International Conference on
Computer System and Education, IISc, 1994

# Computing On Reconfigurable Bus Architectures*

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

## Abstract

Recently, several parallel computer architectures that employ reconfigurable buses as the communication medium have been proposed. These include reconfigurable meshes, reconfigurable networks, and shared and distributed memory bus computers. In this paper, we review these architectures. Programming principles for these architectures are illustrated using sorting as an example.

## 1 Introduction

Communication plays a central role in the development of effective multiprocessor architectures. For many problems, the theoretical limits to parallel performance are determined by the cut width as well as the diameter of the interconnection network used. The use of buses to provide flexible and high speed communication among processors has been proposed by several authors. For example, fixed topology buses have been used to augment mesh connected computers. Stout [24, 25] and Bokhari [4] (among others) have considered the addition of a global bus that connects all mesh processors. This bus enables any one processor to broadcast data to all other processors in one time unit. The diameter of the resulting augmented mesh becomes one rather than $n$ for an $n \times n$ mesh. Aggarwal [1] has considered meshes augmented with several global buses. Prasanna Kumar and Raghavendra [19, 20] have proposed the augmentation of the standard mesh interconnections with multiple buses; one for each row and column of the mesh. The resulting augmented mesh is called the *mesh with multiple broadcasting.* While the mesh augmentations cited above employ different numbers of buses and buses with different topolgy (global vs. row vs. column), they have one thing in common; the buses have a static topology. In other words, the (sub)set of processors connected together by a bus as well as the number of disjoint buses do not change from program-to-program or during the execution of a single program.

In a reconfigurable bus computer, it is pos-

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

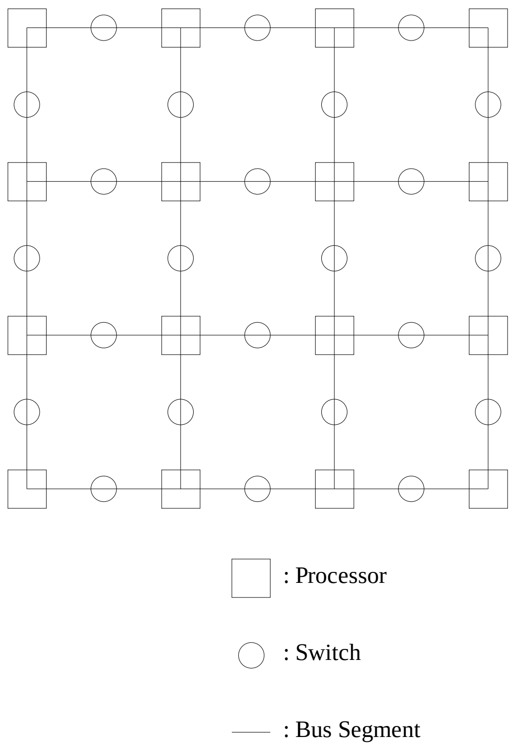

Figure 1: $4 \times 4$ RMESH

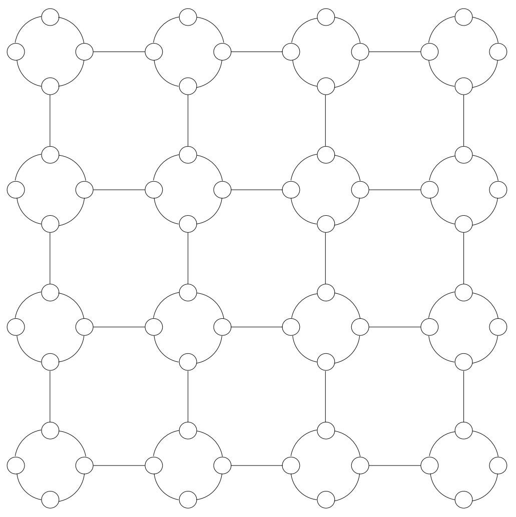

Figure 2: $4 \times 4$ PARBUS

sible to change the subset of processors connected by a bus dynamically and under program control. The idea of using reconfigurable buses to enhance parallel computer performance is quite old. For example, Rothstein [23] used reconfigurable buses in conjunction with cellular automata. He referred to the resulting automata as *bus automata* and demonstrated that bus automata could recognize regular languages in constant time.

More recently, reconfigurable buses have been proposed to replace (*not augment*) the interconnection network of a mesh computer. The resulting parallel computer architecture, the *reconfigurable mesh* (RM), is a synchronized computer that comes in several different flavors. Figures 1 and 2, respectively, show a $4 \times 4$ RMESH and PARBUS.

In a two-dimensional RMESH computer [14, 15, 16], the processors are located at the grid points of a two-dimensional grid. This grid has a bus superimposed on it. The bus spans every row and every column of the grid (see Figure 1). The portion of this bus that lies between two grid adjacent processors is referred to as a *bus segment*. Each bus segment has a switch on it. This switch may be used to break the bus at the segment. Thus, by opening the switches on all column segments and keeping the row segment switches closed, we form independent row buses, one for each row of processors. The segment switches are set dynamically by the processors at the end-points of each segment. In a properly functioning RMESH program, there will be no switch setting conflicts (a conflict arises when the two processors at the ends of a segment try to set the segment switch in different states). Since the RMESH is a synchronized architecture. all processors are able to set their respective switches at the same time to create the desired bus (subbus) configurations. In the tradition of PRAMs, one may define different RMESH computers by restricting/allowing different modes of bus access. This results in EREW, CREW, ERCW, and CRCW models. The most commonly studied model is the CREW model.

The PARBUS version of the RM [14, 15, 16, 30] is like the RMESH except that the bus segments are joined to the ports of a

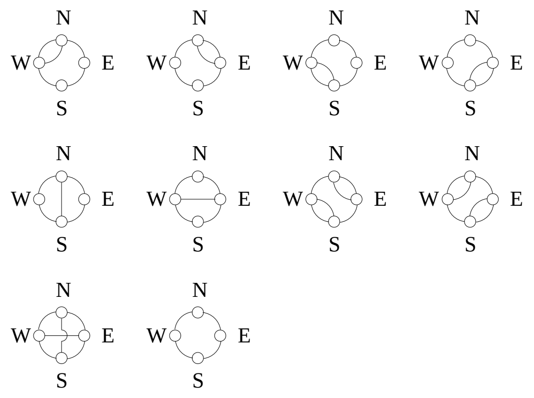

Figure 3: 4 × 4 MRN

four port switch in each processor; there are no segment switches. The four port switch in each processor is able to connect together arbitrary subsets of the four bus segments connected to it. So, for example, if each processor connects together its north and south segments and also its east and west segments, then we obtain row and column buses simultaneously. Note that in the RMESH model, it isn't possible to set the segment switches such that a processor is simultaneously on two disjoint buses. The PARBUS model is very closely related to the polymorphic torus of Maresca and Li [9, 10]. The only difference is that in the polymorphic torus, there is row and column wraparound of the bus segments. So, there is a bus segment connecting the righmost (botttom) processor in each row (column) to the the leftmost (top) processor in that row (column).

Ben-Asher et al. [2] have defined a class of reconfigurable networks called RN. This class includes a mesh reconfigurable network (MRN) that is very similar to a PARBUS. The switches of an MRN are however constrained so as to realize only the configurations of Figure 3.

Greater communication flexibilty is obtained by replacing each bus segment of the RM by several bus segments, each with capacity 1 bit (say). While each bus segment of the RM is several bits wide, the switches used are unable to change the order of bits on either side of the switch. So, for example, in a PARBUS processor it isn't possible to route bit 1 of the south bus segment to bit 2 of the west bus segment and bit 2 of the south segment to bit 3 of the east segment. This can be done when each bit is carried by a different bus segment. Lin [11] and Kao, Horng, and Tsai [7] propose a limited version of such an architecture. In this, the bits can only be shifted as they pass through the switch of a processor. So, all the south (say) bus segments of a processor must be connected to (say) all the north segments. In this connection the bits of the south segments may be rotated when conected to the north segments. Lin refers to his architecture as *reconfigurable bus system with shift switches (REBSIS)* while Kao et al. refer to theirs as *reconfigurable array processor (RAP)*.

The reconfigurable bus architectures described so far have all been related closely to the mesh architecture. Like the two-dimensional mesh, the architectures are easily extended to versions in more than two dimensions or restricted to one-dimension. Also, it isn't necessary for the size of each dimension to be the same. However, because of the closeness to the mesh architecture, there is a close tie between the bus structure and the number of processors. The architecture does not permit one to increase the communication capabilities while keeping the topolgy dimensions fixed. As a result, we see, in the literature, RM algorithms that use a large size RM just to get additional bus paths. The additional processors are used only to set switches but not to compute.

Vaidyanathan [29] and Thiruchelvan, Trahan, and Vaidyanathan [27] have proposed architectures in which the communication capability is independent of the processing capability. These architectures employ a collection of reconfiguarble buses for communication and a collection of processors for computation. [29, 27] define both a shared and distributed memory version of their bus architecture. We refer to these as *shared memory bus computer* (SMBC) and *distributed memory bus computer* (DMBC). Our description of the architectures is slightly different from that given in [29, 27]. So, to avoid confusion, we name the architectures diiferent from [29, 27].

Figure 4 shows an eight processor four bus SMBC. The processors are labeled P1 - P8 while the buses are labeled B1 - B4. The square boxes denote bus segment switches. These serve the same purpose as in the RMESH. 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 lines. There is a fuse switch (designated by $\Delta$) 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

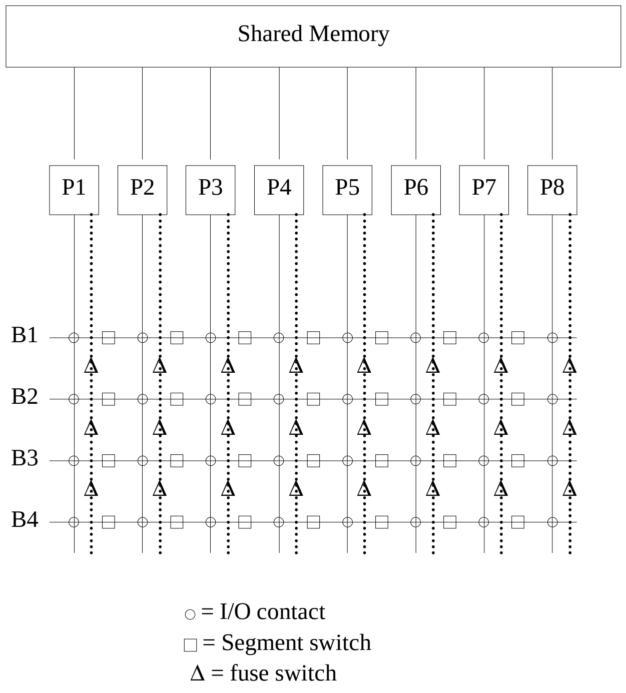

Figure 4: SMBC(8,4,1)

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 SMBC($p, b, io$) to denote an SMBC with $p$ processors, $b$ buses, and $io$ I/O ports/lines per processor.

A DMBC is identical to an SMBC except that each processor has local memory and there is no shared memory at all. A DMBC(8,4,1) is shown in Figure 5. It should be evident that an $n$-processor DMBC with $n$ buses 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.

As in the case of the RM architectures, restricting the bus access methods results in EREW, CREW, ERCW, and CRCW models. A further classification is possible by permitting/not permitting the use of segment

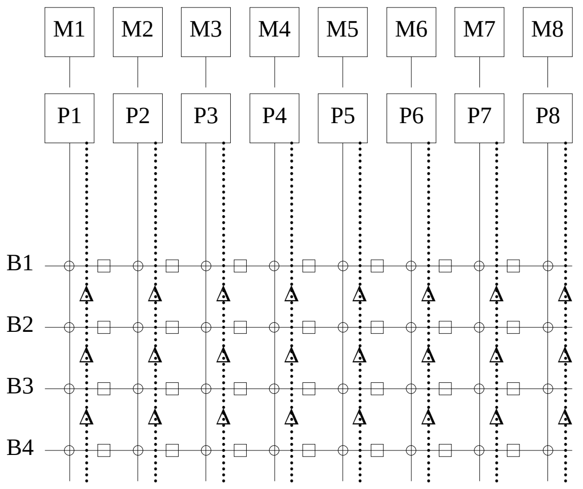

Figure 5: DMBC(8,4,1)

switches and/or fuse switches [29, 27]. We note that the SMBC defined here differs from that of [29] essentially in that the model of [29] has no fuse lines. The concept of fuse lines, is however, due to Thiruchelvan, Trahan, and Vaidyanathan [27] and fuse lines are included in their distributed memory reconfigurable multiple bus machine (RMBS). The fuse lines used in the RMBS model are slightly different from ours. Additionally, the RMBS model makes a distinction between write lines and read lines. The first (i.e., leftmost) I/O line associated with a processor is a write line and the remaining I/O lines are read lines. In addition, a segment switch is added between the write line and the collection of read lines wherever a write line crosses a bus. This permits a processor to write to the left bus segment while reading from the right segment.

Central to all bus models is the assumption that the time to broadcast data on a bus is a constant. One may question the validity of this assumption in the same way that the constant time memory access assumption for PRAMs and single processor computers may be questioned. In the single processor case, for example, consider a problem of size $n$ that uses $M(n)$ memory where $M$ is at least a linear function of $n$. If the memory is organized as a square, then it is bounded by a box of dimension $\sqrt{M(n)}$ and the memory access time varies with $n$. Similarly, at least some of the numbers in use require $\Omega(\log n)$ bits and the arithmetic takes at least this much time per operation. 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. In any case, in keeping with assumptions in the literature, we shall assume a constant broadcast time along buses of a reconfigurable computer.

Some results relating the computational power of the different models described above can be found in [2, 3, 27]. It is interesting to note that certain problems can be solved faster on distributed memory reconfigurable computers than on certain PRAM models. For example, we can, in constant time, find the logical or of $n$ bits stored one to a processor using a $1 \times n$ EREW RMESH or PARBUS or an EREW DMBC($n$,1,1). The algorithm is straightforward. Assume that all segment switches are closed and in the case of a PARBUS assume that the east and west segments are joined at each processor. The result is a single continuous bus stretching from the leftmost to the rightmost processor. If a processor has a 1, then it opens the segment switch to its right (or in the case of a PARBUS, it disconnects the two segments coming to its switch). The switch settings now create bus partitions that have at most one processor with a 1. Next, each processor with a 1 writes to the bus. The leftmost processor reads its bus. If it reads nothing, then the logical or is 0 otheriwse it is 1. It is well known that a CREW PRAM with a polynomial number of processors cannot perform this computation in constant time. Another example that we shall see in this paper is sorting. It is possible to sort in $O(1)$ time on polynomial sized reconfigurable computers while this is not possible on any polynomial sized PRAM [2, 5, 6, 12, 17, 30].

In the remainder of this paper, we illustrate some of the programming principles for reconfigurable architectures using the sorting problem as an example. We consider sorting on an RM in Sections 2 and 3, and on an SMBC/DMBC in Section 4.

## 2 Sorting on an $n \times n \times n$ RM

Constant time algorithms to sort $n$ numbers on an RMESH, PARBUS, and MRN with $n^3$ processors are described in [6, 30, 2]. The algorithms for all three architectures are similar and are based on count sort. The steps of a count sort are:

1. [*Determine Count*] For each element, determine the number of elements that are to appear to its left plus 1.
2. [*Reorder*] Move each element to the position given by its count.

Assume that the initial and final configurations have one element per processor of row 1 of the bottom face of the three-dimensional mesh of processors. To determine the count value for element $i$, we shall use the $i$'th $n \times n$ face of the RM. For this, the data are broadcast up to row one of the remaining faces using $n$ independent buses; each bus connecting a processor on row one of the bottom face with the corresponding processor on each of the remaining faces. Now that each $n \times n$ face has all $n$ elements in its row one processors, each face works independently to determine the count of one element.

In face $i$, the processor at position $(1, j)$ will compare elements $i$ and $j$ and set a local variable $s$ to 1 if element $i$ is less than element $j$ or if the two elements are equal and $j \leq i$. Otherwise, $s$ is set to zero. To accomplish this, the processors in row one form a row bus by closing the segment switches on this row; next the $i$'th processor on this row writes its element to the row bus; all processors in row one read this data from the bus; the two elements in each row one processor are compared and the value of $s$ determined.

Having determined $s$, the count value for element $i$, is obtained by summing the $s$ values on row one. This summing is not accomplished by using the add operation. Instead, it is done using a technique called *bus dropping* in which we configure buses that drop by some fixed number, $r$, of rows at each column where the row one $s$ value is a one. Then, the leftmost processor in row one broadcasts a one which is read by the processors in column $n$. The bottommost processor in column $n$ that reads a one divides its row index less one by $r$ to determine how many times the bus dropped. This yields the sum of the $s$'s. Bus dropping was first used by Miller et al. [MILL88a] to compute the exclusive or of $n$ logical values in constant time.

On an MRN and PARBUS, we may use $r$ = 1. First, the row one processors use column buses to broadcast their $s$ values to all processors on the same column. Next, processors with $s = 1$ (except those in the rightmost column) connect their north and east ports as well as their west and south ports; processors with $s = 0$ connect their east and west ports. As a result of the switch settings, several disjoint buses are created. Each bus has the property that it drops one row when it goes through a column with $s = 1$ and it remains in the same row when going through a column with $s = 0$. As a result, if the leftmost processor of row one writes a one to the bus segment connected to its west port this can be read by only one processor in the rightmost column. Suppose that the rightmost column processor that reads the one, is in row $j$. This implies that the bus has dropped $j - 1$ times. Consequently, there are $j - 1$ row one processors with $s = 1$ not counting the rightmost processor in the row. The sum of the $s$'s is therefore $j - 1 + s(1, n)$ where $s(1, n)$ is the $s$ value of the rightmost processor in row one.

On an RMESH, bus dropping is done twice, once to sum the $s$'s in the row one processors in the even columns and once for the odd columns. The two sums are then added using the standard add operator. We shall describe how bus dropping is used for the odd columns. Our description is from [6]. First, the row one processors in odd columns broadcast their $s$ value down column buses. Next, the odd columns send their $s$ values to their right adjacent even column (if any). Now, each processor has an $s$ value. This is used to set the segment switches as shown in Figure 6. The notation (o,e) denotes a processor in an odd row and an even column. 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 $s = 1$. Once the segment switches have been set as described, the processor in position (1,1) broadcasts a 1 on its bus. All processors in the rightmost column read their bus. If $n$ is odd, then exactly one such processor reads a 1. If this processor is in row $j$, then the number of odd column processors that have $s = 1$ is $(j - 1)/2 + s(1, n)$ where $s(1, n)$ is the $s$ value of the righmost processor in row one. If $n$ is even, then up to two processors may read the 1. The processor in the odd row continues to compute the sum as $(j - 1)/2$.

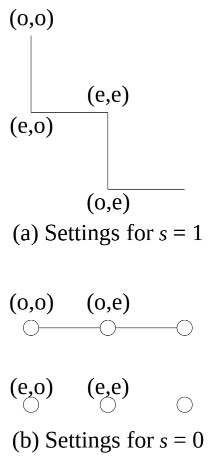

Figure 6: Switch settings to sum for odd columns

Using the above technique, we can arrange for the processor in position (1,$i$) of face $i$ to contain the *count* value for element $i$. These count values are then broadcast down to row 1 of face 1. Now, we are left with the problem of reordering the elements according to their count values. For this, only the processors of face 1 are active. Column buses are used to route element $i$ to the processor in position $(i, i)$, $1 \le i \le n$. Then row buses are used to route these elements from the diagonal processors to position $(i, count(i))$. Finally, column buses are used to route the elements back to row 1. This completes the sort.

One may verify that the total number of broadcast steps, switch setting steps, and computational steps is $O(1)$. Also, note that it is only the row one processors of the $n$ faces that do any computation (i.e., comparisons and/or addition). The remainder are used only for routing. I.e., $n^3 - n^2$ of the $n^3$ procesors do no computation!

## 3 Sorting on an $n \times n$ RM

Algorithms to sort $n$ elements in constant time on an $n \times n$ reconfigurable mesh have been proposed by several authors. Ben-

1. *balance* each vertical slice;
2. *unblock* the mesh;
3. *balance* each horizontal slice as if it were an $N \times N^{1/2}$ mesh lying on its side;
4. *unblock* the mesh;
5. *shear* three times;
6. *sort* rows to the right;

Figure 7: Six steps of a rotate sort [13]

Asher et al. [2] do this for reconfigurable networks and the corresponding result for an MRN follows from their work. Jang and Prasanna [5] and Lin et al. [12] do this for the PARBUS model while Nigam and Sahni [17, 18] do this for all three (RMESH, PARBUS, MRN) RM models. The algorithms of [2, 5, 17] are based on Leighton's column sort method [8]. The algorithm of [12] uses a selection method to sort while one of the algorithms in [18] is based on Marberg and Gafni's rotate sort [13]. In this section, we shall look at Nigam and Sahni's [18] RM adaptation of rotate sort.

Rotate sort was developed by Marberg and Gafni [13] to sort $N^2$ elements on an $N \times N$ mesh connected computer. This is a six step method (see Figure 7). The subtasks used in these six steps are given in Figure 8. The method assumes that the $N \times N$ mesh is such that $N = 2^{2k}$ for some integer $k$. Horizontal (vertical) slices are obtained by tiling the mesh by submeshes of size $N^{1/2} \times N$ ($N \times N^{1/2}$) and blocks are obtained using tiles of size $N^{1/2} \times N^{1/2}$.

Nigam and Sahni have shown how an $n \times n$ RM with $n = N^2$ can simulate each of the six steps of a rotate sort that is sorting $N^2$ elements on an $N \times N$ mesh. The simulation is such that each step runs in constant time on the RM. As a result, the $n \times n$ RM is able to

Subtask: *balance* submeshes of size $v \times w$

1. sort all columns of the submesh downward;
2. rotate each row $i$ of the submesh, $i$ mod $w$ positions right;
3. sort all columns of the submesh downward;

Subtask: *unblock* the whole mesh

1. rotate each row $i$ of the mesh $(i \times N^{1/2})$ mod $N$ positions right;
2. sort all columns of the block downwards;

Subtask: *shear* the whole mesh

1. sort all even numbered rows to the right and all odd numbered rows to the left;
2. sort all columns downward;

Figure 8: Subtasks of rotate sort [13]

sort $n$ elements in constant time. The simulation maps the $N \times N$ mesh being simulated into row one of the $n \times n$ RM using a row-major ordering. Examining, the substeps of Figure 8, we see that we need to simulate the following operations:

1. sort columns or rows of a submesh or block
2. rotate a row of the mesh or of a submesh

Rotation is easily performed in constant time by using a subRM of size $n \times s$ to rotate s elements ($s = N$ if a row is being rotated). Since $s < n$, the rotation can be accomplished by using column buses to route the data in column $j$ of the (sub)mesh row from row 1 of the RM to row $j$ of the $n \times s$ subRM. Next, row buses are used to route this data to the proper destination column of the RM. Finally, column buses are used to get the data back to row 1 of the RM. In the actual simulation, many of the rotations can be coupled with the preceding sort, so that the sort result is in rotated order.

Column and row sorting are similar to each other. A subRM of size $n \times N$ is used to sort each column of the $N \times N$ mesh. Notice that since $n = N^2$, we have enough processors to simulate the sort algorithm of Section 2. To perform the sort, we need to do the following:

1. reorganize the data in row 1 of the RM into column-major order
2. sort each column of $N$ data using a sub RM of size $n \times N$

The row-major to column-major transformation can be done using three broadcasts. First, each element in row 1 of the RM computes its destination processor (note that from a processor index and the value of $N$, one can first compute the row and column of the mesh where the element resides, and from these one can compute the element position in column-major order). Once the destination of each element is known, the procedure outlined above for a rotation may be used to reorder the elements. Note that the reordering is done using an RM of size $n \times n$.

To sort each column of size $N$ using an $n \times N$ subRM, we partition the subRM further into chunks of size $N \times N$. Each chunk represents a face of the three-dimensional RM used in Section 2. Using column buses, the data to be sorted can be sent to rows of each chunk; the $i$'th chunk determines the count value for element $i$ of the column being sorted using the steps outlined in the preceding section; using the count values as destination columns, the data is routed back to row 1 of the RM in sorted order using the strategy employed for a rotation. Actually, since the elements are to be stored in row-major order following the sort, we can transorm the destination column address to columns in row-major order and use the entire $n \times n$ RM to obtain the sorted row-major order without creating a sorted column-major order in row 1 of the RM.

Nigam and Sahni [18] have analyzed an optimized version of the above simulation and determined that the simulation uses 120 broadcasts on an RMESH and 81 on an MRN and PARBUS. They have also shown how to simulate Leighton's column sort [8] on an $n \times n$ RM. This simulation requires 139 broadcasts on an RMESH and 59 on an MRN and PARBUS.

## 4 Sorting on an SMBC$(n^2, n, 1)$ or DMBC$(n^2, n, 1)$

Sorting algorithms for SMBCs and DMBCs have been developed in [29, 26, 28]. These algorithms sort $n$ integers. The algorithm of [29] sorts $O(\log n)$-bit integers on an EREW SMBC (without fuse lines) in

$O(\log n)$ time using $n/\log n$ processors and $n^\epsilon$ buses ($\epsilon$ may be any constant greater than 0) and in $O(\log n/\log\log n)$ time on a common CRCW SMBC (without fuse lines) using $n\log\log n/\log n$ processors and $O(n^\epsilon)$ buses. [26] contains a constant time sorting algorithm for $n$ $O(\log n)$-bit integers using a DMBC with $n^{1+\epsilon}$ processors and the same number of buses. The sorting algorithms in [28] are for DMBCs without fuse lines.

[29] defines an operation, called *neighbor localization* that is central to the sorting algorithm developed. In this, the $i$'th processor contains a value $x_i$, $1 \le i \le n$ and each processor is to determine a value $l_i$ which is the nearest processor to its left (processors are ordered left to right by their index) such that $x_{l_i} = x_i$. In case there is no such $l_i$, then define $l_i$ to be zero.

Before seeing how neighbor localization can be done in constant time on a DMBC/SMBC, let use see how such a computer can shift data one processor left. If the DMBC/SMBC has two buses, then bus 1 (2) can be used to shift data left from odd (even) processors to their adjacent even (odd) processors. For this, the odd processors open the segment switch to their right on bus 1 and close the left adjacent switch on this bus. The even processors do this on bus 2. Now, all odd processors write the data to be shifted to bus 1 and even processors write to bus 2. Next, the odd processors read from bus 2 and the even ones from bus 1. When the SMBC/DMBC has only one bus, the shift can be accomplished in two stages. In the first, odd processors shift to even ones and in the second, even processors shift to odd ones.

When the $x_i$ are integers in the range 1 through $n$ neighbor localization can be done in constant time on an EREW DMBC/SMBC with $n$ processors, $n$ buses, and 1 I/O port per processor by having processor $i$ compute $l_{i+1}$ and then shift the computed $l$ value right by one. The steps are:

1. Shift the $x_i$ left by one processor. Let $r_i$ be the values read by the processors during the shift. I.e., $r_i = x_{i+1}$, $1 \le i < n$.

2. Processor $i$ breaks bus $x_i$ at its left by opening the left adjacent segment switch on bus $x_i$, $1 < i \le n$.

3. Processor $i$ writes $i$ to bus $x_i$, $1 \le i < n$.

4. Processor $i$ reads from bus $r_i$ into variable $d_i$, $1 \le i < n$. If nothing is read from the bus, $d_i$ is set to zero.

5. The $d$ values are shifted right by one to get $l_i$, $1 < i \le n$. $l_1$ is set to zero.

Rather than describe how one may sort $O(\log n)$-bit integers on SMBCs and DMBCs, we describe a constant time algorithm to sort arbitrary elements. The algorithm for SMBCs and DMBCs is the same; requires $n^2$ processors, $n$ buses, and one I/O port per bus; and works for the CREW model. Since the required SMBC/DMBC configuration needs $O(n^3)$ area, the constant time sort is not area-time ($AT^2$)-optimal [8]. Note that the constant-time $n \times n$ RM sort of Section 3 is $AT^2$-optimal as an $n \times n$ RM can be realized using $O(n^2)$ area.

Our implementation of count sort on an SMBC/DMBC 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 $j$th 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 $i$, $1 \leq i \leq n$.

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 $j$th sorted element.

To sum the $s$ values in each $n$-processor block (Step 5), the bus dropping scheme of Jenq and Sahni [6] described in Section 2 is adapted. First, the $n$-processor blocks partition the $n$ buses into independent blocks of buses by opening the segment switches to the left of the first processor in each block. Now, each block can work independent of the others. The summing in each block is accomplished in three passes. 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 $2n/3$ $s$-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 processor $2i - 1$ 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 6. This is 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 $q$.

## 5 Conclusion

In this paper, we have reviewed different reconfigurable bus architecures and sorting methods for these. The discussion of the sort methods illustrated some of the techniques used to program reconfigurable bus computers.

The shared memory and distributed memory architectures described here are slight variations of corresponding architectures introduced in [29, 27].

The reconfigurable bus architectures described are powerful in that they can solve certain problems faster than theoretically possible by polynomial-size PRAMs. In addition, these architectures appear to be simpler to program.

Closely related to the content of this paper is the issue of randomized sorting algorithms for reconfigurable bus machines. These are discussed in [21, 22].

## References

[1] Aggarwal, A., Optimal bounds for finding maximum on array processors with $k$ global buses, *IEEE Trans. Comput.*, 35, 1986, pp. 62-64.

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

[3] Y. Ben-Asher, K. Lange, D. Peleg, and A. Schuster, The complexity of reconfiguring network models, *Technical Report*, Haifa University, Haifa, Israel.

[4] Bokhari, S., Finding maximum on an array processor with a global bus, *IEEE Trans. Computers*, 33, 1984, pp. 133-139.

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

[12] 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.

[13] Marberg, J., and Gafni, E., Sorting in Constant Number of Row and Column Phases on a Mesh, *Algorithmica*, 3, 1988, 561-572.

[14] 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.

[15] 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.

[16] 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.

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

[18] M. Nigam, and S. Sahni, Sorting $n$ Numbers on $N \times N$ Reconfigurable Meshes with Buses, *Journal of Parallel & Distributed Computing*, 1994.

[19] V. Prasanna Kumar and C. Raghavendra, Image processing on enhanced mesh connected computers, *Proc. Computer Architecture for Pattern Analysis and Image Database Management*, 1985, pp. 243-247.

[20] V. Prasanna Kumar and C. Raghavendra, Array processor with multiple broadcasting, *Jr. of Parallel and Distributed Computing*, 4, 1987, pp. 173-190.

[21] Rajasekaran, S., Mesh connected computers with fixed and reconfigurable buses: Packet routing, sorting, and selection, *European Conference on Algorithms*, 1993.

[22] Rajasekaran, S., and McKendall, T., Permutation routing and sorting on the reconfigurable mesh, *Parallel Processing Letters* , 1993.

[23] Rothstein. J., On the ultimate limits of parallel processing, *Proc. 1976 International Conference on Parallel Processing*, 1976, pp. 206-212.

[24] Stout, Q., Broadcasting on mesh connected computers, *Proc. 1982 Conf. on Info. Sci. and Sys.*, 1992, pp. 85-90.

[25] Stout, Q., Mesh connected computers with broadcasting, *IEEE Trans. Comput.*, 32, 1983, pp. 826-830.

[26] 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.

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

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

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

[30] 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.