COMP 322: Fundamentals of Parallel Programming

Lecture 9: Memoization

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

Contact email: vsarkar@rice.edu

http://comp322.rice.edu

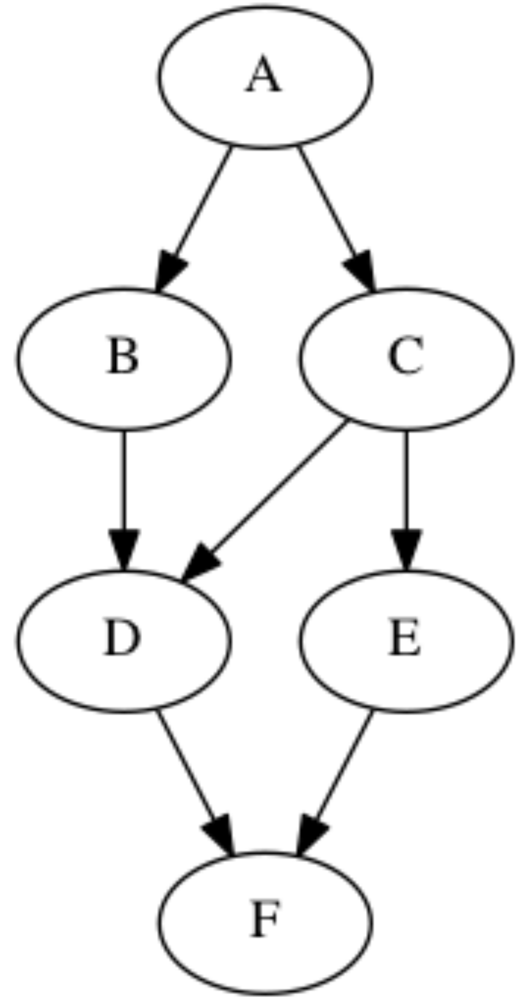
Worksheet #5: Computation Graphs for Async-Finish and Future Constructs

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 waits for both B and C, while E waits only for C.

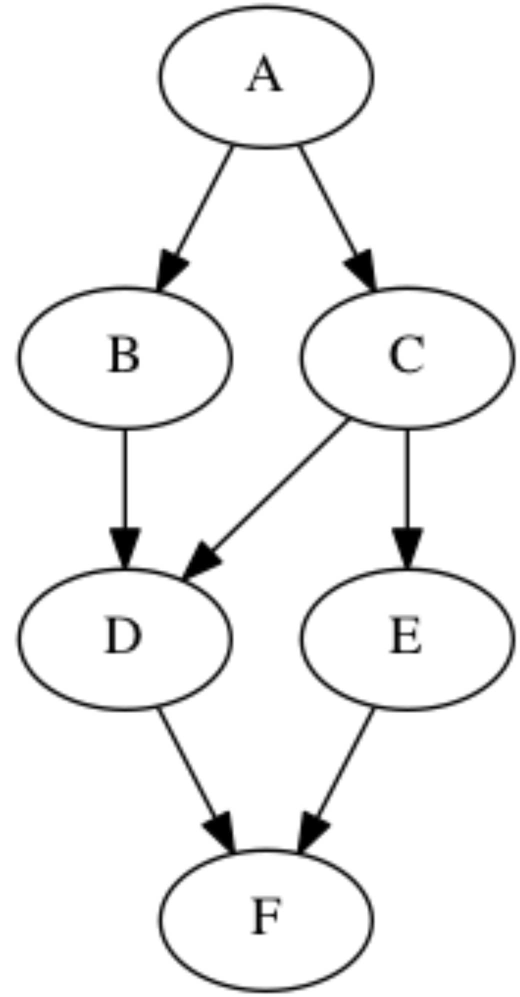
2) Can you write pseudocode with `future async-get` constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

Yes, see program sketch with void futures. A dummy return value can also be used.



Worksheet #5 solution (contd)

```
HjFuture<Void> A = future(() -> {
      return "A"; });
HjFuture<Void> B = future(() -> {
      A.get(); return "B"; });
HjFuture<Void> C = future(() -> {
      A.get(); return "C"; });
HjFuture<Void> D = future(() -> {
      // Order of B.get() & C.get() doesn't matter
      B.get(); C.get(); return "D"; });
HjFuture<Void> E = future(() -> {
      C.get(); return "E"; });
HjFuture<Void> F = future(() -> {
      D.get(); E.get(); return "F"; });
F.get();
```



Background: Functional Programming

- Eliminate side-effects
  - emphasizes functions whose results that depend only on their inputs and not on any other program state
  - calling a function, f(x), twice with the same value for the argument x will produce the same result both times

Helpful Link: http://en.wikipedia.org/wiki/Functional_programming

Example: Binomial Coefficient

- The coefficient of the x^k term in the polynomial expansion of the binomial power $(1 + x)^n$
- Number of sets of k items that can be chosen from n items
- Indexed by *n* and *k*
    - written as C(n, k)
    - read as "n choose k"
- Factorial Formula: C(n, k) = $\left(\dfrac{n!}{k!(n-k)!}\right)$
- Recursive Formula

  C(n, k) = C(n – 1, k – 1) + C(n – 1, k)

  Base cases: C(n, n) = C(n, 0) = C(0, k) = 1

Helpful Link: http://en.wikipedia.org/wiki/Binomial_coefficient

Example: Binomial Coefficient (Recursive Sequential version)

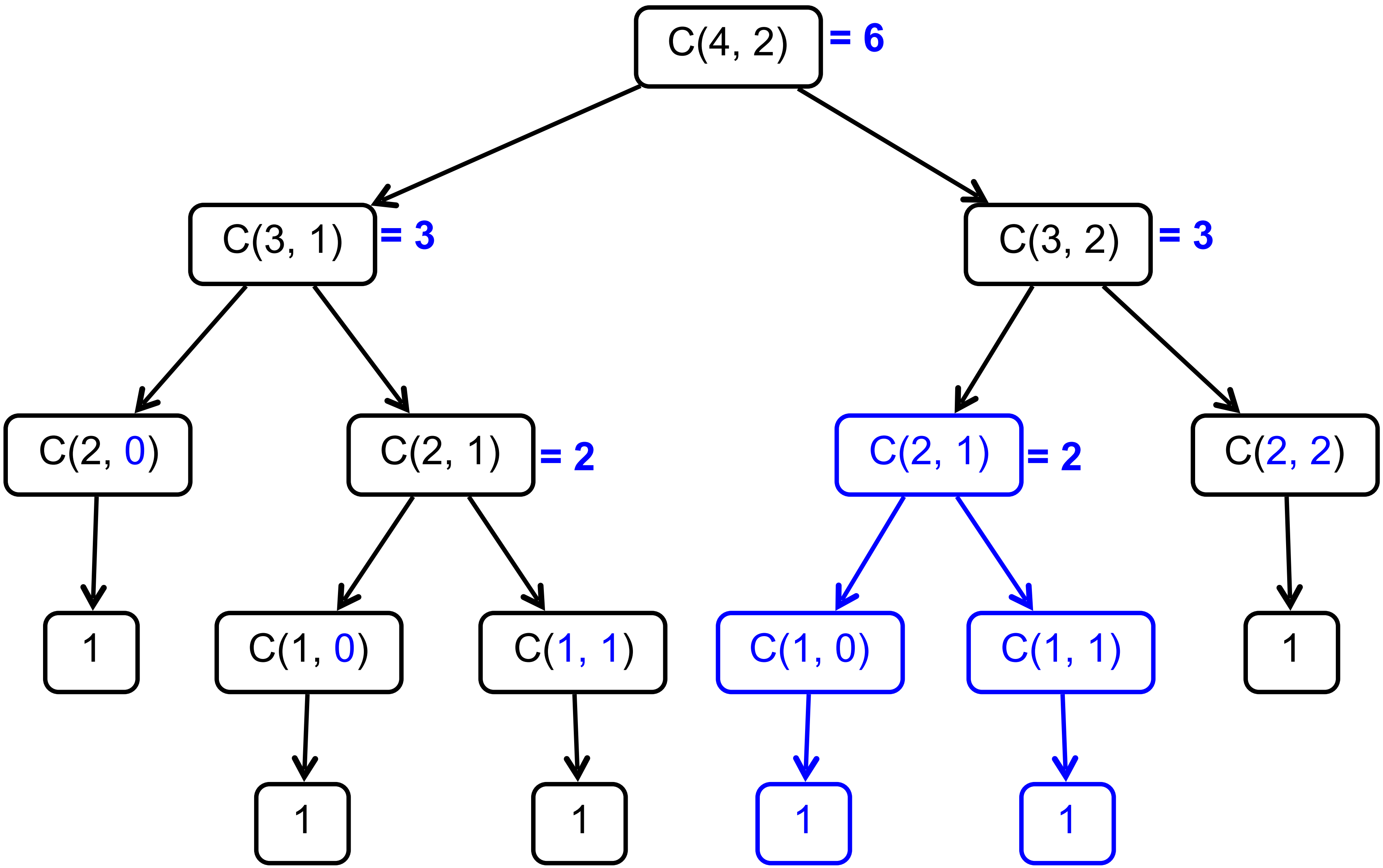
```
int choose(int N, int K) {
    if (N == 0 || K == 0 || N == K) {
        return 1;
    }
    int left  = choose (N-1, K - 1);
    int right = choose (N-1, K);
    return left + right;
}
```

Example: Binomial Coefficient (Parallel Recursive Pseudocode)

```
int choose(int N, int K) {
    if (N == 0 || K == 0 || N == K) {
        return 1;
    }
    future<int> left  =
              future { return choose (N-1, K-1); }
    future<int> right =
               future { return choose (N-1, K); }
    return left.get() + right.get();
}
```

- Use of futures supports incremental parallelization with low developer effort

What inefficiencies do you see in the recursive Binomial Coefficient algorithm?



Memoization

- Memoization - saving and reusing previously computed values of a function rather than recomputing them
    - A optimization technique with space-time tradeoff
- A function can only be memoized if it is *referentially transparent*, i.e. functional
- Related to caching
    - memoized function "remembers" the results corresponding to some set of specific inputs
    - memoized function populates its cache of results transparently on the fly, as needed, rather than in advance

Helpful Link: http://en.wikipedia.org/wiki/Memoization

Pascal's Triangle is an example of Memoization

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

N \ K	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

Example: Binomial Coefficient (sequential memoized version)

```
final Map<Pair<Int, Int>, Int> cache = new ...;

int choose(int N, int K) {
    Pair<Int, Int> key = Pair.factory(N, K);
    if (cache.contains(key)) {
        return cache.get(key);
    }
    if (N == 0 || K == 0 || N == K) {
        return 1;
    }
    int left  = choose (N - 1, K - 1);
    int right = choose (N - 1, K);
    int result = left + right;
    cache.put(key, result);
    return result;
}
```

Example: Binomial Coefficient (parallel memoized version w/ futures)

```
final Map<Pair<Int, Int>, Int> cache = new ...;
int choose(final int N, final int K) {
    final Pair<Int, Int> key = Pair.factory(N, K);
    if (cache.contains(key)) {
        return cache.get(key);
    }
    if (N == 0 || K == 0 || N == K) {
        return 1;
    }
    future<int> left = future { return choose(N - 1, K - 1); }
    future<int> right = future { return choose(N - 1, K); }
    int result = left.get() + right.get();
    cache.put(key, result);
    return result;
}
```

- Assumes availability of a “thread-safe” cache library, e.g., ConcurrentHashMap

Example: Binomial Coefficient (parallel memoized version w/ futures - better)

```
final Map<Pair<Int, Int>, future<Int>> cache = new ...;
int choose(final int N, final int K) {
    final Pair<Int, Int> key = Pair.factory(N, K);
    if (cache.contains(key)) {
        return cache.get(key).get();
    }
    future<Int> f = future {
      if (N == 0 || K == 0 || N == K) return 1;
      future<int> left = future { return choose (N-1, K-1); }
      future<int> right = future { return choose (N-1, K); }
      return left.get() + right.get();
    }
    cache.put(key, f);
    return f.get();
}
```

- Assumes availability of a “thread-safe” cache library, e.g., ConcurrentHashMap