4.1 Divide and Conquer

Most divide and conquer algorithms can easily be easily parallelised. We won't go into much detail about these algorithms since we have already discussed several of these, such as:

- Quick sort
- Merge sort
- Merge
- Integer multiplication
- Matrix inversion
- Matrix multiplication
- Summation

4.2 Contraction

Contraction is most common in parallel algorithms (but used infrequently in sequential algorithms). It differs from divide and conquer in that it contains only a single recursive call.

4.2.1 Process

Algorithms belonging to this paradigm usually have the following steps:

1. Reduce to a smaller problem (**Reduction**)
2. Solve the smaller problem (often recursively)
3. Use the results to solve the original larger problem (**Expansion**)

4.2.2 Example: Prefix Sum Problem

Consider the array, $A = [1, 4, 2, 1, 3, 1, 2, 5]$. The prefix sum of A is the array $PS(A)$ which contains for it's i-th elements the sum of all elements in array A till position i. Though this looks like a sequential algorithm, it can be made parallel:

In the first step, we can pair off adjacent elements and find the sum of such pairs to get array A' (reduction phase). Not that this can be done in $O(1)$ depth and $O(n)$ work. Let see we can recursively compute the prefix sum, $PS(A')$ of A'.

$A' = [5, 3, 4, 7]$
$R' = PS(A') = [0, 5, 8, 12]$

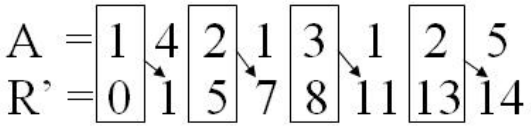

Figure 4.2.1: Expansion Step

We now note that prefix sum of A' contains the odd numbered elements of the prefix sum of A. To fill up the even numbered elements, we can simply use the relation $PS(A)[2*i-1] = PS(A')[i-2]+A[2*i-2], i \geq 2$ (expansion phase). Therefore, these missing even numbered elemts can also be filled up in depth $O(1)$ and work $O(n)$. So, we have the recursion $W(n) = W(n/2) + O(n) \implies W(n) = O(n)$.

4.3 Partitioning

Partitioning is similar to a 2-level divide and conquer algorithm, but it contains no recursive calls.

4.3.1 Process

1. Partition the problem into k pieces
2. Now solve the pieces with a different algorithm (this may be sequential)
3. Combine the results

4.3.2 Example: Merging Using Partitioning

Given two arrays, A and B, of size n, divide each into $n/\log n$ pieces, each of size $\log(n)$. For each piece P in A, use binary search to find where the endpoints of P fit in B. For each segment and its corresponding span in the other array, insert the larger piece into the smaller (inserting each element in the correct place).
Binary search has work $W(n) = O(n/\log n) \cdot \log n = O(n)$ and depth $D(n) = \log n$.
The sequential merges have work $W(n) = O(n/\log n) \cdot \log n = O(n)$ (as long as we insert the larger portions of the arrays into smaller portions) and depth $D(n) = \log n$.

4.4 Pointer Jumping

Pointer jumping creates shortcuts between pointer structures. It is generally not work efficient but can be effective as a substep in another algorithm.

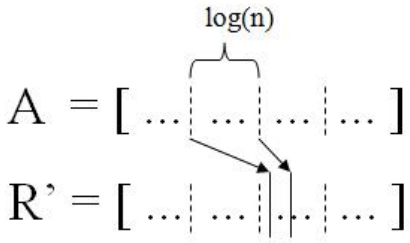

Figure 4.3.2: Merging Array Portions

4.4.1 Process

1. Call $p = *p$ in parallel for all pointers
2. Repeat as needed

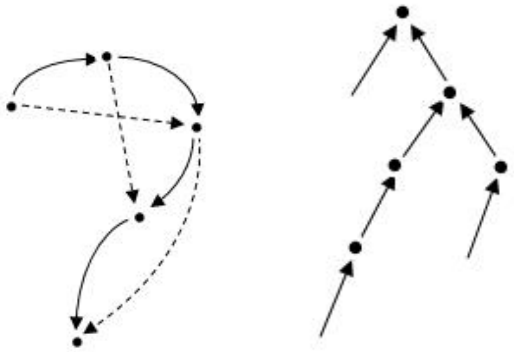

Figure 4.4.3: Pointer Jumping, and Finding the Root of a Tree

4.4.2 Example: Find Root in a Tree

Given a tree with each node pointing to its parent, simply repeat $p = *p$ for all nodes several times. Eventually all nodes will point to the root.
The worst case runtime occurs when the tree is a linked list, although this still only takes $O(\log n)$. However, since we update *all* nodes on each iteration, this is not work efficient, with $W(n) = O(n \log n)$ and $D(\log n)$.

4.5 Symmetry Breaking

Symmetry breaking is used to make a (random) choice among elements that look identical. It is use in scenarios where it can be very difficult to find a deterministic algorithm, as a deterministic algorithm might need additional information to break the symmetry.

4.5.1 Example: Independent Set

Suppose we are given a circular linked list. Finding the maximal independent set would be trivial with a sequential algorithm, which must have a fixed starting point (simply select every second element). However, since we are running in parallel, nothing distinguishes each node from the others. We can use randomized symmetry breaking with simulated coin flipping. Each node randomly recieves a value, either heads or tails.

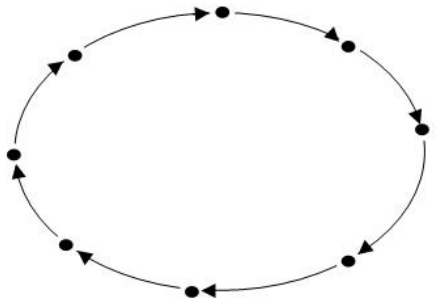

Figure 4.5.4: A Symmetrical, Circular Linked List

- If a node flips heads and the node in front of it has tails, then the current node is included in the independent set
- If a node flips heads and the node in front also flips heads, then the node is not in the set
- If the node flips tails, then it is in selected for the set

This guarantees that, with a high probability, a constant fraction of nodes in a maximal independent set will be selected ($1/4$ of the nodes) in each round. Also, each round runs in constant time. Thus in polylogarithmic number of rounds, we would have obtained a maximal independent set with high probability.

4.6 Pipelining

Most pipelining algorithms are either not work efficient or have a high depth.

4.6.1 Process

Pass partial results to a task, which will deal with them in parallel

4.6.2 Example: Prime Number Generation

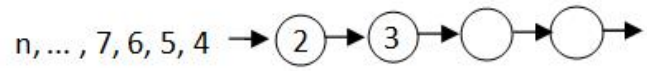

Figure 4.6.5: Finding Primes Via Pipelining

Begin with a stream of numbers $[2, ..., n]$ and a series of processes. Feed the stream into the processes, starting with 2. Store the first value in the first process. For each subsequent value, if it is a multiple of any value through which it passes, it is discarded. If it reaches an empty space, it is stored there, and the process continues. In this way, the non-prime numbers are filtered out. Many values are concurrently "pipelined" through the system at once.

A sequential method can run the same process for n numbers in time $O(n\log n)$. Even if our parallel algorithm stops at the $\sqrt{n}^{\text{th}}$ term, it still has work $W(n) = O(\frac{\sqrt{n}}{\log n} \cdot n) = O(\frac{n^{3/2}}{\log n})$. The parallel algorithm is almost certainly slower than the sequential version, even with unlimited processors.