(2)

VLSI Memo No. 89-536
May 1989

Four Vector-Matrix Primitives

Ajit Agrawal, Guy E. Blelloch, Robert L. Krawitz, and Cynthia A. Phillips

Abstract

This paper describes four APL-like primitives for manipulating dense matrices and vectors and describes their implementation on the Connection Machine hypercube multiprocessor. These primitives provide a natural way of specifying parallel matrix algorithms independently of machine size or architecture and can actually enhance efficiency by facilitating automatic load balancing. We illustrate their use in three numerical algorithms: a vector-matrix multiply, a Gaussian-elimination routine and a simplex algorithm for linear programming. We describe implementations of the primitives assuming load-balanced embeddings of matrices and vectors on a hypercube multiprocessor architecture. The primitives may indicate a change from one embedding to another. The implementations are efficient in the frequently occurring case where there are fewer processors than matrix elements. In particular if there are $m > p \lg p$ matrix elements, where p is the number of processors, then the implementations of some of the primitives are asymptotically optimal in that the processor-time product is no more than a constant factor higher than the running time of the best serial algorithm. Furthermore, the parallel time required is optimal to within a constant factor.

We have implemented the primitives on the Connection Machine System, and this implementation improved the performance of the simplex implementation by almost an order of magnitude over a naive implementation, from 52 Mflops to 500 Mflops. We give Connection Machine timings for the primitives and the algorithms.

Microsystems Research Center, Room 39-321 | Massachusetts Institute of Technology | Cambridge, Massachusetts 02139 | Telephone (617) 253-8138

Acknowledgements

To appear in *First Annual ACM Symposium on Parallel Algorithms and Architectures,* June 1989. This research was supported in part by Thinking Machines Corporation and the Defense Advanced Research Projects Agency under contract number N00014-87-K-0825. Phillips: was supported in part by International Business Machines Corporation graduate fellowship.

Author Information

Agrawal: Yale University, New Haven, CT 06520.
Blelloch: Carnegie Mellon University, Pittsburgh, PA 15213.
Krawitz: Thinking Machines Corporation, Cambridge, MA 02139.
Phillips: Laboratory for Computer Science, Room NE43-338, MIT, Cambridge, MA, 02139. (617) 253-7583.

Four Vector-Matrix Primitives[6]

Ajit Agrawal[2]
Guy E. Blelloch[3,4]
Robert L. Krawitz[4]
Cynthia A. Phillips[5,4]

Abstract

This paper describes four APL-like primitives for manipulating dense matrices and vectors and describes their implementation on the Connection Machine[1] hypercube multiprocessor. These primitives provide a natural way of specifying parallel matrix algorithms independently of machine size or architecture and can actually enhance efficiency by facilitating automatic load balancing. We illustrate their use in three numerical algorithms: a vector-matrix multiply, a Gaussian-elimination routine and a simplex algorithm for linear programming. We describe implementations of the primitives assuming load-balanced embeddings of matrices and vectors on a hypercube multiprocessor architecture. The primitives may indicate a change from one embedding to another. The implementations are efficient in the frequently occurring case where there are fewer processors than matrix elements. In particular if there are $m > p \lg p$ matrix elements, where p is the number of processors, then the implementations of some of the primitives are asymptotically optimal in that the processor-time product is no more than a constant factor higher than the running time of the best serial algorithm. Furthermore, the parallel time required is optimal to within a constant factor.

We have implemented the primitives on the Connection Machine System, and this implementation improved the performance of the simplex implementation by almost an order of magnitude over a naive implementation, from 52 Mflops to 500 Mflops. We give Connection Machine timings for the primitives and the algorithms.

1 Introduction

When implementing a parallel algorithm, it is convenient to have a high-level parallel language which provides the convenience one has come to expect from well-established serial languages. One wishes to concentrate on the details of the algorithm, allowing the language to abstract away details of the machine including the number of processors, the interconnection network, and the embedding of data elements into processors. However, since parallel machines are currently very expensive and used for huge, computationally intensive applications, users often will not give up performance for ease of programming and portability. Techniques for mapping high-level descriptions of algorithms onto efficient code for various parallel machines have therefore become very important and will probably consume a large portion of computer science research in the next decade.

This paper provides such a technique by showing how very high-level descriptions of a broad class of dense matrix algorithms can be mapped onto a real machine, the Connection Machine, with no performance loss over hand coded versions. It presents 4 high-level APL-like primitives and illustrates code for a vector-matrix multiply, a Gaussian-elimination routine and a simplex algorithm for linear programming based on these primitives. The algorithms are straight-forward implementations of the classic algorithms. The code is high-level, it works for any sized matrices[7], and contains no information on how data is mapped onto processors nor on how the data should be communicated. Therefore it is concise: none of our routines contain more than 20 lines of code. The paper then discusses our implementation of the primitives on the Connection Machine and gives

[1] Connection Machine System is a registered trademark of Thinking Machines Corporation

[2] Yale University
[3] Carnegie Mellon University
[4] Thinking Machines Corporation
[5] MIT Laboratory for Computer Science

[6] This research is supported in part by Thinking Machines, in part by the Defense Advanced Research Projects Agency under Contract N00014-87-K-0825. Cynthia Phillips is supported in part by an IBM graduate fellowship.

[7] Our current implementation is restricted to sizes which are powers of two.

Scalar Instructions:
+, −, ×, ...
Global Instructions:
Broadcast, g-min, g-max, ...
Elementwise Vector and Matrix Instructions:
p-+, p--, p-*, p-÷, ...
Vector-Matrix Instructions:
insert, extract, distribute, reduce

Figure 1: The instructions we use in the algorithms described in this paper. *Global* instructions compute a single value from all elements of a vector or matrix or broadcast a single value across a vector or matrix. *Elementwise* instructions perform an operation elementwise over corresponding elements of equal sized matrices or vectors.

actual timings for both the primitives and two of the algorithms. The simplex and vector-matrix multiply execute faster than any other code for these applications for the Connection Machine. With 256 matrix elements per processor, an iteration of simplex runs at 500 Mflops and Matrix-Vector multiply runs at over 1 Gflop on a 64K CM-2 (single precision). Versions of the primitives we implemented are now included in PARIS, the parallel instruction set of the Connection Machine.

The four primitives we consider *extract* a vector from a row or column of a matrix, *insert* a vector into a row or column, *distribute* a vector across the rows or columns, and *reduce* the rows or columns into a vector using a binary associative operator such as +, maximum, or minimum. It is convenient to also use an operation which *spreads* a row or column across its matrix (*extract* followed by *distribute*). We also assume the existence of various other simple primitives—these primitives are summarized in Figure 1.

Although the *extract* and *insert* primitives might seem trivial, their implementation is actually as complex as the implementation of the *distribute* and *reduce* primitives. They involve rearranging the data among the processors by communicating along trees embedded within subcubes occupied by the rows or columns of a matrix. This data transfer guarantees optimal load balancing of any row or column when extracted from a matrix. Thus after extraction each processor holds (within one) the same number of vector elements ($= \lceil m/p \rceil$ where m is the number of matrix elements and p is the number of processors). Load balancing the vectors improved the performance of the simplex algorithm from under 100MFlops to over 500Mflops.

There has been considerable research on implementing dense matrix algorithms on hypercube based machines [14, 9, 12, 7, 3, 4]. This paper concentrates on how to get similar results without having to code for the particular machine. The final execution of the linear systems solver is similar to the algorithm suggested for a hypercube by Johnsson [12]. Our implementations of the primitives are simple, clean subcases of the n edge-disjoint spanning binomial trees (NSBT) algorithms due to Johnsson and Ho [13]. Our *extract* implementation is a version of what they call one-to-all personalized communication within subcubes and our *distribute* implementation is an example of all-to-all broadcasting. Other related algorithms for hypercubes are discussed by Fox and Furmaski [8], Stout and Wager [17], and Deshpande and Jenevin [6].

To motivate some of the decisions we made in the selection and implementation of primitives, let us examine, as an example, the solution of linear programs using the Dantzig simplex method. The standard form of a linear programming problem is as follows:

$$\text{minimize} \quad c^T x \quad \text{such that} \quad \begin{cases} Ax = b \\ x \geq 0 \end{cases}$$

where c is an m_2-dimensional integer *objective function* vector, A is an $m_1 \times m_2$ integer *constraint matrix*, b is an m_1-vector of integers, and x is a real m_2-vector of unknowns. All information needed to perform a step of the computation is kept in a *tableau* which is initially the constraint matrix A augmented by the column vector b and the row vector c. A single step of the computation is then one step of Gaussian elimination on the entire tableau where the pivot column has the most negative value of vector c and the pivot row has the smallest positive value of b/a within the pivot column. Section 2 explains some of the ideas behind the method—for now, an understanding of the functionality suffices.

If we have as many processors as tableau elements, a straight-forward implementation of one simplex step can be written as follows:

Algorithm Simplex

```
;tableau T ((m1 + 1) × (m2 + 1))
;initially matrix A augmented by
;column vector B and row vector C

repeat forever:
1  selecting processors that initially held vector C
2    pivotcolnum = column # of processor
       holding g-min(T)
     (if g-min(T) ≥ 0, exit Simplex successfully)
3  selecting processors that initially held vector B
4    send value of T to pivot column
       (store locally as B)
5  selecting processors in
       Pivot column with T > 0
       (if none, exit Simplex unsuccessfully)
6    Ratio = p-+(B, T)
7    pivotrownum = row # of processor
       holding g-min(Ratio)
8  pivotelement = A[pivotcolnum][pivotrownum]
9  selecting processors in the pivot row
10 T = p-÷(T, Broadcast(pivotelement))
```

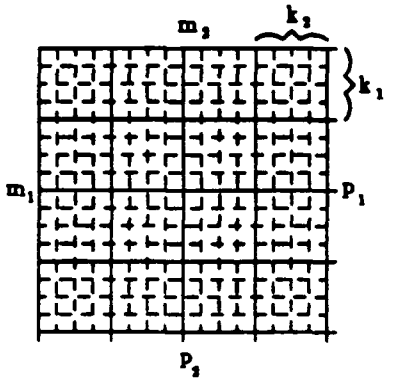

Figure 2: Each processor in an P processor hypercube viewed as a $p_1 \times p_2$ grid of processors holds a $k_1 \times k_2$ submatrix of the $m_1 \times m_2$ matrix ($P = p_1 p_2$, $m_1 = p_1 k_1$ and $m_2 = p_2 k_2$).

```
11 send T value of pivot row to all rows
       (store in Pivotrow)
12 send T value of pivot column to all columns
       (store in Pivotcol)
13 T = p-−(T, p-*(Pivotrow, Pivotcol))
```

In general, however, we would like to run problems in which the tableau is larger than the number of processors. Figure 2 shows how we might map an $m_1 \times m_2$ matrix onto a machine where the processors can be logically configured as a grid. Each processor holds a $k_1 \times k_2$ submatrix which is as square as possible (an aspect ratio of at most 2). The runtime environment can keep track of how a matrix is mapped onto the processors, and can automatically loop over all the elements in each processor when operating on the matrix.

This embedding is *load balanced* with respect to the matrix since each processor holds an equal number of matrix elements. Any given row or column, however, when viewed as a vector is not well-balanced at all as shown in Figure 3. In particular, when performing the divisions in lines 6 and 10 and the minimums in lines 2 and 7 in the simplex algorithm above, a small subset of the processors must perform many computations while the rest of the processors are idle. When computing on a single column, as in line 6, it may be more efficient to use the underlying interconnection network to spread the work from the one active processor in each row to the idle processors in its row. For example, in computing on the column selected in Figure 3, the two active processors distribute one operation to each of the three idle processors in their rows so that each processor performs one operation. Furthermore, it may be beneficial to always leave the B and C vectors stored in this load-balanced fashion and to update them in place. In our implementation on the Connection Machine System, a carefully coded implementation similar to the

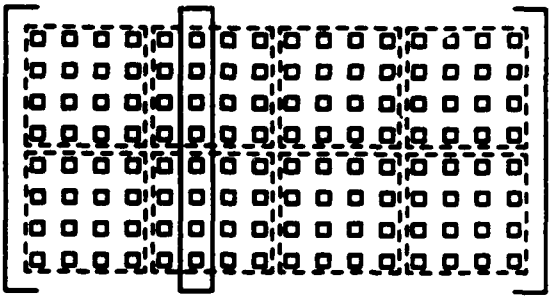

Figure 3: All the elements of a single column are located in a small subset of the processors. In the example they are located in 2 of the 8 processors. When extracting a column, the elements are distributed across the processors so that each processor gets one element of the column.

code above ran at 52 Mflops. After adding the load balancing, the loop ran at 500 Mflops.

In seeking to generalise from this experience, we found that the four classes of primitives provide expressive power and potentially automatic efficiency improvement. In the simplex example, our inefficiencies came from treating vectors as matrices. Distinguishing between vector and matrix computations is not only efficient, but also fits well into the way programmers think about computing on these objects.

Section 2 gives examples of the use of the primitives in several applications including the simplex method for linear programming. The four classes of primitives provide cues that computation is shifting from a full matrix to a vector related to that matrix (eg. a row or column), indicating that it may be a proper time for load balancing. Bookkeeping related to the looping over matrix or vector elements within a processor is best executed at runtime rather than compile time because the binary code should be independent of the number of processors in a machine. A compiler, however, can frequently decide whether is is better to load balance during an extract or reduce operation or to leave the elements in place.

Section 3 discusses the implementations of the primitives for *hypercube* multiprocessors with $N = 2^n$ processors. Each processor has a unique address which is $\lg N = n$ bits long and each processor is directly connected to all processors whose address differs from its own in exactly one bit position. Two connected processors whose address differs in bit i are called *ith-dimensional neighbors*. The algorithms we present in this paper work on any hypercube multiprocessor, but they are simple enough to run on a *data parallel architecture* in which each processor executes the same instruction broadcast from a *front end* computer (also known as a single instruction multiple data architecture).

Section 4 gives timings for an implementation of our primitives on the Connection Machine System [10]. The current implementations are restricted in two ways. Firstly, the number of rows and columns in a matrix must be a power of two. Secondly, a vector extracted from a row of a matrix can only be distributed to or deposited in a row of a matrix of equal size as the matrix is was extracted from. In Section 3 we discuss techniques for avoiding the first restriction. The Connection Machine has a router for arbitrary interprocessor communication and its processors have the ability to indirectly address their memory; our algorithms, however, do not require the power of these features.

Remarkably, although our implementations are very simple, the reduce and distribute primitives are optimal for a hypercube in two senses: parallel time, and processor-time product (work), provided the matrix is sufficiently large relative to the number of processors and each processor is capable of sending only a constant number of messages in unit time.

2 Example Applications

In this section, we present CM implementations of matrix-vector multiplication, LU decomposition with partial pivoting, solution of a linear system from an LU decomposition and a vector b, and a simplex method for linear programming. These have been implemented using the primitives we described and are competitive with all other CM implementations to date.

2.1 Matrix-Vector Multiply

The first example we present is a matrix-vector multiply. Using a distribute, we spread the input vector over all rows of the matrix, multiply the two element-wise, and reduce all rows of the result by summing across the rows, returning a column vector containing the matrix-vector product. Thus,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}$$

becomes

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & a_{2n} \\ \vdots & & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} b_1 & b_2 & \cdots & b_n \\ b_1 & b_2 & & b_n \\ \vdots & & & \vdots \\ b_1 & b_2 & \cdots & b_n \end{bmatrix}$$

and the sums of the elementwise multiplications are taken along the rows to yield the final result.

One complication is that our algorithm accepts a row vector as input but returns a column vector. This condition causes less of a problem if we want to immediately use the vector in another matrix-vector multiplication, as we would if we were investigating a Markov process. At some cost in storage, we can store both the transition matrix and its transpose, then alternate which one we use. Alternatively, we can pay lg n time to transpose either the vector or the matrix.

Algorithm Matrix-Vector Multiply (MVM)
;*matrix* A $(m_1 \times m_2)$
;*vector* B
;*temporary matrix* $TEMP$ $(m_1 \times m_2)$

1 $Temp$ = distribute-row(B)
2 $Temp$ = p-*($A, Temp$)
3 return(+-reduce-to-column($Temp$))

2.2 Linear System Solution by LU Decomposition

The next example demonstrates solution of a linear system by LU decomposition with partial pivoting and back solving. Given an $m \times m$ matrix A, to compute upper triangular matrix U and lower triangular matrix L such that $A = LU$, we perform m steps of Gaussian elimination. Moving left to right through the matrix, we extract the columns in sequence. We find the element with the greatest absolute value, and extract its row. We then divide the pivot column by the pivot element itself, and distribute the pivot row and column across the matrix. We replace the part of the pivot column below the pivot element, where it will serve as a column of L. Finally, we perform Gaussian elimination on the square below and to the right of the pivot row and column. Rather than physically exchanging the two rows, which would yield a true lower and upper triangular matrix, we record in a separate vector which row was eliminated in which iteration and use this information to find the diagonal in the solution phase.

Algorithm LU Decomposition (LUD)

;*matrix* A $(m \times m)$
;LU decomposition with partial pivoting
;Returns LU in A, permutation vector in P

1 For i from 1 to m Do
2 selecting rows and columns of A
that have not been pivoted on
3 Col = extract-column(A, i)
4 $pivotrow$ = row # of g-max(p-abs)
5 Row = extract-row($A, pivotrow$)
6 $P[pivotrow] = i$
7 $pivotelement = A[i, pivotrow]$
8 Col = p-÷(Col, Broadcast($pivotelement$))
9 A = deposit-column(A, Col, i)
10 $Colmatrix$ = distribute-column(Col)
11 $Rowmatrix$ = distribute-row(Row)
12 selecting positions in A that are not
in the pivot row or column

13 $A = \text{p-}-(A, \text{p-}*(Rowmatrix, Colmatrix))$

The forward and back solution phase is also straightforward. Given a vector b, we first solve $Ly = b$, divide y and U through by the diagonal such that the system has a unit diagonal, and then solve $Ux = y$. A solution step consists of extracting a column, multiplying it by the diagonal element, and subtracting from b.

Algorithm Solve Linear System from Decomposition

```
;matrix LU (m × m)
;vector B
;permutation vector P
;Solve LUx = B

1  For i from 1 to m Do
2   Column = extract-column(LU, i)
3   pivot = B[k] such that i = p[k]
4   selecting the elements B[j] and Column[j]
       such that P[j] > i
5     B = p-−(B, p-*(Column, Broadcast(pivot)))
6  send logical diagonal of LU to first col of Temp
7  Diag = extract-column(Temp, 1)
 ;At this point B contains y from Ly = b
 ;Divide U and B by the diagonal.
8  B = p-÷(B, Diag)
9  Temp = distribute-column(Diag)
10 Selecting positions in the logical
       upper triangle of LU
11  LU = p-÷(LU, Diag)
12 For i from m downto 1Do
13  Column = extract-column(LU, i)
14  pivot = B[k] such that P[k] = i
15  selecting the elements B[j] and Column[j]
       such that P[j] ≤ i
16    B = p-÷(B, p-*(Column, Broadcast(pivot)))
17 unpermute B
```

2.3 Simplex method for linear programming

Our final example illustrates the Simplex method for solving linear programming problems. The standard form of a linear programming problem is as follows:

$$\text{minimize} \quad c^T x \quad \text{such that} \quad \begin{cases} Ax = b \\ x \geq 0 \end{cases}$$

where c is an m_2-dimensional integer *objective function* vector, A is an $m_1 \times m_2$ integer *constraint matrix*, b is an m_1-vector of integers, and x is a real m_2-vector of unknowns. Generally we have $m_1 < m_2$.

A vector x such that $Ax = b$ and $x \geq 0$ is called a *feasible solution* because it satisfies all the constraints. If a linear program has an optimal solution, we can always find one such that m_1 of the entries in vector x are equal to 0 [15]. Such vectors, called *basic feasible solutions*, correspond geometrically to corners of the convex $(m_2 - m_1)$-dimensional polytope of all feasible solutions. The simplex method for solving linear programs due to Dantzig [5] starts at a basic feasible solution and *pivots* to a new basic feasible solution which improves the objective function. Algebraically, we increase one of the zero-valued *nonbasic* variables (the *entering* variable) until one of the non-zero *basic* variables becomes zero. In the Dantzig method of pivoting, the entering variable is the one that will decrease the objective function by the most (per unit increase in the variable).

All the information necessary to perform the pivoting is kept in a *tableau* where the objective function and all nonbasic variables are represented in terms of the basic variables. Since columns corresponding to the basic variables always form the identity matrix, we save space by not representing these columns (the code in Section 1 does not include this optimization). At the start, the tableau is the constraint matrix A augmented by the column vector b and the row vector c. Vectors b and c are also maintained in vector representation. We then use Gaussian elimination to eliminate all columns corresponding to basic variables. To form the tableau for which one basic variable is replaced by a nonbasic variable then involves one step of Gaussian elimination.

The implementation of Simplex with Dantzig's rule is fairly straightforward. We first find the index of the most negative coefficient of the objective function; pivoting on this variable will give us the most rapid improvement in the solution per unit increase in the entering nonbasic variable. If there are no negative coefficients, then we cannot make any improvement, and thus have finished successfully. We then extract the indexed column, and select the positions corresponding to real constraints, i. e. only positive coefficients correspond to basic variables that decrease as the entering variable increases. If there are no positive coefficients in the column, then the system is unbounded; we can increase the value of variable improving the objective function without limit and never violate a constraint. To find the limiting constraint, we divide the b vector by the positive elements of the pivot column elementwise and find the index of the smallest ratio. The two indices define the pivot element. We then perform a Gaussian elimination step.

Algorithm Simplex (space-saving)

```
;tableau A ((m1 + 1) × (m2 + 1))
;constraint vector B
;objective function vector C

repeat forever:
1  pivotcolnum = col # of element
       holding g-min(C)
       (if g-min(C) ≥ 0, exit Simplex successfully)
2  Pivotcol = extract-column(A, pivotcolnum)
```

```
3 selecting positions in Pivotcol with values > 0
        (if none, exit Simplex unsuccessfully)
4   Ratio = p-÷(B, Pivotcolumn)
5   pivotrownum = row # of element
        holding g-min(Ratio)
6  Pivotrow = extract-row(A, pivotrownum)
  ;update pivot row and column
7  pivotelement = A[pivotcolnum][pivotrownum]
8  Pivotrow = p-÷(Pivotrow, Broadcast(pivotelement))
9  Rowmatrix = spread-row(Pivotrow)
10 Colmatrix = spread-column(Pivotcol)
  ;update constraint vector and objective function
  ;on their own, even though updated later
11 value = A[m, n]
12 B = p--(B, p-*(Pivotcolumn, Broadcast(value)))
13 C = p--(C, p-*(Pivotcolumn, value))
  ;Update the tableau
14 A = insert-row(A, Pivotrow, pivotrownum)
15 selecting positions of A that
        are not part of pivot row or column
16 A = p--(A, p-*(Pivotrow, Pivotcolumn))
```

3 Implementation of Primitives

In this section we present efficient implementations for the four vector-matrix primitives on hypercube architectures and analyze the time and processor-time complexities of these implementations. The implementations are based on a particular embedding of matrices and vectors on the hypercube. We show that both the time complexity and the processor-time complexity of the reduce primitives are within a constant factor of optimal provided that the matrix is sufficiently large relative to the number of processors. Finally, we discuss generalizations to higher dimensional matrices, matrices whose dimensions are not powers of 2, and matrices where rows or columns are favored.

Figure 2 shows how we map an $m_1 \times m_2$ matrix on an N-processor hypercube. Each processor holds a $k_1 \times k_2$ submatrix which is as square as possible. The processors can themselves be viewed in a grid (perhaps in row-major order with respect to hypercube addresses), but we use the full power of the hypercube connections in our implementation. The processors holding a given row of the matrix form a $\lg p_2$-dimensional subcube of the n-dimensional hypercube and the given row is mapped to the same submatrix row in each processor. To map a length 2^v *vector* onto an n-dimensional hypercube configured as a $p_1 \times p_2$ grid, we map 2^{v-n} vector elements to each processor with row vectors mapped in column-major order and column vectors in row-major order. Figure 4 illustrates how row vectors are mapped onto the virtual grid both when $v > n$ and when $v < n$. The embedding of vectors as described above is *load balanced* in that each processor holds an equal number of elements.

Based upon the matrix and vector embeddings described above, we describe the implementation of the row version of the extract primitive. The implementation of the other primitives are similar and are pictured in Figures 5, 6, and 7. The deposit primitive is basically the inverse of the extract primitive. The reduce primitive is similar to the extract primitive but as well as swapping data across the cube to load balance, the data is summed along the way. The distribute primitive is basically the inverse of the reduce primitive.

All of the primitives are implemented by stepping through the dimensions of the hypercube with each processor simultaneously communicating with its neighbor in that dimension. In the extract and reduce primitives, the number of data elements communicated typically halves on each dimension step (each step halves the complexity). In the deposit and distribute primitives, the number of data elements communicated typically doubles on each dimension step. For the extract and deposit primitives, only one processor from each pair of communicating processors sends data. For the reduce and distribute primitives, both neighbors generally send equal amounts of data. None of the primitives require indirect addressing on the processors, and the control is strictly data parallel.

3.1 Extract

The extract operation takes a matrix and an index r within the bounds of the matrix and extracts row r as a row vector (see Figure 4). In the extract procedure, we step through the dimensions, starting at the highest dimension, and each processor containing elements of r sends half of its elements to its empty neighbor in that dimension. The result of the extract is that the vector elements originally held in the processor at grid address (r_p, j) are evenly distributed among the processors in column j.

Algorithm Extract-Row

```
;matrix
;row number r (processor row r_p, offset r_m)

1 Let r_0, r_1, ... r_{lg p_1 - 1} be the bit representation
      of processor row r_p.
2 For i = 0 to lg p_1 - 1 Do
3   If no processor has > 1 element of row r
      then stop.
4   Selecting processors with elements of row r
5     if r_i = 0
6       then send last k_2/2^{i+1} elements
          of row r to neighbor in dimension i.
7       else send first k_2/2^{i+1} elements
          of row r to neighbor in dimension i.
```

The second if statement guarantees that the row is kept in a fixed order. Figure 4a illustrates the case where the number of elements per row in each processor is greater

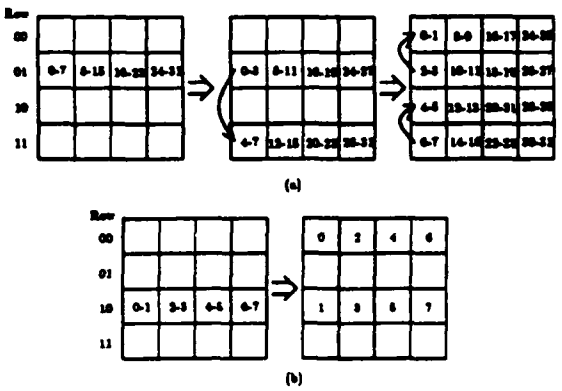

Figure 4: Examples of the algorithm extract-row. The grid indicates the grid of processors. Number ranges indicate elements of the row vector being extracted. At the start, all elements are in a single row of processors. In case (a), each processor ends up with more than one vector element. In case (b), some processor rows do not get any vector elements. In both cases, the final vector is in column-major order.

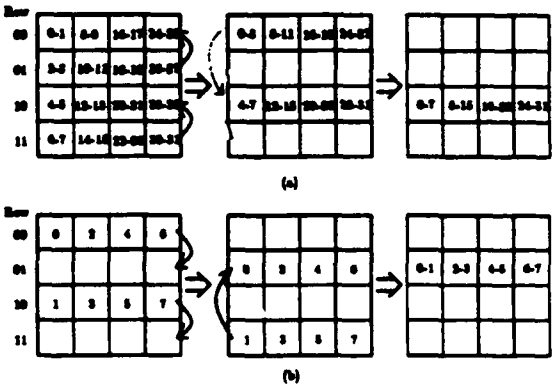

Figure 5: The deposit-row primitive deposits a row vector into a row of a matrix. The algorithm is basically the inverse of the extract-row algorithm.

than the number of processors in each column of the processor grid. In this case the row elements are dispersed in column-major order with each processor getting the same number of matrix elements. Figure 4b illustrates the case where the number of row elements per processor is less than the number of processors in a column of the processor grid. In this case, each processor has at most one matrix element. Which rows are empty depends upon which row is extracted. If no exchanges are performed across the last j dimensions, then only processors that match the extracted row in the last j bits of column address will contain vector elements. A modified version can put the vector in canonical form with at most $\lg p_1$ extra steps.

We now count the number of *messages* sent where a message is the sending of one matrix value by each active processor. During the first communication phase, each active processor sends $k_2/2$ messages, then $k_2/4$ messages, then $k_2/8$ and so on. Each time a message is sent, each sending processor sends one element to a neighbor which has fewer elements. Thus, each message phase reduces the maximum number of elements per processor by one. Therefore, the total number of messages sent is equal to the reduction in maximum number of elements per processor. Since at most $\lceil k_2/p_1 \rceil$ elements of r are left in any processor, the number of messages sent is $k_2 - \lceil k_2/p_1 \rceil$.

Using similar arguments the message complexities of deposit and distribute are at most $k_2 - \lceil k_2/p_1 \rceil + \lg p_1$, $k_2 - \lceil k_2/p_1 \rceil + \lg(\lceil p_1/k_2 \rceil)$ respectively. The reduce primitive has the same message complexity as distribute and it has an operation complexity of $k_2 k_1 - \lceil k_2/p_1 \rceil +$

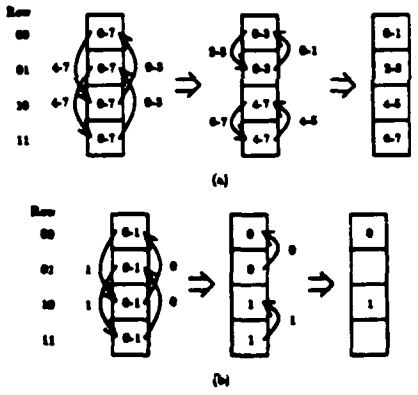

Figure 6: The reduce-to-row operation focusing on one column of processors. In the example we are reducing 7 columns. The ranges indicate the processors which contain the partial sums of the given indices. In each step of the reduce-to-row operation, each processor sends half its elements to its neighbor in the ith dimension and accumulates what it receives into the half it keeps. The process terminates when all dimensions are processed. (a) If processors contain > 1 final sum, then all processors have the same number of sums. (b) If all dimensions have not yet been crossed but processors have at most 1 partial sum, processors with higher addresses send their partial sum.

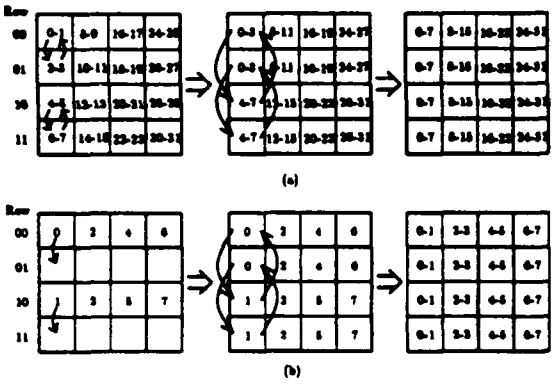

Figure 7: The distribute-row primitive replicates a row vector across a matrix. Communication proceeds across dimensions of the processor row address least significant bit to most significant bit. Each Processor sends all information to its neighbor and receives an equivalent amount of information, thus doubling data on each step. In case (b) where not all processors hold vector elements, the first steps replicate single values until all processors have one vector element. Then the number of values sent doubles on each successive step.

$\lg(\lceil p_1/k_2 \rceil)$ since processors must accumulate a local subtotal before cube swapping.

3.2 Analysis

In this section we argue that our algorithms for the reduce-to-row operations is within a constant factor of optimal in two ways. First we show that the parallel time (messages plus computation) required to execute algorithms reduce-to-row is within a constant factor of optimal. This is also true of the distribute-row implementation provided the embedding is nontrivial. Then we argue that the total work which is the product of parallel time (T) and number of processors (P) is within a constant factor of the work required for any sequential implementation of the reduce operation provided $k_1 k_2 \geq \lg p_1$. In other words, our algorithm is optimal if the number of matrix elements per processor is at least as large as the number of dimensions of the hypercube (in fact, a constant fraction smaller than $\lg n$ is also sufficient). To argue optimality in terms of time, we state lower bounds for the number of messages and operations required by any parallel implementation of reduce-to-row and compare them to the operation counts argued earlier in this section. We then compare minimum number of operations with the PT product of our implementation of reduce-to-row. To achieve the lower bounds we use a model where each processor can only send out one message in unit time.

3.2.1 Parallel Operation Count

To prove that our implementation of reduce-to-row performs at most a constant factor more than the optimal number of parallel operations, we first prove that any parallel algorithm for reduce-to-row must send at least $\Omega(\lg p)$ messages regardless of how the matrix elements are embedded in the hypercube. To complete the optimality proof for reduce-to-row, we finally show a lower bound of $k_1 k_2$ parallel arithmetic operations on an $N = p_1 p_2$ processor machine.

We now argue that any algorithm that computes reduce-to-row must send $\Omega(\lg p)$ messages. Let us assume that we have $m \geq p/2$, because otherwise we could run our algorithm on a subcube of the machine. Also, let us assume without loss of generality that $m_1 > m_2$, so that $m_1 > p/4$. Suppose some processor contains $\Omega(\lg p)$ elements from some column. Then it must accumulate at least half of them locally or send at least half of the elements to other processors, thus yielding the lower bound. Suppose, instead that no processor contains more than $\lg p$ elements of any one column. Then some column must be embedded in at least $m_1 / \lg p$ processors. Therefore, to accumulate the column requires $\lg m_1 - \lg \lg p \geq \lg(p/4) - \lg \lg p_1$ messages. Therefore, we must send $\Omega(\lg p)$ messages.

We now count the minimum number of parallel arithmetic steps on machine with $N = p_1 p_2$ processors. The number of arithmetic operations that must be performed is $m_2(m_1 - 1)$. Since there are only $p_1 p_2$ processors to perform that operation, we must use at least $(m_1 m_2 - m_2)/p_1 p_2 = \Omega(k_1 k_2)$ time.

Combining the separate lower bounds, we have that the minimum time to perform a reduce-to-row is $\Omega(\lg p + k_1 k_2)$. Since the number of messages plus operations executed by our implementation of reduce-to-row is $O(\lg p + k_1 k_2)$, our implementation is optimal in terms of asymptotic parallel execution time.

We can use arguments similar to those above to show an $\Omega(k_2 + \lg p)$ lower bound on message complexity for the distribute operation provided we require a processor holding e elements of a column to record e local copies. If the processors need only have one copy for each column of which it contains any elements, then we can show the lower bound provided the matrix is embedded nontrivially. By nontrivial, we mean that no one processor can contain more than $(1 - 1/p)m$ of the m matrix elements. The basic idea is that if a processor contains all or most of a column, then distributing within that column is trivial. It then, however, costs that processor considerably to distribute along rows.

3.2.2 Processor-Time Product

In this section we show that the product of arithmetic operations times number of processors is within a constant factor of the number of operations required by any sequential reduce-to-row implementation. As argued above, any sequential algorithm must perform

$m_2(m_1 - 1)$ arithmetic operations.

The PT product for reduce-to-row is

$$\begin{aligned} PT &= p_1 p_2 (k_1 k_2 + \lg(\lceil p_1/k_2 \rceil) - \lceil k_2/p_1 \rceil) \\ &= m_1 m_2 + p_1 p_2 \lg(\lceil p_1/k_2 \rceil) - p_1 p_2 \lceil k_2/p_1 \rceil \\ &\leq m_1 m_2 + p_1 p_2 k_1 k_2 - p_1 p_2 \lceil k_2/p_1 \rceil \end{aligned}$$

because we assume that $k_1 k_2 \geq \lg p_1$. Therefore we have that

$$\begin{aligned} PT &\leq 2m_1 m_2 - p_1 p_2 \lceil k_2/p_1 \rceil \\ &\leq 3m_1 m_2 - 3m_2. \end{aligned}$$

This proves that the total work as indicated by the processor-time product is within a constant factor of optimal provided that $k_1 k_2 \geq \lg p_1$.

3.3 Computing on a Single Row or Column

In this subsection we discuss the cost of computing on a single extracted row of a matrix. In particular we consider whether it is always worth load balancing rather than just moving the row, or column, locally within the processors that contain it. We consider four cases that might appear in practice:

1. We extract a row, operate on it, and distribute it across the other rows.
2. We extract a row, operate on it, and deposit it back in place.
3. We extract a row, operate on it, and deposit it back into another row.
4. We extract a row, operate on it, and throw the row away (for example, if we wanted to find the maximum in a row).

The load balancing advantage costs nothing in the first case. Since spreading a row across to all others is efficiently implemented as an extract followed by a distribute, we simply break the spread into its two pieces and operate on the load-balanced representation in the middle. The decision of whether or not to load balance the vector in the second and fourth cases should be a compile-time or run-time decision, and will depend on how many operations need to be performed, and the relative time of communication and computation. We have found in the applications we have studied that the first case occurs frequently, so it is therefore often worth load balancing when extracting a row. We analyze the second case to give an example of the considerations required for deciding whether it is worth load balancing or not.

In the second case we want to extract a row, operate on it, and put it back. Let a be the time it takes to perform a single arithmetic operation of interest (e.g. a divide) and let s be the time to send a matrix element over a hypercube wire. Let $q = \lceil k_2/p_1 \rceil$ be the maximum number of elements per processor after load balancing. Then the time it takes to compute without load balancing is $k_2 a$. If we extract first, compute and run extract in reverse, the cost of the entire computation is $2(k_2 - q)s$ for the messages plus qa for the computation. The load balancing is advantageous exactly when

$$\begin{aligned} k_2 a &> 2(k_2 - q)s + qa \\ (k_2 - q)a &> 2(k_2 - q)s \\ a &> 2s. \end{aligned}$$

Thus a compiler need only estimate the amount of time to perform the arithmetic and the amount of time to send a matrix element over a cube wire. If the compiler determines that it is not advantageous to do load balancing in such a situation, it can do a *lazy extract* which simply copies the row to a local array. If we want to store the result vector back to another row of the matrix, the decision is not entirely configuration independent since we must figure the cost of sending a row directly (worst case $\lg p_1$) vs. the cost of an extract plus deposit (worst case $2k_2 + \lg p_1$), making load balancing less favorable. In this analysis we assumed there is no pipeline startup cost for executing multiple arithmetic or communication steps. With a pipeline start up cost, the decision could not be made at compile time.

3.4 Extensions

In this subsection, we discuss extensions to higher dimensional matrices, matrices whose dimensions are not powers of 2, and matrices where rows or columns are favored. We also discuss ways to represent vectors in a canonical so that we no longer distinguish between row and column vectors.

If we have t-dimensional matrices, the address of each matrix element is now divided into t pieces corresponding to indices in each of the dimensions. All implementations will extend directly by operating on the appropriate set of address bits. For example, to extract in dimensions 1 and 2, use the concatenated bits of the processor addresses for these dimensions in place of the row address in algorithm extract.

For matrices where m_1 and m_2 are not powers of 2, we embed the matrix in a $p_1 \times p_2$ matrix such that each processor has at most $k_1 = \lceil m_1/p_1 \rceil$ rows and $k_2 = \lceil m_2/p_2 \rceil$ columns. If either k_1 or k_2 is not a power of 2, the first communication phase of an extract and the last communication phase of a deposit, etc, will have fewer messages than normal, namely $\lfloor k_i/2 \rfloor$. Otherwise, the implementations are unchanged.

If we wish to optimise operations on rows, taking a penalty for operations on columns, we can configure the processor grid such each processor holds a minimum number of elements from the same row (at most $\lceil m_2/p \rceil$).

Another possible extension is to store vectors in a canonical form. In our current implementation a vector

extracted from a row of a matrix (a row vector) will be ordered on the processors differently from a vector extracted from a column of a matrix (a column vector). It is possible to store all vectors in the same ordering, a canonical form, with no additional cost to our primitives. This, however, requires that the rows and columns of a matrix are mapped onto the processors in an noncontiguous order—one row of a matrix will no longer be adjacent to the next. This will make nearest neighbor communication on a grid more expensive. We believe that the better choice is to keep the two representations and swap between them when necessary. This could be hidden from the user.

4 Timings

We present the timings for each of the primitives on the CM-2 along with the timings for the vector matrix multiply and Simplex algorithms. The timings are for a square matrix in a 16384 processor machine running at 6.7 MHz. Timings reflect the total time that the CM-2 was busy. Each element of the matrix is a single precision (32 bit) floating point number. Timings for the full 65,536 processor machine are extrapolated from the figures for the smaller machine. We give the timings for different values of k_1 and k_2, the intraprocessor matrix dimensions. The times for more than 1 element per processor scale sublinearly with the total number of elements in each processor. The times are presented in milliseconds, and the flop rate in Mflops.

	Reduce			
Elements	16K		64K	
per proc	msec	Mflop	msec	Mflop
1 × 1	0.70	23	0.86	75
2 × 2	0.98	66	1.14	229
4 × 4	1.75	149	1.91	558
8 × 8	4.30	243	4.46	939
16 × 16	13.27	315	13.46	1246

	Distribute			
Elements	16K		64K	
per proc	msec	Mflop	msec	Mflop
1 × 1	0.44	-	0.44	-
2 × 2	0.69	-	0.69	-
4 × 4	1.30	-	1.30	-
8 × 8	3.09	-	3.09	-
16 × 16	8.81	-	8.92	-

	Extract			
Elements	16K		64K	
per proc	msec	Mflop	msec	Mflop
1 × 1	0.69	-	0.85	-
2 × 2	0.88	-	1.04	-
4 × 4	1.21	-	1.37	-
8 × 8	1.95	-	2.11	-
16 × 16	3.46	-	3.65	-

	Vector-Mtx Mpy			
Elements	16K		64K	
per proc	msec	Mflop	msec	Mflop
1 × 1	1.87	17	2.19	59
2 × 2	2.67	48	2.99	174
4 × 4	4.37	110	5.05	415
8 × 8	11.19	187	11.51	728
16 × 16	32.90	254	33.39	1004

We now provide an example of performance improvement from use of these primitives. A carefully coded naive implementation of Simplex, using $k_1 = 8$ and $k_2 = 8$ ran at a speed of 190 milliseconds per iteration, which is equivalent to 44 Mflops. The version using the primitives discussed in this paper takes 17 ms per iteration, which is equivalent to about 500 Mflops.

The distribute-row implementation used for the matrix-vector multiply timing propagated row values through the intraprocessor matrix. Performance can be improved by propagating only one copy of each value per processor. We estimate the time spent distributing the values across the intraprocessor matrix to be as high as 85% of the overhead for 16 × 16 intraprocessor matrices. Hence, we could gain considerable performance by implementing this optimization.

5 Conclusions

In this paper we discuss a set of powerful primitive matrix operations which allow easy specification of parallel matrix routines. We demonstrate via our hypercube implementation that the additional expressive power need not reduce performance and can, in fact, improve performance by providing automatic load balancing in the case where there are more matrix elements than processors. We describe some routines based on these primitives and other simple parallel operations and give some timings for the primitives and routines for our implementation on the Connection Machine.

In the future we hope to generalize our implementation of the primitives so they work on processor grids whose row and column sizes are not powers of two, and to allow a vector extracted from a row of a matrix to be distributed to or deposited in either a row or column of another matrix. We also hope to make our primitives available to higher level languages so that they can be easily used.

We hope that this paper spurs interest in developing a small set of simple matrix primitives which could then be efficiently implemented with a consistent interface on a large number of machines.

Acknowledgements

Thanks to Charles Leiserson of MIT who gave us comments on an early draft and to Lennart Johnsson of Thinking Machines Corporation and Yale who gave us a list of relevant references.

References

[1] Guy E. Blelloch. Applications and algorithms on the Connection Machine. TR87-1, Thinking Machines Corporation, Cambridge, MA, November 1987.

[2] Guy E. Blelloch. *Scan Primitives and Parallel Vector Models.* PhD thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, November 1988.

[3] Peter R. Capello. Gaussian elimination on a hypercube automation. *J. Parallel and Dist. Comput.*, 4:288–308, July 1987.

[4] Vladimir Cherkassky and Ross Smith. Efficient mapping and implementation of matrix algorithms on a hypercube. *The Journal of Supercomputing*, 2(1):7–27, September 1988.

[5] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, Princeton, NJ, 1963.

[6] Sanjay R. Deshpande and Roy M. Jenevin. Scaleability of a binary tree on a hypercube. In *1986 International Conference on Parallel Processing*, pages 661–668, IEEE Computer Society, 1986.

[7] Geoffrey C. Fox, S. W. Otto, and A. J. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4(1):7–31, 1987.

[8] Geoffrey C. Fox and Wojtek Furmanski. *Optimal communication algorithms on hypercube.* Technical Report CCCP-314, California Inst. of Technology, Pasadena, CA, July 1986.

[9] G. A. Geist and M. T. Heath. Matrix factorization on a hypercube multiprocessor. In *Proceedings of the First Conference on Hypercube Multiprocessors*, pages 161–180, August 1985.

[10] William D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[11] Kenneth E. Iverson. *A Programming Language.* Wiley, New York, 1962.

[12] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Comput.*, 4(2):133–172, April 1987.

[13] S. Lennart Johnsson and Ching-Tien Ho. *Spanning graphs for optimum broadcasting and personalized communication in hypercubes.* Technical Report YALEU/DCS/RR-500, Dept. of Computer Science, Yale Univ., New Haven, CT, November 1986. Revised November 1987, YALEU/DCS/RR-610. To appear in IEEE Trans. Computers.

[14] C. Moler. Matrix computation on a distributed memory multiprocessor. In *Proceedings of the First Conference on Hypercube Multiprocessors*, pages 161–180, August 1985.

[15] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes.* Cambridge University Press, Cambridge, 1986.

[17] Quentin F. Stout and Bruce Wager. Passing messages in link-bound hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.