5/4

Parallel Algorithms: (vs. traditional sequential (or serial) algorithm)

There are numerous types of parallel computers in operations and no longer we can adopt one "generic" model of computation and hope that it adapts to all parallel computers. In this short class we can not cover most of parallel computing. We just give some examples. You can take other classes on this topic.
esp.

As we have seen in non-deterministic computers having an unlimited number of machines can help us. Often the more processors we use, up to a certain limit, the faster the algorithms become. However since in real-world the number of computers (processors) is limited, it is important to use the processors effectively. Another important issue is communication among the processors. Generally, it takes longer to exchange data between two processors than it does to perform simple operations on the data. Therefore it is important to minimize communications and to arrange it in an effective way. Yet another important issue is synchronization, which is a major problem for parallel algorithms that run on independent computers connected by some communication network.

Speed-up: We denote the running time of an algorithm by $T(n,p)$, where n is the input size and p is the number of processors. The ratio $S(p) = \frac{T(n,1)}{T(n,p)}$ is called the speed up the algorithm. If $S(p) = p$, we call it a perfect speedup. The value of $T(n,1)$ is the best known sequential algorithm.

We can fix p and then minimize $T(n,p)$. However if p changes, we required a new algorithm. It is more desirable to find an algorithm that works for any values of p.

In general, we can modify an algorithm with $T(n,p) = X$ to an algorithm with $T(n, \frac{p}{k}) = kX$ by replacing each step of the original algorithm with k steps in which one processor emulates (simulates) the execution of one step of k processors. However there are some exceptions when k does not divide p, etc.

Various models of parallel computation and communication

Shared-memory model: assume that there is a random-access shared memory such that any processor can access any variable with unit cost. The assumption of unit cost access regardless of the number of processors or the size of the memory is unrealistic but it is a good approximation. We may need additional locking for data (to avoid overwrit

<u>Interconnection network</u>: Can be represented by a graph such that the vertices correspond to the processors and two vertices are connected if the corresponding processors have a direct link between them. Each processor has a quick-access local memory, but the communication is done through messages (though maybe by traversing several links to arrive at their destinations.

A ~~several~~ subtype of parallel algorithms which is in-use in our modern world and over the internet is a distributed algorithm which often uses interconnection networks.
one of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behaviour of the independent parts of the algorithm in the face of processor <u>failures and unreliable communication links</u>.

<u>Maximum-Finding Algorithm in parallel</u>:

<u>The problem</u>: Find the maximum among n distinct numbers, given in an array.
As we have seen $T(n,1) = n-1$. An efficient way to run a parallel algorithm is to use a binary tree. The processors are divided into pairs for the first round (with possibly one processor sitting out, in case of an odd number of players), all the winners are again divided into pairs and so on until the finals. The number of rounds is $\lceil \log_2 n \rceil$ and the number of processors is $\lfloor \frac{n}{2} \rfloor$. Thus $T(n, \lfloor \frac{n}{2} \rfloor) = \lceil \log_2$
and $S(P) = O(\frac{n}{\log n})$ which is very good. Note that we use the shared-memory mod

A Generalization: <u>The Parallel-Prefix Problem</u>:
let $\cdot$ be an arbitrary associative binary operation, namely it satifies $x \cdot (y \cdot z) = (x \cdot y) \cdot z$, which we simply call product. For example $\cdot$ can represent addition, multiplication, or max of two numbers.
<u>The problem</u>: Given a sequence of numbers $x_1, x_2, \cdots x_n$, compute the product $x_1 \cdot x_2 \cdot \cdots \cdot x_k$ for all k, such that $1 \leq k \leq n$.
we denote by $PR(i,j) = x_i \cdot x_{i+1} \cdot \cdots \cdot x_j$. The goal is to compute $PR(1,k)$ for all $1 \leq k \leq n$.
The sequential version of the problem can be solved trivially by simply computing the prefixes in order. The parallel case is harder and needs divide and conquer. Say n is power of 2

IH: we know how to solve the problem for $\frac{n}{2}$ elements.
The basis for one element is trivial. The algorithm proceeds by dividing the inp
in half, and solving each half by induction. Thus we obtain the values of
$PR(1,k)$ and $PR(\frac{n}{2}+1, \frac{n}{2}+k)$ for all k, $1 \le k \le \frac{n}{2}$. The values for the first half can b
used directly. The values $PR(1,m)$, for $\frac{n}{2} < m \le n$ can be obtained by computing
$PR(1,\frac{n}{2}) \cdot PR(\frac{n}{2}+1, m)$. Both terms are known by induction (notice that we use
the associativity of the operation). This step can be done in one step if we have
n processors. Thus overall $T(n,n) = O(\log n)$ and $S(n) = O(\frac{n}{\log n})$ which is very good.