TECHNIQUES FOR PARALLEL MANIPULATION OF SPARSE MATRICES (Extended Abstract)

Clyde P. Kruskal
Computer Science Department
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

Larry Rudolph
Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

Marc Snir
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

ABSTRACT
We present new techniques for the manipulation of sparse matrices on parallel MIMD computers. This yields efficient algorithms for the following problems: matrix addition, matrix multiplication, row and column permutation, matrix transpose, matrix vector multiplication, and Gaussian elimination.

1. INTRODUCTION
Many practical problems have computational solutions that involve solving large systems of linear equations as well as other types of manipulations of large matrices. In many cases these matrices are sparse, i.e. the number of nonzero entries is small. Consequently, it is desirable to use data representations and algorithms that avoid wasting space and time for zero entries. Sparse matrix algorithms for sequential machines have been extensively studied.

We will consider parallel algorithms that perform the same arithmetic operations that would be performed by a sequential algorithm. This does not preclude the achievement of significant speedups. Simple matrix algorithms contain many independent arithmetic operations that can be executed in parallel. The issue is the overhead required to allocate the arithmetic operations evenly to the processors of a parallel computer and to compute the locations of the operands to these operations. We desire overhead per processor to be proportional to the amount of arithmetic processing done, so that the total execution time is equal, up to a (small) multiplication constant, to the time required to perform the arithmetic operations, using all available processors. Such algorithms will exhibit optimal speedup.
The algorithms use a compact representation of matrices, where only the nonzero entries are stored. In order to avoid sequential bottlenecks, the entries are not necessarily stored in row or column order, rather as a set of tuples. We show how radix sort and prefix operations can be used to organize such a bag of entries according to the needs of the computation and to distribute work evenly to all processors.

It is fairly easy to obtain a matrix addition algorithm that executes with optimal speedup. The difficulty is to use space proportional only to the sizes of the input and output matrices. Our algorithm achieves this.

For Gaussian elimination we introduce a new technique of "lazy evaluation". A redundant matrix representation is used, which may have several entries with the same row and column indices; the "true" value of the matrix entry is the sum of all these entries. Entries are summed only when the true value is needed, or when there is a sufficient backlog of work to justify a scan and compaction of the entire data structure.

2. FOUNDATIONS

We assume the EREW PRAM computation model with \( p \) autonomous processors, all having access to shared memory: At each step each processor performs one operation from its instruction stream. An instruction may involve access to shared memory. Concurrent access to the same memory location by several processors is forbidden.

Let \( m \) denote the size of the problem, that is, the number of nonzero entries in all the matrices involved, or, equivalently, the number of nonzero values in the input and in the output. Let \( T(m) \) be the sequential time to solve a problem, and \( T_p(m) \) be the parallel time with \( p \) processors. The speedup of a parallel algorithm is \( T(m) / T_p(m) \) and its efficiency is \( T(m) / p T_p(m) \). An algorithm is efficient if its efficiency is bounded away from zero, so that it achieves a speedup proportional to \( p \), for large enough \( m \). We also wish to use an amount of memory that does not exceed the size of the input or the output by more than a constant factor.

Matrices are assumed to be sparse (most entries are zero), but not too sparse. Let \( d \) denote the sum of the lengths of sides of the matrices. Then, formally, we assume \( m = \Omega(d^\varepsilon) \) for some positive \( \varepsilon \). In most cases of interest the number of nonzero entries will be at least linear in the dimensions of the matrices, so this assumption is quite reasonable. We also assume that the problem size is somewhat larger than the number of processors used, or, more formally, that \( m = \Omega(p^{1+\delta}) \) for some positive constant \( \delta \). Both assumptions are likely to be fulfilled by problems that are large enough to justify the use of parallel computers.

The canonical representation of a sparse matrix stores the matrix as a set of quadruples, one quadruple for each nonzero element. The four components of each quadruple are matrix name, row index, column index, and value. The quadruples in the list can be stored in arbitrary order. The name of the matrix need not be explicitly stored at each element; algorithms that operate on two matrices (e.g. matrix addition or matrix multiplication algorithms) will need an extra bit per element to differentiate the matrices. A row major representation of a matrix stores the nonzero elements in row major order. The matrix is represented by a list that contains for each row the number of elements in this row, followed by an ordered list of the nonzero elements in this row, represented by column index and value. This represen-
Parallel Manipulation of Sparse Matrices

Parallel manipulation is more space efficient whenever the matrix is not too sparse, i.e., whenever there is at least one entry per row, on the average. The column major representation of a matrix is defined similarly.

3. BASIC PARALLEL ROUTINES

The sparse matrix algorithms are easy to explain and understand given several basic routines. They all require time $O(n/p + \log p)$ when using $p$ processors.

**Parallel Prefix:** Given $n$ numbers $x_1, \ldots, x_n$ stored in consecutive locations in an array and an associative operation $*$, the parallel prefix problem is to compute the products $x_1 * \cdots * x_n$, for $i=1, \ldots, n$. This can be accomplished in time $O(n/p + \log p)$ and space $n + O(p)$ (see [1] or [2]). Parallel prefix can be computed within the same time bounds when the items are stored on a linked list [3,4].

**Summing by Groups:** Assume that $n$ items are to be summed and are divided into groups; items that belong to the same group are contiguous and the first item in each group is marked. The summing by groups problem is to compute the sum within each group. This is handled as a parallel prefix computation, by defining a suitable sum operation that does not carry across set boundaries [2]. The computation yields the initial sums within each subgroup.

**Broadcasting and Packing:** The broadcast problem is to create $n$ copies of an item. This again is parallel prefix, with the product defined as $a*b = a$. Broadcast by groups is executed in a similar manner. Given an array of length $n$, the packing problem is to move the nonempty entries in the array to the head of the array, while preserving their order. Packing can be done by ranking the nonzero entries, and then moving each to the location indicated by its rank. Ranking can be done by assigning nonempty elements the value one and empty elements the value zero, and then applying parallel prefix using addition.

**Sorting and Merging:** Radix sort can be used to sort $n$ integers in the range 1 to $R$ in time $T_p(n) = O((n/p)(\log R / \log(n/p)))$ and space $O(pn^\varepsilon + n)$ for any constant $\varepsilon > 0$ [5]. The time is $O(n/p)$ whenever the number of distinct key values $R$ is at most polynomially larger than $n/p$. Two sorted lists of sizes $m \leq n$ can be merged in time $O((m+n)/p + \log n)$ [6].

### 3.1. Cross Product

Consider the problem of forming the cross product (i.e. Cartesian product) of two sets $S$ and $T$. The result is to be placed in two arrays $S$ and $T$, where $S[i] \times T[j]$ is the $i$th element of the cross product.

First determine the cardinalities $s$ and $t$ of the two sets. We need $t$ copies of each element of $S$, and $s$ copies of each element of $T$. Consider $S$ and $T$ as being two dimensional arrays of size $s \times t$, stored in row major order. Row $i$ of $S$ will consist of $t$ copies of element $i$ in $S$, and, similarly, column $i$ of $T$ will consist of $s$ copies of element $i$ in $T$.

The following operations will produce $S$: Place element $i$ of $S$ into location $(i,t)$ of array $S$. Broadcast each element to the next $t-1$ locations of $S$. To produce $T$, replace $i+i$ by $j+i$.

A cross product computation makes use of parallel prefix on arrays of size $s \times t$ or smaller; it can be done in time $O(s \cdot t + \log s)$ and space $O(s \cdot t + p)$.
Cross products can also be done by groups: Assume that the sets \( S_k \) are packed to the top of array \( S \) and the sets \( T_k \) are packed to the top of array \( T \). The cross products \( S_k \times T_k \) are to be placed in the arrays \( S \) and \( T \).

1. Place the cardinality of each \( S_k \) into \( a_k \) of a temporary array \( a \) and the cardinality of each \( T_k \) into \( a_k' \) of a temporary array \( a' \).
2. Form the products \( a_k \times a_k' \) to determine the cardinalities of the cross product sets.
3. Perform a parallel prefix with addition on these products to determine the starting location of each cross product in the final answer.
4. Copy the starting location of the \( k \)th cross product along with the cardinality of \( T_k \) to the first element in each set \( S_k \).
5. Broadcast the starting location the \( k \)th cross product along with the cardinalities of \( S_k \) and \( T_k \) to all elements within each set \( S_k \), using broadcast by groups.
6. Determine the rank of each element within each set (parallel prefix by groups).
7. Set all locations of \( S \) to empty.
8. Place each element \( i \) of each set \( S_k \) into location \( i \cdot | T | + \text{the starting location of the } k \text{th cross product of array } S \).
9. Broadcast each element in each set \( S_k \) to the next \( | T_k | - 1 \) locations of \( S \), using broadcast by groups.

**Analyses:** Recall that we make two assumptions about the problem size: (1) \( m = \Omega(d^e) \) for some positive \( e \); and (2) \( m = \Omega(p^{1+e}) \) for some positive constant \( e \).
Thus, parallel prefix on the \( m \) elements of matrix takes time \( O(m/p) \). Also, for \( n \times n \) matrices, the integer range for sorting is \( R = n^2 \). Using radix sort, the time to sort is \( O(n^e/p) \). The space is \( O(pm^{e} + m) \) for any constant \( e > 0 \). By assumption (2), there is a \( e \) such that the space is \( O(m) \) (namely any \( e < \delta \)). More generally, for 2-dimensional matrices that are not necessarily square, the length of each side of a matrix is bounded by \( d \). The integer range for the sorting will be at most \( d^2 \), so the radix sort time and space remain the same.

### 3.2. Format Conversion, Matrix Transpose, and Matrix Addition

It is easy to see that a canonical representation of a \( q \times r \) matrix can be computed from its full matrical representation in time \( O(qr/p) \): Each processor is allocated \( qr/p \) entries of the matrix; it creates a list of quadruples representing the nonempty entries in its set. These lists are then packed in time \( O(qr/p) \). Conversely, a canonical representation can be converted into a full matrical form in time \( O(m/p) \) if the matrix is already initialized to zero, and time \( O(qr/p) \) otherwise. A sparse matrix in canonical representation is transposed by inverting in each quadruple the row and column indices, in time \( O(m/p) \) and space \( O(m) \).

Suppose we are given two sparse matrices \( A \) and \( B \), and wish to form their sum \( C \). We need to pair together elements of \( A \) and \( B \) with the same indices (i.e. in the same row and column) and then replace each such pair by the sum of their values. The pairing of values can easily be done by collecting the \( A \) and \( B \) quadruples together, and sorting using an element's index as the key. The total time for the algorithm is \( O((m/p)(\log^d)/\log(m/p)) \) and the space is \( O(pm^{e} + m) \). Under our assumptions, these simplify to time \( O(m/p) \) and space \( O(m) \). If the two matrices are already sorted the addition can be done using merging rather than sort-
4. MATRIX MULTIPLICATION

We now show how to form the matrix product **C** of two sparse matrices **A** and **B**. Every nonzero element of column **k** in **A** must be multiplied by every nonzero element of row **k** in **B**. This means that elements of column **k** in **A** need to be grouped together with elements of row **k** in **B**. Each such group is composed of a set of elements from **A** and a set of elements from **B**. Within each group, form the cross product of the two sets and multiply the **A** and **B** values forming each pair. The product of the pair **a** and **b** contributes to **c**. Thus, the pairs need to be sorted by the key (i, j), where **i** is the row of the **A** element and **j** is the column of the **B** element. All products with the same key (i, j) are summed to form the **c** , which are packed to the top of the array.

The algorithm works as follows:

1. Place the two matrices **A** and **B** together into one set of quadruples (matrix name, row index, column index, value).
2. Sort the quadruples using, for **A**, the column index and using, for **B**, the row index.
3. Within each group of elements with the same index from step (2), form the cross product of the elements from set **A** with the elements from set **B**.
4. Multiply the **A** value with the **B** value in each cross product pair, and form a new **C** element with that value, and with row index from the **A** element and the column index from the **B** element.
5. Sort the **C** elements by row and column index.
6. Sum the values of all **C** elements with the same row and column indices (using summing by groups), and place the value into the first element of the group.
7. Pack the first element of each group into the final answer **C**.

Let T be the number of nontrivial terms occurring in the matrix product. The serial matrix multiplication algorithm uses O(T) time and O(m + M) space, where m is the input size and M is the output size. The parallel algorithm presented here requires time O((T/p)log(d)/log(T/p)) and space O(p^{1−γ}T^* + T), which simplifies under our assumptions to time O(T/p) and space O(T). Although the time for our parallel algorithm is optimal, the space may be significantly larger than optimal. We now refine the parallel algorithm so as to reduce the amount of space.

The main idea is to process the cross products in blocks. A partial result matrix **C** is kept of the accumulated sums computed so far. Whenever a new block is processed, a matrix of values produced from the cross products in that block is formed. The partial result matrix **C** is then updated by adding to it the new matrix. After all of the blocks are processed, **C** will be the desired product matrix. The goal is to choose the block sizes small enough so as not to use extra space, but large enough so that significant time is not wasted updating **C**.

Assume that the cross products are formed from groups **U** of elements from **A** and groups **V** of elements from **B**.

1. Form array **R** of the cross product sizes | **U** | | **V** |. Assume, wlog, that each **R** > 0.
(2) Use parallel prefix to form $S_i = \sum_{j=1}^{i} R_j$ (and set $S_0 = 0$).

(3) Let the product matrix $C = 0$ (by initializing it to the empty list). Let $i_0 = 0$.

Repeat for $k = 1, 2, \ldots$ until all of the blocks are processed.

(4) Let $b = \max \{ m, |C| \}$.

(5) Let $i_k = \min \{ i : S_i - S_{i-1} > b \}$ (if the condition never holds, let $i_k = \text{index of the last group}$).

(6) Form the product matrix using groups $U_j$ and $V_j$, for $i_{k-1} < j \leq i_k$.

(7) Update $C$ by adding to it the new matrix.

At each iteration, the time to compute the new product matrix is at least as large as the time to add the two matrices, since the number of terms in the block is always chosen to be at least as large as the size of the current product matrix $C$. Let $i_k$ be the number of products computed at the $k$th iteration of the above algorithm ($i_k = S_{i_k} - S_{i_{k-1}}$). Then, not counting the time to determine the next block, the total time for the algorithm is

$$
\Sigma t_k \leq T.
$$

where $\Sigma t_k = T$. By our assumptions, this simplifies to $\Sigma O \left( t_k \log \left( \frac{t_k}{p} \right) \right) = O \left( \frac{T}{p} \right)$.

The value of $i_k$ can be determined at each iteration by just one processor using unbounded binary search: Conservatively, the processor starts at $i = i_{k-1} + 1$ and continues increasing $i$ by doubling the difference $i - i_{k-1}$ until $S_i - S_{i_{k-1}} \geq b$; after that, a traditional binary search (between the last two values of $i$) will find $i_k$ in total time $O \left( \log \left( i_k - i_{k-1} \right) \right)$. Since $i_k - i_{k-1} \leq b$ , the time to determine $i_k$ is bounded by $O \left( \log i_k \right)$. By our assumptions, $p \leq i_k - i_{k-1}$, which implies

$$
\frac{i_k}{p} \geq t_k = O \left( \log i_k \right).
$$

The time to broadcast $i_k$ to all of the processors is $O \left( \log p \right)$. Similar reasoning to the above shows that $\log p = O \left( \frac{t_k}{p} \right)$. Thus the searching and broadcasting time is dominated by the computation time.

Finally, we need to show that the space usage is not large: At each iteration, the total size of the new matrix, not counting the last cross product group in the block, is proportional to the maximum of $m$ and the current size of the product matrix $C$. This is clearly $O \left( m + M \right)$. All of the terms in a given cross product group are associated with a distinct element of the product matrix. Thus, the last group has size $O \left( M \right)$. So, the total size of the matrices at each iteration is $O \left( m + M \right)$. The sorting algorithm uses space $O \left( p \left( m + M \right) + (m + M) \right)$. By our assumptions this is simply $O \left( m + M \right)$.

5. GAUSSIAN ELIMINATION

Standard algorithms for the inversion or decomposition of an $n \times n$ (dense) matrix involve a sequence of $n$ stages (one stage for each row of the matrix). Most of the computations done at each stage are independent vector operations; these can be computed efficiently in parallel. The computations done at successive stages are
strongly data dependent. In order to reduce the time required to invert or decompose a matrix in parallel below time $O(n)$ it is necessary to use different algorithms (e.g., [7]). Such algorithms seem to be both numerically unstable and inefficient with respect to the number of operations performed. We shall, therefore, consider parallel versions of the standard serial algorithms. We first show how to do Gaussian elimination assuming that the pivots are known in advance, and then discuss pivot selection.

Let $A$ be a square $n \times n$ (sparse) matrix with $m$ nonzero entries. Assume that $m \geq n$; otherwise the matrix is singular. We shall consider the problem of solving the system of linear equations $Ax = b$ using Gaussian elimination. Computing an LU decomposition for the matrix $A$ is done essentially in the same manner, and will not be discussed.

Gaussian elimination consists of an elimination phase and a back substitution phase. The elimination phase consists of a sequence of $n$ stages that modify the entries in the extended matrix $[Ab]$. A pivot element is chosen in this matrix, and suitable multiples of the row containing this element are subtracted from the remaining rows.

Each row of the matrix is stored as a set of pairs, (column, value). The column indices need not be distinct so that the actual value of an element in row $r$ and column $c$ is the sum of the values in the row $r$ set having column index $c$. Similarly, each column is stored as a set of pairs, (row, value), where the row indices need not be distinct. Thus, the value of an element can be determined from its row or column set. The actual values of the elements in some particular row (column) are resolved, i.e., the values of the elements with the same column index (row index) are summed when the new pivot element is from that row (column). Thus, the algorithm uses lazy evaluation to avoid wastefully accessing all the elements of a row or column when only a few elements need to be resolved. In order not to waste space storing each element as the sum of many pairs, all of the row and column sets are resolved whenever the total number of pairs in the row and column sets reaches a certain threshold (which is dynamically set).

The algorithm consists of $n$ stages. At each stage, the next pivot element is selected. The values in its row and column sets are resolved (independently). The values on these two sets are then used to determine the set of values needed to zero out the column of the pivot element while keeping the other matrix values consistent. This set is unioned into the row and column sets.

More formally, let $E$ be the extended matrix $[Ab]$ and let $<i_1, j_1>, <i_2, j_2>, \ldots, <i_n, j_n>$ be the successive pivots, then the algorithm works as follows:

**Forward Elimination:**

\[
\text{let } \text{prev\_size} = |E|; \\
\text{for } k := 1 \text{ to } n \text{ do begin} \\
(1) \text{ resolve row } i; \text{ place it in row vector } U; \\
(2) \text{ resolve column } j; \text{ place it, without the pivot } (a_{i_0, j_0}), \text{ into row vector } V; \\
(3) \text{ let } Q = \frac{1}{a_{i_k, j_k}} U \times (V - (U \times V)_{ij}); \\
(4) \text{ let } E = \text{prev\_size} \cup Q; \\
\text{end} 
\]
(5) if \(|E| = 2 \times \text{prev size}\) then 
\[\text{begin resolve } E; \text{let } \text{prev size} = \max (m, |E|) \text{ end if;}\]
end for.

**Back Substitution:**

for \(k := n\) downto 1 do 
(1) calculate \(x_{it}\) of the solution vector: 
\[x_{it} := \frac{b_i}{a_{it}};\]
(2) subtract from \(b\) the vector \(x_i E_{ij};\)
end for.

**Analysis:** Temporarily assume that there is no space limitation. Each row and column set is stored as a packed array with a header giving its size (i.e. the number of pairs in it). Each such array has length \(n\), sufficient for a maximum size row or column. Step (1), resolving a row, is accomplished by first sorting the set by column index, then summing by groups the values having the same column index, then packing the totals to the top of the array, and finally, updating the set size. Step (2) can be accomplished similarly. The time for both steps is bounded by the time to execute a radix sort: 
\[O\left(\left(\frac{u + v}{p}\right) \left(\frac{\log n}{\log(2 + (u + v)/p)}\right)\right), \text{ where } u = |U| \text{ and } v = |V|.

Step (3) is a tensor product followed by the multiplication of a constant times a matrix. This is simply a cross product calculation, then a broadcast of the pivot value, and finally, a local calculation at each pair of the cross product. This takes time \(O(u(v-1) + \log p)\).

Step (4) (updating the row and column sets) can be accomplished by first unioning \(Q\) into the row sets and then unioning it into the column sets. We consider only the former problem (unioning into the row sets) as the latter problem can be accomplished similarly. A straightforward way to do this is to determine for each element in \(Q\) a distinct location for inserting it into the proper row array of \(E\): Sort \(Q\) by row index. For each distinct row index in \(Q\), copy to the first element of \(Q\) with that index, the size of the corresponding row of \(E\). Assign to all other elements in \(Q\) the index 1. Use parallel prefix by groups on the rows of \(Q\) to produce a distinct index for each element in a row. Assign (about) \(q/p\), \(q = |Q|\), elements of \(Q\) to each processor. Copy each element to the designated location within \(E\). All of this takes time \(O(q/p + \log p)\).

Step (5) (resolving the entire matrix) can be accomplished by first working on the rows and then on the columns. As usual, we show only how to resolve the row sets. Copy the row sets into contiguous locations of a temporary (scratch) array. Radix sort the elements primarily on the row index and secondarily on the column index. Sum by groups the values of the elements with the same row and column indices. Within each such group, pack the elements containing the sum of the values. Finally, copy each row back into \(E\). The time is dominated by the time it takes to sort, which is \(O\left(\left(\frac{e}{p}\right) \left(\frac{\log n}{\log(e/p)}\right)\right)\), where \(e = |E|\).

This completes the forward elimination phase. To analyze the total execution time, we consider each step of the loop separately. In each case, the cost is amortized over all \(n\) iterations of the loop.

First consider the work at Step (1). Let \(t_p\) be the number of nonzero elements initially in row \(r\) of the matrix plus the total number of nonzero entries inserted into
Parallel Manipulation of Sparse Matrices

row \( r \) of the matrix (at Step (4)). Let \( T = \sum_{r} t_r^R \). The execution time of the serial algorithm is \( O(T) \). The execution time of Step (1) is dominated by the time required to sort the entries in each row \( r \), which is \( O((t_r^R/p) \log(n)/\log(t_r^R/p)) \). The total execution time is

\[
O\left(\sum_{r} \frac{t_r^R \log n}{p \log(t_r^R/p)}\right).
\]

This is maximized when all the \( t_r^R \) are equal to each other, and therefore are all equal to \( T/n \). Thus the total execution time of Step (1) is bounded by

\[
O\left(\frac{T \log n}{p \log(T/n^p)}\right).
\]

This is optimally \( O(T/p) \), provided that \( T/n = \Theta(p^{1+\epsilon}) \) for some positive \( \epsilon \). A similar analysis applies to Step (2).

Next consider the time to execute Steps (3) and (4). Let \( q_k \) be the size of array \( Q \) at stage \( k \) of the algorithm. Then \( T \geq \sum_k q_k \). The total execution time of the parallel algorithm at steps (3) and (4) is

\[
O\left(\sum_{k} \frac{q_k}{p} + \log p\right) = O\left(\frac{T}{p} + n \log p\right),
\]

which is dominated by Steps (1) and (2).

Finally, consider the work done at Step (5). Since, before each sort, the number of new elements inserted is at least \( m \), each sort is executed efficiently (in time proportional to the number of elements in the full matrix divided by \( p \)). Furthermore, the matrix \( A \) is resolved only when the number of entries has at least doubled, which means that the time to sort is (no more than) proportional to the time it took to insert the elements in the first place. Thus, Step (5) increases the overall running time by at most a constant factor.

The back substitution is much easier. It uses only the columns of \( A \). Extract from \( A \) the vector \( b \) (i.e., the \( n+1 \)-st column), and store it as a dense array. This takes time \( O(n/p + \log p) \). Step (1) only requires accessing the pivot element and then executing a local computation. This takes time \( O\left(1/|A_{ij}|/p + \log p\right) \). Step (2) executes a local computation between each distinct element of column \( A_{ij} \), the corresponding element of \( b \), and the pivot \( a_{ij} \). This also takes time \( O\left(1/|A_{ij}|/p + \log p\right) \). Summing over all \( n \) iterations of the back substitution phase, this sums to \( O(T/p + n \log p) \), which is dominated by the forward elimination phase.

5.1. Efficient Memory Usage

We now show how to implement the Gaussian elimination algorithm with less memory. The implementation uses a dynamic memory allocation routine that partitions memory into equal size blocks. We first describe a "one level" routine, where the block size \( B = \Theta(p \log l) \), and then extend it to more levels but smaller \( B \) in order to save space.

Assume temporarily that all of the memory needs are known in advance. Let \( M \) be the maximum size of the matrix during the Gaussian elimination algorithm. The algorithm uses a chunk of memory of size \( \Theta(M) \) to perform scratch calculations such as sorting the pairs in a row or column. It uses another chunk of size
Each row set of the matrix is stored in a linked list of blocks, where each block except possibly the last one stores \( \Theta(B) \) pairs. Pointers to the first and last blocks in each row list, along with a count as to the exact number of pairs in each list, are stored in a one dimensional array of size \( n \). The columns of the matrix are stored similarly.

Now consider each of the steps within the for loop of the elimination phase. To resolve a row at Step (1), the processors copy the pairs to the scratch array, then do all of the calculations as previously described. To copy the pairs, the processors traverse the blocks one at a time, copying all of the pairs from one block before continuing to the next. After the pairs in a block are copied, the block is placed back onto the free list. The time to copy the pairs from a single block is the time to broadcast the block pointer to all of the processors plus the time to copy the pairs in parallel, which is \( \Theta(B + \log p) \). Since \( B = \Theta(p \log p) \), this is \( \Theta(B / p) \). Thus the copying is done completely efficiently. Resolving the columns at Step (2) is handled similarly.

The calculations at step (3) are executed completely in the scratch array, and therefore require no further discussion. Step (4) unions the pairs back into the matrix. We show only how to union back into the row sets, as the column sets are handled similarly. Sort the pairs in \( Q \) by row index. Assign to all elements in \( Q \) the index 1. Use parallel prefix by groups on the rows of \( Q \) to produce a distinct index for each element in a row. Copy to the first element of each row in \( Q \) the size of the corresponding row of \( E \) along with the pointer to the last block in that row. Broadcast these two items to all of the pairs within the row of \( Q \). Now it is easy for each pair to decide locally whether it will be inserted at the end of the last block of the row in \( E \) or into some new block, and to determine the exact location within its block. Pairs that will use the first location of a block are responsible for removing the new block from the free list. Pack those pairs to the top of a separate scratch array. Copy the free list pointer to the first such pair. Use parallel prefix to give each pair a unique index into the free list starting from the initial value of the pointer. Let each pair copy its designated block pointer from the free list. Reset the free list pointer. Link together blocks from the same row. Let the first pair in each row link its block to the end of the block list of \( E \), and let the last pair reset the last block pointer of \( E \). Returning to all of the pairs of \( Q \), broadcast by groups the block pointer to all of the pairs in \( Q \) sharing the same block. Finally, copy all of the pairs of \( Q \) to their proper locations in \( E \). All of this uses only local computations and parallel prefix, and so it takes time \( O (|Q| /p + \log p) \).

Resolving the rows at Step (5) has three steps: (a) copying the pairs of \( E \) to the scratch matrix, (b) actually resolving the pairs, and (c) copying the new pairs back into \( E \). Step (b) can be done as previously described, since it is all done within the scratch matrix. Step (c) is just like the copying back into \( E \) part of Step (4), except now it is a little easier since all of the rows of \( E \) are empty. Step (a) can be done by essentially reversing the steps of Step (c), so we will not go into the details. Note, however, that it is important not to simply traverse the blocks of \( E \) one at a time, as we were willing to do in Step (1), since that will require \( \Omega(n) \) time. Resolving the columns at Step (5) is accomplished similarly.

The back substitution does not present any further difficulties and can easily be accomplished in the required time and space bounds using the above techniques.
Now consider if the total amount of space needed is not known in advance. Start with $\Theta(m)$ space for the scratch array and $\Theta(m + nB) = \Theta(m + np \log p)$ space for the free list of blocks. Whenever space runs out, acquire twice the space being used and copy everything to the new space. This increases the total execution time by at most a constant factor, and uses at most twice the necessary space.

The algorithm as so far described uses space $\Theta(M + np \log p)$. To reduce this, instead of storing the rows and columns as linked lists of blocks, they can be stored as a linked list of trees of blocks. In the two-level scheme the block size is $B = \Theta(\sqrt{p \log p})$, and each tree has depth two. The root of each tree has (at most) $O(B)$ children, and the pairs are packed into the children. All of the trees in the linked list are full – except possibly the last one, which is packed into as few blocks as possible. Copying a row or column to the scratch array, as done in Steps (1) and (2) of the forward elimination phase, is not much harder than before. The processors have the extra complexity of broadcasting children pointers by groups (of $\sqrt{p}/\log p$ processors). The added difficulty of unioning the rows or columns into $E$, as done in Step (4), is having to obtain the blocks. First any pair needing a new block does so as described previously. Then some of the new blocks will need new roots. But, this extra layer of work can be done in the same fashion and will increase the time only additively. The space used in this version is $O(M + n \sqrt{p \log p})$. The scheme can be generalized to any (constant) number of levels with only a constant factor time penalty; so, in general the space can be reduced to $O(M + np^\delta)$ for any $\delta > 0$.

To summarize, assuming the pivots are known in advance, Gaussian elimination can be executed in time $O\left(\frac{T}{p \log(T/np)}\right)$ and space $\Theta(M + np^\delta)$ for any $\delta > 0$.

The time is optimally $O(T/p)$, provided that $T = \Omega(np^{1+\varepsilon})$ for some positive $\varepsilon$, and the space is optimally $O(M)$, provided that $M = \Omega(np^\delta)$ for some $\delta > 0$.

### 5.2. Pivot Selection

Pivot selection methods for Gaussian elimination in sparse matrices attempt both to control numerical stability, and reduce the number of nonzero elements created (the fill). Let $r_i$ be the number of nonzero elements in row $i$ and $c_j$ be the number of nonzero elements in column $j$. Then one common criterion is to select a pivot $a_{ij}$ that minimizes the Markowitz count $(r_i - 1)(c_j - 1)$ over all entries that fulfill $|a_{ij}| \geq u \max_i |a_{ii}|$ for some fixed threshold $u$. The second condition corresponds to partial pivoting; the first one attempts to minimize the amount of fill created by the pivot.

An exhaustive search over all the possible entries is too expensive; serial algorithms are usually content with an approximation to the above scheme. For example, Zlatev [8] suggests confining the search to a fixed number of rows (three).

Our method does not allow us to keep population counts for rows and columns. Instead, we suggest using the number of entries in each underlying unresolved row and column as an approximation to the correct count. We assume that such count is associated with each row and column, and updated whenever the row or column is updated (under the conservative assumption that whenever an element is inserted into a row or column it fills in a zero location). Also, the rows are stored in a priority queue, according to their count. It is easy to see that row and column counts,
and the priority queue can be maintained when the matrix is updated with a constant factor overhead.

We use the following pivot selection algorithm: (1) Pick three rows with least row count; resolve rows (and update $r_i$s). (2) Find in each row the element with maximum absolute value; create for each row the set of elements that fulfill the condition. (3) Compute the Markowitz count of all these elements; pick the element with least count as pivot.

The most straightforward way to implement this is to sort the rows by count (of nonzero entries) at each stage. This will increase the total execution time by $O(n^2/p)$. More sophisticated parallel priority queue algorithms will reduce this time.

6. CONCLUSIONS

We have presented algorithms for basic computations on sparse matrices. These algorithms are theoretically optimal in the domain of interest. We believe these algorithms are also practical - although a practical implementation would use many "local" optimizations not mentioned in this introductory work. These techniques can be used for other sparse matrix algorithms. For example, it is easy to obtain efficient parallel algorithms for Cholesky factorization on sparse matrices.

REFERENCES