VLSI DESIGN
1999, Vol. 9, No. 1, pp. 69–81
Reprints available directly from the publisher
Photocopying permitted by license only

Multiplication of Matrices with Different Sparseness Properties on Dynamically Reconfigurable Meshes

MARTIN MIDDENDORF [a,*], HARTMUT SCHMECK [a], HEIKO SCHRÖDER [b] and GAVIN TURNER [c]

[a] *Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, D-76128 Karlsruhe, Germany;*
[b] *Department of Computer Studies, University of Loughborough, Loughborough, LE11 3TU United Kingdom;*
[c] *Department of Computer Science, Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand*

(Received 5 May 1997)

Algorithms for multiplying several types of sparse $n \times n$-matrices on dynamically reconfigurable $n \times n$-arrays are presented. For some classes of sparse matrices constant time algorithms are given, *e.g.*, when the first matrix has at most kn elements in each column or in each row and the second matrix has at most kn nonzero elements in each row, where k is a constant. Moreover, $O(k\sqrt{n})$ algorithms are obtained for the case that one matrix is a general sparse matrix with at most kn nonzero elements and the other matrix has at most k nonzero elements in every row or in every column. Also a lower bound of $\Omega(\sqrt{kn})$ is proved for this and other cases which shows that the algorithms are close to the optimum.

Keywords: Matrix multiplication, sparse matrices, reconfigurable arrays

1. INTRODUCTION

Mesh-connected arrays with dynamically reconfigurable buses have gained considerable attention recently because of their ability to perform a number of interesting operations significantly faster than other standard computational models [8]. In particular, constant time algorithms have been designed for problems which need nonconstant time on a CRCW-PRAM with bounded fan-in (*e.g.*, for the parity function). This is achieved by appropriately transferring major parts of the computation into the reconfiguration of buses. Problems studied so far include semigroup and parallel prefix computations, sorting, binary addition, graph problems, and image processing.

In [12] Park *et al.*, present a constant time algorithm for computing the product of $n \times n$-matrices on an $n^2 \times n^2$ reconfigurable mesh. Although this is optimal with respect to the VLSI-complexity measure AT^2 [13], such an algorithm has only little practical relevance, since it can only be realised reasonably for relatively small matrices. But for small matrices, the "con-

*Corresponding author. Fax: + +49-721-693717, e-mail: {mmi, hsch}@aifb.uni-karlsruhe.de

stant" time is prohibitively large. On $n \times n$-arrays we have systolic algorithms for multiplying $n \times n$-matrices in time $O(n)$ which is AT^2-optimal again (see *e.g.*, [14]) and of significant practical relevance. Therefore, on $n \times n$ arrays we cannot hope to get better time performance by using reconfigurable buses. For sparse matrices, though, we have a different situation, since the lower bound for AT^2 no longer applies. If the number of nonzero elements is $O(n)$, the number of arithmetic operations for matrix multiplication is reduced to at most $O(n^2)$, *i.e.*, one should hope for algorithms with improved time performance on $n \times n$-arrays.

In this paper we present constant time algorithms for multiplying various types of sparse $n \times n$-matrices on reconfigurable $n \times n$-arrays. This is an improvement over the algorithm of Middendorf *el al.* [7] which achieves constant time only for a very restricted class of sparse matrices. Moreover, we give $O(k\sqrt{n})$ algorithms for the case that one matrix is a general sparse matrix with at most kn nonzero elements and the other matrix has at most k nonzero elements in every row or every column. We also derive a lower bound of $\Omega(\sqrt{kn})$ for these cases which shows that our algorithms are not far from the optimum. Our algorithms are faster than the algorithm of ElGindy [2] who gave an $O(k^2\sqrt{n})$ algorithm for the special case that the first matrix has at most k nonzero elements in each column and the second matrix has at most k nonzero elements in every row.

Our algorithms should also be compared with those of Kruskal, Rudolph, and Snir [3] and Manzini [6], who essentially give $O((n/p)\log p)$ time algorithms for sparse matrix multiplication on PRAM's and hypercubes with p processors.

2. MODEL OF COMPUTATION

Our model of computation is an SIMD $n \times n$-array of processing elements (PE's) with dynamically reconfigurable buses as depicted in Figure 1. Various possibilities for configuring the buses are

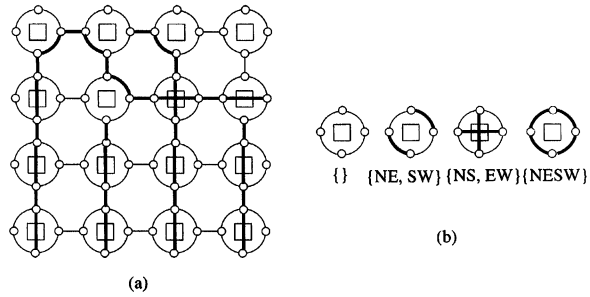


FIGURE 1 Reconfigurable mesh-connected array.

indicated by using thick lines. The bus switch setting for a particular PE is described by a set of strings like {NS, EW} for simultaneous north-south and east-west connection (see Fig. 1(b)). Except for minor modifications it is enough to assume for all algorithms in this paper that every PE can connect at most two of its ports at a time.

Every PE can read from every bus it is connected to, but only one PE at a time can write the value of one of its registers on a bus, *i.e.*, we have CREW-buses. If no PE writes on a bus then its value is 0. Every PE has a constant number of registers of length $(1+\varepsilon)\log n$ (Note that we do not allow that the number of registers depends on the parameters of the problem instances). Every PE knows its row and column indices. Within one time step every PE can locally configure the bus, write to and/or read from one of the buses it is connected to, and perform some local computation. Signal propagation on buses is assumed to take constant time regardless of the number of switches on the bus. This is the standard assumption for this model of computation, some technological justification is given in [5].

3. SPARSE MATRICES

Let A be an $n \times n$-matrix. As is obvious from looking *e.g.*, at [3], [6] and [11], there is no generally agreed upon definition of the sparseness of A, although the minimal requirement is that the number of nonzero elements is significantly less

than n^2. We use the characterisation introduced in [7].

Let $r_A(c_A)$ denote the maximal number of nonzero elements per row (column) of A and let k_A be the smallest integer such that the number of nonzero elements in A is at most $k_A \cdot n$. Then A is called

- *weakly sparse* iff $k_A \in O(1)$,
- *row sparse* iff $r_A \in O(1)$,
- *column sparse* iff $c_A \in O(1)$,
- *uniformly sparse* iff it is row and column sparse.

Furthermore, we assume that the number of nonzero elements of sparse $n \times n$-matrices is $\Omega(n)$, *i.e.*, it is not too small.

The main reason for studying sparse matrices is that they occur quite frequently in numerical computations and that the time complexity of these computations is often determined by the complexity of operations on matrices. Therefore, it would be very profitable to have reduced time and space complexities for operations on sparse matrices compared with full matrices. In particular, the multiplication of weakly sparse matrices needs at most $O(n^2)$ arithmetic operations. If both matrices are uniformly sparse, this is even further reduced to $O(n)$ and the product matrix is uniformly sparse, too. Systolic algorithms for matrix multiplication on mesh-connected $n \times n$-arrays cannot exploit the potential sparseness of its operands. As we show in the following section, the situation is different for arrays with reconfigurable buses.

4. MULTIPLICATION OF SPARSE MATRICES

In this section we present several algorithms for the multiplication of sparse $n \times n$ matrices A and B on reconfigurable arrays. While the algorithm of Middendorf *et al.* [7] achieves constant time multiplication only for uniformly sparse matrices, the algorithms presented below achieve constant time for a much wider range of sparse matrices. In particular the algorithms need time $O(c_A \cdot c_B)$ if both A and B are column sparse, $O(r_A \cdot r_B)$ if A and B are row sparse, and $O(r_A \cdot c_B)$ if A is row sparse and B is column sparse. For the case that A is column sparse and B is row sparse we show a lower bound of $\Omega(\sqrt{\max\{c_A, r_B\} \cdot n})$. ElGindy [2] gave an $O(c_A \cdot r_B \cdot \sqrt{n})$ algorithm for this case. We obtain a faster algorithm with time $O(\sqrt{c_A \cdot r_B \cdot n})$. But this algorithm is only a special case of more general algorithms described here for the case that one of the matrices is weakly sparse. These algorithms need time $O(\sqrt{k_A \cdot c_B \cdot n})$, $O(\sqrt{k_A \cdot r_B \cdot n})$, $O(\sqrt{c_A \cdot k_B \cdot n})$, and $O(\sqrt{r_A \cdot k_B \cdot n})$, respectively.

Initially, the elements a_{ij} and b_{ij} of matrices A and B are stored in processing elements (i,j) of the array. This situation may occur whenever A and B are the result of some previous computation performed on the array. We have to compute the product $C = A \cdot B$ such that afterwards element c_{ij} of C is stored in PE(i,j). In the following we refer to elements of matrices A, B, and C stored in a PE as A-, B-, and C-elements.

For the first algorithm we assume that A and B are column sparse, *i.e.*, $c_A, c_B \in O(1)$. The algorithm is illustrated in Figure 2.

Algorithm CC

In all the PEs initialise the C-element to 0.

REPEAT

1) FOR k:=1 TO n PARDO

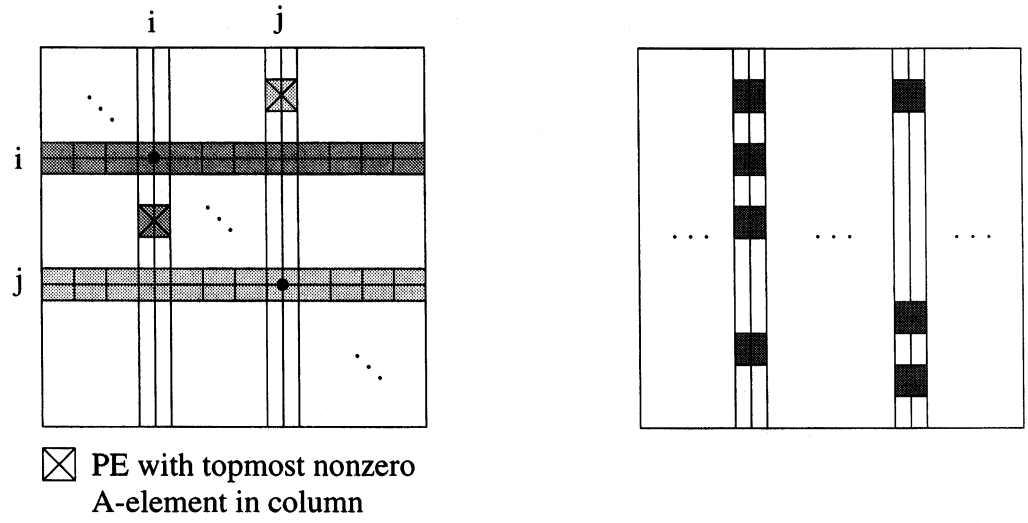
   Broadcast the topmost nonzero A-element of column k together with its row index to all the PE's in row k (*i.e.*, configure all PEs (i, i) {NESW} and all other PEs {NS, EW}).

   PAREND

2) FOR all PEs PARDO

   Multiply the broadcasted A-element a with the local B-element b, *i.e.*, compute a product $p = a \cdot b$

   PAREND



a) Step 1 of CC: Send topmost nonzero A-element of column i to all PE's in row i, i=1,...,n

b) Step 3 of CC: In each column at most c_B p's have to be send to final row

FIGURE 2 Illustration of Algorithm CC.

3) Route the maximally c_B p's of each column along column buses to their final row and add them to the current C-values.

   Discard the topmost nonzero A-element of each column (*i.e.*, the next element becomes the new topmost element).

UNTIL all the nonzero A-elements have been discarded.

Steps (1) and (2) take constant time. Since there are at most c_B nonzero p's in every column, Step (3) needs at most time $O(c_B)$ by making use of the reconfigurable buses in a straightforward way. Therefore, the time complexity of Algorithm CC is $O(c_A \cdot c_B)$.

To show the correctness of the algorithm, consider one iteration of the loop: Let a_{ik} be the current topmost nonzero A-element in column k. a_{ik} is broadcast to all the PE's in row k and subsequently multiplied with all the nonzero B-elements in this row, *e.g.*, with some b_{kj}. The product $a_{ik} \cdot b_{kj}$ is then moved to row i and added to the current value of c_{ij}. In this way all the scalar products c_{ij} are computed correctly. Observe, that each column of the product matrix may contain $c_A \cdot c_B$ nonzero elements. Hence, it seems difficult to improve on the running time of Algorithm CC.

Obviously, the algorithm can easily be adapted to the case of row sparse matrices A and B, where it would need time $O(r_A \cdot r_B)$.

If A is row sparse and B is column sparse, we suggest to use the following algorithm:

Algorithm RC

In all the PE's initialise the C-element to 0.

REPEAT

1) Broadcast the topmost nonzero B-element of every column together with its row index over its column.

   REPEAT

   2) Broadcast the leftmost nonzero A-element of every row together with its column index over its row.

   3) FOR all PEs PARDO

      If the PE received matching indices, multiply the broadcasted A-

and B-elements and add the product to the local C-element.

PAREND

4) Discard the leftmost nonzero A-element of every row.

UNTIL all the nonzero A-elements have been discarded.

5) Discard the topmost nonzero B-element of each column and "reactivate" all the nonzero A-elements.

UNTIL all the nonzero B-elements have been discarded.

All the Steps (1) to (5) need constant time each. Since Steps (2), (3), and (4) are repeated r_A times for every iteration of the outer loop, the total time of Algorithm RC is $O(r_A \cdot c_B)$. This could be reduced to $O(r_A + c_B)$, if all the nonzero elements of a column of B could be stored in every PE of this column. But since we do not allow the number of registers of a PE to depend on problem parameters r_A or c_B, this can not be done in general.

The correctness of Algorithm RC can be seen as follows:

Let b_{kj} be the topmost nonzero B-element of column j. After Step (1) this element is known to all the PE's in column j. Let a_{ik} be the leftmost nonzero A-element of row i. In Step (3) this element is multiplied with all the B-elements broadcasted in Step (1) having row index k. In particular, a_{ik} will be multiplied with b_{kj}, and the product is added correctly to the C-element at this PE(i, j). When all the A-elements have been discarded, the B-elements broadcasted in Step (1) have been combined with all matching A-elements. Therefore, these B-elements are discarded (in Step (5)), whereas the A-elements have to be reactivated to be combined with the B-elements broadcasted in the next execution of Step (1). In this way, all the necessary computations of the matrix multiplication are performed correctly.

Except for the multiple broadcast of matrix A Algorithm RC could be seen as an asynchronous version of one of the standard systolic algorithms for matrix multiplication, where matrices A and B are moved simultaneously over the rows and columns of the array, respectively.

The multiplication of a column sparse matrix A with a row sparse matrix B is considerably more difficult: If we try to use Algorithm CC, we have to move c_B products to their final positions (in Step (3)), but there may be $\Omega(n)$ nonzero products. In Algorithm RC we would even end up with a time of $O(n^2)$. The following lower bound result implies that any algorithm for the multiplication of a column sparse matrix A with a row sparse matrix B needs time $\Omega(\sqrt{\max\{c_A, r_B\} \cdot n})$. This improves the first lower bound of $\Omega(\sqrt{n})$ for this case given in [10]. The idea for the improvement is due to Kunde [4].

THEOREM *Any algorithm for the multiplication of a column sparse matrix A and a row sparse matrix B on an $n \times n$-array with reconfigurable buses needs in the worst case time $\Omega(\sqrt{r_B \cdot n})$, even if $c_A = 1$, and $\Omega(\sqrt{c_A \cdot n})$, even if $r_B = 1$.*

Proof For the case $c_A = 1$ consider the two matrices depicted in Figure 3: A has exactly one nonzero element (which is 1) in every column and n nonzero elements altogether. B has r_B nonzero elements in every row and $r_B \cdot n$ nonzero elements altogether. If the nonzero elements of B are pairwise different, then, in the product matrix C, for every c_{ij} with $i, j \leq \sqrt{r_B \cdot n}$ there is exactly one nonzero B-element $b_{kj} = c_{ij}$. Therefore, all these $r_B \cdot n$ elements of C are nonzero. To compute these we have to move at least $\Omega(r_B \cdot n)$ elements of B into an area of the array having a boundary of length $O(\sqrt{r_B \cdot n})$. This takes at least time $\Omega(\sqrt{r_B \cdot n})$. The case $r_B = 1$ is similar and is omitted. ■

In the rest of this paper we consider the case that one of the matrices is only weakly sparse. First we get the following lower bound result.

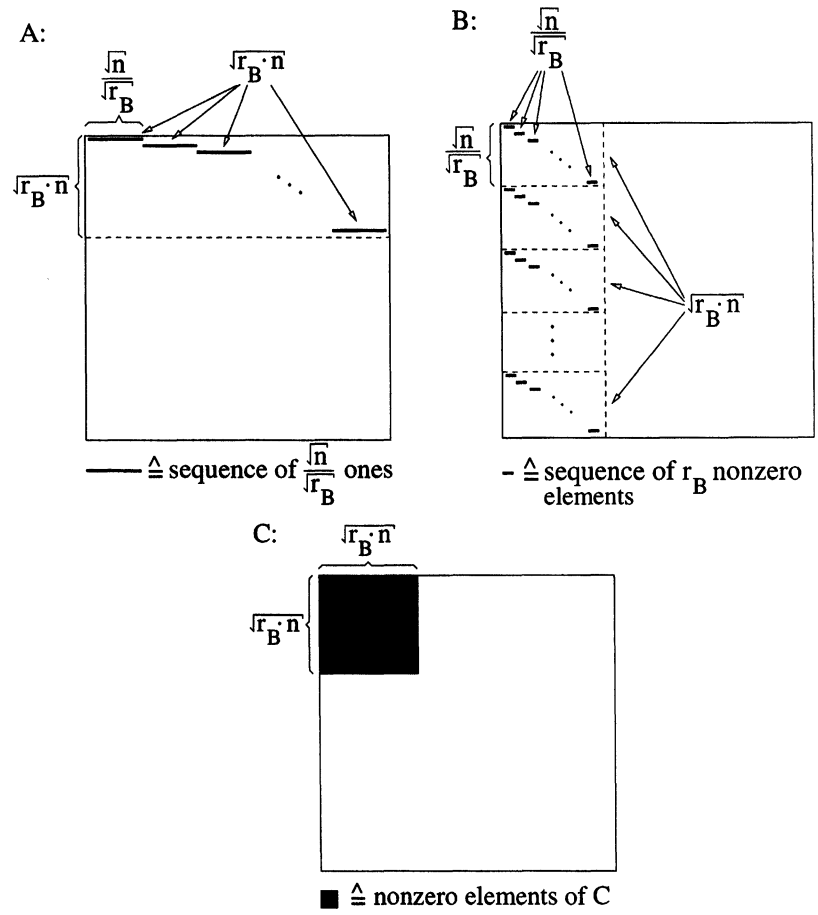


FIGURE 3 Worst case matrices A, B, and C.

COROLLARY *On an $n \times n$ array with reconfigurable buses we have the following lower bounds for the multiplication of weakly sparse matrices:*

(*a*) *for A weakly sparse and B row sparse we need time $\Omega(\sqrt{max\{k_A, r_B\} \cdot n})$ in the worst case*

(*b*) *for A column sparse and B weakly sparse we need time $\Omega(\sqrt{max\{c_A, k_B\} \cdot n})$ in the worst case*

(*c*) *for A weakly sparse and B uniformly sparse we need time $\Omega(\sqrt{k_A \cdot n})$ in the worst case even if $r_B = c_B = 1$*

(*d*) *for A uniformly sparse and B weakly sparse we need time $\Omega(\sqrt{k_B \cdot n})$ in the worst case even if $r_A = c_A = 1$.*

Proof We obtain (a) and (b) immediately from our theorem. For the Case (c) consider the matrices depicted in Figure 4. A similar argument as in the proof of the theorem shows the lower bound. Case (d) is easily obtained from Case (c). ■

Now we give algorithms for the case that one of the matrices is only weakly sparse. For the first algorithm we assume that A is weakly sparse and B is row sparse. A special case of this is that A is column sparse.

For ease of description we assume that A has exactly $k_A \cdot n$ nonzero elements, k_A, r_B and c_B divide n, and $\sqrt{k_A \cdot n}$, $\sqrt{k_A \cdot r_B \cdot n}$ are integers. Let M'

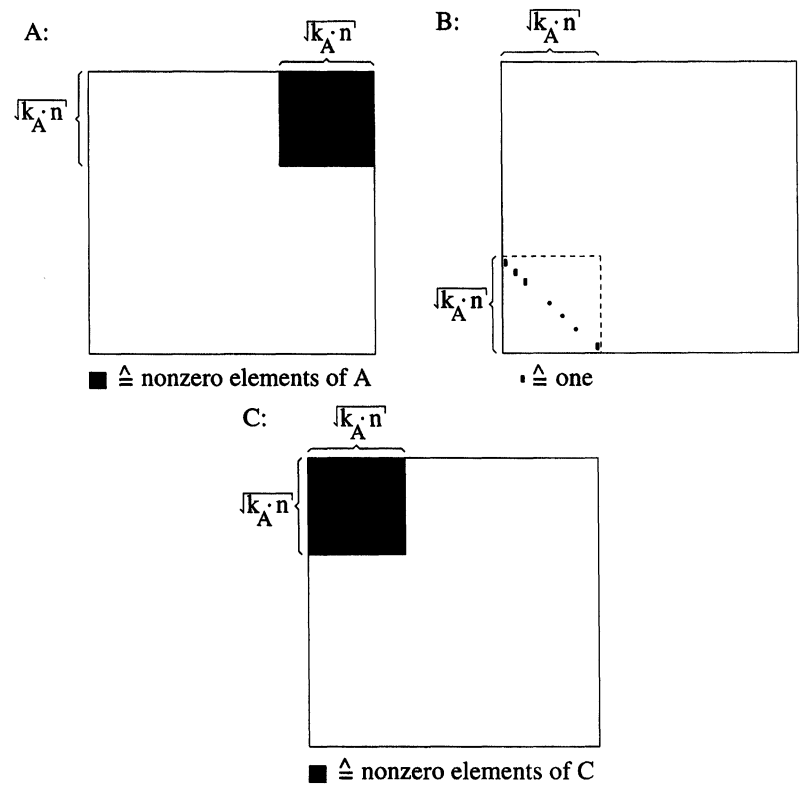


FIGURE 4 Worst case matrices A, B, and C.

be the submesh of the reconfigurable mesh M that consists of all PE's in columns $1 + i \cdot r_B$ for $i \in [0, k_A - 1]$. We have $k_A \cdot r_B < n$ (otherwise contradiction to assumption k_A, $r_B \in O(1)$). The reader is referred here to the example given after the description of the algorithm.

Algorithm WR

(1) Rearrange the nonzero A-elements such that each processor in the submesh M' contains one nonzero A-element.
(2) Sort the nonzero A-elements in lexicographic order with respect to their indices such that they are in row-major order in the submesh M'.
(3) Mark each PE(i, j) if there is a nonzero A-element with column index j in row i of the mesh.
(4) Transpose matrix B.
(5) Compute all products $a_{pq} \cdot b_{qr} \neq 0$, $p, q, r \in [1, n]$ as follows:

FOR $j = 1$ TO r_B DO

In each column of the mesh send the jth nonzero B-element (if present) to all marked PE's in the column.

FOR $i = 1$ TO k_A DO

In each row of the mesh send the B-element that has been received by the ith marked PE, say b_{qr}, to all PE's in the row that contain an A-element with column index q. Compute the product of the A-element with the received B-element and send the product to a PE in the same row that has not yet received any such product. /⋆ Observe, that this is possible since there are $r_B - 1$ PE's between any two PE's of M' in each row of the mesh. ⋆/

END

END

(6) In each row of the mesh sum all products with the same destination.

(7) Identify each index for which there exist products in more than one row of the mesh which have this index as their final row index. Call each such index *distributed.*

/* Observe that products with the same distributed row index occur in neighboring rows of the mesh. Also, each row of the mesh contains products with at most two different distributed row indices. Note also that all products with a non-distributed row index already represent a C-element. */

(8) In each row of the mesh send the products with a distributed row index into the final column.

/* Note, there are at most two such products with the same column index in each row. */

(9) In each column of the mesh sum all products (with a distributed row index) that have the same destination.

/* Now all elements of C have been computed and it remains to send them to their final destination. */

(10) Route all C-elements with a non-distributed row index as follows:

FOR $i = 1$ TO $k_A \cdot r_B$ DO

In each row of the mesh send the ith C-element in the row that has a non-distributed row index in three steps to its final destination:

Send it on a row bus to the diagonal PE.
Send it on a column bus to its final row.
Send it on a row bus to its final column.

END

(11) In each column of the mesh mark the $\sqrt{k_A \cdot r_B \cdot n}$ uppermost C-elements with a distributed row index as *white* and all other C-elements with a distributed row index as *black*.

(12) FOR $i=1$ TO $\sqrt{k_A \cdot r_B \cdot n}$ DO

In each column of the mesh route the ith white C-element to its final row.

END

/* Since C contains at most $k_A \cdot r_B \cdot n$ nonzero elements there can be at most $t \leq \sqrt{k_A \cdot r_B \cdot n}$ columns $j_1, j_2, \ldots, j_t$ of the mesh containing a black element. */

(13) FOR $i=1$ TO $\sqrt{k_A \cdot r_B \cdot n}$ DO

Route the black elements in column j_i (if defined) in three steps to their final destinations:

Send them on a row bus to the diagonal PE.
Send them on a column bus to their final row.
Send them on a row bus to their final column.

END

Before we analyse the algorithm we give an example that illustrates how the algorithm works. Let A and B be the matrices presented in Figure 5. A has $4n-1$ nonzero elements and is weakly sparse with $k_A = 4$. B is row sparse with $r_B = 5$. Hence, the submesh M' consists of the processors in columns 1, 6, 10, and 16. Figure 6 shows the distribution of the nonzero A-elements after Step (2), *i.e.*, they have been send to the processors of M' and sorted in row major order in submesh M'. Figure 7 shows for exemplary rows 1 and 4 which processors have been marked in Step (3) and which nonzero B-elements are sent to the marked processors in Step (5). The products that are computed in these rows during Step (5) are depicted in Figure 8(a). The situation after Step (6) – when for every row products with the same destination are summed up – is presented for rows 1 and 4 in Figure 8(b). Products with a distributed row index are identified in Step (7) which is depicted in Figure 9. During Step (8) products with a distributed row index are sent to their final column (see Fig. 10(a)). Then, during Step (9), in each column products with the same destination are summed (see Fig. 10(b)). Afterwards, in Step (10), products with a non-distributed row index are routed to their final destination. In Step (11) the $\sqrt{k_A \cdot r_B \cdot n} = \sqrt{20 \cdot n}$ uppermost C-elements

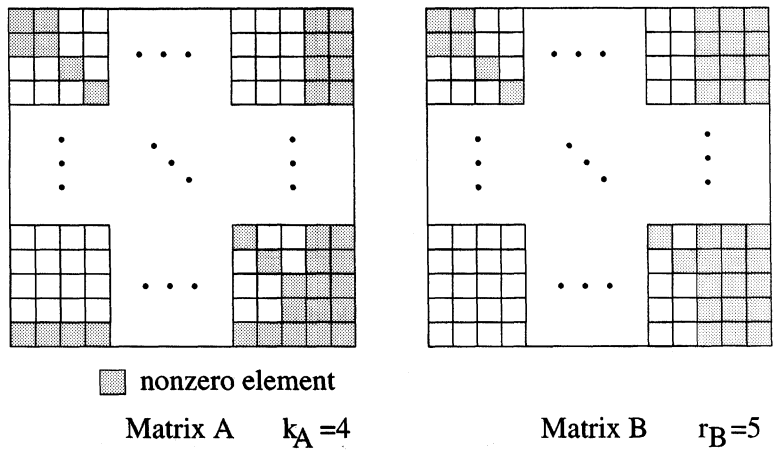


FIGURE 5 Matrices A and B of the example for Algorithm WR.

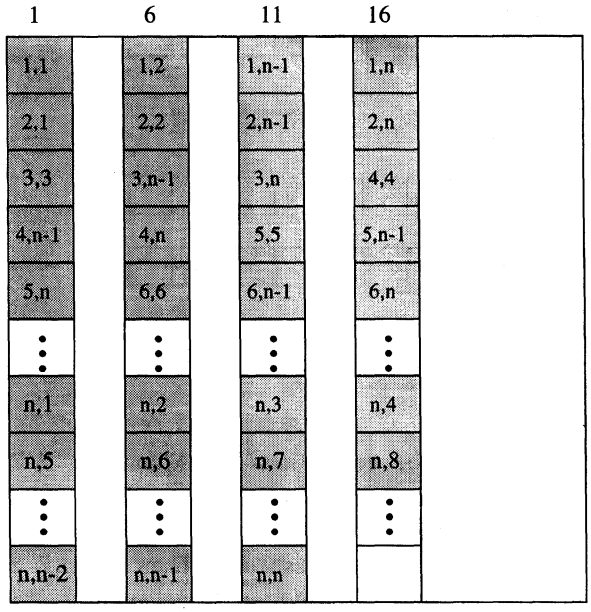


FIGURE 6 After Step 2: Nonzero A-elements in submesh M'.

with distributed row index are marked white. Observe that there are at least $((n-3)/2)$ C-elements with a distributed row index in each of the columns $n-2$, $n-1$, and n. Thus columns $n-2$, $n-1$, and n contain black elements if $((n-3)/2) > \sqrt{20 \cdot n}$. Finally, the white C-elements are routed during Step (12) and the black ones during Step (13).

Now let us analyse the time complexity of Algorithm WR. Step (1) will take time $O(\sqrt{k_A \cdot n})$ using standard operations for the reconfigurable mesh. We give a sketch: Let M'' be the submesh of M containing all processors not in M'. Perform at most $\sqrt{k_a \cdot n}$ times the following two steps: (i) Identify the leftmost nonzero element of A in each column of M'' and send these elements to different processors of M' that don't contain a nonzero element of A. (ii) Similar to (i) but with the topmost nonzero elements of A in each row of M''. For Step (2) recall that sorting n numbers on the reconfigurable mesh can be done in time $O(1)$ [9]. To sort the $k_A \cdot n$ numbers stored in M' repeat $\log k_A$ times the following steps:

a. For $i = 1$ to k_A: Sort all numbers in column i of the submesh M' from the bottom if i is odd and from the top if i is even;
b. In parallel sort the numbers in each row of M' with k_A steps of odd-even transposition sort (see *e.g.* [1]).

As a result the $k_A \cdot n$ numbers are sorted columnwise in M'. Now, we give each number an additional index in such a way that after sorting the numbers again but this time according to the added indices, the numbers will be sorted rowwise in M'. Altogether, this sorting takes time $O(k_A \cdot \log k_A)$. Steps (3), (4), and (5) need time

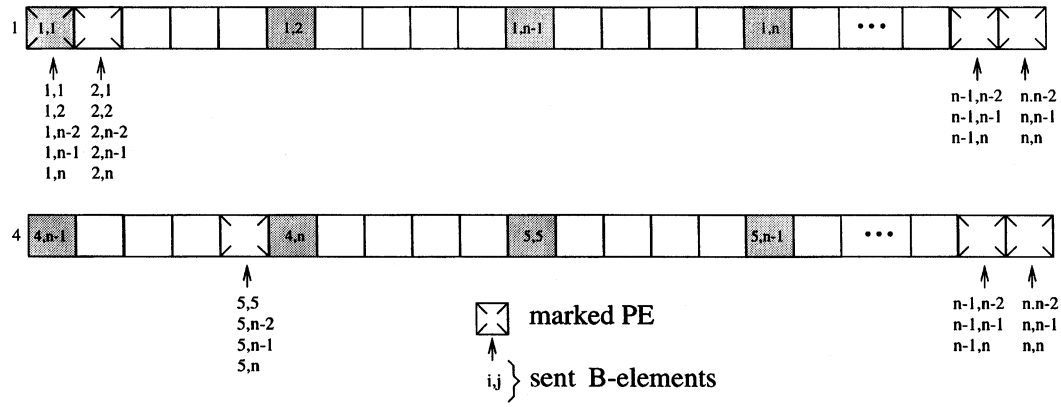
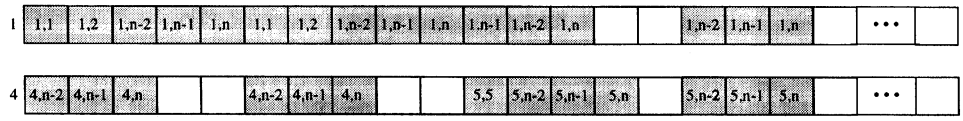
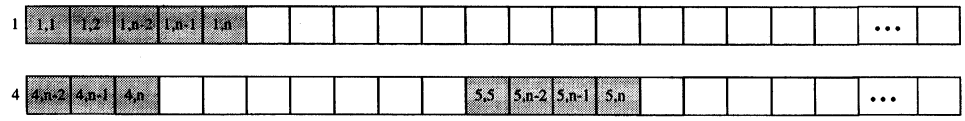


FIGURE 7 Nonzero A-elements and sent B-elements in rows 1 and 4 during Step 5.



a) After step 5



b) After step 6

FIGURE 8 Computed products in rows 1 and 4 after Steps 5 and 6.

$O(k_A)$, $O(r_B)$, $O(r_B \cdot k_A)$, respectively. Step (6) can be done in time $O(k_A \cdot r_B)$ since in every row there are at most k_A products with the same destination. Step (7) needs time $O(1)$. Since at most $k_A \cdot r_B$ products are in each row Step (8) can be done in time $O(k_a \cdot r_B)$. For Step (9) time $O(\log(n/k_A))$ is enough, since products with the same destination can occur in at most $(n/k_A)+1$ neighboring rows of the mesh. Step (10) takes time $O(k_A \cdot r_B)$. Each of the final Steps (11), (12), (13) needs time $O(\sqrt{k_A \cdot r_B \cdot n})$. Altogether we see that the total running time of Algorithm WR is $O(\sqrt{k_A \cdot n_B \cdot n})$. This shows that the algorithm is not far from the lower bound. In particular, it is faster and more general than the algorithm of ElGindy [2].

To show the correctness of Algorithm WR, consider two nonzero elements $a_{i,j}$ and $b_{j,h}$. After Steps (1) and (2) element $a_{i,j}$ is stored in some PE(p, q) of the submesh M'. Then PE(p,j) is marked in Step (3). In Step (4) element $b_{j,h}$ is moved from PE(j,h) to PE(h,j). In Step (5) $b_{j,h}$ is first sent over a column bus to PE(p,j), since it is marked. Then, $b_{j,h}$ is sent to PE(p, q), multiplied with $a_{i,j}$, and the product is sent to another PE, say (p, r), in the row. In Step (8) other products stored in row p, which have the same destination, are added and the combined product is stored in a PE(p,s). First let us assume that i is a non-distributed row index. Then the product already equals C-element $c_{i,h}$ and is routed in Step (10) to its final destination (it is sent first to PE(p,p), then

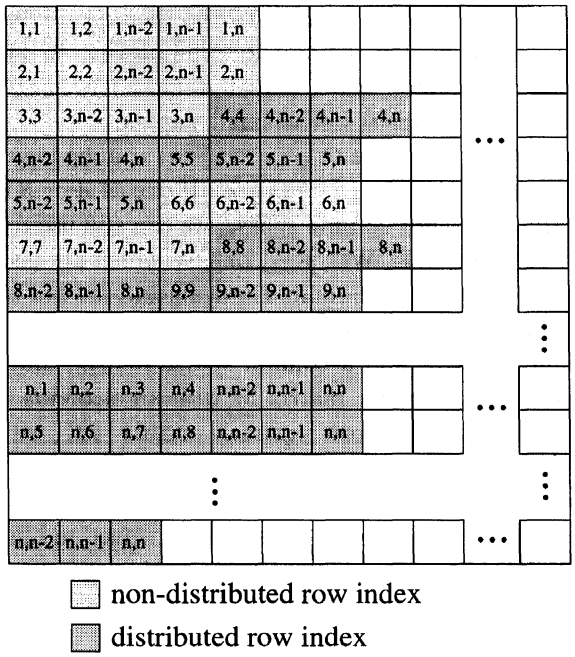


FIGURE 9 Products after Step 7.

to PE(i,p), and finally to PE(i, h)). On the other hand, assume that p is a distributed row index. Then the product is sent to PE(p,h) in Step (8). In Step (9) it is added to all other products with the same destination. Afterwards it equals C-element $c_{i,h}$ and is stored in a PE in column h, say PE(o,h). If $c_{i,h}$ is one of the $\sqrt{k_A \cdot r_B \cdot n}$. uppermost C-elements in row h, then it is sent in Step (12) directly to its destination which is PE(i,h). Otherwise, it is routed in Step (13) to PE(i,h) as follows: First it is sent to PE(o,o), then to PE(i,o), and finally to PE(i,h).

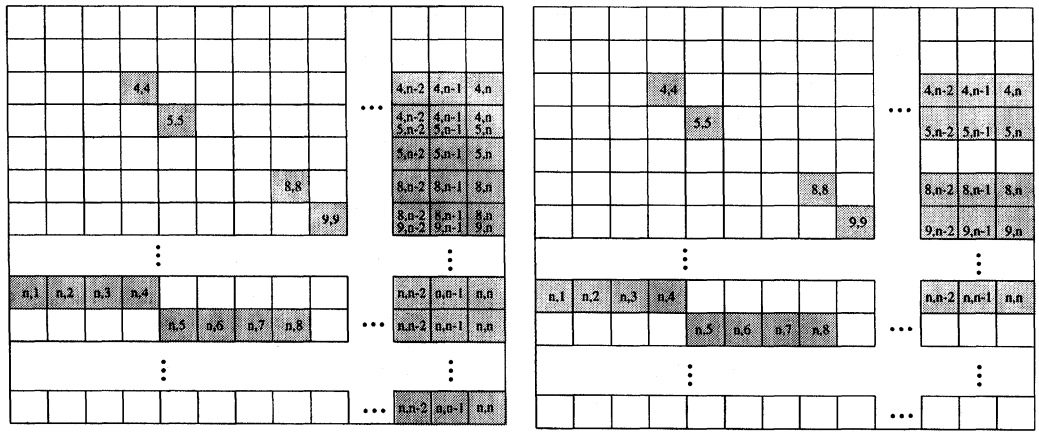
It is easy to obtain an Algorithm CW with time $O(\sqrt{c_A \cdot k_B \cdot n})$ for the case that A is column sparse and B is weakly sparse (Hint: Essentially interchange in Algorithm WR A with B and column with row).

The next Algorithm WC for the case that A is weakly sparse again but B is column sparse is simpler than Algorithm WR.

Algorithm WC

In all the PE's initialise the C-element to 0.

1) Determine the columns $j_1, j_2, \ldots, j_t$ of A with at least $\sqrt{k_A \cdot n}/\sqrt{c_B}$ nonzero elements and let $h_1, h_2, \ldots, h_{t'}$ be the other columns of A. Observe, that $t \leq \sqrt{k_A \cdot c_B \cdot n}$ must hold. Let $A_1(A_2)$ be the $n \times n$ matrix that match on A in columns $j_1, j_2, \ldots, j_t$ ($h_1, h_2, \ldots, h_{t'}$) and has only zero elements in all other columns. Thus, we have $A = A_1 + A_2$.



a) After step 8 b) After step 9

FIGURE 10 Elements with distributed row index during Steps 8 and 9.

2) Compute $A_1 \times B$ as follows:

   FOR $i = 1$ TO t DO

   Send each element of column j_1 of A_1 to all PE's in its row.
   Send each element of row j_1 of B to all PE's in its column.
   In each PE that has received an A_1-element and a B-element multiply both elements and add the result to the current local C-value.

   END

3) Determine $A_2 \times B$ with Algorithm CC (but without initialising the local C-elements to zero).

The columns $j_1, j_2, \ldots, j_t$ can be identified easily in time $O(\sqrt{k_A \cdot n}/\sqrt{c_B})$. Step (1) takes time $O(t) \leq O(\sqrt{k_A \cdot c_B \cdot n})$. Step (2) takes time $O(c_{A_2} \cdot c_B) = O((\sqrt{k_A \cdot n}/\sqrt{c_B}) \cdot c_B)$. Therefore the time complexity of Algorithm WC is $O(\sqrt{k_A \cdot c_B \cdot n})$.

For the correctness consider nonzero elements $a_{i,j}$ and b_j, h. If $j \in \{j_1, j_2, \ldots, j_t\}$ then $a_{i,j}$ is sent in Step (1) to all PE's in row i and $b_{j,h}$ is sent to all PE's in column h. Hence, PE(i, h) correctly receives both elements, multiplies them, and adds the result to the local C-value. If, on the other hand, $j \notin \{j_1, j_2, \ldots, j_t\}$ then $a_{i,j}$ is multiplied in Step (2) by Algorithm CC with $b_{j,h}$ and added to the corresponding C-value.

One easily obtains an Algorithm RW with time $O(\sqrt{r_A \cdot k_B \cdot n})$ for the case that A is row sparse and B is weakly sparse, since this is just the transposed case.

5. CONCLUSION

In this paper we have described several algorithms for multiplying different types of weakly sparse matrices on dynamically reconfigurable arrays. Our constant time algorithms could be combined into a polyalgorithm which would check initially the values of c_A, r_A, and c_B, r_B and then choose the algorithm having the smallest complexity. In the case of a column sparse matrix A and a row sparse matrix B there is still a small gap between our algorithm with time $O(\sqrt{c_A \cdot r_B \cdot n})$ and the lower bound of $\Omega(\sqrt{\max\{c_A, r_B\} \cdot n})$. If one of A or B is neither row nor column sparse, we can still use the Algorithms WR, WC, RW, or CW which are also close to the lower bound. These algorithms can be used as long as at least one of the matrices is the sum of a row and a column sparse matrix. Since each weakly sparse matrix M with kn elements is the sum of a "row sparse" matrix M' with $r_{M'} = \sqrt{k \cdot n}$ and a "column sparse" matrix M'' with $c_{M''} = \sqrt{k \cdot n}$ the case that both matrices A and B are only weakly sparse can be solved in time $O(\min\{k_A^{1/2} \cdot k_B^{1/4}, k_A^{1/4} \cdot k_B^{1/2}\} \cdot n^{3/4})$.

References

[1] Akl, S. G. (1989). *The design and analysis of parallel algorithms*, Englewood Cliffs: Prentice-Hall.

[2] ElGindy, H. (1996). "A sparse matrix multiplication algorithm for the reconfigurable mesh architecture", Technical Report 96-08, Dept. of Comp. Sci. and Software Eng., University of Newcastle, Australia.

[3] Kruskal, C. P., Rudolph, L. and Snir, M. (1989). "Techniques for parallel manipulation of sparse matrices", *Theoret. Comput. Sci.*, **64**, 135–157.

[4] Kunde, M., Personal communication.

[5] Lie, K. T. and Schröder, H., "A fault tolerant reconfigurable array", *Pacific Rim Conference on Fault Tolerant Systems*, December 1993.

[6] Manzini, G. (1994). "Sparse matrix vector multiplication on distributed architectures: Lower bounds and average complexity results", *Inform. Process. Lett.*, **50**, 231–238.

[7] Middendorf, M., Schmeck, H. and Turner, G. (1995). "Sparse Matrix Multiplication on a Reconfigurable Mesh", *Australian Computer Journal*, **27**, 37–40.

[8] Miller, R., Prasanna-Kumar, V. K., Reisis, D. I. and Stout, Q. F. (1993). "Parallel Computations on Reconfigurable Meshes", *IEEE Trans. Comput.*, **42**, 678–692.

[9] Nigam, M. and Sahni, S. (1995). Sorting n numbers on $n \times n$ reconfigurable meshes with busses", *J. Par. Distr. Comput.*, **23**, 37–48.

[10] Schmeck, H., Schröder, H. and Turner, G. (1995). "Efficient sparse matrix multiplication on a reconfigurable mesh", *Mitteilungen–Gesellschaft für Informatik e. V., Parallel-Algorithmen und Rechnerstrukturen*, **13**, 89–96.

[11] Østerby, O. and Zlatev, Z. (1983). *Direct Methods for Sparse Matrices*, Berlin, Heidelberg, New York: Springer-Verlag.

[12] Park, H., Kim, H. J. and Prasanna, V. K. (1993). "An $O(1)$ time optimal algorithm for multiplying matrices on reconfigurable mesh", *Inform. Process. Lett.*, **47**, 109–113.

[13] Savage, J. G. (1981). "Area time tradeoffs for matrix multiplication and related problems in VLSI models", *JCCS*, **22**, 230–242.
[14] Ullman, J. D. (1984). *Computational Aspects of VLSI*, Rockville: Computer Science Press.

Authors' Biographies

Martin Middendorf studied Mathematics and received the diploma and Dr rer nat at the University of Hannover, Germany. Currently he is an Assistent Professor at the University of Karlsruhe, Germany. His research interests include design and analysis of parallel algorithms, VLSI-Design, and combinatorial problems in molecular biology. He is member of EATCS and GI. His URL is http://www.aifb.uni karlsruhe.de/Staff/mmi.html.

Hartmut Schmeck received his Ph.D. in Computer Science from the University of Kiel, Germany. He held visiting positions at Queen's University, Kingston, Canada; Universities of Hildesheim and Münster, Germany; Technical University of Denmark at Lyngby; and University of Newcastle, Australia. Currently, he is Professor of Applied Computer Science at the University of Karlsruhe, Germany. His research interests include parallel algorithms and architectures and evolutionary computation. He is a member of ACM, IEEE CS, and GI, and member of programme committees of ARCS'97, EuroPar'97, PART'97, and IPPS'98. His URL is http://www.aifb.unikarlsruhe.de/Staff/schmeck.html.

Heiko Schröder studied Mathematics, Physics and Computer Science at the University of Kiel (Germany) where he received his Ph.D. in Computer Science in 1977. He was Assistant Professor of Computer Science at Kansas University (USA), Senior Research Fellow in Canberra (Australia) and Professor of Microelectronics in Newcastle (Australia). He is currently Professor of CS and HoD in Loughborough (UK). His main research interest include massively parallel architectures and algorithms for problems related to sorting, image processing, visualisation, optimisation and fault tolerance.

Gavin Turner received a Ph.D. in Computer Science at the University of Newcastle, Australia in 1996, and is currently a lecturer in Computer Science at Victoria University of Wellington, Wellington New Zealand. His main research interests lie in the area of design and analysis of algorithms, in particular algorithms and architectures for special-purpose parallel computing. In addition, he is interested in a wide variety of problems from image processing, load balancing and combinatorics.