15-492: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Introduction **Date:** 08/28/2007
Scribe: Kanat Tangwongsan

1.1 Course Announcement

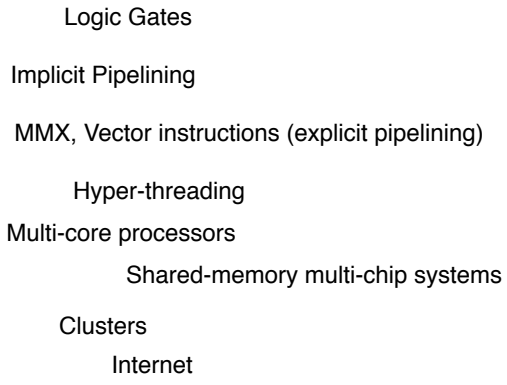
In this course we will study parallel algorithms, with an emphasis on algorithms that can be used on shared-memory parallel machines such as multicore architectures. We will devote the first half of the semester to the theory of parallel algorithms and spend the second half learning about various parallel-programming platforms. An approximate schedule can be found on the course's web site.

- **Instructor:** Guy Blelloch (`guyb@cs.cmu.edu`), WeH 7125
- **TA:** Kanat Tangwongsan (`ktangwon@cs.cmu.edu`), WeH 7102
- **Website:** `http://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/`
- **Course Mailing List:** `15492-f07@lists.andrew.cmu.edu`

Course Requirements and Grading: scribing 2 lectures (15%), a take-home midterm (15%), 5 assignments (50%), and a final Project (20%).

1.2 Ubiquitous Parallelism

Parallel computing has been a subject of active study for decades. From logic gates inside a microprocessor to millions of machines on the Internet, some level of parallelism is available.



Different levels of parallelism are shown in the diagram above. Inside a processor, logic gates can process signals simultaneously. In modern processors, multiple instructions are executed at the same time. We call this (implicit) pipelining. Some modern processors also have vector instructions (e.g., MMX instructions). This is a form of explicit pipelining—the user explicitly specifies that certain operations should be carried out in parallel. For this course, we will focus on small-scale parallelism (i.e., how a single user can exploit parallelism).

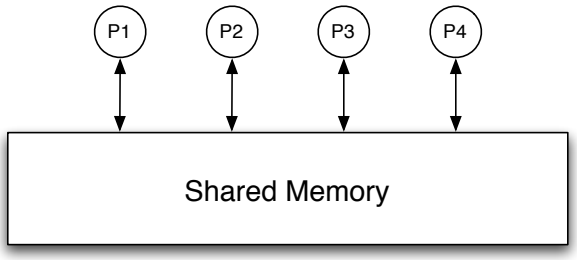
1.3 Models of Parallelism

Computer scientists invent abstract models of machines so that a theoretical study can accurately predict the performance of an algorithm on an actual computer without having to deal with the complicated details of an actual machine. For sequential machines, the RAM model has been the standard model for theoretical analysis of algorithms. In this section, we will review the RAM model and introduce few models of parallel machines, which will be examined later in the course.

The RAM Model. The RAM model has successfully been used to study the performance of sequential algorithms. In the RAM model, instructions are executed in a sequence one after another, with no concurrent operations. A random access machine (RAM) consists of a program counter (PC), a number of registers, and some (random-access) memory. See CLRS for more details.

The Circuit Model. In parallel computing, an important aspect we want to capture is dependencies. For a large class of computations, this can be modeled as a directed acyclic graph (DAG), where a directed edge from u to v indicates that the computation at v depends on the outcome of u. Then the cost of a computation can be characterized by the size (number of nodes) and the depth (length of the longest path) of the DAG. The circuit model, however, is less than ideal, because it is static and lacks the ability to model dynamic decisions that occur in most algorithms.

The PRAM Model. With the success of the RAM model, we are tempted to extend it to model parallel machines. We will ignore the nitty-gritty of parallel machines for now. In the simplest form, a PRAM operates on a shared memory, as depicted in the figure below. All processors run synchronously under the control of a single clock.



1.4 Matrix Multiplication

Let us assume for simplicity that we always deal with square matrices. Given matrices $A \in \mathbb{F}^{n \times n}$ and $B \in \mathbb{F}^{n \times n}$, we want to computer the matrix $C = AB$. This is one of the most fundamental problems in computing. If we let A_{ij} denote the entry at the j-th column of row i, then the definition of matrix multiplication gives that $C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$. This summation is exactly the dot product of the i-th row of A with the j-th column of B. From this, one can come up with the most natural sequential algorithm for matrix multiplication like the one below. Note that lines 3 to 6 is nothing more than computing the dot product (as mentioned earlier).

1: **for** $i = 1$ to n **do**
2: **for** $j = 1$ to n **do**

3: $C_{ij} = 0$;
4: **for** $k = 1$ to n **do**
5: $C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$;
6: **end for**
7: **end for**
8: **end for**

Now looking more closely at the sequential code above, we quickly realize that if have n^2 processors, each C_{ij} can be computed separately and in parallel with other entries. We just designate each processor to run lines 3 to 6 of the code above, one processor for each $(i, j) \in \{1, 2, \ldots, n\} \times \{1, 2, \ldots, n\}$. Using this scheme, the algorithm has work $O(n^3)$ and depth $O(n)$. That is $O(n^2)$ speed-up from the naïve sequential algorithm.

Can we do even better? Clearly if we can speed up the dot-product computation to, say, $O(\log n)$ depth, we will have improved the algorithm to $O(\log n)$ depth. The crucial observation here is that $+$ is associative. For example, we know that $((a+b)+c)+d = (a+b)+(c+d)$. Let $a_k = A_{ik} \cdot B_{kj}$, so $C_{ij} = \sum_{k=1}^{n} a_k$. One can imagine building a perfect binary tree, where all a_k's sit at the leaves, and each internal node is the sum of the two children. Figure 1.4 shows an example with $n = 8$. We know that a perfect binary tree has depth $O(\log n)$. Since an internal node depends only on its two children, the depth of the dot-product calculation is $O(\log n)$.

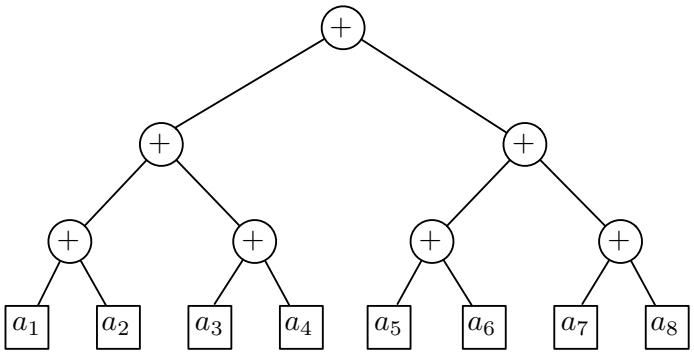


Figure 1.4.1: A perfect binary tree for summing $n = 8$ numbers.

1.5 Quick sort

Quick sort is a popular divide-and-conquer, comparison-based sorting algorithm. When the pivots are chosen uniformly at random, we know that quick sort makes $O(n \log n)$ comparisons with high probability[1]. Below is a pseudo-code for sequential quick sort:

$\mathsf{QSort}(A) =$
if $|A| \leq 1$ **return** A

[1] An event $\mathcal{E}$ occurs with high probability if $\mathbf{Pr}[\mathcal{E}] \geq 1 - O(n^{-c})$ for some constant $c \geq 1$.

else
  $p = A[\mathsf{rand}(|A|)]$;
  return $\mathsf{QSort}(\{x \in A : x < p\})$++$\{p\}$++$\mathsf{QSort}(\{x \in A : x > p\})$;

If we draw the recursion tree for quick sort, we will see that a call only depends on their predecessors. Therefore, a simple thing is try to parallelize the recursive calls. Assuming that quick sort always splits the array in half every time, this gives us a parallel algorithm with depth $D(n) = D(n/2) + O(n)$ and work $W(n) = 2W(n/2) + O(n)$. The recurrence relations[2] solve to $D(n) = O(n)$ and $W(n) = O(n \log n)$. With a more careful analysis, we can show that randomized quick sort which spawns a new thread for each recursive call has work $O(n \log n)$ and depth $O(n)$ with high probability.

After all this is not so impressive—having infinite processors only yields $\log n$ speed up. When we examine this code more, we realize that perhaps we want to parallelize the partitioning routine. As we will see later in the course, partitioning (i.e., constructing the array $\{x \in A : x < p\}$) can be accomplished in parallel with depth $O(\log n)$ and work $O(n)$. This will give us a version of parallel quick sort with depth $O(\log^2 n)$ and work $O(n \log n)$.

[2]There are many ways to solve these recurrences. E.g., one can use the master formula (see e.g. CLRS) or simply unfold them.