15-750: Parallel Algorithms

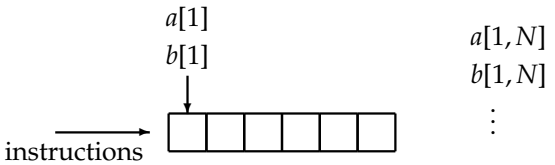
Scribe: Ilari Shafer

March {18,21} 2011

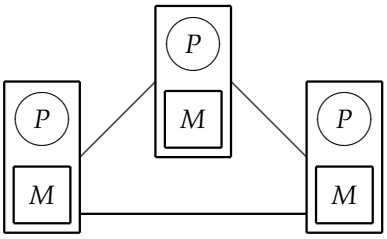
1 Introduction

1.1 A Few Machine Models of Parallel Computation

SIMD Single instruction, multiple data: one instruction operates on multiple data items. Examples: vector machines, graphics cards



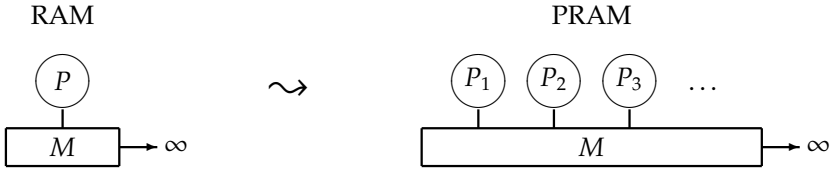
Distributed Memory Each processor P has local memory M; processors communicate over some links. Examples: Google's clusters.



PRAM Parallel Random Access Machine: an extension of the RAM model to multiple processors operating on the same shared memory.

- Infinite number of processors, each with some it's own local registers (as many as you want). Each processor knows it's index.
- A shared memory (Infinitely large)
- A shared global clock
- In each clock tick, each processor can read or write one memory location, or do some basic operation (arithmetic operation).

PRAM machines can either have concurrent/exclusive reads and writes (CRCW, CREW, ERCW!, EREW). The model has to specify what happens when there is a concurrent write: usually it is either junk, or one of the entries written.

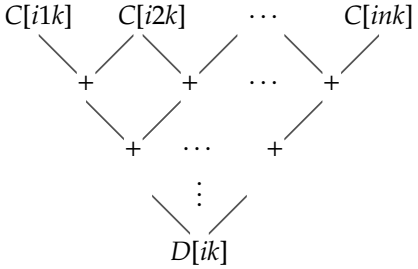


1.2 Example: Matrix Multiplication

For the CREW PRAM model: given matrices $A_{n\times n}$, $B_{n\times n}$, produce a new 3-dimensional matrix C:

$$C[ijk] = A[ij] \times B[jk]$$

where the left hand side consists of one write operation per location, and the right hand side consists of concurrent reads. This can be done with one cycle with n^3 instructions, one on each of n^3 processors. Then, to combine the results, sum in a tree:



where $D = AB$. This tree has height $O(\log n)$ (in fact, $\lceil \lg n \rceil$), and has that many necessary synchronization points. Assuming that each timestep provides us one synchronization point, we can then multiply two $n \times n$ matrices with n^3 processors in time $O(\log n)$.

For an algorithm in the PRAM model, we care about the total number of processors it uses, and the number of cycles the algorithm takes to complete. An algorithm is considered parallel if the number of cycles it takes is polylogarithmic in the size of its inputs.

1.3 Complexity class NC

A decision problem is in NC if there exists a parallel algorithm that runs in time $O(\log^c n)$ with $O(n^k)$ processors for some constants c and k. It is easy to see that $NC \subseteq P$ (since we can simulate an algorithm designed for PRAM on a RAM with polynomial increase in the runtime), but it is unknown whether $NC = P$.

PRAM model is unrealistic:

- Uniform memory access cost is too unrealstic an assumption for parallel machines.
- A globally synchronized clock for all processors is not true for most machines.
- Model is too hardware specific, we want to describe algorithms in a more abstract programming model so that it can be mapped on to different kinds of machines.

2 Programming Models

While PRAM model is widely used to describe algorithms, it is not necessary to tie the description of a parallel algorithm to a specific machine model. A more general and elegant way to think about a parallel algorithm is to see them a collection of instructions with some dependencies (precedence constraints) between them. In other words, we can think of them as Directed Acyclic Graphs.

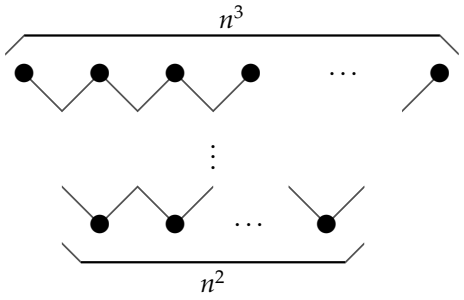
2.1 DAGs

We express computation as a directed acyclic graph with vertices (nodes) representing instructions and directed edges representing the precdence constraints: if there is an edge from v_1 to v_2, then v_1 has to be executed before v_2. The cost of each instruction (weight of vertex in the graph) depends on the machine on which the computation is executed.

Two common measures for estimating the complexity of the program (graph) are *Work* and *Depth*. If instructions (nodes) have weights w_i, then we define:

$$\text{Work} = \sum w_i$$
$$\text{Depth} = \max_{\text{path}} \sum_{w_x \in \text{path}} w_x$$

For example, the DAG for the matrix multiplication example above looks like:

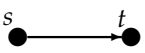
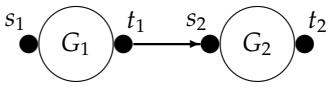
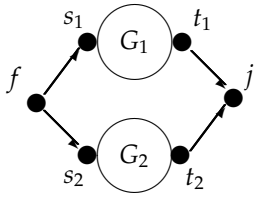


2.2 SP Digraphs

We refine our attention to a subset of DAGs: series-parallel digraphs. These are sufficiently general to encode interesting parallel problems, and lend themselves to easier analysis than general DAGs. Further, they model programming languages with the *fork-join* construct that allows functions to be evaluated in parallel:

```
fork
  f1 (args...)
  f2 (args...)
join
```

They are acyclic by construction; the rules for constructing a SP digraph are shown graphically below. They consist of the base case, series composition $G_1 \& G_2$, and parallel composition $G_1 \| G_2$. Note that SP digraphs are acyclic by construction.

Rule	Work $Wk(G)$	Depth $Dp(G)$
	$w_s + w_t$	$w_s + w_t$
$G_1 \& G_2$ 	$Wk(G_1) + Wk(G_2)$	$Dp(G_1) + Dp(G_2)$
$G_1 \| G_2$ 	$w_f + w_j +$ $Wk(G_1) + Wk(G_2)$	$w_f + w_j +$ $\max\{Dp(G_1) + Dp(G_2)\}$

The diagram above illustrates the three inductive rules for a SP digraph, where G_1 and G_2 are themselves SP digraphs with sources and sinks s_1 and t_1, respectively. The work and depth of the graph given work w_i for node i follow naturally.

Higher-level constructs like the "parallel for" pfor($i \to j$) can be described in terms of SP constructs; either we can either envision an extension of fork-join to $G_i, G_{i+1}, \ldots G_j$, or recursively build down to computation nodes with the two-way fork-join $G_1 \| G_2$.

2.3 Parallel QuickSort

We now look at a description of QuickSort and see how parts of it are already parallel. We will parallelize the other part with out much effort later on. 1.

Algorithm 1: QuickSort **QS**

Data: (A, n): A totally-ordered list A of length n
Result: A in sorted order

```
1 p ← pivot[A]
2 low(A) ← {x|x ∈ A, x ≤ p}
3 high(A) ← {x|x ∈ A, x > p}
4 return QS(low(A), length(low(A)))++QS(high(A), length(high(A)))
```

In Alg. 1, sorting the arrays $low(A)$ and $high(A)$ can be done in parallel. We can use the fork-join construct to evaluate these functions in parallel as in Alg. 2. For now, we will use sequential algorithms for the evaluation of $low(A)$ and $high(A)$ which have work and depth $O(n)$.

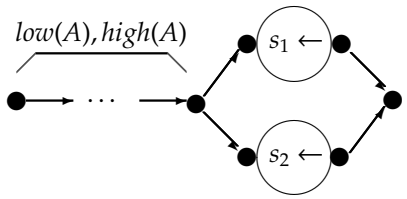
The work and depth of **QS** of Alg.2 are (assuming we some how a pivot that splits the array in half):

$$W(n) = O(n) + 2W(n/2) \quad \rightsquigarrow \quad W(n) \in O(n \log n)$$
$$D(n) = n + D(n/2) \quad \rightsquigarrow \quad D(n) \in O(n)$$

Algorithm 2: QuickSort **QS**

Data: (A, n): A totally-ordered list A of length n
Result: A in sorted order

```
1 p ← pivot[A]
2 low(A) ← {x|x ∈ A, x ≤ p}
3 high(A) ← {x|x ∈ A, x > p}
4 fork
5    S1 ← QS(low(A), length(low(A)))
6    S2 ← QS(high(A), length(high(A)))
7 join
8 return S1++S2
```



2.4 Scheduling a DAG/Executing a parallel program

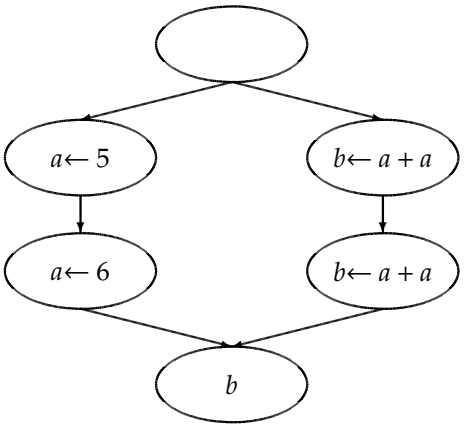
To execute a parallel program represented as a DAG on a parallel machine, we assigned the instructions to the machines processors in some order as long as the order obeys the precedence constraints. That is, if instructions v_1 and v_2 are such that there is a path from v_1 to v_2 in the DAG ($v_1 \prec v_2$), then v_2 can not be assigned to a processor before v_1 is completed.

2.5 Deterministic Parallelism

- Instructions A and B are concurrent if neither $A \prec B$ nor $B \prec A$.
- If A and B are concurrent and they access the same location, and one of the accesses is a write, then there is a race.
- A sequential ordering of instructions is a list of all instructions such that if $A \prec B$, then A occurs before B.
- If a DAG (program) is race free, then any sequenial order has the same output. To reason about the correctness of the program, look at any sequential ordering.

We refer to such parallelism (with out races) as deterministic parallelism. Writing prorgams with out "side effects" is a natural way to write such programs. Note that this does not preclude randomized algorithms: any node can still take a random number for it's input.

The diagram below shows an example of a program with races program that references variables a and b. Observe that different schedules of the instructions at left and right could produce different values for b.



v_1, v_2 concurrent

v_1	v_2
W	W
R	W
W	R

3 Scheduling a Graph

To execute a graph (DAG or SP digraph) on a machine, we consider how it might be executed on a PRAM. This allows us to determine the runtime of an algorithm on a particular machine model given the abstract cost model given by work and depth.

For some notation, denote by $V(G) = \langle s, t \rangle$ the source and sink of an SP digraph G, and $v_1 < v_2$ if computation v_1 must be scheduled before v_2. If $V(v_1) = \langle s_1, t_1 \rangle$ and $V(v_2) = \langle s_2, t_2 \rangle$, then $t_1 < t_2$. A greedy schedule is one that makes an instruction v ready to fire if all the predecessors of v are complete, and never leaves a processor idle if there is a ready-to-fire instruction. In other words, if we have an instruction that is ready to execute and at least one free processor, we will assign it to some processor.

3.1 Brent's Theorem

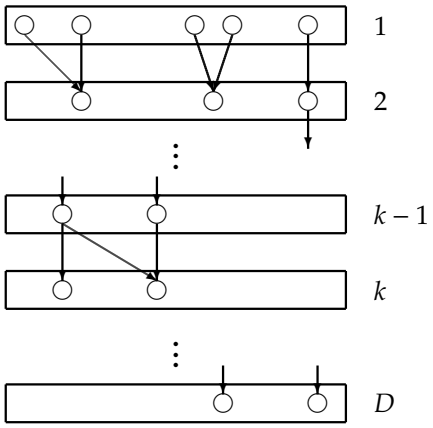
Brent's Theorem provides us with an upper bound for scheduling the execution of a DAG on a machine assuming the greedy schedule described above. Note that Brent's bound is not necessarily tight; it makes no assumptions about

Theorem 1. *A DAG G with depth D and work W, when scheduled on a machine with p processors, will take time at most $W/p + D$.*

Proof. As in Sec. 1.2, we assume that each instruction requires one unit of time. Consider partitioning the nodes (instructions) in to levelsL

1. Instructions at level 1 are ready to be executed immediately upon starting the program (have no incoming edges in the DAG)
2. Instructions at level k may have incoming edges from levels in $[1, k-1]$.
3. An instruction in level k must have at least one incoming edge from level $k-1$.

Rule 3 ensures that each instruction is in exactly one level. A depiction of this assignment is shown below. Note that we have D levels in total, by definition of depth; each instruction in the longest path from the start to end of the computation will have a different level number.



Now, consider any time step t (such that the program has not been completely executed by time t). Suppose that by time $t-1$, the schedule has finished all instructions in levels until k, but there are some instructions in level $k+1$. One of two cases may occur:

1. There are more than p ready-to-fire instructions at level k. In this case, we can keep all processors busy by assigning up to p units of work.
2. If not all p processors have work, we have no unassigned ready-to-execute nodes, since the schedule is a greedy one (otherwise, we would have assigned work to a processor). In this case, we finish off this level. Since there are D levels, we can spend at most D time in this case.

We can spend at most W/p time in case 1, and D time in case 2. An upper bound on the time for the entire computation is then the sum of these bounds: $W/p + D$.

Note further that $\max\{W/p, D\}$ is a lower bound on the run time of any schedule. □

4 Parallel Primitives

There are a number of useful "building blocks" that are useful for algorithms can be effectively parallelized. Recall that algorithms that have depth polylogarithmic in their input are typically of interest. We will consider a few of these parallel primitives.

4.1 Prefix-Sum

The prefix-sum of a list is a list where each element is the sum of all preceding elements in the input list. Formally, for an input list a and output list b = **prefix-sum**(a),

$$b[i] = \sum_{j=1}^{i-1} a[j]$$

a[**length**(a) − 1] is unused. For example,

$$\begin{array}{lllllll} a = & [6 & 4 & 3 & 10 & 8 & 7] \\ b = & [0 & 6 & 10 & 13 & 23 & 31] \end{array}$$

To perform prefix-sum in parallel, we take the sums of neighboring elements and apply prefix-sum recursively:

$$\begin{array}{lcccccc} a = & [6 & 4 & 3 & 10 & 8 & 7] \\ pairs = & [10 & & 13 & & 15 &] \\ s_2 = & [0 & & 10 & & 25 &] \\ b = & [0 & 6 & 10 & 13 & 23 & 31] \end{array}$$

The depth of prefix-sum is $D(n) = O(1) + D(n/2) = O(\log n)$, and work is $W(n) = O(n) + W(n/2) \rightsquigarrow Wk(n) = O(n)$.

We can use prefix-sum to compute the arrays $low(A)$ and $high(A)$ in the QuickSort algorithm in Alg. 2 in order to make it highly parallel. As an example, consider the parallelization of $low(A) \leftarrow \{x | x \in A, x \leq p\}$. Suppose we have:

$$\text{pivot} = p = 10 \qquad\qquad A = [4\ \ 13\ \ 8\ \ 9\ \ 12\ \ 6\ \ 7\ \ 19]$$

We compute l in parallel on n processors, which is defined by $g[i] = \mathcal{I}(A[i] \leq p)$, where $\mathcal{I}$ is the indicator function. Then, we find **prefix-sum**(g):

$$\begin{array}{rcccccccc} A = & [4 & 13 & 8 & 9 & 12 & 6 & 7 & 19] \\ g = & [1 & 0 & 1 & 1 & 0 & 1 & 1 & 0] \\ \textbf{prefix-sum}(g) = & [0 & 1 & 1 & 2 & 3 & 3 & 4 & 5] \end{array}$$

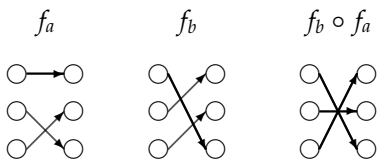
We observe that the check to see if an element in A with index i is less than the pivot can be easily parallelized. We then use **prefix-sum**$(g)[i]$ as the index to place $A[i]$ if it belongs in $low(A)$:

$$low(A) \leftarrow [4\ \ 8\ \ 9\ \ 6\ \ 7]$$

Now, we can compute $low(A)$ and $high(A)$ in $O(n)$ work and $O(\log n)$ depth. This parallel primitive (computation like $low(A)$) is often referred to as **pack** or **filter**. Though we do not prove it here, with a randomly-chosen pivot this leads to $D(n) \in O(\log^2 n)$ (with high praobability) and $E[W(n)] \in O(n \log n)$ for parallel quicksort.

4.2 Parallelism for Associative Operations

For prefix-sum, we used the associative property of addition. We can use the associativity of other operations to produce parallel algorithms for other computations. For example, suppose we have k functions (function composition is associative) and would like to compute their composition $f_k \circ f_{k-1} \circ \cdots \circ f_1(x)$. If we have a compact representation for $f_j \circ f_{j-1} \in F$ where $f_j \in F$ (for some family of functions F), we can compute the composition with the same technique as prefix-sum.

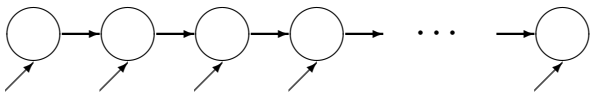


An application of the function composition computation is the problem of computing a sequence of state transitions. Consider the state transition diagrams shown in the left two diagrams above; they can be represented as functions f_a and f_b, with $f_b \circ f_a$ the result of transitioning through f_a then f_b. Yet another example arises with linear recurrences:

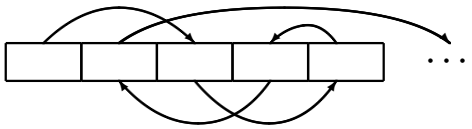
$$f_1 = a_1 x + b_1 \qquad f_2 = a_2 x + b_2 \qquad f_2 \circ f_1 = a_1 a_2 x + a_2 b_1 + b_2$$

4.3 List Ranking

In computing prefix-sum, we have a well-ordered array. That is, given an index, we can easily find the element in the array at that position. In a linked list, we have no such luxury. Suppose we have a (singly) linked list of elements and a handle to each element:



The list-ranking problem is to find the distance from a given element to the end of the list. The problem seems simple at first; indeed, a trivial $O(n)$ sequential algorithm would simply walk the list until it reaches the end. However, in memory, our linked list might look like



so we cannot, for example, determine easily if we are at an even or odd index in the list given only the handle to an element.

Wyllie's algorithm is a way to compute the list ranking in $O(n \log n)$ work and $O(\log n)$ depth. This is non-optimal, since the sequential algorithm only does $O(n)$ work, but the logarithmic depth is appealing. In addition to the linked list pointers, we maintain another value ℓ at each node, which records the number of hops in the original linked list that its current link traverses. For example, suppose we had the original list shown at left below. Then, for the purposes of list ranking, the list at right stores an equivalent amount of information.

Maintaining this invariant is the intuition that underlies Wyllie's algorithm. At each step, for each node the algorithm coalesces the next two pointers (successors) into a single pointer, with an ℓ value that is the sum of the pointers. This pointer manipulation can be done in parallel over the node list: Alg. 3 implements the technique.

In Alg. 3, $J\ell$ and J**succ** are temporary so that we can maintain the original state across the synchronization between the parallel portions. If these two parallel executions were not separated, we would not successfully isolate the parallel operations.

Algorithm 3: Wyllie's Algorithm

Data: A list of nodes with handles $i \in [1, N]$ with successor pointers **succ**$[i]$ and number of hops represented by the pointers $\ell[i]$.

Result: A modified structure, with $\ell[i]$ the distance from element i to the end of the list, and **succ**$[i] \forall i$ pointing to the last element

```
1 for j ← 1 ... ⌈lg n⌉ do
2     pfor i ← 1 ... n do
3         Jℓ[i] ← ℓ[succ[i]]
4         Jsucc[i] ← succ[succ[i]]
5     pfor i ← 1 ... n do
6         ℓ[i] ← Jℓ[i]
7         succ[i] ← Jsucc[i]
```