Efficient Bitonic Sorting of Large Arrays on the MasPar MP-1†

Jan F. Prins

TR91 – 041
Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

Abstract

The problem of sorting a collection of values on a mesh-connected distributed-memory SIMD computer using variants of Batcher's Bitonic sort algorithm is considered for the case where the number of values exceeds the number of processors in the machine. In this setting the number of comparisons can be reduced asymptotically if the processors have addressing autonomy (locally indirect addressing), and communication costs can be reduced by careful placement of the data values. We report on the implementation of several related adaptations of Bitonic sort on a MasPar MP-1.

Keywords: SIMD computing, parallel sorting, Bitonic sort, virtualization, domain decomposition, addressing autonomy, MasPar MP-1.

1. Introduction

Many parallel sorting algorithms have been developed to sort P values on a distributed-memory SIMD machine with P processors connected in an $\sqrt{P}\times\sqrt{P}$ mesh. The most widely known of these algorithms are adaptations of Batcher's Bitonic sort [Bat68] or closely related algorithms such as odd-even merge [Bat68] to the mesh [Kum83, Nas79, Tho77]. All of these approaches perform $O(\log^2 P)$ parallel comparisons and make these comparisons using the mesh interconnections in a way that yields communication costs proportional to $O(\sqrt{P})$.

---

†This work supported in part by the Office of Naval Research, Contracts N00014-89-J-1873 and N00014-86-K-0680.

We are interested in the systematic adaptation of algorithms to settings where the number of data elements *N* is larger than *P*, and have chosen Batcher's Bitonic sort for this study because it is relatively simple and well-understood.

There are, however, other sorting algorithms that might be considered. A number of algorithms have been developed specifically to reduce the communication cost toward the lower bound of $4\sqrt{P}$ communication steps on a mesh without wraparound or diagonal connections [Pla89, Sch89, Kun88]. Other algorithms reduce the number of comparisons when $N \gg P$ to achieve optimal speedup with respect to the optimal sequential algorithm. Although one of the variants of bitonic sort also achieves this performance, it does so for values of *N* exponential in the number of processors, while other algorithms, such as the parallel radix sort or one of the probabilistic samplesorts [Rei85] can achieve this performance for smaller values of *N*. As these latter algorithms depend on general routing however, they incur a large overhead on a mesh-connected machine without hardware routing support. In the conclusions section we briefly compare the performance of our Bitonic sort implementations with implementations of these algorithms.

The target of this work is the MasPar MP-1, a mesh-connected SIMD computer. The MP-1 evolved from the MPP [Bat81] and Blitzen [Ble88] but includes some additional features. The aspects of this machine that are important for our purposes are:

- <u>integrated controller</u>
  The parallel portion of the machine consists of a fully-programmable controller and an array of processing elements. The instruction stream specifies operations for the processing elements in the array as well as operations for an integrated 32-bit general-purpose scalar processor. The scalar processor permits algorithms with complex sequential control to be executed without interaction with the host processor. The MPL programming language [Mas90] is a data-parallel extension of C that is used to program the parallel portion exclusively, and is a language roughly at the level of *Lisp for the Connection Machine.

- <u>nearest-neighbor and global communication</u>
  Processors are arranged on a 2-D torus, with each processor connected directly to its eight nearest neighbors. Short distance regular communication is very efficient using this network. In addition, a short-diameter multi-stage crossbar router network is available for arbitrary communication. Since its bandwidth is 1/16 that of the mesh, it is primarily attractive for non-regular communication or long-distance regular communication. The router provides a circuit-switched two-way connection between processors.

- <u>indirect addressing</u>

Each processor has full addressing autonomy within its local memory. However, an indirect (locally developed) reference can take approximately 2-3 times longer than a SIMD-style (same address at all processors) reference.

To sort arrays that have more elements than there are processors, a virtualization technique must be employed. Unlike the Connection Machine, virtualization is not supported directly in the MP-1 instruction set or the MPL language. Instead, the physical machine size is exposed, so that virtualization strategies must be programmed. This yields additional flexibility while retaining efficiency because MPL programs run directly on the controller. Automatic virtualization can be provided by compilation from higher level languages such as the data-parallel Fortran dialect.

2. Abstract Bitonic sort

Bitonic sort of $N=2^n$ elements can be understood as a relatively simple algorithm operating on data arranged in an n-dimensional boolean hypercube A (an n-dimensional array, where each axis has length two). An index $v \in \{0,1\}^n$ specifies a single element in the hypercube. Our convention is that axes are numbered 1 to n and the coordinates on each axis are placed from right to left when forming an index. Figure 1 illustrates the arrangement and indices of the values in a 3-dimensional boolean hypercube holding 8 values. Here $A[101]$ = 'f'.

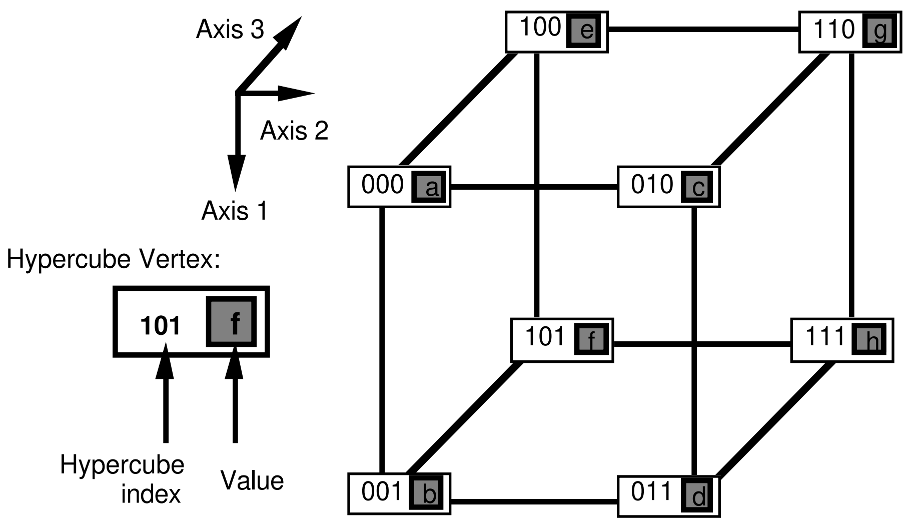

Figure 1. Arrangement of values in a 3-dimensional hypercube

We identify values arranged in a hypercube with the sequence of values obtained by arranging the values in order of increasing index interpreted in base 2 ("odometer" order). For the hypercube in Figure 1, this yields the sequence [a,b,c,d,e,f,g,h]. The expression

$A[v{\approx}]$ for $|v|{\le}n$ yields the sequence (in order of increasing index) of elements whose index is vw for some $w \in \{0,1\}^{n-|v|}$. For the hypercube in Figure 1, we have $A[0{\approx}] = [a,b,c,d]$.

The Bitonic sort algorithm is expressed on the boolean hypercube as follows:

```
for  i:= 1 to n do   {inv: I1}
    for j := i downto 1 do   {inv: I2}
        compare-exchange on axis j of A
        where (index(A) at axis i+1 is 1) do
            exchange on axis j of A
```

To explain the algorithm we define some terms and state the invariants. The compare exchange (CE) operation on axis j compares, for every $v \in \{0,1\}^{n-j}$ and $w \in \{0,1\}^{j-1}$ the elements $A[v0w]$ with $A[v1w]$ and exchanges the two values if the former is larger than the latter. The proposition $s\Uparrow$ holds if s is a monotonically non-decreasing sequence, $s\Downarrow$ holds if s is a monotonically non-increasing sequence, and $s\Updownarrow$ holds if s is a bitonic sequence, i.e. s is a circular shift of uv where $u\Uparrow$ and $v\Downarrow$. For sequences s and t the relation $s \le t$ holds if *every* value in s is less than or equal to *every* value in t..

The invariant of the outermost iteration,

I1: for all $a \in \{0,1\}^{n-i}$: $A[a0{\approx}]\Uparrow$ and $A[a1{\approx}]\Downarrow$

states that at each iteration successive (i–1)-dimensional subcubes alternately hold increasing and decreasing values. From this it follows that all i-dimensional subcubes of A are bitonic. The invariant of the inner iteration,

I2: (for all $a \in \{0,1\}^{n-j}$: $A[a{\approx}]\Updownarrow$)

and

(for all $v,w \in \{0,1\}^{i-j}$ and $a \in \{0,1\}^{n-i-1}$

$v<w \Rightarrow$

$(A[a0v{\approx}] \le A[a0w{\approx}]$ and $A[a1v{\approx}] \ge A[a1w{\approx}]))$

states that at each iteration successive subcubes of dimension j are bitonic, and that these cubes are in order relative to each other (≤) although they may not be in order internally. On termination of the inner loop, with j=0, we have, in each subcube of dimension i, bitonic sequences of length 1 in order, yielding totally-ordered subcubes of dimension i. These subcubes are alternately in increasing and decreasing order, allowing i to increase while maintaining *I1*. On termination of the outer loop, with i=n+1, the first n-dimensional subcube of A (which is all of A) is in increasing order.

When $A[a\approx]\Updownarrow$ for $a\in\{0,1\}^{n-j}$, the compare-exchange operation on axis j guarantees $A[a0\approx] \le A[a1\approx]$ and $A[a0\approx]\Updownarrow$ and $A[a1\approx]\Updownarrow$. To maintain *I2* under decrement of j we must insure the direction of the ordering is correct for the value of index i+1. If the results of the CE operations are exchanged on axis j, the inequality above is reversed and we end with a decreasing sequence when j=0. The *index(A)* operation yields a hypercube of size and shape of A in which the elements are the corresponding indices of A, hence the **where** condition holds for those elements that are in the larger index subcube of dimension i on axis i+1. Since the CE and the exchange operate on the same values, they can be combined into a single operation and we shall count them as such.

Consequently, a simple analysis of the algorithm above confirms that it performs

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \frac{n(n+1)}{2} = \frac{\log^2 N + \log N}{2} = O(\log^2 N)$$

parallel compare-exchange operations.

To adapt the abstract Bitonic sort to our target machine we have to solve two kinds of problems. First, with $N > P$ elements we have to choose how to *virtualize* the algorithm to apply when each processor holds multiple values in A.. Second, since the algorithm is formulated on a hypercube and we are interested in implementing it on a machine with a mesh topology, we have to *embed* the values on the hypercube into the mesh. We consider first the virtualization strategies.

3. Virtualization strategies

We assume that we have $P = 2^m$ processors, and $N = 2^n$ elements with $m<n$. We consider three different adaptations of the Bitonic sort algorithm reflecting three different treatments of multiple values per processor.

The first and simplest virtualization strategy is to reduce the n dimensional hypercube to an m dimensional hypercube, each element of which is an $n-m$ dimensional hypercube. Figure 2 gives an example of the decomposition. The idea is that the m-dimensional hypercube can be embedded in the P processor mesh while the $n-m$ dimensional elements will reside within each processor memory. We refer to this approach as *hypercube virtualization* because it preserves the hypercube structure of the original algorithm, although we must alter the interpretation of the CE operation. Each parallel CE operation on an axis in the m-dimensional hypercube corresponds to N/P successive comparisons of $P/2$ parallel pairs of corresponding components of elements at each end of the axis. A parallel CE operation on an axis contained within the elements consists of $N/2P$ successive comparisons of P parallel pairs of components of elements. Since CE operations are performed most frequently on lower numbered axes, the natural choice is to place these axes within elements since fewer sequential comparisons are involved in CE operations on axes within elements.

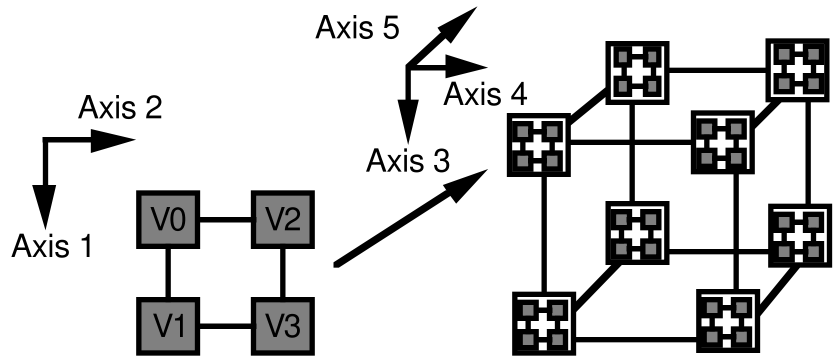

Figure 2. Hypercube virtualization of 5-dimensional hypercube into a 3-dimensional cube of 2-dimensional cube elements

There are $\log^2 N$ parallel CE operations in the abstract algorithm and each CE operation involves $O(N/P)$ parallel comparisons, hence the total number of parallel comparisons using P processors is

$$O\left(\frac{N}{P}\log^2 N\right) \tag{1}$$

which is not work efficient relative to an $N \log N$ sequential sort.

A second virtualization strategy is to again reduce the problem to an m-dimensional hypercube, but choose the elements to be sequences of size N/P. We refer to this strategy as *sequence virtualization*.; an illustration is given in Figure 3.

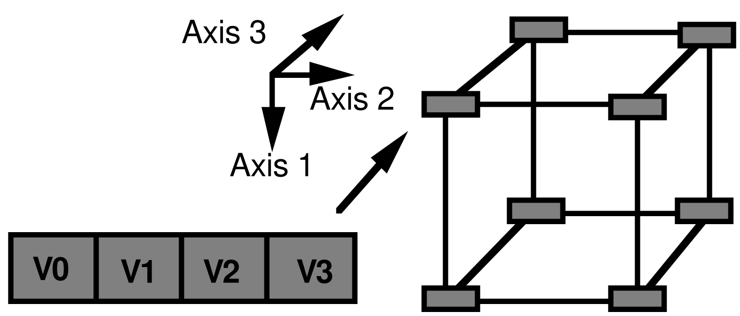

Figure 3. Sequence virtualization of 5-dimensional hypercube into a 3-dimensional cube of length 4 sequence elements

For this strategy we change the interpretation of CE so that it operates on sequences of length *N/P*. For *s* and *t* sequences, CE(*s,t*) operation yields sequences *v, w* such that vw = perm(*st*) and $v \leq w$ (using the definition of ≤ for sequences). Thus a CE operation partitions two sequences into the smallest *N/P* values and the largest *N/P* values of the combined sequences. This definition satisfies the properties required of CE in the abstract Bitonic sort algorithm to establish and maintain invariants I1 and I2. On termination we are guaranteed that the elements of the *m*-dimensional hypercube are totally ordered with respect to ≤. If, in addition, we know that each of the sequences is ordered with respect to the ≤ relation on sequence elements, then we can conclude that all *N* values are sorted.

If we start with a sorted sequence at each vertex of the *P* element hypercube, then the CE operation on sequences can be implemented by a merge followed by a partition in half, with each half remaining a sorted sequence. Note that, in general, the merge step will lead to a different merge order at each processor, hence the pattern of references must be locally determined and requires addressing autonomy to implement. An advantage of this virtualization strategy is that *N/P* need not be a power of 2; any data-set of size *c•P* can be sorted on a *P* processor machine for any positive integer *c* using this strategy.

A Bitonic sort using sequence virtualization performs O($\log^2 P$) CE-operations, each of which requires O(*N/P*) compare operations to perform the merge. In addition, we must sort the initial sequence of length *N/P* at each processor. With indirect addressing, this could be done with an O((*N/P*) log (*N/P*)) comparison sequential merge sort algorithm which can be adapted for SIMD execution because the size and location of subsequences to be merged in each step is fixed. Thus, sequence virtualization performs

$$O\left(\frac{N}{P}\left(\log\frac{N}{P} + \log^2 P\right)\right) \qquad (2)$$

compare operations which is an asymptotic improvement over (1) for large values of *N* compared to *P*, and achieves optimal speed-up with respect to the sequential algorithm.when *P* is less than $(\log N / \log\log N)^2$ [Bau78].

The final virtualization strategy is a variant of hypercube virtualization in which the algorithm is transformed as follows to always perform CE operations on a the first $k = n–m$ axes.

```
for  i:= 1 to n do
    for j := i downto 1 do
        if j≥k then
            transpose axis j and k of A
            CE on axis k
```

transpose axis j and k of A
if j<k **then**
CE on axis j

Since the first *k* axes are placed in memory under hypercube virtualization, every CE operation can be realized with N/2P successive parallel comparisons of *P* values. Thus fewer parallel comparisons are made using this approach. Since the virtualization technique varies the organization of data in the hypercube, we call it a *varying hypercube* virtualization.

Figure 4 illustrates the data movement required in the hypercube to transpose an element axis *k* (axis 2 in this case) with a hypercube axis *j* (axis 5 in this case). In each element there are N/2P values whose indices on axis j and k differ. These values must be exchanged with N/2P values in the element at the opposite side of axis j. Hence a total of N/P values move; this is the same number as are moved in bringing together the values for a CE operation on a hypercube axis.

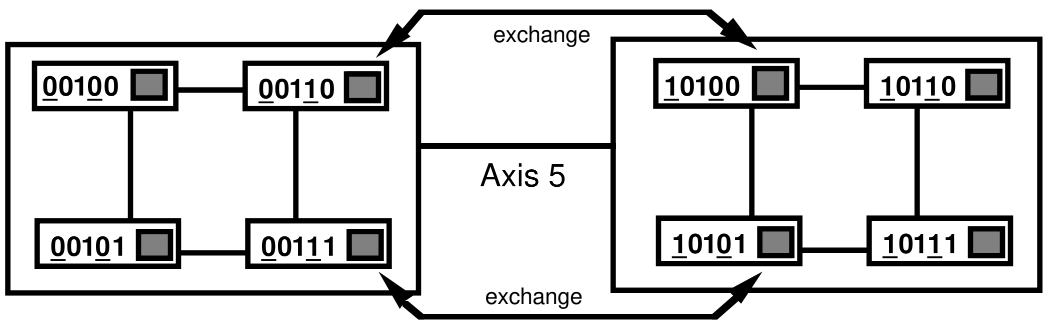

Figure 4. Data movement required for the transposition of a hypercube axis (axis 5) with an element axis (axis 2).

4. Embedding

Now we consider the embedding choices. To choose an embedding that minimizes communication, we must examine the operations in the abstract algorithm and assess their communication cost. The only operations that involves communication are the CE operations on hypercube axes and the transposition of hypercube axes. Since the CE operation is applied more frequently to lower dimensions of the hypercube, an embedding that places values along lower-numbered axes closest together is desirable.

All the virtualizations have reduced the hypercube to the leading m axes, so that we are considering the embedding of an m-dimensional hypercube in $P = 2^m$ processors arranged in a $2^{m/2} \times 2^{m/2}$ mesh. We assume m is even; it is simple to extend the following to the case where m is odd. For simplicity we will renumber the hypercube axes $1..m$.

A *row-major* embedding of a boolean hypercube maps each successive dimension to processors successive powers of two apart along the first row, wrapping across rows when the stride exceeds the number of processors per row on the mesh. Hence the stride on the mesh between elements along hypercube dimension *i* is given by:

$$C_{rm}(i) = \begin{cases} 2^{i-1} \text{ direction East, if } i\text{-}1 < m/2 \\ 2^{i-m/2-1} \text{ dir South, if } i\text{-}1 \geq m/2 \end{cases}$$

The row-major embedding can be improved by using the short-distance strides in both directions on the mesh first. This is the *balanced-axis* embedding, in which the stride between the elements along the mesh between elements along hypercube dimension *i* is:

$$C_{ba}(i) = \begin{cases} 2^{((i-1)/2)} \text{ direction East, if i odd} \\ 2^{((i-2)/2)} \text{ direction South, if i even} \end{cases}$$

Since

$$\sum_{i=1}^{n} \sum_{j=1}^{i} C_{rm}(i) \geq \sum_{i=1}^{n} \sum_{j=1}^{i} C_{ba}(i)$$

a balanced axis embedding results in lower communication cost than row major embedding, although the final sorted sequence is not in row-major order.

Thus far we have treated the MP-1 only as a mesh-connected machine, and have ignored diagonal connections and the torus topology. The *xnet* embedding uses these connections on a square mesh [Law90]. Its stride function is given by

$$C_{xn}(i) = \begin{cases} 1 \text{ direction NE, if } i=1 \\ 1 \text{ direction SE, if } i=2 \\ 1 \text{ direction E, if } i=3 \\ 2^{((i-2)/2)} \text{ direction NE, if i even} \\ 2^{((i-3)/2)} \text{ direction SE, if i odd} \end{cases}$$

and results in lower communication cost than either of the other embeddings.

Finally, as pointed out in [Fla82], it is not necessary that the embedding be fixed throughout the course of the algorithm. We can use this observation to perform an additional transformation of the varying hypercube virtualization of Bitonic sort with $k = n\text{-}m$::

The idea is that a function $M[j]$ records which original hypercube axis currently occupies axis *j*. The function *M* changes whenever a transposition is performed. To implement the transpose between axis k and j, we find the axis $M^{-1}j]$ currently occupied by j and perform the transpose with axis k:

```
let M be the identity function on 1..n
for  i:= 1 to n do
    for j := i downto 1 do
        if j≥k then
            transpose axis M^-1[j] and axis k of A
            M[k],M[M^-1[j]] := j, M[k]
            CE on axis k
        if j<k then
            CE on axis j
```

For a given initial embedding of the hypercube, the total communication distance for the transformed algorithm is the same as that obtained if the embedding were fixed. The advantage lies in the fact that only one transpose operation is required per CE instead of two, hence half the values are moved.

5. Analysis

The cost of Bitonic sort under each choice of virtualization and embedding is given by the expression

$$T_{i,j}\,(N,P) = (a_{i,j}\,C_i \;+\; b_{ij}T_j\,) \bullet L_i$$

where a_{ij} and b_{ij} are implementation-specific constants and the remaining factors are determined as follows:

virtualization i	comparisons C_i	loading L_i
sequence	$\log N/P \;+\; \log^2 P$	2N/P
hypercube	$(\log^2 N)$	2N/P
var.-hypercube	$(\log^2 N)$	N/P

embedding j	unit shift steps T_j
row-major	$(\log 16\sqrt{P})\sqrt{P} - 3/2 \log P - 4$
balanced-axis	$7\sqrt{P} - 2 \log P - 7$
xnet	$5\sqrt{P} - \log P - 5$

5. Implementations

Figure 5 illustrates the space of possible implementations corresponding to the virtualization and embedding strategies discussed. The implementations actually undertaken are identified in the Figure. Some implementations were not attempted because their performance would clearly be inferior.

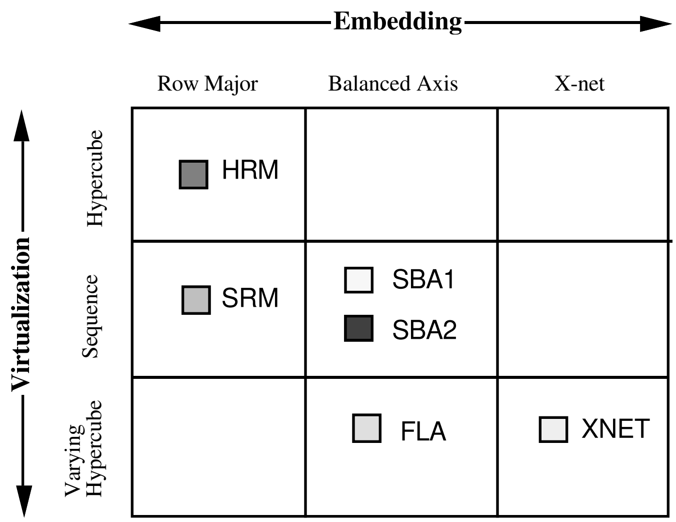

Figure 5. Implementation choices and implementations attempted.

The Sequence Row Major (SRM) and Hypercube Row Major (HRM) variants were implemented to show how CE operations affect performance. The two Balanced Axis (SBA1 and SBA2) implementations were used (1) to show how decreased communication affects performance and (2) to study more closely the costs of indirect addressing. The Varying Hypercube virtualization was implemented with two intitial embeddings (FLA and XNET). The following remarks are based on the implementations described above.

<u>initial sort</u>

For the initial one-time intraprocessor sort in sequence virtualization we used a merge-exchange sort [Dij87] [Bat68], which is an O($n \log^2 n$) sort. Although it is possible to write an asymptotically better merge sort as suggested in the previous section, the constants are worse for achievable values of *N/P* with the current per-processor memory size. A radix sort becomes competitive at the largest values of *N/P*, but all such sorts require extra space. The pre-sort is the only operation in sequence virtualization that need be non-linear in *N/P* (each CE in sequence virtualization is a linear-time merge) and we find that only at the very largest feasible inputs can we notice this asymptotic effect, suggesting that the comparisons at the pre-sort stage are dominated by the comparisons in the bitonic stage. The merge-exchange sort is oblivious [the comparison pattern is independent of the comparison outcomes] and hence need not use indirect addressing, yet unlike bitonic sort works for any sequence size, not just those with length a power of 2.

sequence compare-exchange

In sequence virtualization the CE operation is a merge of two sequences followed by a split in half. The two sequences can be merged most rapidly into a fixed destination sequence. Each step places one element in the next position of the destination sequence, hence the destination sequence can be directly addressed. By including sentinels at the end of the two sequences to be merged we can eliminate "run-out" sections of code which have surprising performance implications in this setting. Consider that in different processors different merge orders are found hence it is possible that a merge loop followed by the two cases of run-out loops all run the maximum number of iterations (four times the length of the sequences to be merged).

The half of the merged sequence that is returned to the other processor on the hypercube axis depends on the value of *index(A)* at axis $i+1$. A separate copy step is used in SBA1 to make this split in all processors. An alternative implementation in SBA2 used three rotating buffers for the merge. The conditional reversal of the merged sections was then accomplished by a pointer swap. This makes the number of references small but all are indirect. Note that sequence virtualization requires storage at each processor that is three (SBA2) or four (SBA1) times the space required for hypercube virtualization.

hypercube compare-exchange

In the transpose and hypercube CE operation the axes of the hypercube elements are mapped into memory identically at each processor, hence these operations need not use any indirect addressing. Since the N/P element CE can be carried out as N/P single element communication and compare steps, no extra space is needed for these implementations.

communication

We use the mesh connections for all regular communication, because of their great bandwidth advantage over the router. On a 64x64 processor machine even the longest stride movement (length 32) is competitive with the routing time. The router was used to restore layouts produced by the non row-major order embeddings (SBA1, SBA2, FLA, XNET) to row-major order. We decided arbitrarily that row-major order was required of the result.

6. Performance

It is useful to analyze performance in relation to the virtualization ratio VPR = N/P. For a given size of machine and choice of embedding the *communication costs* are linear in VPR and independent of the virtualization technique. The *comparison costs* grow non-linearly with increasing VPR, and are independent of the embedding. In sequence virtualization the only non-linear component of comparison cost is in the initial sort, while in hypercube virtualization the cost grows proportional to $\log^2 N$. Since both virtualization approaches incur comparison costs from the inter-processor CE operations, the effect of comparison costs becomes pronounced when N is significantly larger than P.

The sort time of the implementations described on a 4096 processor MP-1 sorting 32-bit integer data values are shown in Figure 6 and displayed graphically in Figure 7. The times are in milliseconds/VPR.

	Hypcube	Sequence			Var. Hypcube	
	rowmaj	rowmaj	balaxis	balaxis	balaxis	xnet
VPR	HRM	SRM	SBA1	SBA2	FLA	XNET
2	19.8	21.4	21.4	30.3	6.8	5.0
8	19.5	20.7	19.5	25.9	5.8	4.7
32	19.4	19.6	18.0	23.7	5.9	4.9
128	20.4	19.7	18.1	23.3	6.5	5.4
512	22.1	19.7	18.2	23.5	7.1	5.9
2048	24.0				7.8	6.5

Figure 6. Implementation timings in milliseconds/VPR on 4096 processor MP-1

It is clear from the numbers that the comparison operations contribute a relatively small amount to the cost of all implementations. This is likely due to the basic speed of the ALU compared with the speed of communication. Only at large input problem sizes do comparison costs differ significantly between virtualization techniques and the magnitude of this effect is small by comparison to the improvements due to superior embeddings. On large machines the point at which sequence virtualization becomes significantly faster will be reached at higher VPR than on smaller machines due to higher communication costs.

It is interesting to note that SBA2 replaces 4 indirect references and 6 direct references within a CE (per element of the sequence) with 6 indirect references and zero direct references yet SBA2 runs 34% slower than SBA1. This suggests a factor higher than 2-3 for indirect vs. direct memory references. It is likely that this represents limitations of the current MPL compiler or conflicts in overlapping indirect memory references with computation. But it also suggests that any speedup of indirect addressing will improve the relative performance of sequence versus hypercube virtualization.

With the cost to route the sorted results to row-major embedding included, the balanced axis (SBA1) and the varying balanced axis (FLA) implementations still are superior to a bitonic sort using a row-major embedding.

The performance of FLA and XNET may appear anomalous compared to the other implementations. Even given that FLA has half the communication cost of HRM and uses a better embedding, a performance improvement ratio of >3 seems high. It is likely that some of the difference can simply be attributed to programming. FLA and XNET's transpose operations fits the machine particularly well and both implementations were programmed last.

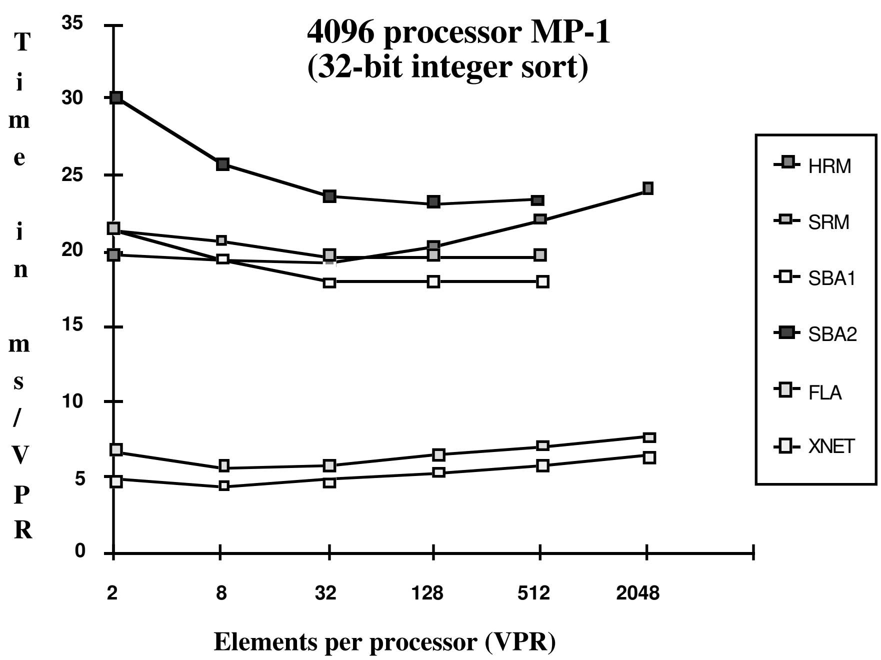

Figure 7. Performance of implementations

7. Conclusion

The bitonic sort algorithm is suitable for distributed memory implementation precisely because the structure of the hypercube guarantees that each value is involved in exactly one CE operation at a time (no contention). It is suitable for a mesh-connected SIMD computer because the regularity of the CE operations on the hypercube can be preserved in the embedding into lower-dimensional meshes.

In the course of this work it became clear that the notions of virtualization and embedding as we defined them were orthogonal, and their impact on performance could be separated. Although sequence virtualization offers the asymptotically best sort, for the range of VPR that we can accommodate on our current machine, the improvement is relatively small and the space penalty severe. Hence our conclusion is that the most reasonable large-array sort for this machine is a varying-hypercube virtualization with the processor axes transposed dynamically within an xnet embedding. However, the xnet embedding requires a square mesh, and thus MasPar machines with an odd power of 2 processors (2048, 8192) could not run this implementation. Thus the vbsort implementation that we distribute uses the balanced axis embedding and runs on all MasPar machines. As can be seen in Figure 7, this implementation achieves nearly the same performance as the xnet embedding.

The vbsort implementation of bitonic sort is fast and the most usable sort we have developed on MasPar MP-1 machines to date. We also undertook implementation of radix and samplesort [BL+91]. Both of these algorithms have significantly higher memory requirements and, in fact, samplesort is not realistically implementable on current MasPar machines since the splitter table used at each processor has size proportional to the number of processors and exceeds the local memory capacity on MasPar machines with large numbers of processors. Radix sort becomes competitive with vbsort when *n/p* > 128 and has better scaling behavior thereafter. There two disadvantages of radix sort are that it performs poorly at small n/p and that it is difficult to efficiently sort arbitrary-sized records. Bitonic sorting offers some significant advantages over both of these sorts: in-place operation and deterministic performance independent of input distribution. Hence the vbsort implementation has developed significant use in practice.

Acknowledgement

We wish to thank Peter Lawrence of MasPar for providing an initial implementation of bitonic sort using hypercube virtualization and for many interesting and helpful subsequent discussions on parallel sort implementations.

References

[Bat68] K.E. Batcher, "Sorting networks and their applications," in *Spring Joint Computer Conf., AFIPS Proc.*, vol.32, 1968, pp.307-314.

[Bat81] K.E. Batcher, "Design of a Massively Parallel Processor", *IEEE Trans. on Computers*, C-29(9), 1981, p836-840.

[Ble88] D.W. Blevins, E.W. Davis, R.A. Heaton, J. Reif, "BLITZEN: A highly integrated massively parallel machine", Proc. Frontiers88, IEEE 1988.

[BL+91] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha; A Comparison of Sorting Algorithms for the Connection Machine CM-2; *3rd Annual ACM Symposium on Parallel Algorithms and Architectures,* July 21-24, 1991, Hilton Head, SC.

[Bau78] G. Baudet and D. Stevenson, "Optimal sorting algorithms for parallel computers", *IEEE Trans. Comput.*, vol.C-27, no.1, pp.84-87, Jan. 1978.

[Dij87] E.W. Dijkstra, "A heuristic explanation of Batcher's Baffler", EWD953 U. Texas Austin.

[Fla82] P.M. Flanders, "A unified approach to a class of data movements on an array processor", *IEEE Trans. Comput.*, vol.C-31, no.9, Sep. 1982.

[Kum83] M. Kumar and D.S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," *IEEE Trans. Comput.*, vol.C-32, Mar. 1983.

[Knu&3] D.E. Knuth, "The Art of Computer Programming, vol-3 Searching and Sorting" Addison-Wesley, 1973.

[Kun88] M. Kunde Routing and sorting on mesh connected arrays", in VLSI Algorithms and Arch., J.H. Reif, ed. , LNCS 319, Springer-Verlag 1988.

[Law90] P. A. Lawrence, personal communication.

[Mas90] MasPar Computer Corporation, "MasPar Parallel Application Language (MPL) Reference Manual", 2nd beta edition, Doc #9302-0000-0690, Feb. 1990.

[Nas79] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. Comput.*, vol.C-27, no.1, pp.2-7, Jan. 1979.

[Sch79] I.D. Scherson and S. Sen, "Parallel sorting in two-dimensional VLSI models of computation," *IEEE Trans. Comput.*, vol.38, no.2, pp.238-249, Feb. 1989.

[Sto71] H.S. Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. Comput.*, vol C-20, pp.153-161, Feb.1971.

[Tho77] C.D.Thompson and H.T.Kung, "Sorting on a mesh-connected parallel computer," *CACM*, vol.20, pp.263-271, Apr. 1977.