registers ac*

- Retrieval

```
ac.get();
```

  - *Either returns initial value before end-finish or final value after end-finish*
  - *get() is nonblocking because finish provides the necessary synchronizatio*

Use of Finish Accumulators to count solutions in Parallel NQueens

```
final FinishAccumulator ac =
                    newFinishAccumulator(Operator.SUM, int.class);
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;
       if (ok(depth+1,b)) nqueens_kernel(b, depth+1);
     } // for-async
} // nqueens_kernel()
```

Error Conditions with Finish Accumulators

1. Non-owner task cannot access accumulators 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 accumulators with a finish

```
// T1 allocates accumulator a
accumulator a = newFinishAccumulator(...);
async {
  // T2 cannot register a with finish
  finish (a) { async a.put(1);  }
}
```