ELSEVIER

Parallel Computing 22 (1996) 595–606

PARALLEL COMPUTING

Data broadcasting and reduction, prefix computation, and sorting on reduced hypercube parallel computers [†]

Sotirios G. Ziavras [*], Arup Mukherjee

Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102, USA

Received 3 March 1994; revised 7 December 1994, 29 June 1995, 31 October 1995

Abstract

The popular hypercube interconnection network has high wiring(VLSI) complexity. The reduced hypercube (*RH*) is obtained by a uniform reduction in the number of channels for each hypercube node in order to reduce the VLSI complexity. It is known that the *RH* achieves performance comparable to that of the hypercube, at much lower hardware cost, through hypercube emulation. The reduced complexity of the *RH* permits the construction of powerful, massively parallel computers. This paper proposes algorithms for data broadcasting and reduction, prefix computation, and sorting on the *RH*. These operations are fundamental to many parallel algorithms. A worst case analysis of each algorithm is given and compared with that of equivalent algorithms for the hypercube. It is shown that the proposed algorithms for the *RH* yield performance comparable to that of the hypercube.

Keywords: Hypercube architecture; Reduced hypercube interconnection network; VLSI complexity; Data broadcasting; Data reduction; Prefix computation; Sorting

1. Introduction

The n-dimensional (direct binary) hypercube, or n-cube, is a general purpose interconnection network in parallel processing and has been widely researched [1,9]. It has 2^n nodes. Two nodes are neighbors iff their n-bit addresses differ in a single bit. The n-cube has small diameter, equal to n. It can also emulate very

[*] Corresponding author. Email: ziavras@hertz.njit.edu

[†] This work was supported in part by the National Science Foundation under Grants CCR-9109084, CDA-9121475, and DMI-9500260.

efficiently other widely used topologies [10–14]. Nevertheless, the total number of communication channels increases dramatically with an increase in the total number of processors in the hypercube system [1,17]. This VLSI constraint prevents manufacturers from building powerful, massively parallel hypercube systems. The high VLSI complexity of the hypercube has led many researchers to look into hypercube-like topologies with lower VLSI complexity [1,2,4,7,16–18]. The reduced hypercube (RH) is such a variation [2]. Algorithms for embedding linear arrays, meshes, and binary trees into RHs were presented in [15]. We develop here algorithms for data broadcasting and reduction, prefix computation, and sorting on RHs. All of the proposed algorithms for the *RH* are compared with relevant algorithms for the (regular) hypercube.

2. The reduced hypercube

The reduced hypercube (*RH*) interconnection network was proposed by Ziavras [2] in order to reduce the large VLSI complexity of the hypercube, and thus to facilitate the construction of large(powerful), massively parallel systems. A *RH* can be viewed as a hierarchical structure. The properties of *RH* systems with two levels were studied in [2]. Generalized *RH* systems with many levels were investigated in [18]. The algorithms developed in this paper assume the interesting class of *RH*s with two levels. Such a *RH* is formed by uniformly removing several edges from the hypercube with the same number of nodes. The *reduced hypercube* $RH(k,n)$ contains a total of N nodes, where $N = 2^{k+2^n}$, with $k \geq n$ and $n > 0$. Each node(processor) in the $RH(k, n)$ is attached to $k + 1$ edges. In the hypercube with the same number of nodes each node is attached to $k + 2^n$ edges. Therefore, each node in the N-node *RH* has $k + 2^n - (k + 1)$ or $2^n - 1$ less edges than each node in the N-node hypercube.

The N-node $RH(k, n)$ is constructed from the N-node hypercube by uniformly removing $2^n - 1$ edges from each of its nodes. To accomplish this, the $(k + 2^n)$-bit addresses of hypercube nodes are first partitioned into two fields, the $0th$ and $1st$ fields, as follows. The $0th$ field contains the k least significant bits of the $(k + 2^n)$-bit node address. This field represents the address of the node within a complete k-cube, which is referred to as a *building block (BB)*. The $1st$ field contains the 2^n most significant bits of the node address. It represents the address of the *BB* that contains the node. In addition, a subfield is identified in the $0th$ field, the $0th$ *subfield*. It contains the n most significant bits of the k-bit $0th$ field. It represents the address of a $(k - n)$-dimensional subcube, which is referred to as a *subblock (SB)*, within the k-cube *BB* that contains the node. Therefore, the address format is as given below, where the symbol '$\cdot$' denotes concatenation:

$$\underbrace{\overbrace{BB\_address}^{2^n\ bits}}_{1st\ field} \bullet \underbrace{\overbrace{\overbrace{SB\_address}^{n\ bits} \bullet \overbrace{Node\_address\_in\_SB}^{k-n\ bits}}^{k\ bits}}_{0^{th}\ field}$$

For simplicity, let the term $k + 2^n$ be denoted by ν from now on. In order to reduce the ν-cube into the $RH(k, n)$, out of the ν edges of each hypercube node the following two sets are kept, leaving $k + 1$ edges to each node.

Set 1. The k edges of the ν-cube that traverse the k lowest dimensions (i.e. dimensions 0 through $k - 1$) and connect the referenced node with k distinct nodes are kept. As a result, a complete k-cube building block (*BB*) that includes the referenced node is kept.

Set 2. This set contains only one edge which is also present in the original ν-cube. This edge is the one that connects directly the referenced node with the node whose address differs from its own address only in the *mth* bit of the 1*st* field, where m is the decimal value in the 0*th* subfield and $0 \le m \le 2^n - 1$.

In general, the resultant $RH(k, n)$ contains 2^{2^n} k-cube *BB*s. It can also be viewed as a 2^n-cube of k-cube *BB*s. A *BB* address forms the 2^n most significant bits (i.e. the 1*st* field) of the ν-bit addresses for contained nodes. Each *BB* is divided into 2^n subblocks (*SB*s) and distinct *SB*s implement connections in distinct dimensions of the 2^n-cube; each *SB* is a $(k - n)$-cube. Connections between pairs of *SB*s in different *BB*s are as follows: A node in a particular *SB* of a particular *BB* is connected directly to the node with the same 0*th* field address which belongs to the *BB* whose 2^n-bit address differs only in the *mth* bit, where m is the decimal value in the 0*th* subfield of the former node. An impressive property of the *RH* is that it can emulate simultaneously with optimal dilation (i.e. 1) [2] several popular cube-connected cycles networks [7]. Fig. 1 shows the structure of the $RH(k, 2)$ where the large squares represent the k-cube building blocks. The numbers above the squares represent in decimal the *BB* addresses and the numbers within the

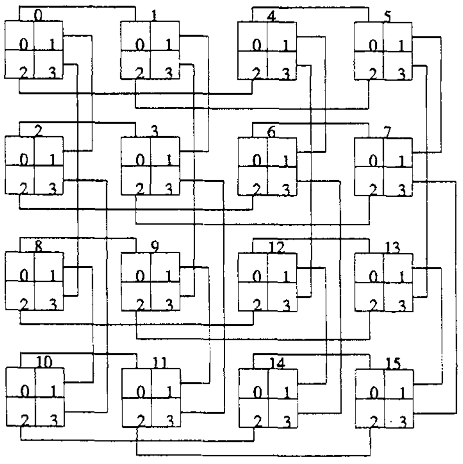

Fig. 1. Structure of the $RH(k, 2)$.

quadrants of large squares are the *SB* addresses in decimal. To keep the figure simple, the nodes within the *BB*s are not shown. Each line between a pair of *BB*s represents 2^{k-2} bidirectional communication channels; this is also the total number of nodes in each *SB*. It is implied that each node in a *SB* is connected directly to the node with the same *0th* field address in the *SB* where the connection line leads.

The reduced number of edges in the RH, when compared to the hypercube with the same number of nodes, may degrade its performance for algorithms designed explicitly for the hypercube. The algorithms given in this paper are not pure hypercube algorithms. Nevertheless, the emulation of the hypercube by the *RH* was investigated in [2] where it was shown that the resultant average dilation of edges is small in practical cases.

3. Data broadcasting

We assume a SIMD message-passing multicomputer environment in this paper. Broadcasting is very common in parallel algorithms, such as matrix-vector multiplication, Gaussian elimination, shortest path, vector inner product, etc. Initially one processor has a value that needs to be broadcast. At the end of the broadcasting procedure, there is a copy of this value in every other processor. Section 3.1 introduces a broadcasting procedure for the $RH(n, n)$. In Section 3.2 this broadcasting procedure is generalized to include the $RH(k, n)$, for $k > n$.

3.1 Broadcasting on the RH(n, n)

The special case of $k = n$ is considered here. In the first phase of the algorithm the 2^n most significant bits of nodes' addresses are used to map a (complete) binary tree with 2^n levels onto the 2^n-cube of *BB*s. The binary tree is double-rooted (using a spacer node) to utilize all *BB*s in an one-to-one mapping [3]. For example, Fig. 2 shows the double-rooted binary tree of depth 2 that utilizes all nodes in the 3-cube. Only the 2^n most significant bits of node addresses are considered in the first phase. Each virtual node in the mapping is actually an n-cube *BB*. Therefore, one of each *BB*'s internal nodes will receive the broadcast value from its parent (except for the root) and up to two other internal nodes will have to transmit the received value to their children located in two other *BB*s.

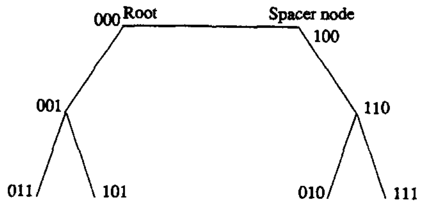

Fig. 2. Double-rooted binary tree with three levels.

In the second phase each node in each *BB* determines whether it is the Node-of-Entry (*NOE*) or a Node-of-Exit (*NOX*) for the implementation of connections to parent and child *BB*s in the previously mapped binary tree. An algorithm also for broadcasting a value from the *NOE* to all other nodes in a *BB* must be introduced. In the third and final phase, without loss of generality, the value is broadcast starting from the node with address 0 in the root *BB* in the tree of *BB*s. The aforementioned phases are described in detail below.

3.1.1 Phase I: Setting up the binary tree configuration of BBs

The 2^n-level binary tree of *BB*s is obtained by applying an algorithm for an one-to-one mapping of a binary tree with $2^{2^n}-1$ nodes onto the 2^n-cube [3] of *BB*s. This phase starts by setting up 2^{2^n-3} three-level double-rooted binary trees having the predetermined configuration of Fig. 2. That is, every *BB* becomes a member of a three-level tree; its position in the tree is determined by the values of bits 0, 1, and 2 in its 2^n-bit address. The algorithm is run by all 2^{2^n+n} nodes in the $RH(n, n)$ using only the 2^n most significant bits of their address.

At each successive iteration of the algorithm, trees are merged to form larger trees until eventually a single binary tree is formed that contains all *BB*s. The merging of two equal-sized trees with spacer nodes requires a new spacer node while the old spacer nodes become internal nodes. With the introduction of a single two-degree node as the child of its root, and thereby stretching (or equivalently double rooting) it, the tree can be made to utilize a hypercube completely. The spacer node so introduced is used only for communication between the root and one of its children. At the end of this phase each node in each *BB* knows which *BB* (if any) is its parent, and which *BB*s (if any) are its children, and also their virtual and physical addresses [3].

Because of the binary tree mapping, each *BB* corresponds to one of the following cases:

(1) It has two children and no parent. This is the root *BB*.
(2) It has a parent and a single child. This is the spacer *BB*.
(3) It has a parent and two children. These are all internal *BB*s, excluding the spacer *BB*.
(4) It has a parent and no children. These are all leaf *BB*s.

All nodes within a *BB* produce the same parent and/or child *BB* addresses in Phase I.

3.1.2 Phase II: Determining the nodes-of-entry and nodes-of-exit

Each node then determines whether it is connected directly to a parent or child *BB*. It does this by comparing its *BB* address with the address of parent (if any) and child (if any) *BB*s computed in Phase I. If such a comparison shows a difference in a single bit with offset equal to the value stored in the 0*th* subfield of the node's address, the node is connected directly to the corresponding parent or child *BB*. Each node which is connected directly to a parent *BB* marks itself as Node-of-Entry (*NOE*). Each node which is connected directly to a child *BB* marks itself as Node-of-Exit (*NOX*). Each *SB* in the $RH(n, n)$ contains a single node, so there is no ambiguity in relation to the chosen node. Each *BB* will have up to one *NOE* node and up to two *NOX* nodes according to the classification presented earlier. Each *BB* is then configured as a binary tree with the *NOE* node as the root, using the algorithm [3] presented for the first phase.

3.1.3 Phase III: Broadcasting the value to all nodes

Assume, without loss of generality, that the value to be broadcast is stored in the node with address 0 in the root *BB* with address 0. The value is then broadcast to the *BB*s using the binary tree of *BB*s configured in the first phase. In each *BB* the *NOE* node receives the value first and passes on the value to its children following a binary tree mapping for the n-cube *BB*. If a node that receives the value is a *NOX*, it passes on the value to the neighbor in the next level of the binary tree of *BB*s, and also passes on the value within the same *BB* using the internal binary tree mapping. If an internal node is not a *NOX*, it just passes on the value to its two children in the same *BB* using the internal binary tree mapping. To broadcast a value from a node other than 0 in *BB* 0, simple transformation of addresses is needed because of the symmetry in the n-cube *BB*s and in the 2^n-cube of *BB*s.

3.1.4 Analysis of the algorithm

For Phase I, according to [3] the binary tree setup algorithm requires time $O(2^n)$ for the 2^n-cube of *BB*s. For Phase II, it takes time $O(1)$ for each node to determine whether it is the *NOE*, a *NOX*, or neither. The mapping of a binary tree onto the n-cube [3] *BB* consumes time $O(n)$. So this phase takes time $O(n)$. For Phase III, broadcasting on the 2^n-cube of *BB*s requires time $O(2^n)$ because of the binary tree mapping. Broadcasting within a single n-cube *BB* requires time $O(n)$ because of the binary tree mapping. Therefore, this phase takes time $O(n\ 2^n)$.

Therefore, the overall time complexity of the algorithm is $O(n2^n)$. In contrast, broadcasting on the $(2^n + n)$-cube with the same number of nodes requires time $O(2^n + n)$ or $O(2^n)$. However, in practical cases the value of n is small, that is 1, 2, 3, or 4 [2], and therefore broadcasting on these two systems requires comparable amounts of time.

3.2 Broadcasting on the RH(k, n), where $k > n$

We generalize the broadcasting procedure given in Section 3.1 to make it applicable to the $RH(k, n)$, where $k > n$. For $k > n$ the $RH(k, n)$ is viewed as 2^{k-n} distinct $RH(n, n)$s where all nodes with the same address in the 2^{k-n} distinct $RH(n, n)$s are connected to form a $(k-n)$-cube [2]. The nodes' addresses in the latter hypercube become the least significant $k-n$ bits of the nodes' addresses in the $RH(k, n)$. This property is used here in order to basically follow the broadcasting algorithm of Section 3.1

Without loss of generality, assume broadcasting from the node with address 0. All nodes with zeros in the $2^n + n$ most significant bits of their address participate in the first phase of the algorithm. A $(k-n)$-level binary tree with a spacer node is mapped to a $(k-n)$-cube in *BB* 0, using the mapping algorithm in [3]. This hypercube contains all nodes that have all zeros in the 2^n+n most significant bits of their addresses. Broadcasting is then carried out in this binary tree within *BB* 0. Ignoring the $k-n$ least significant bits of node addresses, broadcasting is then implemented independently within the 2^{k-n} distinct $RH(n, n)$s. Broadcasting begins with that node in each $RH(n, n)$ whose all 2^n+n most significant bits in the address are zeros, using the procedure of Section 3.1.

3.2.1 Analysis of the algorithm

Broadcasting the value within the $(k-n)$-cube of BB 0 requires time $O(k-n)$ because of the binary tree mapping. The parallel broadcast of the value within the distinct $RH(n, n)$s requires time $O(n2^n)$, as shown in Section 3.1. Therefore, the overall time complexity of the algorithm for the $RH(k, n)$ is $O((k-n)+n2^n)$ or $O(k+n2^n)$. In contrast, broadcasting on the (2^n+k)-cube with the same number of nodes requires time $O(k+2^n)$. In practical cases the value of n is small, and therefore broadcasting on these two systems requires comparable amounts of time.

4. Data reduction

In data reduction an associative operator is applied to values stored one per processor, in order to produce a single value as the result. The common operators are logical OR, logical AND, maximum, minimum, and add. Data reduction often facilitates barrier synchronization on message-passing parallel computers for the separation of different phases in concurrent algorithms.

Many-to-one mapping of a binary tree is suitable for the implementation of the reduction operation on a hypercube. A binary tree of height d can be mapped with maximum dilation 1 onto a hypercube with 2^d nodes as follows [5,8]:

(1) The root of the tree is mapped to any hypercube node.
(2) For each node i at depth j (the root is at depth 0), the left child of i is mapped to the same hypercube node i, and the right child of i is mapped to the hypercube node whose address is obtained by complementing the bit $p-j+1$ of node i's address, where p is the offset of the most significant bit.

Nodes from any single level of the binary tree are mapped to distinct hypercube nodes according to this mapping. Since we determine the right child of a node by complementing one bit of its address, there is an edge in the hypercube that connects directly these two nodes. Also, the leaves are consecutively numbered.

The data reduction algorithm for the *RH* proceeds as follows. A binary tree is first mapped onto each k-cube *BB*, according to this many-to-one manner. Each node at depth $k-1$ does a reduction operation with its right child which is a leaf. Then, each node at depth $k-2$ does a reduction operation with its right child, and so on, until we reach the root, which is chosen to be node 0 in the *BB*. Each *BB* now has a node with the result of the reduction operation for the entire *BB*. The

2^n most significant bits of node addresses are then used to map in a many-to-one manner a (2^n+1)-level binary tree onto the 2^n-cube of *BB*s, using again the algorithm given above. At most n hops are required within a *BB* to go from the node which has the reduction operation value for that *BB* to the node (whose $k-n$ least significant bits are zeros) that implements a connection to its parent *BB* in the binary tree of *BB*s, and then at most $n+1$ hops to go from the latter node to the node having the reduction operation value for the parent *BB*. This node then performs the reduction operation. If it is the right child of its parent, it passes on the value to the parent *BB* as indicated above. So, there is a dilation of at most $2n+1$. At the end of this stage the node with address 0 has the final result.

The time complexity of the algorithm is found as follows. The parallel reduction operation within *BB*s requires time $O(k)$. The reduction operation between pairs of *BB*s requires time $O(n)$ because the dilation is $O(n)$. Although the dilation is often ignored in the estimation of the time complexity of algorithms, and often justifiably so because of the incorporation of wormhole routing [9], it is taken into consideration here. There are $O(2^n)$ levels in the binary tree, thus the reduction operation among *BB*s requires total time $O(n2^n)$. Therefore, the total time required is $O(k+n2^n)$. The reduction algorithm for the hypercube with the same number of nodes has time complexity of $O(k+2^n)$ because a (2^n+k+1)-level binary tree will be mapped in a many-to-one manner. Since the value of n is small in practical cases and wormhole routing may be used, it takes comparable amounts of time for the implementation of the reduction operation on these two systems.

5. Prefix operation

Prefix computation is commonly used in parallel algorithms, including the evaluation of polynomials, ranking and packing problems, solution of linear recurrences, carry look-ahead addition, finding convex hulls in images, and scheduling problems. Given p numbers $n_0, n_1, \dots, n_{p-1}$, the prefix computation problem is to compute $s_j = n_0 \oplus n_1 \oplus \cdots \oplus n_j$, for all j between 0 and $p-1$, where $\oplus$ denotes an associative operator. Initially n_j resides in the processor with address j, and at the end of the procedure the same processor holds s_j. Binary trees are used frequently in prefix computations [5]. The algorithm for the *RH* goes through two phases. In the first phase the prefix operation is carried out within each *BB*. In the second phase the prefix operation is carried out among *BB*s.

In the first phase each processor in each *BB* maintains two parameters namely *rslt* and *msg* [8]. These parameters are initialized with the value n_j that the processor contains. k steps follow for the k-cube *BB*. In step i each processor sends its *msg* parameter to its neighbor in dimension i, for $i=0,1,2,\dots,k-1$. Its new *msg* value is obtained by applying $\oplus$ to the old *msg* value and the one received. If the incoming value comes from a lower-addressed neighbor, then it assigns to *rslt* the value obtained by applying $\oplus$ to the old *rslt* value and the one received.

In the second phase the connections between the subblocks of different *BB*s are employed. Only the processor with address $2^k - 1$ in each *BB* is involved in this phase; this processor contains the prefix result for the entire *BB*. The algorithm used in the first phase is now implemented for the 2^n-cube of *BB*s. However, each processor involved in this phase does not include its own value in the calculation of *rslt*. Therefore, 2^n steps are required, where each step consumes time $O(n)$ because the maximum dilation is $2n + 1$. At the end, the value of *rslt* in the processor with address $2^k - 1$ in each *BB* is broadcast to all processors in the corresponding *BB* and the associative operation is applied to the incoming value and the partial prefix value calculated in the first phase.

The time complexity is found as follows. The first phase takes time $O(k)$. The second phase has $O(2^n)$ steps, where each step takes time $O(n)$. The value at the end is broadcast to all nodes in each *BB* in $O(k)$ steps. Therefore, the time complexity for the second phase is $O(k + n2^n)$, which is also the overall time complexity. A hypercube with the same number of nodes takes time $O(k + 2^n)$. In practical cases, the overhead due to missing links in *RH*s may not be significant because of the small value of n. With advanced routing, such as wormhole routing [9], the complexity for the *RH* is close to $O(k + 2^n)$.

6. Sorting

Sorting is a very common operation in parallel processing. Many algorithms require sorted data because they are easier to manipulate than randomly ordered data. Sorting is defined as the task of arranging an unordered collection of elements into monotonically increasing (or decreasing) order; without loss of generality, the increasing order is assumed here. Let $S = \langle a_1, a_2, \ldots, a_p \rangle$ be a sequence of p elements in arbitrary order; sorting transforms S into a monotonically increasing sequence $S' = \langle a'_1, a'_2, \ldots, a'_p \rangle$, such that $a'_i \leq a'_j$ for all $1 \leq i < j \leq p$, where S' is a permutation of S. The global order assumed by our algorithm is as follows: *BB*s are ordered according to their 2^n-bit addresses while the nodes inside a *BB* are ordered according to their k-bit addresses.

Let $\mathrm{N} = 2^{2^n + k}$ be the number of processors in the $RH(k, n)$, with $k \geq n$. Let p be the number of data elements to be sorted, where $p \geq N$. Initially each processor is assigned a block of p/N elements. The algorithm consists of three phases. In the first phase data elements in each *BB* are sorted. In the second phase data are sorted in the 2^n-cube of *BB*s. In the third phase sorted data are distributed in parallel within *BB*s.

6.1 Phase I: Sorting within BBs

All processors sort their p/N elements internally using merge sort [5]. Each k-cube *BB* then sorts all its data using the bitonic sort algorithm [6,8]. To prepare for the second phase, each processor in any *BB* then sends its p/N sorted elements to that processor in the same *BB* whose address differs from its own address only in the n subblock address bits, which are all zeros. This can be done in $O(n(p/N))$ communication cycles using the E-cube routing algorithm for n-dimensional subcubes; it takes much less time with wormhole routing and involvement of DMA controllers. All receiving processors in *SB* 0 concatenate the incoming elements to their own elements in the increasing order of source *SB* addresses. This preparation is required because the bitonic sort algorithm in the second phase for the 2^n-cube of *BB*s begins with dimension 0 data transfers and only *SB* 0 implements connections in this dimension.

6.2 Phase II: Sorting among BBs

The algorithm takes advantage of the fact that a *RH* can be viewed as a hypercube of hypercubes (i.e. *BB*s), therefore bitonic sort can be applied to the former hypercube [6]. The bitonic sort algorithm implements compare-exchange steps in dimension 0 first (this is the reason that all data in each *BB* were moved to *SB* 0), then in dimensions 1 and 0, in this order, then in dimensions 2, 1 and 0, in this order, and so on. In each step the elements are passed to that *SB* in each *BB* which implements connections in the respective dimension. Each physical processor involved in this phase can be viewed as 2^n virtual processors because it contains sequences that came from 2^n *SB*s. Each time the $(k-n+1)$-cube formed by two *SB*s in two neighboring *BB*s applies again the bitonic sort algorithm in order to implement the compare-exchange steps, assuming a virtual $(k+1)$-cube (i.e. corresponding to a pair of *BB*s). At the end of the last step, the sorted elements in each *BB* are in the *SB* with address zero, because the last dimension used by the bitonic sort algorithm is dimension 0.

6.3 Phase III: Distribution of sorted values in BBs

The *SB* with address 0 in each *BB* has the sorted sequence for the *BB* at the end of phase II. The sequence of elements in each processor of such a *SB* is actually composed of 2^n subsequences of consecutive elements for distribution to the other 2^n-1 *SB*s in the *BB*, so that global order is achieved. E-cube routing is used to distribute the subsequences to all processors (i.e. the last step of the first phase is reversed).

6.4 Analysis of the algorithm

Initially each processor sorts its p/N elements internally in time $O(p/N(\log(p/N)))$, using merge sort. It takes time $O(p/N(\log^2 2^k))$ or $O(p/N(k^2))$ for the values to be sorted in each k-cube *BB* using the bitonic sort algorithm [8]. It takes time $O(n(p/N))$ for these sorted values to accumulate in the lowest-addressed *SB* in each *BB*, because up to n dimensions may be traversed by each datum (DMA controllers and wormhole routing can reduce this time dramatically). In the second phase it takes $O((p/N)2^n)$ communication cycles each time to transfer sequences between neighboring *SB*s because $(p/N)2^n$ is the number of elements in each active processor. Bitonic sort in the $(k-n+1)$-cubes formed by neighboring *SB*s in neighboring *BB*s assumes virtual $(k+1)$-cubes and takes time $O(2^n(p/N)\log^2 2^{k+1})$ or $O(2^n(p/N)k^2)$ time; each physical processor corresponds to 2^n virtual processors. Let us denote the term $(p/N)2^n$ by the symbol β and the term $(p/N)2^nk^2$ by the symbol α. The total asymptotic time complexity for the second phase of the algorithm is on the order of

$$\overbrace{\alpha}^{step\ 0:\ dim\ 0} + \overbrace{\beta+\alpha}^{step\ 1:\ dim\ 1} + \overbrace{\beta+\alpha}^{dim\ 0} + \overbrace{2\beta+\alpha}^{step\ 2:\ dim\ 2} + \overbrace{\beta+\alpha}^{dim\ 1} + \overbrace{\beta+\alpha}^{dim\ 0} + \cdots$$

where step i starts with the *ith* dimension of the 2^n-cube of *BB*s for $i=0, 1, \ldots, 2^n-1$. This can be expressed by the summation $O(\Sigma_{i=0}^{2^n-1}[(i+1)\alpha + 2i\beta])$ which simplifies to $O(4^n(\alpha+\beta))$ or $O((p/N)8^nk^2)$, where $N=2^{2^n+k}$. The third phase takes time $O(n(p/N))$. Thus, the total time complexity is $O(p/N(8^nk^2 + \log(p/N)))$. In contrast, bitonic sort on the hypercube with the same number of nodes consumes time $O(p/N(\log(p/N)+\log^2 N))$ or $O(p/N(\log(p/N)+4^n+k^2+k2^n))$. The additional time for the $RH(k, n)$ is only $O(p/2^k)$.

7. Conclusions

The *RH* has significantly lower VLSI complexity, and comparable diameter and average internode distance, when compared to the hypercube with the same number of nodes [2]. Mappings onto the *RH* of frequently used topologies, such as the ring, torus, and binary tree have been shown to be efficient [15]. The main focus of this paper was to develop algorithms for operations on the *RH* which are very frequently used in parallel processing. The algorithms which were developed are for data broadcasting and reduction, prefix computation, and sorting. The results show that the *RH* achieves performance comparable to that of the regular hypercube in practical cases. Since the *RH* technology also allows the construction of systems larger than hypercubes, the *RH* is a viable interconnection network for building massively parallel systems.

References

[1] S.G. Ziavras, On the problem of expanding hypercube-based systems, *J. Parallel Distrib. Computing* (Sep. 1992) 41–53.

[2] S.G. Ziavras, RH: A versatile family of reduced hypercube interconnection networks, *IEEE Trans. Parallel Distrib. Systems* (Nov. 1994) 1210–1220.

[3] S.R. Deshpande and R.M. Jenevin, Scalability of a binary tree on a hypercube, *Int. Conf. Parallel Proc.* (Aug. 1986) 661–668.

[4] K. Ghose and K.R. Desai, The HCN: A versatile interconnection network based on cubes, *Supercomputing '89 Conf.* (Nov. 1989) 426–435.

[5] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures* (Morgan Kaufmann, San Mateo, CA, 1992).

[6] G.C. Fox et al., *Solving Problems on Concurrent Processors* (Prentice Hall, Englewood Cliffs, NJ, 1988).

[7] F.P. Preparata and J. Vuillemin, The cube-connected cycles: A versatile network for parallel computation, *Comm. ACM* 24 (May 1981) 300–309.

[8] V. Kumar et al., *Introduction to Parallel Computing* (Benjamin/Cummings, CA, 1994).

[9] W.J. Dally and C.L. Seitz, Deadlock-free message routing in multiprocessor interconnection networks, *IEEE Trans. Computers* (May 1987) 547–553.

[10] T.H. Lai and W. White, Mapping pyramid algorithms into hypercubes, *J. Parallel Distrib. Comput.* 9 (1990) 42–54.

[11] S.G. Ziavras and M.A. Siddiqui, Pyramid mappings onto hypercubes for computer vision: Connection Machine comparative study, *Concurrency: Practice Experience* 5 (Sep. 1993) 471–489.

[12] C.T. Ho and S.L. Johnsson, Spanning balanced trees in Boolean cubes, *SIAM J. Sci. Stat. Comput.* (July 1989) 607–630.

[13] T.F. Chan and Y. Saad, Multigrid algorithms on the hypercube multiprocessor, *IEEE Trans. Computers* 35 (Aug. 1988) 969–977.

[14] S.L. Johnsson, Communication efficient basic linear algebra computation on hypercube architectures, *J. Parallel Distrib. Comput.* (Apr. 1987) 133–172.

[15] S.G. Ziavras and M.A. Sideras, Facilitating high-performance image analysis on reduced hypercube (RH) parallel computers, *Int. J. Pattern Recogn. Artif. Intell.* (special issue, Aug. 1995) 679–598.

[16] H.P. Katseff, Incomplete hypercubes, *IEEE Trans. Computers* (May 1988) 604–608.

[17] S.G. Ziavras, A class of scalable architectures for high-performance, cost-effective parallel computing, *6th IEEE Symp. Parallel Distrib. Processing*, Dallas. Texas (Oct. 1994) 162–169.

[18] S.G. Ziavras, Generalized reduced hypercube interconnection networks for massively parallel computers, in: D.F. Hsu, A. Rosenberg, and D. Sotteau, eds., *Series in Discrete Math. and Theor. Computer Science* 21 (American Math. Soc. 1995) 307–325.