COMP 322: Parallel and Concurrent Programming

Lecture 16: Parallelism for Recursive Data Structures

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Some of the slides adopted and modified from: EPCC, The University of Edinburgh, www.epcc.ed.ac.uk

The Divide and Conquer Paradigm

- Important general technique for designing algorithms
- In general, it follows the steps:
  - divide the problem into subproblems
  - recursively solve the subproblems
  - combine solutions to subproblems to get solution to original problem
- Works well for parallelization too
  - ***IF*** the subproblems are independent

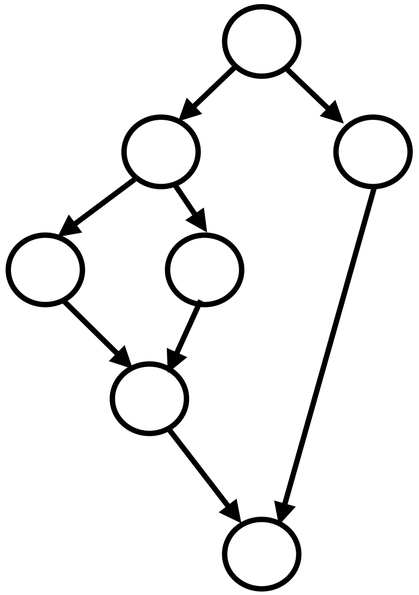
Lab 4: Recursive Task Parallelism

```
private static double recursiveMaxParallel(final double[] inX, final int start, final int end)
        throws SuspendableException
{
    if (end - start == 2) {
        doWork(1);
        return 1/inX[end - 1] + 1/inX[start];
    } else if (end == start + 1) {
        return 1/inX[start];
    } else {
        var bottom = future(() -> recursiveMaxParallel(inX, start, (end + start) / 2));
        var top = future(() -> recursiveMaxParallel(inX, (end+start) / 2, end));
        var bVal = bottom.get();
        var tVal = top.get();
        doWork(1);
        return bVal + tVal;
    }
}
```

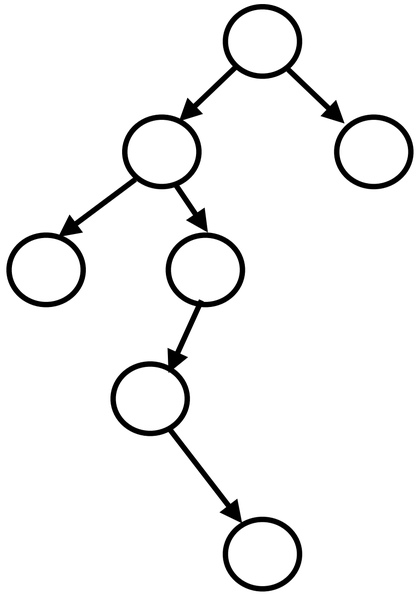

Recursive Data

Given a problem described by an algorithm which involves moving through a recursive data structure in a seemingly sequential way, how can the algorithm be modified to expose parallelism?

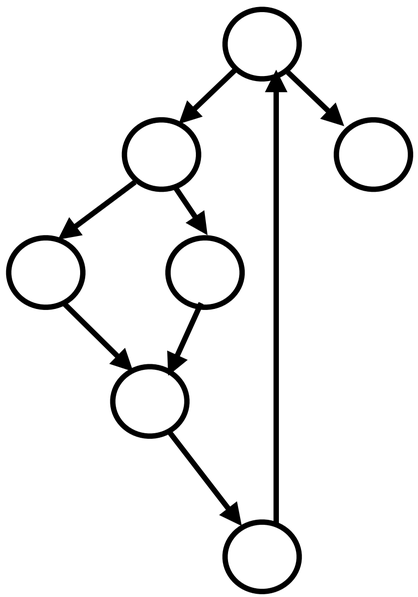
DAGs



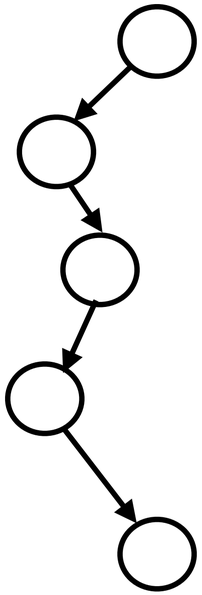
Trees



Graphs



Lists



Context

- Many problems with recursive data structures can be solved with Divide & Conquer

  If this can be used, use it.

  Other algorithms appear to have to move sequentially through the data structure and computing the result at each element.

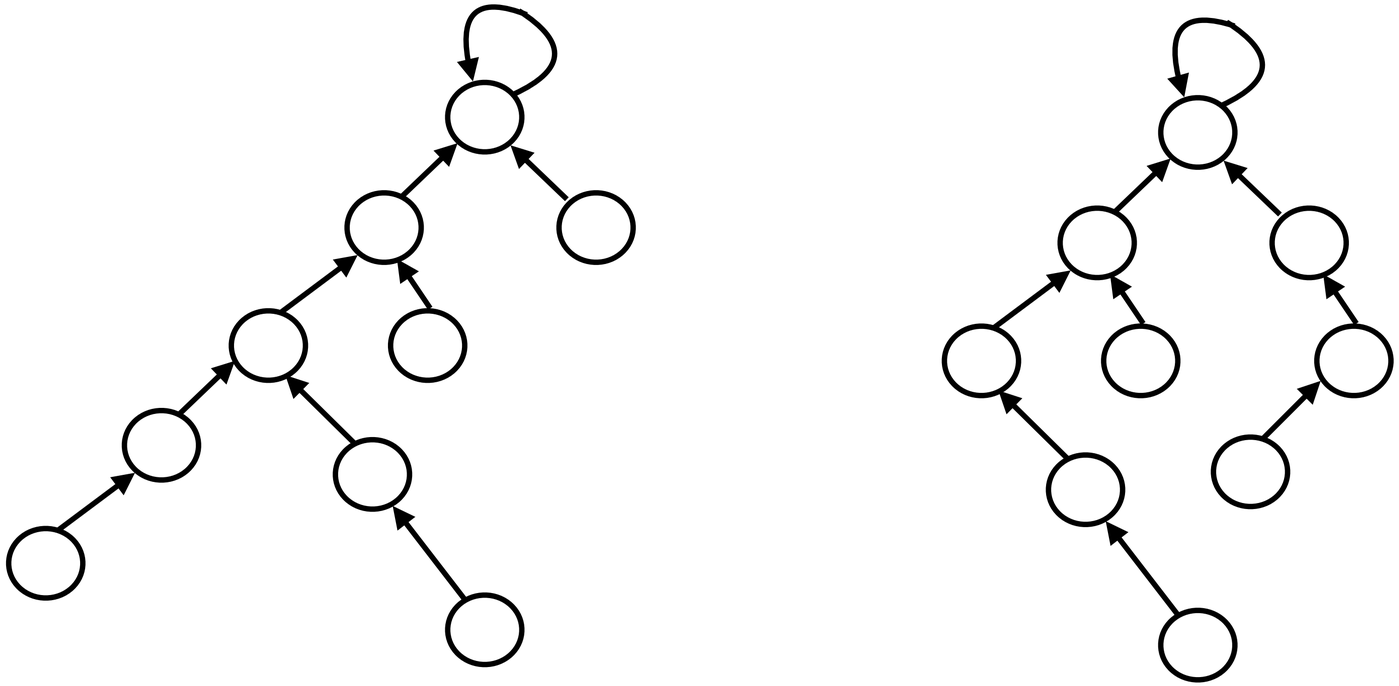
- It's often possible to re-cast your calculation so that instead of acting on each element in the data structure in turn, you can perform modified operations in parallel

  Also known as "Pointer Jumping" or "Recursive Doubling"

Recursive Data: an Example

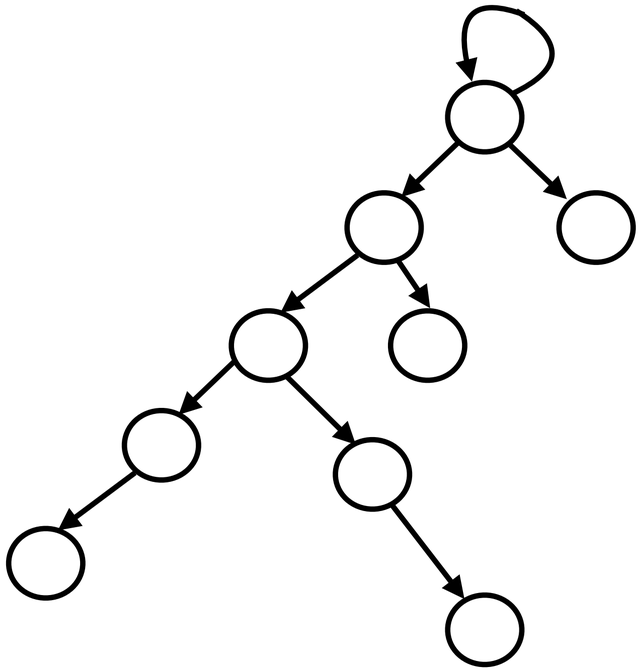
- Finding roots in a forest

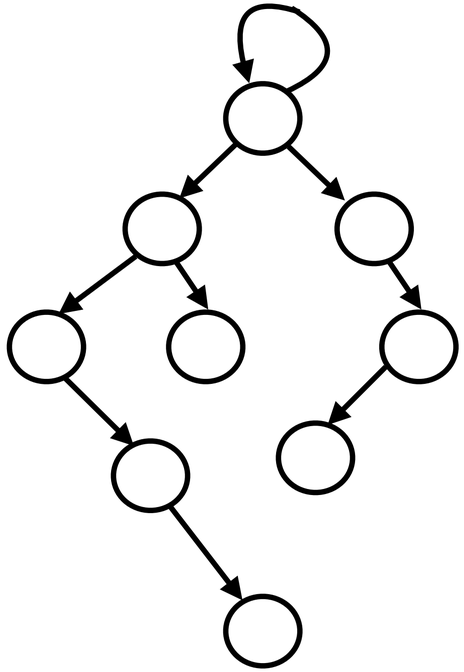
  For each node in a forest, compute the root of the tree that contains that node
  Joseph JáJá: “An Introduction to Parallel Algorithms”, 1992



Sequential Algorithm. O(n)

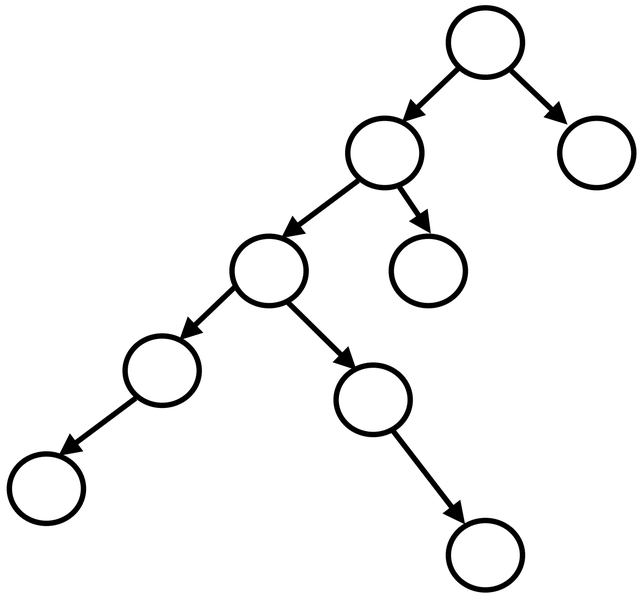
- Step 1: find the roots. O(n)
- Step 2: do a DFS, starting with the roots. Point each node to root. O(n)

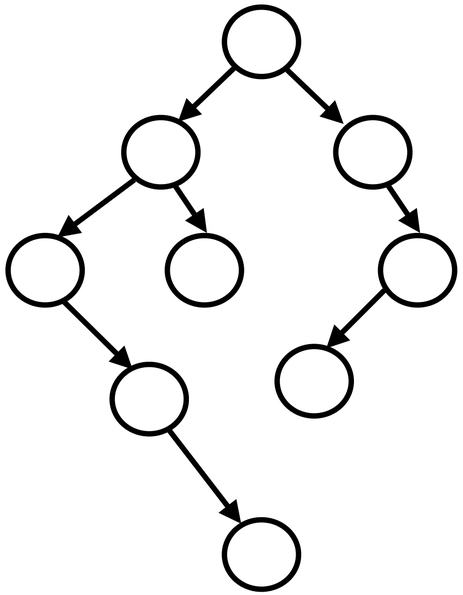




Parallel Algorithm: Naive

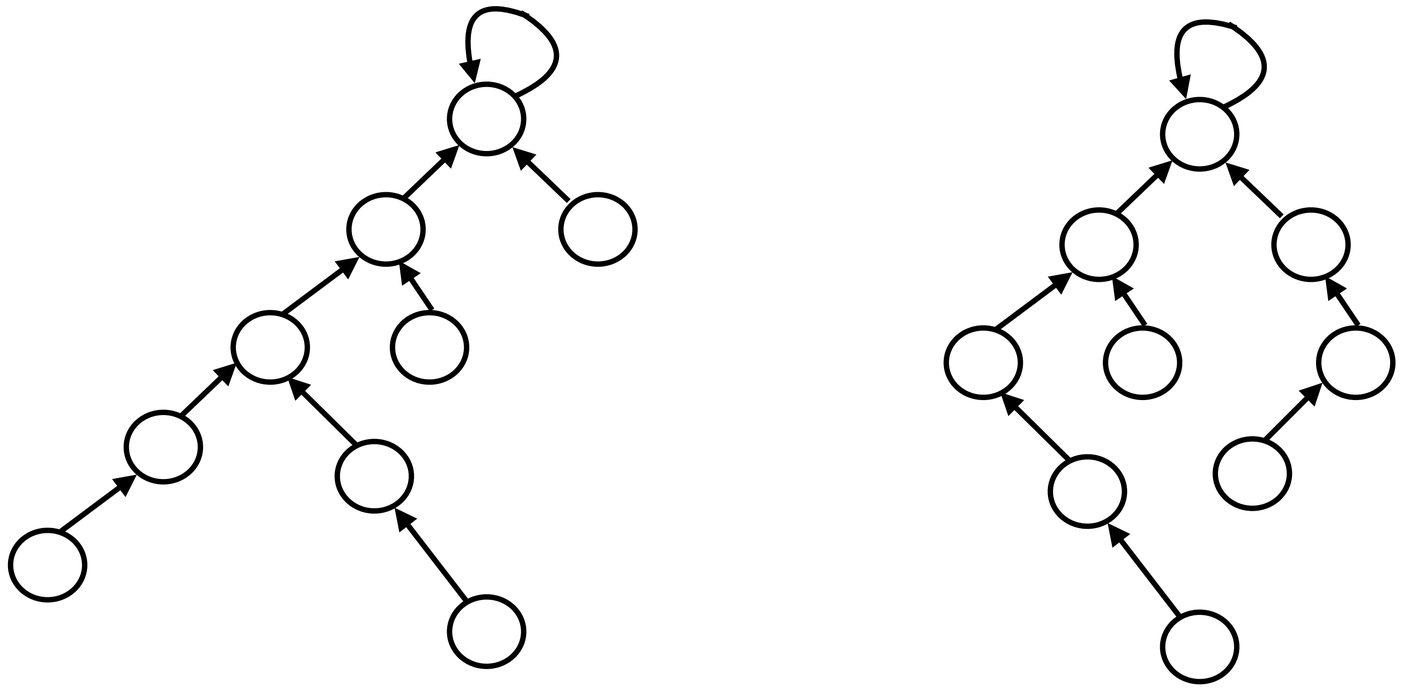
- Make a DDF out of each node's "root computed" feature
- Make a data-driven task that depends on parent
- Inside of the DDT, set the node's root to its parent's root
- Will process trees and subtrees in parallel, but performance heavily reliant on what the trees look like
- WORK: O(n). CPL: O(h), where h is the height of the tallest tree





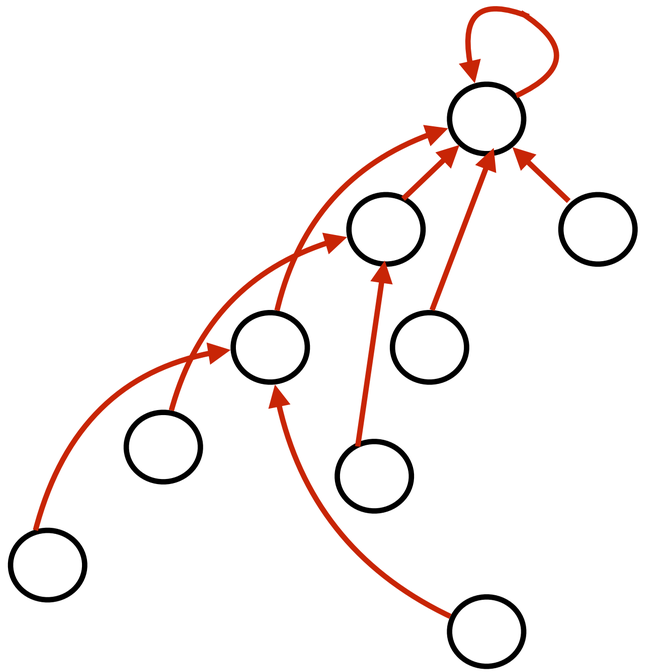
Better Parallel Algorithm

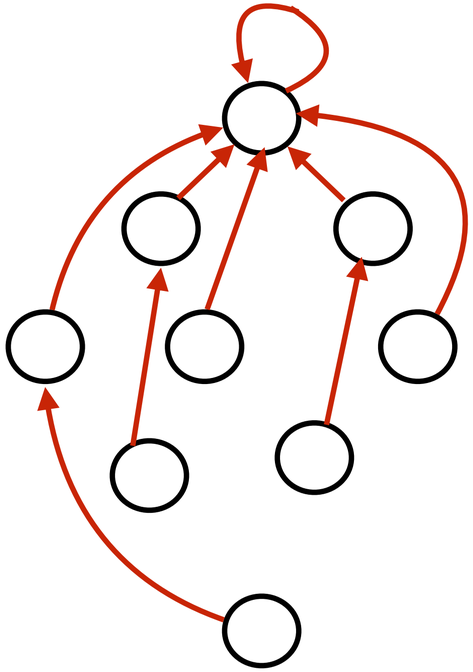
- Set each node's root to its parent
- For each node, set its root to its parent's root, if it exists
- This can all be done in parallel using N tasks



Better Parallel Algorithm

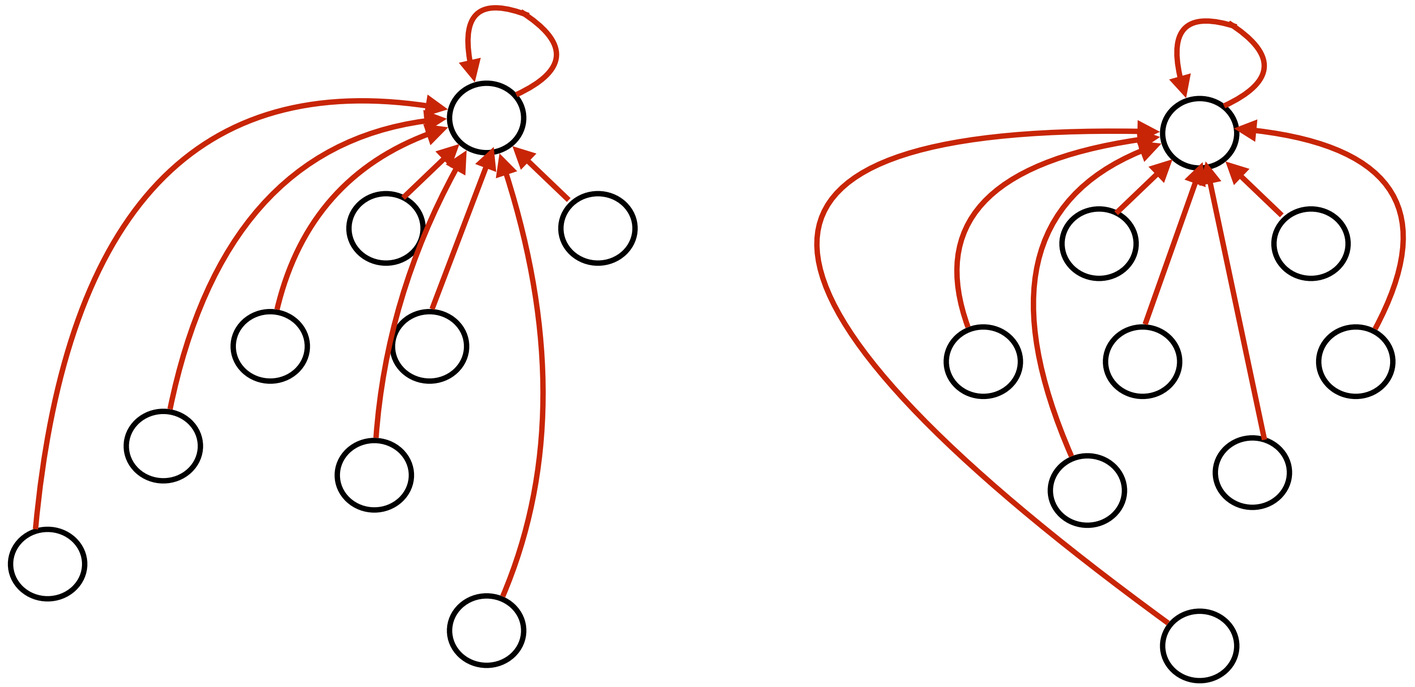
- For each node's root starts as its parent
- For each node, set its root to its parent's root, if it exists
- This can all be done in parallel using N tasks





Better Parallel Algorithm

- Again:
- For each node, set its root to its parent's root, if it exists
- This can all be done in parallel using N tasks again
- Stop when no more updates can be done



Better Parallel Algorithm

- N tasks in each phase of the computation
- log(h) phases
- Total WORK: O(N log(h))
- CPL: O(log(h))
- By reshaping the algorithm, we have exposed additional parallelism
- We are doing more work, but (hopefully) the added parallelism makes up for it
- But be careful: there is a tipping point

  N = 1024, and h = N, ideal parallelism = 1024

  With 1 cycle per task, and 1024 processors, we'll complete the work in 10 cycles

  But with less than 10 processors, we'll need more than 1024 cycles. Worse than sequential!

  10 is the "break even" point

Considerations

- Reformulating the problem to ensure that parts of the data structure can be operated on independently usually increases the total amount of work to be performed
  - You have to consider this trade-off
- Reformulating the problem may be difficult
  - In some cases maybe even impossible
  - Often results in less-than intuitive design
    - Harder to understand and maintain
- Exposed parallelism may be difficult to exploit
  - Too fine-grained
  - Requires too much copying

Summary

- Divide and Conquer is a great general technique for designing algorithms
- Works well in parallelization too
- Futures and future tasks are a great tool for implementing parallel Divide and Conquer solutions
- Recursive data structures often lend themselves to Divide and Conquer and recursive task parallelism
- Sometimes, the most intuitive algorithm would require traversing the recursive data sequentially
  - In this case, “pointer jumping” might help in parallelization
  - But, it usually creates more work, so there might be a “breaking even” point