Data Manipulation On The Distributed Memory Bus Computer *

Sartaj Sahni
Computer & Information Sciences Department
University of Florida
Gainesville, FL 32611, USA

Abstract

We consider fundamental data manipulation operations such as broadcasting, prefix sum, data sum, data shift, data accumulation, consecutive sum, adjacent sum, sorting, and random access reads and writes, and show how these may be performed on the distributed memory bus computer (DMBC). In addition, we study two image processing applications: shrinking and expanding, and template matching. The DMBC algorithms are generally simpler than corresponding algorithms of the same time complexity developed for other reconfigurable bus computers.

1 Introduction

The distributed memory bus computer (DMBC) which is a derivative of the distributed memory reconfigurable multiple bus machine of Thiruchelvan, Trahan, and Vaidyanathan [19] was proposed in [17]. Figure 1 shows an eight processor four bus DMBC. The processors are labeled P1 - P8 while the buses are labeled B1 - B4. The square boxes denote bus segment switches. By opening a bus segement switch, the bus it is on gets disjointed at that point. There are two lines (one solid and one dotted) from each processor to all the buses. The solid line is an I/O line that has contact switches wherever it crosses a bus. By setting the appropriate contact switch, a processor can read from or write to a bus. An I/O line can be connected to at most one bus at any time. In a general setting, the shown I/O lines may each represent many independent I/O lines so that a processor can simultaneously read from or write to several buses. In this case, the reading and writing is done to/from designated I/O buffers connected to the individual I/O lines. These buffers can, however, only be accessed serially by the individual processors. The dotted lines represent fuse

*This work was supported in part by the National Science Foundation under grant MIP-9103379

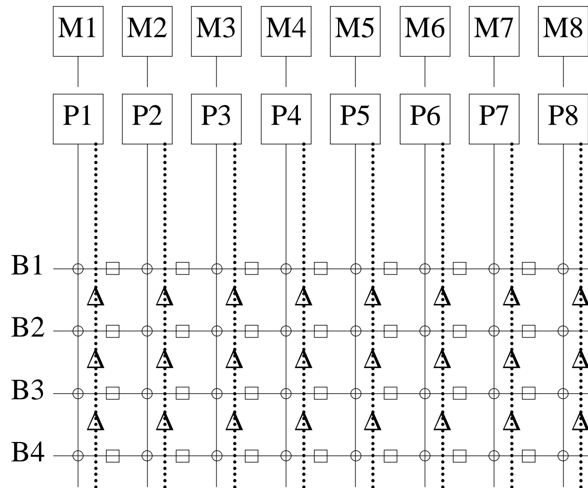


Figure 1: DMBC(8,4,1)

lines. There is a fuse switch (designated by Δ) on each segment of the fuse line. Fuse switches are used to partition the fuse line into disjoint sections. Each disjoint fuse section "fuses" together the bus segments it crosses into a single bus. Fuse switches on a particular fuse line may be set only by the processor to which the fuse line is connected. Segment switches may be set by either of the processors the switch lies between. Fuse and segment switches may either be set individually by an appropriate processor or by a mask broadcast down a fuse line or a segment switch setting line that connects all switches in the same column. In either case, the switch setting time is assumed to be constant. Some of the forms a mask may take are: all switches on line; all switches on odd segments; all switches on even segments. We use the notation DMBC(p, b, io) to denote a DMBC with p processors, b buses, and io I/O ports/lines per processor.

Architecturally, the DMBC is also closely related to the reconfigurable networks of Ben-Asher et al. [1] and the various reconfigurable mesh architectures proposed in [6, 8, 9, 10, 12, 13, 14, 22]. A DMBC($n, 1, 1$), for example, is identical to a one-dimensional RMESH, PARBUS, and MRN (see [17], for example, for a definition of these variants of a reconfigurable mesh). A DMBC($n, m, 1$) may be viewed as a generalization of a one-dimensional RMESH to the case of multiple buses and a DMBC($n, n, 1$) is essentially an $n \times n$ RMESH with all processors in the same column collapsed into one and with a capability to set all switches in the same vertical position of the RMESH by adjacent collapsed processors.

Like other reconfigurable bus architectures, the DMBC is an SIMD computer in which the processors operate in synchrony. By placing different but obvious restrictions on bus access, we obtain EREW, CREW, ERCW, and CRCW models. In this paper, we are concerned primarily with

the CREW model. A further classification of the DMBC is possible by permitting/not permitting the use of segment switches and/or fuse switches [21, 19]. The DMBC model may be enhanced by making the fuse lines and/or buses wraparound.

Central to all bus models is the assumption that the time to broadcast data on a bus is a constant. The validity of this assumption depends on the context. When analyzing uni-processor algorithms, we assume that memory access time is constant. This assumption is realistic as on commercial processors memory cycle times are set so that all words of memory can be accessed in this much time regardless of their physical distance from the processor. As a result, when we analyze an algorithm such as merge sort and determine its complexity to be $O(n \log n)$, there is good correspondence with observed run-time increase as we change n while keeping the computer invariant.

If we switch to a VLSI model and assume that memory is organized as a square, then since the merge sort algorithm uses $O(n)$ memory, the memory occupies a $\sqrt{n} \times \sqrt{n}$ chip space and memory access times are really $O(\sqrt{n})$ (under the assumption that we don't change the drivers and wire widths as n increases). The merge sort algorithm now has a complexity that is $O(n^{1.5} \log n)$. If we further take into account the size of the numbers we need to deal with, the complexity becomes even larger. Is this more detailed analysis a more accurate reflection of reality? Even in the VLSI model, it may not be so as we can change wire widths and driver size so as to hold access times fixed as n increases. In the case of commercial computers simple experiments with merge sort indicate that the run-time is, in fact, $O(n \log n)$ and not $O(n^{1.5} \log n)$ or anything higher.

When analyzing algorithms, we implicitly fix the computer on which they are being run and determine the growth in run time as problem size changes under the assumption that the computer has enough resources (say memory) to run the program on larger instances. For parallel algorithms, we may do the same. However, now our analysis regards processors and buses as resources much in the same way as memory is regarded as a resource. If there aren't enough of these, the algorithm fails. If the computers cycle time is large enough to accomodate transmission on the longest configurable bus, then the assumption of constant bus broadcast times becomes valid. The real question, then, becomes "Will it ever be technologically feasible to build reconfigurable bus computers with competitive cycle times?". The constant time broadcast assumption is also consistent with the assumption of constant instruction broadcast times in SIMD computers and constant data transmission times along wires of a sorting circuit or interconnection network (note that the longest wires in these circuits/networks have a length that increases as the circuit/network size increases.

The DMBC has some merits and demerits when compared with existing reconfigurable mesh (RM) architectures. Some of these are stated below.

1. An n processor RM has a VLSI implementation that uses $O(n)$ area. However, in this much area, we can realize only an n processor DMBC with a constant number of buses. On many problems to attain the same asymptotic complexity as on an n processor RM, we need an

n processor DMBC with $O(n)$ buses. This requires $O(n^2)$ area. Such a DMBC has n^2 segment and fuse switches while the corresponding RM has only $O(n)$ switches. We note that, in practice, if the VLSI technology is fixed, then to meet the constant time bus broadcast requirement, we will need to use wider wires and/or larger drivers as n increases. So, the area requirements will grow more rapidly than stated.

2. There are problems for which similar complexity solutions require an $O(n^2)$ processor RM or a DMBC$(n, n, 1)$. In these solutions, the additional RM processors are used only to obtain adequate bandwidth to support the needed data movement. Both computers take $O(n^2)$ area but the DMBC uses asymptotically fewer processors. If processors are significantly more expensive than switches, then the DMBC becomes relatively attractive.

3. The DMBC can be expanded in units of one processor and/or one bus. As a result we have greater flexibility than in an RM.

4. The communication bandwidth of a DMBC is not coupled to the number of processors, whereas in an RM it is. As a result, one can configure DMBCs that are compute and/or communication rich.

5. DMBC's with a sufficient number of buses can perform certain data manipulation tasks faster and with simpler algorithms than an RM.

In Section 2, we look at DMBC algorithms for the following fundamental data manipulation problems:

1. window broadcast
2. prefix sum
3. data sum
4. shift
5. data accumulation
6. consecutive sum
7. adjacent sum
8. sorting
9. random access reads and writes.

A unified treatment of these operations for hypercube computers can be found in [16] and for reconfigurable meshes in [12, 4]. Two image processing applications (image shrinking and expanding, and template matching) are considered in Section 3.

2 Data Manipulation

Most of the stated data manipulation operations can be performed in a rather straightforward manner on a suitable size DMBC. The others are only modestly more complex.

2.1 Window Broadcast

Assume that we have n processors and that the first w of these have data. The linear arrangement of n processors can be tiled with windows of size w (with the exception of the last window which may be smaller). The objective is to broadcast the data in the first window to the remaining windows such that the i'th processor of each window has the data initially in processor i of the first window. This operation can be performed with a single broadcast on a DMBC$(n, m, 1)$ with $m \geq w$. For this, we simply use bus B_i to broadcast the data from processor P_i, $1 \leq i \leq w$; following the broadcast, the j'th processor in each window reads bus j. Note that window broadcast does not use the segmenting and fusing features. Also, when $m < w$, the broadcast can be accomplished with $\lceil w/m \rceil$ broadcasts.

2.2 Prefix Sum and Data Sum

The prefix sum of n numbers distributed one to a processor can be computed in $O(\log n)$ time on a DMBC$(n, 1, 1)$ using bus splitting as in [12]. All processors can compute, in $O(\log n)$ time the sum of the n numbers using the same technique.

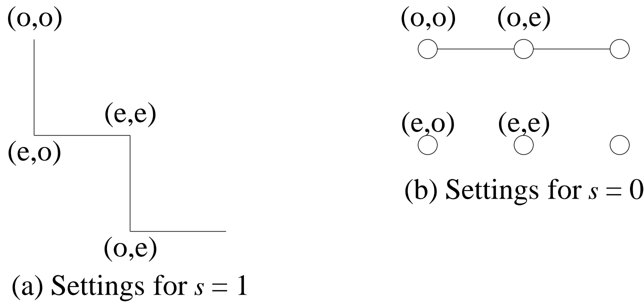
A useful special case of the data sum operation is when the numbers to be summed are either zero or one. In this case, a DMBC$(n, n, 1)$ can sum the n numbers in $O(1)$ time. For this, we adapt the bus dropping scheme used by Jenq and Sahni [4] to perform the same operation on an $n \times n$ RMESH. The summing is accomplished in three passes. Let s denote the zero/one value in each processor. In each pass $n/3$ of the s's are summed. At the end, the three partial sums are added to get the desired result.

To sum $n/3$ s's using n processors and n buses, the following steps are followed:

1. Verify that at least one of the $n/3$ s-values is a one by segmenting bus one at the right of each processor with $s = 1$. Next each processor with $s = 1$ writes a one to bus one and then the leftmost processor reads bus one. In case nothing is read, the sum of the s's is zero and we terminate.

2. Use the first $n/3$ buses to transmit the $n/3$ s-values from their current processors to the $2n/3$ leftmost processors in the block such that each such processor gets exactly one s value and processors $2i - 1$ and $2i$ get the same s value, $1 \leq i \leq n/3$.

3. Establish the segment and fuse switch settings corresponding to the row and column switch settings of Figure 2. The notation (o,e) denotes a position on an odd bus and even fuse line.



(a) Settings for $s = 1$

(b) Settings for $s = 0$

Figure 2: Switch settings for first $2n/3$ processors

So, if $s = 1$ in processor 1, then processor 1 opens its right adjacent segment switches on all odd buses and closes the right adjacent segment switches on even buses; it also closes every alternate fuse switch on its fuse line beginning with the first one and opens the remaining; processor 2 opens its right adjacent segment switches on all even buses and closes the right adjacent segment switches on odd buses; it also closes every alternate fuse switch on its fuse line beginning with the second one and opens the remaining. The remaining notations are correspondingly interpreted. One may verify that the given switch settings create disjoint staircase like buses that drop by two rows for each odd column that has $s = 1$.

The switch settings just described are done only by the first $2n/3$ processors. The remaining processors in the block open their fuse switches and close their segment switches to create horizontal buses that span the last $n/3$ processors in the block.

4. The leftmost processor of the block writes a one to the first bus.
5. Processor $2n/3 + j$ of the block reads bus $2j + 1$, $1 \leq j \leq n/3$.
6. Exactly one of the last $n/3$ processors in the block reads a non-null value. Let this be processor $2n/3 + q$. The sum of the $n/3$ s-values is q.

Note that the bus dropping technique can be used to sum the s's modulo 2 using five buses provided the fuse lines wraparound. For this, the wraparound fuse switch is used to connect bus 5 to bus 1 in even columns and a segment switch on bus one is closed to further connect the bus five segment to the next odd processor. We note that this technique of bus wraparound for modulo counting has been used before for modulo counting on a PARBUS [11].

2.3 Shift

Assume that each processor has a single data item and that the the data items are to be shifted right (or left) by s processors, $0 < s < n$. Thus, processor P_i is to send its data to processor P_{i+s} provided $i + s \leq n$. The senders may be divided into $s + 1$ equivalence classes based on the value of $i \bmod (s + 1)$ where i is the sender's index. All senders in the same equivalence class can complete their transmission simultaneously using a single bus by segmenting this bus at appropriate points.

Hence, a DMBC$(n, s+1, 1)$ can do the required shift with a single broadcast step. This data shifting algorithm may be simulated by a DMBC$(n, 1, 1)$ using $s + 1$ broadcasts.

Circular shifts can be implemented with the above scheme if we have wraparound buses. Techniques for the case with no wraparound are discussed in the next subsection.

2.4 Data Accumulation

In this operation, each processor initially has a single piece of data and processor P_i is to accumulate the data from the next m processors. This is done with wraparound. So, processor P_n accumulates data from processors P_1 through P_m.

Data accumulation may be trivially done with m broadcasts on a DMBC$(n, n, 1)$. For this, processor P_i broadcasts its data using bus B_i on each of m broadcast cycles, $1 \leq i \leq n$. Processor P_j reads data from the appropriate m buses, one bus at a time. This method of accumulation requires no bus segmenting or fusing. Using bus segmenting data accumulation can be done in the same number of broadcast steps on a DMBC$(n, m, 1)$ with wraparound buses when $m > 1$. Now, P_i broadcasts using bus $B_{i \bmod m}$ (B_0 is interpreted as being bus B_m) after opening the right adjacent segment switch on this bus. P_i accumulates $m - 1$ of the data items it needs from the remaining m buses. To get the last remaining item, a left circular shift of one is performed. This requires a single broadcast as $m > 1$. When $m = 1$, data accumulation may be done by shifting data left circularly by one processor. This shift needs two broadcasts when there is only one bus and one broadcast when we have two or more buses. In case the buses do not wraparound, then using m additional broadcast steps, the rightmost m processors can accumulate the remaining data they need. Alternatively, using an additional bus, processor P_1 can write the data initially in $P_1 \cdots P_m$ in time cycles $1, \cdots, m$, respectively, on to the additional bus. The processors at the right end read the data they need during the given time cycle. Note that P_1 would otherwise be idle during the write phase of these cycles and the rightmost processors would otherwise be idle during the read phase of the cycles.

On a DMBC$(n, 1, 1)$ with wraparound buses, data accumulation can be done with $m+1$ broadcast steps. In the i'th step, all processors P_j, $j \bmod (m+1) = i \bmod (m+1)$ open their right adjacent segment switch and then write their data onto bus B_1; next, the processors that did not write data, read the bus and store the value read. When buses do not wraparound, we can use m additional broadcast steps to complete the accumulation or use an additional bus as described above.

2.5 Consecutive and Adjacent Sums

These are similar operations. In both, each processor begins with m numbers. In a consecutive sum, the processors are partitioned into groups of size m left-to-right (for simplicity, assume that m divides n). The i'th processor in each group is to sum up the i'th data in the m processors in its group. In an adjacent sum operation, each processor sums up data from m right adjacent processors

(for this, P_1 is right adjacent to P_n). The sum computed by P_j involves the first data item in P_j, the second in its right adjacent processor, the third from the processor two away, and so on. Both operations can be performed in $O(m)$ time on a DMBC$(n, 1, 1)$. Algorithms for the cases when the number of buses is one, two, or m, and when the buses do or do not wraparound can be developed using the ideas stated in our discussion of the data accumulation operation. The basic strategy is for each processor to initiate a token that is circulated through the processors that contain the data needed for the sum. The tokens add up the needed data and return to the initiating processors. When the number of buses is m, a simpler strategy can be used. For consecutive sum, for example, the buses are segmented to span only the length of individual partitions. On the i'th broadcast, the j'th processor in each partition uses bus $B_{(j+i-2)\bmod m+1}$ to broadcast its $(j+i-2)\bmod m+1$'th data item. Processor j then reads from bus B_j and adds.

2.6 Sorting

Since a DMBC is closely related to the architectures of [21, 18, 20], the sorting algorithms of these papers may be easily adapted for the DMBC. These algorithms sort n $O(\log n)$-bit integers. An arbitrary set of elements can be sorted in $O(\log^2 n)$ time using merge sort and a DMBC$(n, n, 1)$. To merge two sorted sequences L and R, each element in L determines the number of elements in R that are less than it and each element in R determines the number in L that are less than or equal to it. For the former, elements of L are broadcast on independent buses and the processors holding R sample the buses using a binary search. The operation takes $O(\log |L|)$ time and does not use fusing. Bus segmentation is required as many merges need to be done simultaneously when sorting. The run time can be reduced to $O(\log n)$ by simulating Cole's $O(\log n)$ parallel merge sort [2]. This simulation is possible as Cole's method uses n processors and $O(n)$ memory.

We now describe a simple constant time sorting algorithm for a CREW DMBC$(n^2, n, 1)$. Since the required DMBC configuration needs $O(n^3)$ area, the constant time sort is not area-time (AT^2)-optimal [7]. Note that the constant-time $n \times n$ RM sorts of [3, 10, 15] are AT^2-optimal as an $n \times n$ RM can be realized using $O(n^2)$ area.

Our DMBC sorting algorithm is based on count sort. It partitions the n^2 processors into blocks of size n. The n processors in block i are used to determine the count value for element i. Once the counts are known, element i is put onto bus $count(i)$ and then read from this bus by processor $count(i)$. The steps in the sort algorithm are:

1. All segment switches are closed and all fuse switches are opened. Processor i writes element i to bus i, $1 \le i \le n$.
2. The jth processor in each n-processor block reads element j from bus j, $1 \le j \le n$.
3. All processors in block i read element i from bus i, $1 \le i \le n$.
4. Processor j of block i compares its two elements e_i and e_j and sets $s = 1$ if $e_j < e_i$ or if

$e_j = e_i$ and $j \leq i$.

5. Sum the s values in each n-processor block. Store the sum in variable d of the first processor in each block.

6. All segment switches are closed and all fuse switches are opened. Processor 1 of block i writes e_i to bus d_i, $1 \leq i \leq n$.

7. Processor j of the entire SMBC/DMBC reads bus j and obtains the jth sorted element.

Since the s values to be summed in each n-processor block (Step 5) are zero or one, the summing takes $O(1)$ time.

2.7 Random Access Reads and Writes

Random access reads are easily performed in constant time on a DMBC$(n, n, 1)$. Processor P_i writes its data to bus B_i and then reads from bus B_k if it wants to read the data that was with P_k. A random access write in which destination conflicts are to be resolved arbitrarily may be performed with two broadcasts as below:

1. (send left) Let d_i be the destination of the data originating in P_i. If $d_i < P_i$, then P_i opens its right segment switch on bus d_i and broadcasts its data on this bus; following the broadcast, P_i reads bus i, $1 \leq i \leq n$.

2. (send right) If $d_i > P_i$, then P_i opens its left segment switch on bus d_i and broadcasts its data on this bus; following the broadcast, P_i reads bus i, $1 \leq i \leq n$.

3. Processors that received data on both broadcasts discard one of the data items received.

3 Image Processing

3.1 Shrinking and Expanding

The q-step expanding (shrinking is almost identical) of an $N \times N$ binary image I [5] can be done as below:

1. For each pixel $I[i, j] = 0$ determine if there is a 1 in the same row and at most q columns to the left.

2. For each pixel $I[i, j] = 0$ determine if there is a 1 in the same row and at most q columns to the right.

3. The results of the preceding two steps determine a new binary matrix R with $R[i, j] = 1$ iff pixel $I[i, j]$ is a one or there is a one on the same row of I and at most q columns away.

4. For each point with $R[i, j] = 0$ determine if there is a one on the same column and at most q rows above.

5. For each point with $R[i,j] = 0$ determine if there is a one on the same column and at most q rows below.
6. The results of the preceding two steps determine a new binary matrix E with $E[i,j] = 1$ iff $R[i,j]$ is a one or there is a one on the same column of R and at most q rows away. E is the q-step expansion of I.

The q-step expansion of I can be computed in constant time using a fairly simple algorithm and a DMBC($n^2, n, 1$). We begin with the I matrix stored in row-major order, one pixel per processor. The row operations for row i of I (steps 1 and 2) as well as the column operations for column i of R (steps 4 and 5) are done using bus B_i, $1 \leq i \leq n$ (actually, the row operations can be done utilizing a single bus and bus segmentation; however, n buses are needed for the column operations). We consider the first step only. Processors that do not have a pixel value of 1, need to determine if there is a 1 valued pixel at most q processors to the left and on the same row. For this, these processors will first determine the location of the nearest one on their left. Processors in row i with pixel value 1, open their left segment switch on bus B_i and then broadcast their column index on this bus; next, processors on row i with pixel value zero read bus B_i; if nothing is read or the column index read is is more than q away, then there isn't a one within q columns to the left; otherwise, there is.

3.2 Template Matching

In this operation, we begin with an $n \times n$ image and an $m \times m$ template, $m \leq n$. We are to compute a matrix C with the property that $C[i,j]$ is the inner product of the $m \times m$ submatrix of I beginning at $I[i,j]$ and T (the inner product of two matrices of the same dimensions is defined to be the sum of the products of corresponding elements). For this to be well defined, assume that I has row and column wraparound. Assume that in the initial configuration, I is stored in a DMBC($n^2, n, 1$), in row-major order, one element per processor and that T is stored in row-major order one element per processor in the leftmost m^2 processors. The matrix C may be computed in m^2 time by first having each processor accumulate $m-1$ image values on the same row and to the right. This leaves each processor with m image values. This data accumulation can be done in $O(m)$ time by segmenting the n buses at row boundaries. Following the data accumulation, the elements of T are broadcast on a single bus, one at a time, and each processor computes m values which are the inner products of the m rows of T and the m image values held in the processor. This takes $O(m^2)$ time. Finally, the C values are computed using the adjacent sum operation defined over columns of the inner products. For this, each column of C uses a different bus. The total time to compute C is $O(m^2)$.

4 Conclusion

The DMBC architecture is a very flexible architecture that permits the development of simple yet fast algorithms for the commonly studied data manipulation operations. These, in turn, result in correspondingly simple and fast algorithms for higher level applications such as those that arise in image processing [16]. In many cases, the algorithms are conceptually simpler than those developed for reconfigurable meshes and at least as fast.

References

[1] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster, The power of reconfiguration, *Journal of Parallel and Distributed Computing*, 13, 139-153, 1991.

[2] R. Cole, Parallel merge sort, *SIAM Journal of Computing*, 17, 4, 770-785, 1988.

[3] J. Jang and V. Prasanna, An optimal sorting algorithm on reconfigurable meshes, *Proceedings 6th International Parallel Processing Symposium*, IEEE Computer Society, Los Alamitos, CA, 130-137, March 1992.

[4] J. Jenq and S. Sahni, Reconfigurable mesh algorithms for fundamental data manipulation operations, In *Parallel computing on distributed memory multiprocessors*, Ed. F. Ozguner, Springer Verlag, NATO ASI Series F, 1993.

[5] J. Jenq and S. Sahni, Image shrinking and expanding on a pyramid, *IEEE Trans. on Parallel and Distributed Systems*, 4, 11, 1291-1296, 1993.

[6] T. Kao, S. Horng, and H. Tsai, Computing connected components and some related applications on a RAP, *Proc. 1993 International Conference on Parallel Processing*, 1993, pp. III-57 - III-64.

[7] Leighton, T., Tight bounds on the complexity of parallel sorting, *IEEE Trans. on Computers*, C-34, 4, April 1985, 344-354.

[8] H. Li and M. Maresca, Polymorphic-torus architecture for computer vision, *IEEE Trans. on Pattern & Machine Intelligence*, 11, 3, 133-143, 1989.

[9] H. Li and M. Maresca, Polymorphic-torus network, *IEEE Trans. on Computers*, C-38, 9, 1345-1351, 1989.

[10] R. Lin, Reconfigurable buses with shift switching - Radix sort *Proc. 1992 International Conference on Parallel Processing*, 1992, pp. III-2 - III-9.

[11] R. Lin, S. Olariu, J. Schwing, and J. Zhang, A VLSI-optimal constant time sorting algorithm on reconfigurable mesh, *Proc. 9th European Workshop on Parallel Computing*, Madrid, Spain, 1992.

[12] R. Miller, V. K. Prasanna Kumar, D. Resis and Q. Stout, Data movement operations and applications on reconfigurable VLSI arrays, Proceedings of the 1988 International Conference on Parallel Processing, The Pennsylvania State University Press, pp 205-208.

[13] R. Miller, V. K. Prasanna Kumar, D. Resis and Q. Stout, Meshes with reconfigurable buses, Proceedings 5th MIT Conference On Advanced Research IN VLSI, 1988, pp 163-178.

[14] R. Miller, V. K. Prasanna Kumar, D. Resis and Q. Stout, Image computations on reconfigurable VLSI arrays, Proceedings IEEE Conference On Computer Vision And Pattern Recognition, 1988, pp 925-930.

[15] M. Nigam, and S. Sahni, Sorting n Numbers on $n \times n$ Reconfigurable Meshes with Buses, *International Parallel Processing Symposium*, 1993, 174-181.

[16] Ranka, S., and Sahni, S., *Hypercube Computing*, Springer-Verlag, 1990.

[17] Sahni, S., Computing on reconfigurable bus architectures, *International Conference on Computer System and Education*, Indian Institute of Sciences, June 1994.

[18] C. Subbaraman, J. Trahan, and R. Vaidyanathan, List ranking and graph algorithms on the reconfigurable multiple bus machine, *Proc. 1993 International Conference on Parallel Processing*, 1993, III-244 - III-247.

[19] R. Thiruchelvan, J. Trahan, and R. Vaidyanathan, On the power of segmenting and fusing buses, *Proc. 1993 International Parallel Processing Symposium*, 1993, 79-83.

[20] R. Thiruchelvan, J. Trahan, and R. Vaidyanathan, Sorting on reconfigurable multiple bus machines, *Proc. Thirteenth Midwest Symposium on Circuits and Systems*, 1993.

[21] Vaidyanathan, R., Sorting on PRAMs with reconfigurable buses, *Information Processing Letters*, 42, 1992, 203-208.

[22] B. Wang, G. Chen, and F. Lin, Constant time sorting on a processor array with a reconfigurable bus system, *Information Processing Letters*, 34, 4, 187-190, 1990.