Massachusetts Institute of Technology
6.892: Theory of Parallel Systems
Professor Charles E. Leiserson

Lecture 1
February 4, 1997
Scribe: Michael A. Bender

Lecture 1: Data-Parallel Computing

1 Data-Parallel Computing

In data-parallel computing, the parallelism lies in the data, not in the control. Data-parallel computing relies on the *vector model* of parallelism. This model assumes multiple data and serial control. Figure 1 depicts the vector RAM (VRAM) architecture. Notice that the program resides in the scalar memory and that there is one program and one program counter. This model is also called the SIMD model (Single Instruction Stream, Multiple Data Stream), a term that was coined by Flynn in the late sixties.

A competing model is called the MIMD Model (Multiple Instruction Stream, Multiple Data Stream. Programs that assume this model are written in the SPMD style (Single Program, Multiple Data Stream.) We consider the MIMD model in a few lectures.

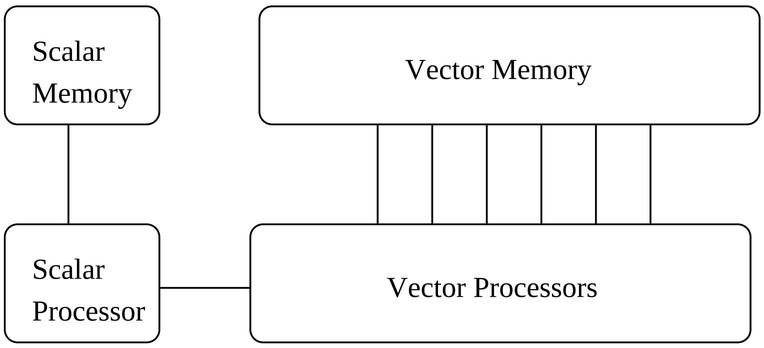

Figure 1: Diagram of a vector architecture. The program resides in scalar memory.

1.1 Vector Operations

In this section we present the major classes of VRAM instructions. We consider vectors A and B, where

$$A = [1\ 2\ 3]\,, \quad B = [4\ 5\ 6]\,.$$

Elementwise operations. Elementwise operations execute one operation on all vector elements.

Example. $A + B \longrightarrow [5\ 7\ 9]$.

Example. $A * B \longrightarrow [4\ 10\ 18]$.

Example. $\text{select}(A,\ B,\ [1\ 0\ 1]) \longrightarrow [1\ 5\ 3]$.

Reduction Operations. Reduction operations take vectors as arguments and output scalars.

Example. $\text{size}(A) \longrightarrow 3.$

Example. $\text{sum}(A) \longrightarrow 6.$

Distribute Operations. Distribute operations take scalars as arguments and return vectors.

Example. $\text{dist}(3,5) \longrightarrow [5\ 5\ 5].$

Routing Operations. Routing operations move data among processors.

Example. $B[2\ 1\ 3] \longrightarrow [5\ 4\ 6].$

Example. $B[2\ 1\ 2] \longrightarrow [5\ 4\ 5].$

Notice that executing $B[2\ 1\ 2]$ is a qualitatively harder operation than executing $B[2\ 1\ 3]$, because in $B[2\ 1\ 2]$ communication is unbalanced. More specifically, $B[2\ 1\ 2]$ requires a *concurrent read*. That is, several processors simultaneously read the value of the second element of vector B. The implementation of concurrent reads in hardware affects performance substantially.

Example. $C[2\ 3\ 1] = B \implies C \longrightarrow [6\ 4\ 5].$

Example. $C[2\ 1\ 2] = B \implies C \longrightarrow ??$

The operation $C[2\ 3\ 1] = B$ involves a write; the operation $C[2\ 1\ 2] = B$ involves a *concurrent write*. In a concurrent write, multiple processors simultaneously attempt to write to the same memory cell.

We can define the memory value after a concurrent write in several ways. If concurrent writes are *illegal*, then on a concurrent write, the computer overheats, catches fire, and blows up. On an *arbitrary* concurrent write, an arbitrary processor succeeds in writing its value. On a *priority* concurrent write, the processor with the highest priority succeeds in writing its value. On a *reduction* concurrent write, all values written to any memory location are *combined* together – e.g., added, anded, or multiplied.

Scan Operations. Scan operations are also called *parallel prefix* operations.

Example. $+\text{-scan}(A) \longrightarrow [1\ 3\ 6].$

Example. $*\text{-scan}(B) \longrightarrow [4\ 20\ 120].$

Multidimensional Operations. Operations on multidimensional arrays are multidimensional operations. Assume

$$C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}.$$

Example. row-sum(C) $\longrightarrow$ [3 7 11].

Ordinary Serial Control. This includes operations such as if, while, procedure calls, etc.

2 Algorithmic Measures

In this section we explore several measures that gauge the efficiency of parallel algorithms.

The *work* of a parallel program is the total number of operations performed collectively by the processors. Alternatively, it is the total time necessary to execute the parallel program on *one* processor. The work is approximately the sizes of the vectors manipulated summed over all computation steps.

The *critical path length* of a computation is the total number of vector operations performed. Alternatively, it is the time necessary to execute the parallel program on an *infinite* number of processors.

A third measure is the *computation time on P processors*. The following extremely important theorem, called Brent's Theorem, relates these three measures.

Theorem 1 (Brent, Graham) *Consider an algorithm having work W and critical path length C. On P processors, one can achieve execution time on P processors T_P, where*

$$T_P \leq \frac{W}{P} + C.$$

Proof. Let S be the size of the vectors on the ith step. That is,

$$W = \sum_{i=1}^{\ell} S_i.$$

We schedule the processors using a greedy strategy. That is, we divide the vector data evenly among the processors. Let t_i be the time to execute the ith step. Then

$$t_i = \left\lceil \frac{S_i}{P} \right\rceil.$$

Then the total time of the computation T_P is

$$T_P = \sum_{i=1}^{C} \left\lceil \frac{S_i}{P} \right\rceil \leq \sum_{i=1}^{C} \left(\frac{S_i}{P} + 1 \right) = \frac{W}{P} + C.$$

$\square$

Falsehoods in Simplification. Several falsehoods derive from the VRAM abstractions. Some of the operations listed in the previous sections are more work-consuming than others. Many of the operations, such as routing operations, have "hidden logs." That is, routing requires a time logarithmic in the number of processors.

2.1 Radix Sort

Now we present a radix sort algorithm that is written for the data-parallel model.

A *stable sort* has the property that if two elements are equal, the sorting algorithm does not change the order of these elements. A stable sort on least significant to most significant bits sorts keys. Figure 2 shows a sample execution of a radix sort.

0 1 0 1		0 1 1 0		1 0 0 0		0 0 0 1
0 1 1 0		1 0 0 0		1 1 0 0		0 1 1 0
0 0 0 1	$\longrightarrow$	1 1 0 0	$\longrightarrow$	0 0 0 1	$\longrightarrow$	1 0 0 0
1 0 0 0		0 1 0 1		0 1 1 0		1 1 0 0
1 1 0 $\underline{0}$		0 0 $\underline{0}$ 1		1 $\underline{0}$ 1 1		$\underline{1}$ 0 1 1

Figure 2: A sample execution of a radix sort.

Figure 3 shows how to implement a radix sort on a vector machine. Note the clever use of scans in the implementation.

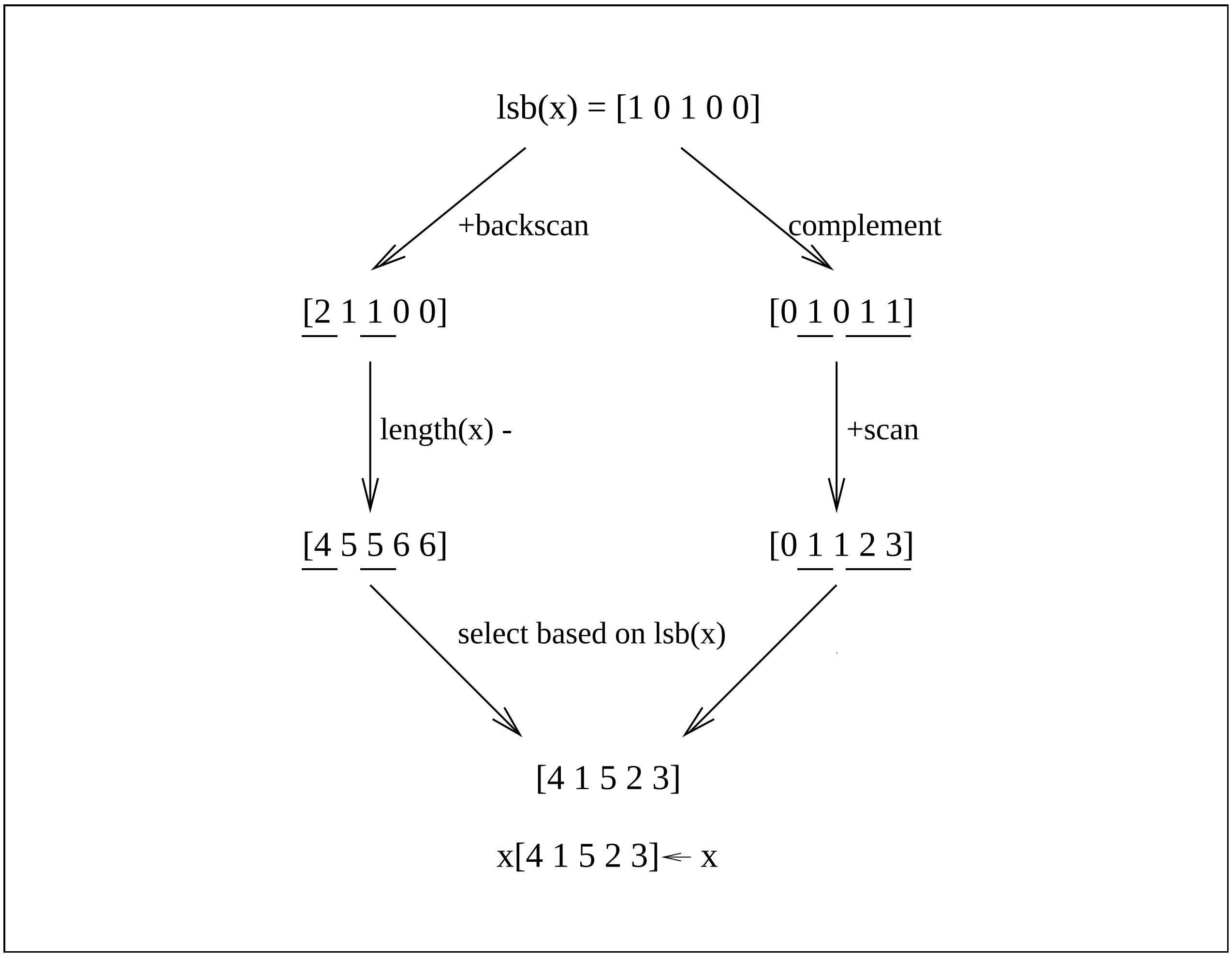

Figure 3: Radix sort for the vector model.