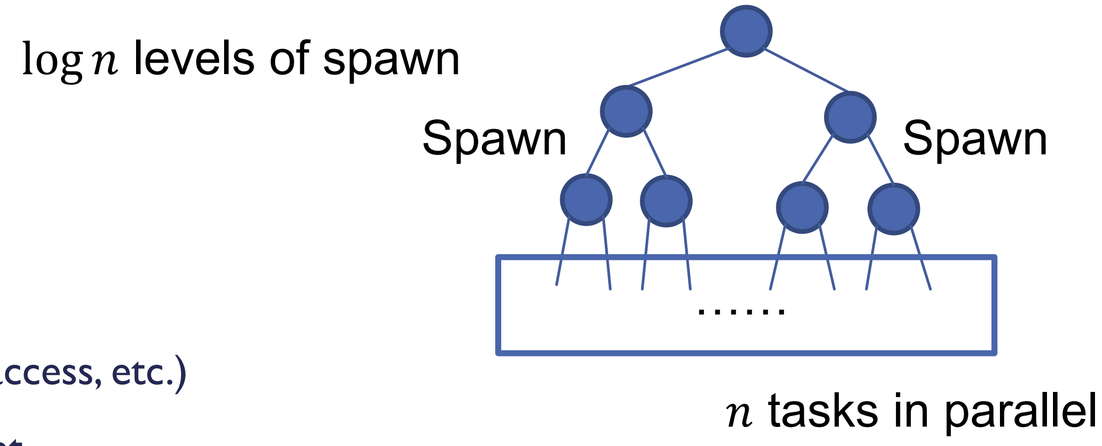
DISCUSSION CLASS

CS 141 F 20

A NEW MODEL TO ANALYZE COMPLEXITY

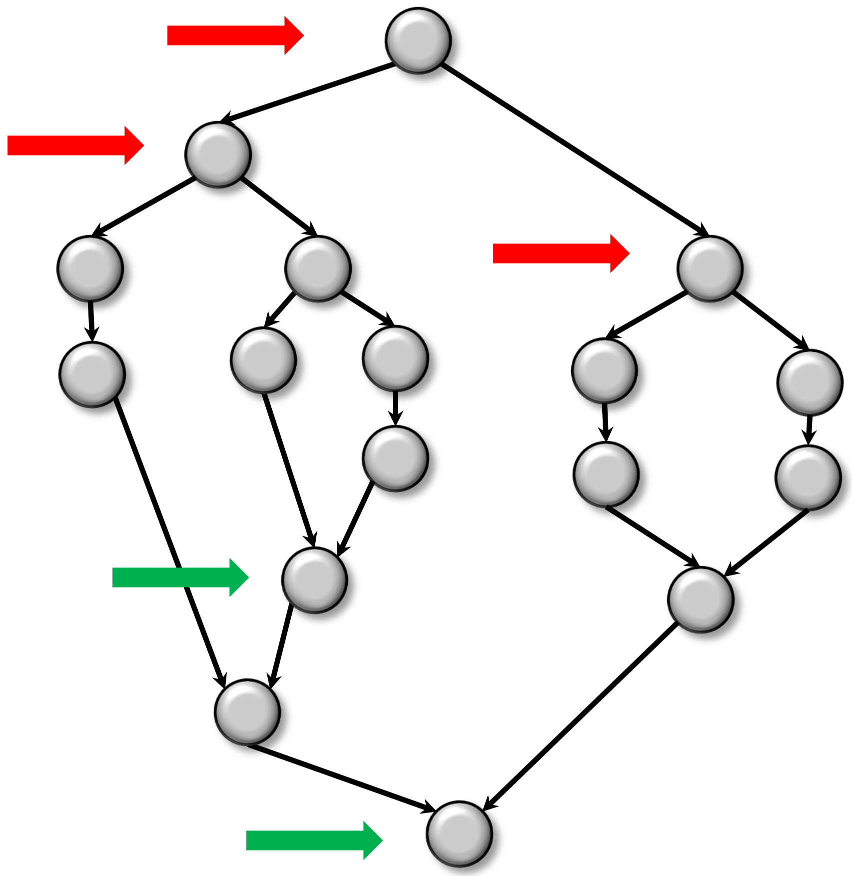
- For sequential programs, we used RAM model
  - Arithmetic operations, memory access done in constant time
  - Worst case is considered
  - There is only one thread
- Need new model to analyze complexity of parallel programs
  - Incorporate parallel operations/multiple threads: Binary fork-join model!

BINARY FORK-JOIN MODEL



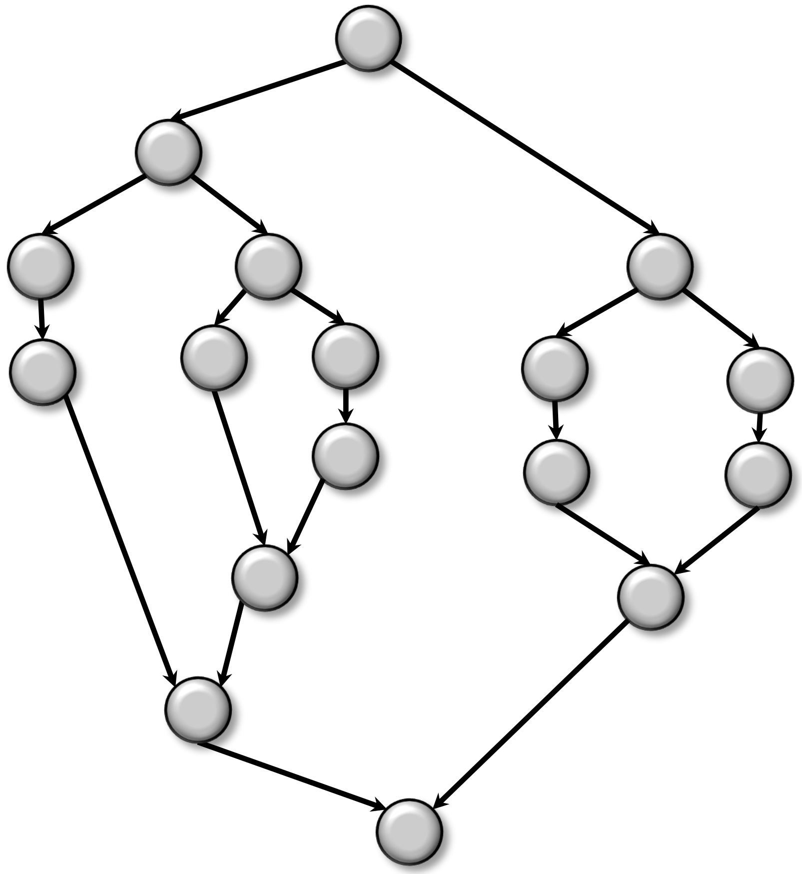
- Computation starts from one thread
- A thread can perform operations, such as:
  - Any sequential programming operations (arithmetic, memory access, etc.)
  - Spawn: start (fork) a new thread working on the next statement
  - Sync: previous forked processors synchronize (join) here
  - Parallel for: can be simulated by using $O(\log n)$ spawns, perform the computation of the for loops in parallel
- No concurrent write to the same memory location (or needs to be specified)

- spawn
- sync



COST MODEL: WORK-SPAN

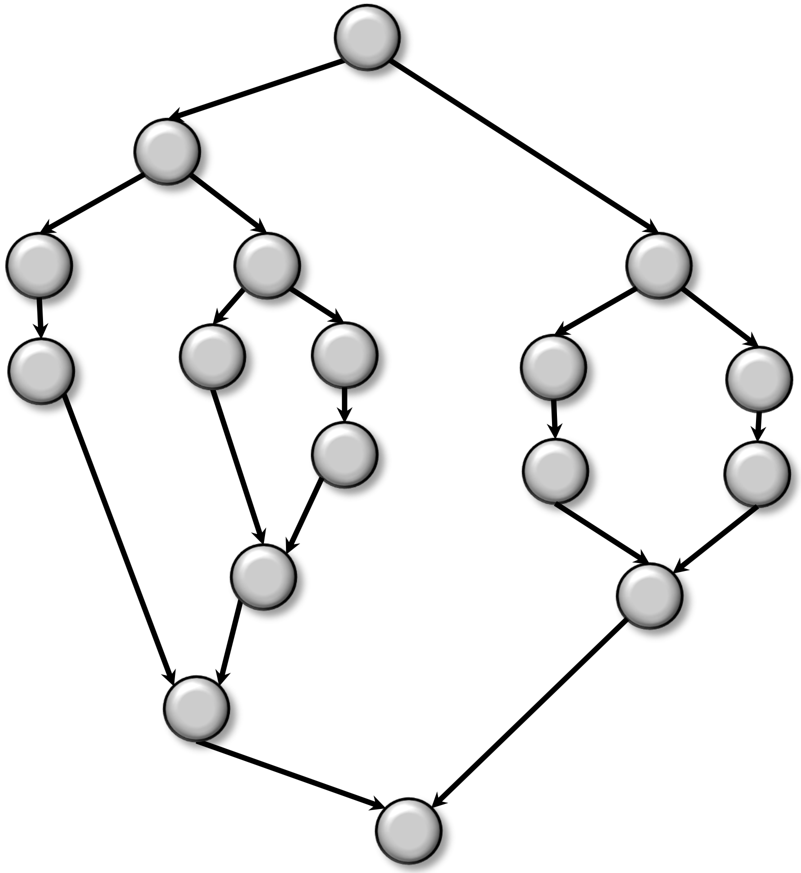
- For all computations, draw a DAG
  - A->B means that B can be performed only when A has been finished
- Work: the total number of operations
- Span (depth): the longest length of chain



DAG shows dependencies in the algorithm

WORK

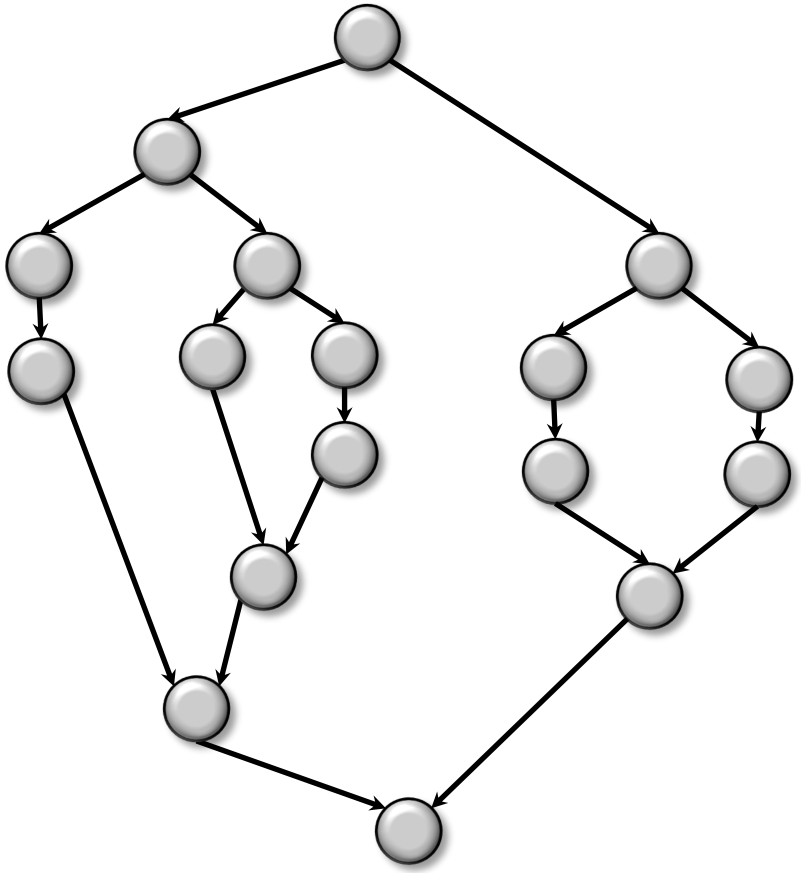
- Work: total number of operations
  - Sequential running time when the algorithm runs on one processor
  - Work-efficiency: work no more than the best sequential algorithm
  - Goal: make the parallel algorithm efficient when a small number of processor are available



We include all the nodes in the tree

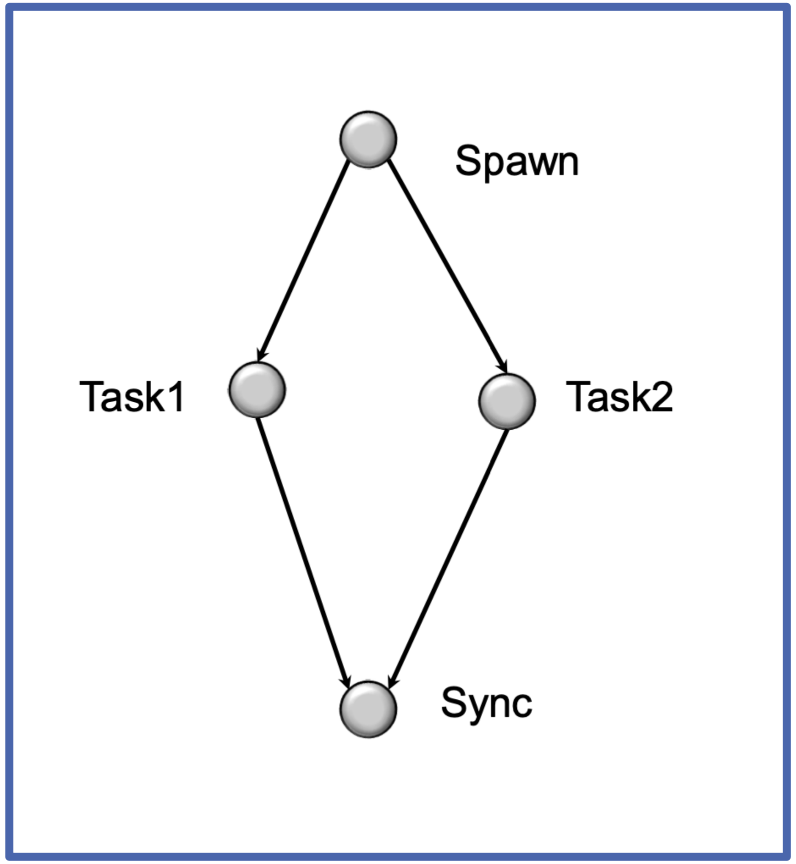
SPAN

- Span(depth): The longest dependency chain
  - Total time required if there are infinite number of processors
  - Make span polylogarithmic (in most of the cases)
  - Goal: make the parallel algorithm faster and faster when more and more processors are available - scalability



Include the depth of the tree (length of the longest path from root to leaf)

COMPUTE WORK AND SPAN



- Assume we have an algorithm in the following form:

  spawn ***Task1***

  Task2

  sync

- Work = work of Task1 + work of Task2
- Span = max(span of Task1, span of Task2)

SCHEDULING A PARALLEL ALGORITHM

- A DAG with work W and span D can be executed using **p** processors in time $O(W/\mathrm{p}+D)$
- Both W and D matter!
  - For small **p**, W is more important
  - For large **p**, D is more important

MERGE SORT (SEQUENTIAL)

```
MergeSort(int *A, int n)
    if (n<=1) return
    MergeSort(A, n/2)
    MergeSort(A + n/2, n-n/2)
    A = merge(A, n/2, A + n/2, n-n/2)
    return
```

MERGE SORT (PARALLEL)

```
MergeSort(int *A, int n)
    if (n<=1) return
    spawn MergeSort(A, n/2)
    MergeSort(A + n/2, n-n/2)
    sync
    A = merge(A, n/2, A + n/2, n-n/2)
    return
```

TIME COMPLEXITY

- Sequential Algorithm
  - $W(n) = 2W(n/2) + O(n) = O(n \log n)$
- Parallel Algorithm
  - $W(n) = 2W(n/2) + O(n) = O(n \log n)$
  - $S(n) = S(n/2) + O(n) = O(n)$

LONGEST PALINDROME

- Given a string, find the length of the longest palindrome
  - madam = 5
  - babad = 3 (bab, aba)
  - dbabad = 3 (bab, aba)
  - cbbd = 2 (bb)
  - a = 1
  - ac = 1 (a, c)

NAÏVE SOLUTION (SEQUENTIAL)

```
int longestPalindrome(String str)
    n = str.length, ans = 1
    for i = 1 to n
        for j = i+1 to n
            if (isPalindrome(str, i, j))
                ans = max(ans, j-i+1)
    return ans
```

NAÏVE SOLUTION (PARALLEL)

```
int longestPalindrome(String str)
    n = str.length, ans = 1
    parallel for i = 1 to n
        parallel for j = i+1 to n
            if (isPalindrome(str, i, j))
                ans = max(ans, j-i+1)
    return ans
```

TIME COMPLEXITY OF NAÏVE SOLUTION

- Sequential Algorithm
  - $W(n) = O(n^3)$
- Parallel Algorithm
  - $W(n) = O(n^3)$
  - $S(n) = (O(\log n)+O(\log n)) * O(n) = O(n \log n)$

DP ALGORITHM (SEQUENTIAL)

```
int longestPalindrome(string str)
    n = str.length, ans = 1
    mem[n][n] = {0}//2d array initialized
    for i = 1 to n
        mem[i][i] =1
    for i = 2 to n
        if str[i]== str[i-1]
            mem[i-1][i] = 1
    for len = 3 to n
        for i = 1 to n-len
            j = i + len -1
            if (str[i] == str[j] && mem[i+1][j-1])
                mem[i][j] = 1
                ans = len
    return ans
```

DP ALGORITHM (PARALLEL)

```
int longestPalindrome(string str)
    n = str.length, ans = 1
    mem[n][n] = {0}//2d array initialized
    parallel for i = 1 to n
        mem[i][i] =1
    parallel for i = 2 to n
        if str[i]== str[i-1]
            mem[i-1][i] = 1
    for len = 3 to n
        parallel for i = 1 to n-len
            j = i + len -1
            if (str[i] == str[j] && mem[i+1][j-1])
                mem[i][j] = 1
                ans = len
    return ans
```

TIME COMPLEXITY OF DP ALGORITHM

- Sequential Algorithm
  - $W(n) = O(n) + O(n) + O(n^2) = O(n^2)$

- Parallel Algorithm
  - $W(n) = O(n) + O(n) + O(n)*O(n) = O(n^2)$
  - $S(n) = O(\log n) + O(\log n) + O(n)*O(\log n) = O(n \log n)$