SPARSE-MATRIX MULTIPLICATION
A SIMPLE AND EFFICIENT PARALLELED ALGORITHM

by

A. Israeli and Y. Shiloach

Technical Report #339
October 1984

* Elbit Computers Ltd., P.O.Box 5390, Haifa 31053, Israel
ABSTRACT

A parallel algorithm for multiplying sparse matrices is presented. It uses the PRAM (shared memory) model of parallel computation.

Nothing is assumed on the particular distribution of nonzero elements in the matrices. Yet the algorithm is simple and efficient.

Let \( N \) be the number of products of a nonzero element in one matrix with a nonzero element in the other, carried out by the regular matrix multiplication algorithm. Our algorithm runs on an \( N-\text{PRAM} \) (PRAM with \( N \) processors), in \( O(\log N) \) time.

A sequential (one processor) variation of this algorithm runs in \( O(N + |A|) \) time, where \(|A|\) is the number of nonzero elements in \( A \).
1.0 INTRODUCTION

Let $A$ and $B$ be two sparse matrices of sizes $m \times n$ and $n \times p$ respectively. We wish to compute the $m \times p$ product matrix $D = AB$. Let $N$ be the number of nontrivial products that are done by the straightforward multiplication algorithm. If a completely random distribution of nonzero elements in $A$ and $B$ is allowed and $N \ll mnp$, then $N$ seems as a lower bound to any multiplication algorithm.

In this sense, our algorithm is nearly optimal. Its sequential version runs in $O(N + |A|)$ time. This running time is the same as that given in G-78, (which is the best we know of).

Unlike the algorithm in G-78, our algorithm has an immediate parallel implementation which runs in depth (=parallel time) of $O(\log n)$ on a PRAM with $N$ processors. Note that even in the extreme case of completely full matrices, the algorithm performance is as good as the existing algorithms in both sequential and parallel versions.

The PRAM model of parallel computation consists of a set of processors that share a common memory. Each of them can access each memory cell. Variants of this model differ in the capability of performing concurrent READs and/or WRITEs from/to the same memory cell. (See SV-81 for a detailed description of this model and its variants). For the algorithm below, a CREW (Concurrent READs Exclusive WRITEs) PRAM is sufficient. The problem of assigning
processors to jobs is also treated carefully. The same algorithm can also be implemented on the Shuffle-Exchange 'Ultracomputer', (see S-80) which is a much weaker model of parallel computation, in time of \((\log N)^2\), (see IS-83).

The next section opens with a brief outline of the algorithm and concludes with its sequential implementation. The last section contains the parallel implementation.
2.0 **THE ALGORITHM AND ITS SEQUENTIAL IMPLEMENTATION**

The algorithm is based on the following simple observation.

**THEOREM:**

Let $A$ and $B$ be two matrices of sizes $m \times n$ and $n \times p$ respectively, and let $C_1, \ldots, C_n$ $(R_1, \ldots, R_n)$ be the columns of $A$ (the rows of $B$). Let $c_j (r_j)$ denote the number of nonzero elements in column $C_j$ $(row R_j)$ for $j = 1, \ldots, n$. If the product $AB = D$ is computed in the usual way, then the total number of nontrivial multiplications of a nonzero element of $A$ with a nonzero element of $B$ is $c_1r_1 + \ldots + c_nr_n$.

**PROOF:**

One can easily verify that each nonzero element of $C_j$ in $A$ is multiplied with each nonzero element of $R_j$ in $B$ exactly once, (see Figure 1). Q.E.D.

The theorem above suggests the following matrix multiplication scheme:

For any fixed $j$, $(1 \leq j \leq n)$, Generate all the products of a nonzero element of $C_j$ with a nonzero element of $R_j$. If $a_{ij}$ and $b_{jk}$ are such a pair of nonzero elements then assign an 'address' $(i,k)$ to their product which means that it should contribute its own value
to the total value of \((D)_{ik}\).

finally sum up all the products that are addressed to the same entry of \(D\).

The input matrices \(A\) and \(B\) are supposed to be sparse and therefore are assumed to be held in a column-wise or row-wise format. The column-wise (row-wise) format is essentially a vector of triples \((i,j,a_{ij})\) \((j,k,b_{jk})\) sorted lexicographically with respect to \(j\) first. (see Figure 2). These triple-vectors are denoted as \(A_{col}\) and \(B_{row}\). The first two entries in each triple denote the location of the (nonzero) element and the third denotes its value. If \(A\) is given in a row-wise format, it can be converted to a column-wise format in \(O(|A|)\) time in a simple way, (see G-78). Here \(|A|\) is the number of nonzero elements in \(A\).

2.1 SEQUENTIAL IMPLEMENTATION

The simplest way to implement this algorithm by one processor takes \(O(N \log N)\) time and \(O(N)\) space. Each product of two triples \((i,j,a_{ij})\) and \((j,k,b_{jk})\) yields a new triple \((i,k,a_{ij}b_{jk})\). These triples are stored in a list which is eventually sorted lexicographically with respect to \(i\) first (or with respect to \(k\) first if a column-wise format of \(D\) is pursued). Then the values of all the triples that contain the same \(i\) and \(k\) are summed up to yield
Allowing more space \(O(mp)\), the time can be reduced to \(O(N+|A|)\) by a way similar to that proposed in G-78.
3.0 PARALLEL IMPLEMENTATIONS

Two implementations will be presented. The first has depth of $O((\log N)^2)$ but uses just $O(N)$ space. The second has depth of $O(\log N)$ but requires $O(mnp)$ space. In both ways, all the $N$ non-trivial products are generated in parallel in constant time. They differ in the way in which the results are stored, gathered and summed.

Let us describe the first implementation and then show how the second achieves a better time by exploiting more space.

3.1 THE FIRST IMPLEMENTATION

Assume that we have an easily computable 1-1 mapping of processors onto pairs $(a_{ij}, b_{jk})$ of nontrivial multiplicands, (such a mapping is given in Sec. 3.3). Then the $t$-th processor, $P_t$, accesses the $t$-th pair and forms a triple of the form $(i, k, a_{ij}b_{jk})$. It stores this triple in the $t$-th entry of a vector of triples of length $N$. This vector is then sorted lexicographically.
graphically with respect to i and k. Then all the triples with the same i and k are summed into a single triple \((i, k, \sum_j a_{ij}b_{jk})\). The resulting vector of triples is the row-wise representation of \(D\).

The bottleneck of this algorithm is the sorting that takes \(O(\log N)\).

Sorting and summing have well known PRAM algorithms and therefore their details are not discussed.

A second implementation is given below that uses a kind of bucket sorting (and thus more space) and saves a logarithmic factor.

3.2 THE SECOND IMPLEMENTATION

This implementation differs from the first in the way in which the product triples \((i, k, a_{ij}b_{jk})\) are stored and gathered.

A vector \(T\) of length \(mpn\) is allocated for this purpose. \(T\) can be viewed as a concatenation of \(mp\) segments of length \(n\), each corresponding to one entry of \(D\). A triple \((i, k, a_{ij}b_{jk})\) is stored in location \(n(p(i-1)+k-1)+j\) of \(T\). This single valued mapping maps all the triples with the same \(i\) and \(k\) into a segment of length \(n\) in \(T\). (see Figure 3). The values in all the triples that fall into such a segment have to be summed up to yield \((D)_{ik}\). The job of summing the values of triples that fall in a given segment is
done by the processors that have generated and stored these triples. Thus we are faced with problem of summing the elements of a sparse vector with as many processors as non-NIL elements, where each processor knows the address of 'its' element, (very important). To do it, we use a binary tree whose leaves correspond to the n elements of the vector. The values to be summed 'climb' the tree one level at a time. Thus, there are $\log n$ steps in this algorithm. At Step $i$, data is moved from Level $i$ to Level $i+1$ in the following way. Left sons of level $i$ that contain nonzero data move it to their fathers. Then right sons at this level that have nonzero data, add it to the values currently stored at their fathers. The responsibility of moving nonzero data from a node to its father lies on the processor that was the last to increment the value at this node. Zeros are not moved upwards. Figure 4 shows how a vector of length 13 that has 5 non-NIL elements is summed up. Similar ideas and structures can be found in the fifth section of SV-82 where they are called 'partial sums trees' and are used for other purposes as well.

3.3 THE ASSIGNMENT PROBLEM

The only problem left, is that of assigning each processor to a different nontrivial pair in logarithmic time. This is done in
the following algorithm. The input to this algorithm is the i.d. number, say $t$, of a processor. It returns the locations in $A_{col}$ and $B_{row}$ from which $P_t$ should retrieve its nontrivial pair. The algorithm is executed in parallel by all the processors. Its correctness is straightforward.

**THE ASSIGNMENT ALGORITHM**

Let $c_j$ ($r_j$) denote the number of nonzero elements in column $C_j$ (row $R_j$) for $j=1,\ldots,n$.

1. Generate the $n$-vectors $C=(c_1,\ldots,c_n)$, $R=(r_1,\ldots,r_n)$ and $CR=(c_1r_1,\ldots,c_nr_n)$.

2. Compute the $n$-vectors $PC$, $PR$ and $PCR$ that are the vectors of partial sums of $C,R$ and $CR$ respectively. E.g.
   \[ PC=(c_1,c_1+c_2,\ldots,c_1+\ldots+c_n). \]

3. Find the (unique) number $j^*$ satisfying:
   \[ PCR(j^*-1)<t\leq PCR(j^*), \quad (PCR(0)=0). \]
   */ At this point we know that $P_t$ is assigned to a nontrivial pair of the form $(a_{i^*j^*},b_{j^*k^*})$. In the following steps, the values of $i$ and $k$ are determined. /*

4. Let $x_t=t-PCR(j^*-1)$. Find $i'$ and $k'$, $1\leq i'\leq c_{j^*}$ and $1\leq k'\leq r_{j^*}$, such that $x_t=(i'-1)r_{j^*}+k'$.
E.g. if \( X_t = 2 \), \( c_j = 3 \) and \( r_j = 5 \) then \( i' = 2 \) and \( k' = 4 \). Note that \( 1 \leq x_t \leq c_j \times r_j \) and this is a standard 1-1 mapping of a set of size \( c_j r_j \) onto the set-product of two sets of sizes \( c_j \) and \( r_j \).

5. The nontrivial pair \((a_{i_j k}, b_{j k})\) associated with \( P_t \) is given by:

\[
a_{i_j k} = Acol(PC(j-1)+i') \quad \text{and} \quad b_{j k} = Brow(PR(j-1)+k').
\]

**EXAMPLE**

Let \( C = (5, 0, 0, 2, 3, 4, 8) \) and \( R = (2, 0, 1, 4, 3, 0, 1) \) and \( t = 25 \).

Then:

\[
CR = (10, 0, 0, 0, 9, 0, 8)
\]

\[
PC = (5, 5, 7, 10, 14, 18) \quad \text{and} \quad PR = (2, 2, 3, 7, 10, 11)
\]

\[
PCR = (10, 10, 10, 18, 27, 27, 35)
\]

Since \( t = 25 \), \( j = 5 \) and \( x_t = 25-10 = 7 \).

Since \( r_j = 3 \) we express \( x_t \) in the form: \( x_t = 2 \times 3 + 1 \) and thus \( j' = 3 \) and \( k' = 1 \). Hence

\[
a_{i_j k} = Acol(PC(5-1)+3) = Acol(10) \quad \text{and} \quad b_{j k} = Brow(PR(5-1)+1) = Brow(8).
\]

**DETAILED IMPLEMENTATION OF THE ASSIGNMENT ALGORITHM.**

The generation of the vectors of partial sums, \( PC \), \( PR \) and \( PCR \) from \( C \), \( R \), and \( CR \) respectively, (Step 2) can be done in \( \log n \) time even in models that are much weaker than the PRAM, such as the Schwartz' 'Ultracomputer', (see S-80).

Step 3 is done by each processor by a simple binary search, and Steps 4 and 5 are constant time arithmetic operations.
It turns out that the generation of C and R from Acol and Brow respectively, (Step I), is what needs further elaboration.

Let V be a vector whose elements may contain 'NIL'-s that represent empty or irrelevant elements. The operation of 'packing' V, denoted as PACK V, results in a new vector containing only the non-NIL elements of V in the same order as they were before.

**Packing a vector V** of length n can be easily accomplished by n processors in log n time even in the much weaker model of the shuffle-exchange ultracomputer, (see S-80). In a PRAM it can be done by generating a vector U such that $U_i=0$ if $V_i=\text{NIL}$ and $U_i=1$ otherwise. If we compute the vector $PU$ of partial sums of U, then each entry in $PU$ that correspond to a non-NIL element of V, is equal to its location in the vector $W = \text{PACK } V$.

**Generation of C from Acol**

Let V denote the vector that consists of the first entry (the column number) of each triple of Acol. The following algorithm computes C from V. The length of V is denoted by q.

1. Generate the vector $U=1,2,\ldots,q$.

2. For $i=1,\ldots,q-1$ do in parallel:
   
   IF $V_i=V_{i+1}$ THEN $V_i=\text{NIL}$ and $U_i=\text{NIL}$.
3. \( V' \) PACK \( V \)

4. \( U' \) PACK \( U \)

5. \( C_j = 0 \) (for \( j = 1, \ldots, n \))

6. For \( j = 1, \ldots, q' = \text{length of } V' \), do in parallel:
   \[ C(V'_j) + U'_j - U'_{j-1}, \quad (U'_0 = 0). \]

Example: Let \( V = (1, 1, 2, 2, 2, 5, 6, 9, 9, 9, 9) \) and \( n = 11 \).
Then after Step 2: \( V = (\text{NIL,1,NIL,NIL,NIL,2,5,NIL,6,NIL,NIL,NIL,9}) \)
and \( U = (\text{NIL,2,NIL,NIL,NIL,6,7,NIL,9,NIL,NIL,NIL,13}) \).
Thus \( V' = (1, 2, 5, 6, 9) \), \( U' = (2, 6, 7, 9, 13) \) and \( q' = 5 \).
The following assignments are done in parallel in Step 6:
\[ C_1 = 2; \quad C_2 = 4; \quad C_3 = 1; \quad C_4 = 2; \quad C_5 = 4. \]
So finally: \( C = (2, 4, 0, 0, 1, 2, 0, 0, 4, 0, 0) \).

REFERENCES


ACKNOWLEDGEMENT

The authors wish to thank Dr. M. Rodeh for the observation expressed in the theorem.


The pair contributes to \( (D)_{i,k} \).

**Figure 1**

Column-wise representation of \( A \):

- **Column 1**
  - ... (nonzero elements)

- **Column 2**
  - ... (nonzero elements)

- **Column \( j \)**
  - \( i, j \) if \( \alpha_{ij} \neq 0 \)

- **Column \( n \)**
  - ...

**Figure 2**