Asian Journal of Information Technology 4 (11): 1027-1030, 2005


# An Efficient Approach for Handling Large Arrays in Parallel Computing Environment Using Bitonic Sorting

Ahmed Shamsul Arefin, Muhammad Arifur Rahman, Mohmmad Mamun Kabir,
Abu MD Zafor Alam, Shamim Al Mamun and M. Lutfar Rahman
Department of Computer Science and Engineering, Daffodil International University,
102 Shukrabad, Mirpur Road, Dhaka, Bangladesh

**Abstract:** In this study we would like to introduce an efficient variant of Bitonic sorting that can be used with sorting large arrays in distributed computing environment. The problem of sorting a collection of values on a mesh-connected distributed-memory computer using our 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.

**Key words:** Bitonic sorting, SIMD, hypercube, distributed computer, parallel algorithm

## INTRODUCTION

Many parallel sorting algorithms have been developed to sort P values on a distributed memory machine with P processors connected in an $\sqrt{P}$ X $\sqrt{P}$ mesh. The most widely known of these algorithms are adaptations of Batcher's bitonic sort[1] or closely related algorithms such as odd-even merge[1] to the mesh[2-4]. 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 $O(\sqrt{(P)})$.

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.

To sort arrays that have more elements than there are processors, a virtualization technique must be employed. Automatic virtualization can be provided by compilation from higher level languages such as the data-parallel Fortran dialect.

## BITONIC SORT FORMULATION

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 shows the arrangement and indices of the values in a 3-dimensional boolean hypercube holding 8 values. Here A(101) = f.

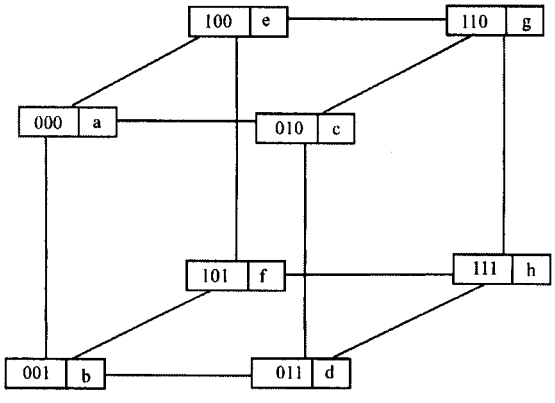


Fig. 1: Arrangement of values in 3D hypercube

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. For the hypercube in Fig. 1, this yields the sequence [a, b, c, d, e, f,g and h]. The expression A[v ≈]for |v| = 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 Fig. 1, we have A[0 ≈] = [a, b, c, d][5-7].

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

**Corresponding Author:** Ahmed Shamsul Arefin, Department of Computer Science and Engineering, Daffodil International University, 102 Shukrabad, Mirpur Road, Dhaka, Bangladesh

1. for i:= 1 to n do
2. for j := i downto 1 do
2.1 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 $s\leftrightarrow$ 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 = t holds if every value in s is less than or equal to every value in t.

Consequently, a simple analysis of the algorithm above confirms that it performs O ($\log^2 N$) parallel compare-exchange operations.

To adapt the abstract Bitonic sort first, 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.

## PROBLEM SOLUTION

We assume that we have $P=2^m$ processors and $N=2^n$ elements with m<n. We consider 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

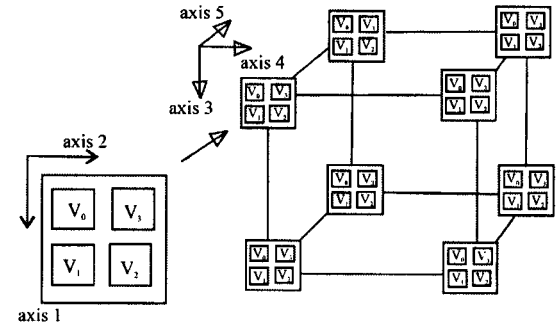


Fig. 2: Hypercube virtualization of 5-dimensional hypercube into 3-dimensional cube of 2-dimensional cube elements

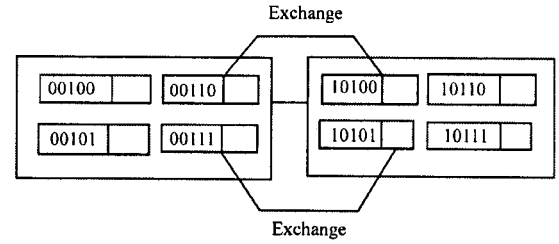


Fig. 3: Data movement required for the transposition of a hypercube axis(axis 5) with an element axis (axis 2)

natural choice is to place these axes within elements since fewer sequential comparisons are involved in CE operations on axes within 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 ($N/P\log^2 N$). Figure 3 shows 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 (Fig. 3). Data movement required for the transposition of a hypercube axis (axis 5) with an element axis (axis 2).

## IMPLEMENTATION

The final virtualization strategy is a variant of hypercube virtualization in which the algorithm is

xtransformed as follows to always perform CE operations on a the first k = n–m axes.

```
1. for i:= 1 to n do
   2.for j := i down to 1 do
2.1.          if j => k then
      transpose axis j and k of A
      Compare-Exchange(CE) on
      axis k
```

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 = 2m processors arranged in a 2m/2 ·2m/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.

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
1.for i:= 1 to n do
2.  for j := i down to 1 do
2.1.     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. 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.

Table 1: Implementation of Timing in milliseconds/VPR

| | | | | | Virtual hypercube | |
|---|---|---|---|---|---|---|
| VPR | HRM | Rowmaj SRM | Sequence balaxis SBA1 | Balaxis SBA2 | Balaxis FLA | Xnet 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 |

HRM = Hypercube row major, SRM = Sequence Row Major, SBA= Sequence Balaxis, FLA = Virtual Hyper Cube, Balaxis, XNet = Virtual Hyper cube Xnet.

Table 2: Cost analysis with virtualization

| Virtualization i | Comparisions C | Loading I.j |
|---|---|---|
| sequence log | $N/P + \log^2 P$ | 2N/P |
| Hypercube | $(\log^2 N)$ | 2N/P |
| Var.-hypercube | $(\log^2 N)$ | N/P |

Table 3: Cost analysis with embedding

| Embedding j | Unit shift steps Tj |
|---|---|
| Row-major | $(\log 16\sqrt{P})\sqrt{P} - 3/2 \log P - 4$ |
| Balanced-axis | $7\sqrt{P} - 2 \log P - 7$ |
| Xnet | $5\sqrt{P} - \log P - 5$ |

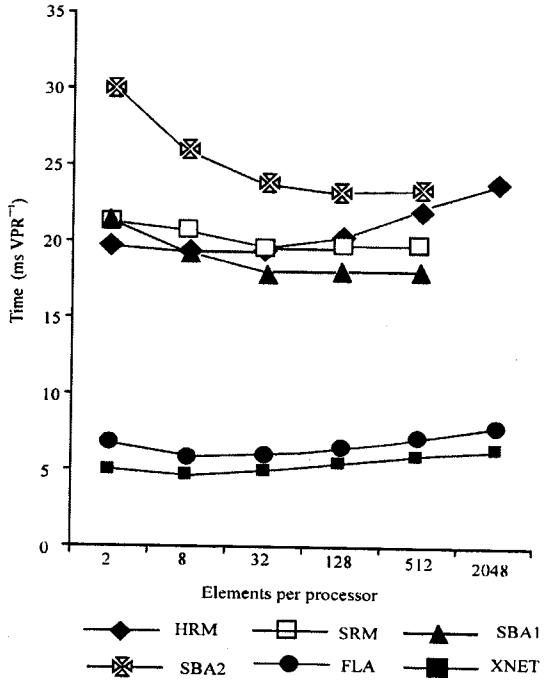


Fig. 4: Performance of implementations

## ANALYSIS

For the initial one-time intra-processor sort in sequence virtualization we used a merge-exchange sort[1,8], which is an $O(n \log^2 n)$ sort. Although it is possible to

write an asymptotically better merge sort, 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[9].

It is useful to analyse performance in relation to the virtualization ratio VPR = N/P (Table 2). 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 nonlinearly with increasing VPR and are independent of the embedding (Table 3). 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 $\log^2 N$[10,11]. 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 SIMD sorting 32 bit integer data values are shown in displayed graphically in (Fig. 4). The times are in milliseconds/VPR (Table 1).

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:

## CONCLUSIONS

Present 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. It is suitable for a mesh-connected distributed computer because the regularity of the CE operations on the hypercube can be preserved in the embedding into lower-dimensional meshes.

## REFERENCES


1. Knuth, D.E., 1973. The Art of Computer Programming, Searching and Sorting. Vol. 3 Addison-Wesley.
2. Kumar, M. and D.S. Hirschberg, 1983. An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes. IEEE Trans. Computers, C: 32.
3. Nassimi, D. and S. Sahni, 1979. Bitonic sort on a mesh-connected parallel computer. IEEE Trans. Computers, 27: 2-7.
4. Thompson, C.D. and H.T. Kung, 1977. Sorting on a mesh-connected parallel computer. CACM., 20: 263-271.
5. Batcher, K.E., 1968. Sorting networks and their applications. Spring Joint Computer Conf. AFIPS Proc., 32: 307-314.
6. Batcher, K.E., 1981. Design of a massively parallel processor. IEEE Trans. Computers, 29: 836-840.
7. Baudet and D. Stevenson, 1978. Optimal sorting algorithms for parallel computers. IEEE Trans. Computers, 27: 84-87.
8. Dijkstra, E.W., A heuristic explanation of Batcher's Baffler. EWD953 U.Texas Austin.
9. Prins, J.F., Efficient bitonic sorting of large arrays on the maspar MP-1.J. University of North Carolina, TR91-041.
10. Bilardi, G. and A. Nicolau, 1989. Adaptive bitonic sorting: An optimal parallel algorithm for shared-memory machines, pp: 18.
11. Brockmann, K. and R. Wanka, 1997. Efficient oblivious parallel sorting on the maspar MP-1. Proceedings of the 30th Hawaii International Conference on System Sciences: Software Technology and Architecture, 1: 200.