Reprint from the proceedings of the ACM Symposium on Parallel Algorithms and Architectures, SPAA92
June 29 - July 1, 1992, San Diego, California.

Supporting the hypercube programming model on mesh architectures

(A fast sorter for iWarp tori)

Thomas M. Stricker
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Many combinatorial problems have simple solutions for parallel processing on highly-connected networks such as the butterfly or the hypercube, whereas the fastest processor-to-processor interconnections are realized in parallel machines with low dimensional mesh or torus topology. This paper presents a method for mapping binary hypercube- algorithms onto lower dimensional meshes and analyzes this method in a model derived from the architecture of modern mesh machines. We outline the criteria used to evaluate graph embeddings for mapping supercomputer communication networks.

Our work was motivated by the need for fast library routines to do parallel sorting, fast Fouriertransformation and processor synchronization. During the design effort of these building blocks, we developed and analyzed a new technique to support a hypercube network embedded onto a two dimensional torus. A direct implementation of the embedding is made possible by logical channels and pathways. A fast merge sorter based on the bitonic network serves as an example to show how a simple hypercube algorithm can outperform most of the asymptotically optimal mesh algorithms for *practical* machine sizes.

In the conventional mesh computation model, processors are allowed to exchange one unit of data with a neighbor in each step. This model needs to be refined since modern mesh computers, such as the iWarp system, have hardware support for fast non-neighbor communication.

The bitonic merge sort, a simple hypercube algorithm, contains a fair amount of fine grain parallelism not found in standard mesh algorithms. This form of parallelism includes pipelined communication, computation overlapped with communication, use of wide instruction words and operands directly read from the communication system through systolic gates.

The measured sorting rate of more than $2 * 10^6$ keys/sec on an iWarp torus with just 64 processors shows the excellent *absolute performance* of our approach. The performance results compare well with much larger parallel computers. In our analysis of the *relative performance* we compare our approach to different sorting methods on meshes. The mapped hypercube algorithm is shown to be best for a wide range of machine and problem sizes.

For the readers mainly interested in complexity results, our approach may seem somewhat surprising, but the analysis of the algorithm in an accurate model for the iWarp machine shows how good speed and good parallel efficiency is obtained from both forms of parallelism, large and fine grain.

1 Introduction

1.1 Hypercubes vs. meshes in parallel algorithms

Many simple and efficient parallel algorithms for combinatorial problems are designed for machines with a hypercube interconnect network. Fast and scalable hypercube networks remain hard to build because of their unbounded connectivity and the large amount of wiring required.

Parallel computers with mesh and torus topologies have been very successful in offering high computational power together with high speed processor interconnections. Past research on parallel algorithms has resulted in a well established model for computing on lower dimensional meshes and tori [TK77] [SS88] [Kun91a]. A large number of algorithms designed and analyzed within that model can be found in the literature. Unfortunately, most of these algorithms lack the simplicity of their counterparts designed for the hypercube models. Most mesh sorting algorithms divide the surface of the mesh into patches of smaller size. The analysis of the asymptotical complexity typically accounts for work within sub-problems in low order terms and concentrates on the work needed to combine the solutions of the sub-problems. Although algorithms with an optimal constant of 3 (in n, the dimension of the mesh) are known, their complexity includes large low order terms, that are dominant for practical machine sizes.

The mesh computation model is quite hard to work with, and probably for this reason several architects of massively parallel machines rely on more powerful networks like hypercubes, butterflies or fat trees for the implementation of mesh and hypercube based algorithms. Examples are the MasPar machine and the earlier Connection Machine models which provide a hypercube or a combinatorial router in addition to the nearest neighbor grid communication [Bla90] [KH88].

*The research was supported in part by the Defense Advanced Research Project Agency, US Department of Defense, monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, and in part by the Office of Naval Research under Contracts N00014-87-K-0385 and N00014-87-K-0533.

1.2 Previous work on sorting in hypercubes

Parallel sorting of m elements on binary hypercubes of p processors has been widely studied. Batcher introduced the bitonic method to sort m elements in $O(\log^2 p)$ time [Bat68]. The bitonic sorting network is discussed in the algorithms collection of [Knu73] and several textbooks.

In our implementation we made use of the bitonic sorting network but did not use the bitonic method to do the local sort within a processor, because bitonic sorting requires a total of $\log^2 m$ passes which results in excessive local bitonic merges.

As pointed out in many earlier publications, the $O(\log p)$ stage networks proposed by [AKS83], and even more recent improved versions, do not provide feasible sorting solutions for practical machine sizes [Pat90].

Sorters in linear time are widely known, for sorting data locally within one processor of a parallel machine. A radix sorter, based on counting sort, is well described in many algorithms textbooks e.g. [CLR90]. An good local sorter based on comparisons is quicksort [Hoa61] with $O(m \log m)$ average case complexity.

Parallel versions of radix sort by rank counting were used successfully on the Connection Machine [BLM$^+$91] and the Cray YMP [ZB91] to break sorting records. Probabilistic sorters based on sampled distributions and permutation routing perform very well on large machines for big problem sizes [RR89] [BLM$^+$91]. On mesh computers, message routing of random permutations is of a complexity similar to sorting [Kun91a]. Reducing sorting to routing is not solving the problem on the mesh unless significantly faster router hardware is provided.

1.3 Previous work on sorting in meshes

A large number of papers have been published on sorting using the mesh model. We compare our sorting algorithm using the model of emulated hypercubes to several sorting algorithms based on the conventional mesh model.

Thompson and Kung show that the complexity of sorting on an $n \times n$ mesh is $O(n)$ and give a $6n + o(n)$ algorithm [TK77]. Unfortunately for practical machine sizes, $o(n)$ hides significant low order terms. Further we have looked into the optimal $3n + O(n^{\frac{3}{4}})$ algorithm by Schnorr and Shamir [SS88]. The algorithm is provably optimal in the sense that it matches a very general lower bound of the 1-1 sorting. This bound, proven in [SS88], applies to a general MIMD model. Because of their simplicity we also considered the sub-optimal algorithms **ShearSort** and **RevSort** described by [SS88]. The latter are less complicated than the optimal algorithms and have no low order terms.

2 The model of emulated hypercube computation

In this paper we describe an approach that uses extra communication bandwidth to offset the mesh's disadvantage in connectivity. By properly overlapping computation and communication, extra bandwidth is made available to the algorithm designer. This is not only the case for computation bound matrix operations but also applies to large combinatorial problems (i.e. sorting, routing, problems in graph theory). In particular, the high bandwidth and the architectural support for logical channels can be effectively used to emulate the hypercube processing model on meshes such as the iWarp torus. Logical channels are a method to share the bandwidth of a high speed physical link between multiple connections.

The mapped hypercube model suggested in this paper provides the programmer with the view of the machine as a hypercube and allows a direct implementation of hypercube algorithms whenever they are simpler than mesh algorithms.

Many recent parallel computers are designed as MIMD distributed memory machines with roughly 50 to 10,000 high performance processors. We refer to this range of processors as *machines of practical* size.

We have encountered the problems of sorting, routing and permuting as part of our real time applications. With signal processing applications in mind we are not interested so much in the largest feasible problem (i.e. sorting all core memory available) but in solving midrange problems faster. The *one million element sorting task* is our primary benchmark. A problem very similar to the one million element sorting task is the Fast Fourier Transform of a 1000 by 1000 pixel full color image into the frequency domain.

3 Embedding binary k-cubes into r-dimensional tori

Mapping a network of logical processors and connections onto a network of physical processors and connection can be viewed as a graph embedding problem. Typical parameters considered for an embedding are:

workload or load: the maximal number of logical processors mapped to physical processors.

dilation: the maximum stretching applied that occurs to a channel during the mapping from logical onto the physical connections.

edge congestion: the maximum number of logical connections that are mapped to one physical connection.

vertex congestion: the maximal number of logical connections that are routed through a physical processor.

For our mapped hypercube model we are interested in embedding a binary k-cube onto an r-dimensional torus with n^r processors. We are only considering mappings with *workload* $l = 1$. The architectural reason for this requirement is the large overhead cost for processor virtualization, especially on modern RISC parallel machines, with many registers and a lot of communication state.

The obvious lower bound for congestion of a binary cube to a ring mapping is $\frac{n}{4 \log n}$. The bound, derived from the bisection bandwidth, is not very tight and suggests for example that a binary 4-cube could be embedded in a 16 cell ring with congestion $c = 1$.

The embedding of binary k-cubes onto an r-torus can be done either in one step, directly reducing the dimension from k to r, or in several steps, by mapping step-by-step groups of hypercube dimensions onto one torus dimension.

Definition 3.1 *A mapping of a binary k-cube into an r-dimensional torus is separable if $\exists k_1, \ldots, k_r : \sum k_i = k$ and there exists a set of mappings $m_1 \ldots m_r$ such that $\bigcup_i m_i$ is a mapping, and m_i maps a k_i-cube to a 2^{k_i} ring.*

3.1 A systematic separable embedding

Earlier work on mesh – hypercube embeddings deals mostly with the reverse problem of embedding meshes into hypercubes [CTH89] [LSBD88]. Such an embedding is much easier to find since the mesh is less connected than the hypercube. Gray code mappings were suggested for simple cases and even improved for irregular sizes by [LSBD88] and others.

We found Gray code mappings to be a good starting point to solve our problem of mapping hypercubes onto meshes. A Gray code is a simple, systematic way to generate a sequence of numbers with a Hamming distance of one. One of the Gray codes is the *binary reflected* Gray code. It is recursively defined:

$$G_1 = 0$$
$$G_n = G_{n/2} \circ (\overline{G_{n/2}} \uplus n)$$

where $\overline{G}$ designates the sequence G written in reverse and where $\uplus$ is an operator adding a constant value to every number. G_{16} lists as:

0 1 3 2 6 7 5 4 12 13 15 14 10 11 9 8

Definition 3.2 *A* Gray code ring mapping *is obtained by mapping the logical processor of the k-cube with cell number x to the node y in the ring with $G_n[y] = x$.*

Definition 3.3 *A* Gray code torus mapping *is a separable mapping when all of the r primitive mappings are Gray code ring mappings.*

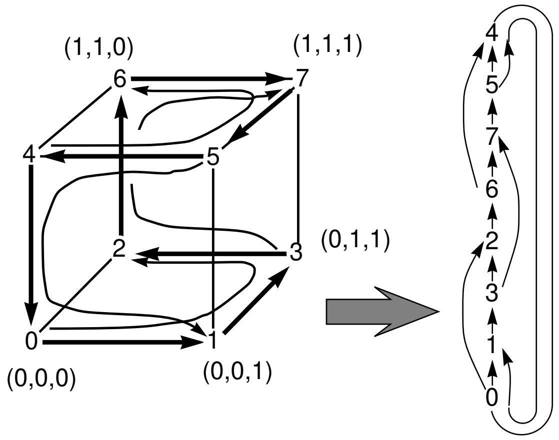

Figure 1: The Gray code ring mapping of a binary 3-cube onto a ring of length 8

The mapping we used for the 64-processor iWarp torus is composed from two Gray code ring mappings, where each of them maps a 3-cube onto a 8-cell ring each. During the analysis and evaluation of the Gray code mapping we can show which properties are important for a good mapping in our emulated hypercube model and its implementation on iWarp.

A **workload** of *one* is very important unless the processors of the parallel system can be used in multitasking mode very efficiently. The **dilation** of the embedding is of secondary importance for our model since the architecture provides non-neighbor communication at a low latency per hop. For the Gray code mapping the dilation is $n/2 - 1$, which is obvious from the recursive structure of the Gray code. The most important factor for our mapped hypercube model is the **edge congestion** c. As we show later in the analysis of the sorter, the edge congestion is an important determinant of the running time of our communication steps. In general we can not avoid congestion, but with modern machines like iWarp we can provide architectural support to handle a certain amount of congestion without degradation of performance.

The algorithms considered in this paper use the hypercube dimensions one at a time. We therefore compute the congestion for every hypercube dimension separately.

Theorem 3.1 *Let $b_n(i)$ be the Gray code mapping of all logical connections along the i-th hypercube dimension. Let $c_n(i)$ be the maximal edge congestion of one dimension of the mapped hypercube i.e. the edge congestion of $b_n(i)$. The congestion $c_n(i)$ can be computed as:*

recursion

$$c_4(2) = 1$$
$$c_n(1) = 1$$
$$c_n(k) = 2 * c_{n/2}(k-1)$$

closed form

$$c_n(i) = \begin{cases} 2^{i-1} & 2^i < n \\ 2^{i-2} & 2^i = n \end{cases}$$

Proof 3.1 *By induction on the recursive structure of the Gray code. Whenever the two subsequences of a Gray coded ring of length 2^j are combined into new Gray coded ring of length 2^{j+1}, a number of 2^{j-2} connections must be rerouted and 2^j new connections are introduced, half of them routed through the wrap around of the ring.*

A similar recursion can be established and solved for the total congestion c_k.

Edge congestion does not capture all of the resource constraints of our architectural model. Logical connections consume resources at every switching node (which distinguishes the logical channels of iWarp from the general concept of virtual channels). The **vertex congestion** of an embedding captures these resource constraints. For example in an iWarp system the number of channels due to outgoing connections, the number of channels due to vertex congestion, and the number of channels permanently allocated to service networks must be smaller than the total number of channels available in the VLSI component.

3.2 Non- separable mapping

The problem of an optimal way to embed a binary 6-cube onto an 8 × 8 torus has been studied earlier. Besides the Gray coded mapping described in this paper, non-separable mappings with better congestion were given by [Sch92] and [She91]. The following table lists the congestion and dilation for the Gray code mappings used in our algorithm versus a slightly improved, non separable mapping given by [Sch92].

	Congestion						Dilation					
HDm	1	2	3	4	5	6	1	2	3	4	5	6
GCM	1	1	2	2	2	2	1	1	3	3	3	3
FGM	1	1	1	1	1	2	2	2	2	2	4	11

Table 1: Parameters of a separable and a non-separable mapping. *HDm:* Hypercube Dimension, *GCM:* Separable Mapping, based on binary reflected Gray codes ($2^3 \times 2^3 \rightarrow 8 \times 8$), *FGM:* Non-separable, mapping based on two folded grids ($2^6 \rightarrow 64$).

In Table 1 GCM, the Gray code mapping, is obtained by Gray coded processor indices for two ring mappings. The hypercube dimensions alternate in both mesh dimensions. The construction of FGM, the folded grid mapping, starts out with a 4 × 4 torus. This torus is isomorphic to the 2^4 cube and results in interconnects with congestion $c = 1$ and dilation $e = 1$ for all four dimensions. Four copies of these tori are interleaved to obtain an 8×8 mesh in the following way: the torus is copied and shifted diagonally by one in the grid. The connections between these two tori can be routed with a congestion of one and a dilation of up to four. This results in an embedding for 32 nodes. The resulting grid graph is again copied, transposed (mirrored on the grid diagonal) and translated to form a properly interleaved 64 node graph. The connections to link the nodes of the graph with its transposed copy are very similar to a grid transpose network. The connections of this network have a length of up to 11 (dilation) and a congestion of up to two. The mesh with the mapped processor indices is shown in Figure 3.2.

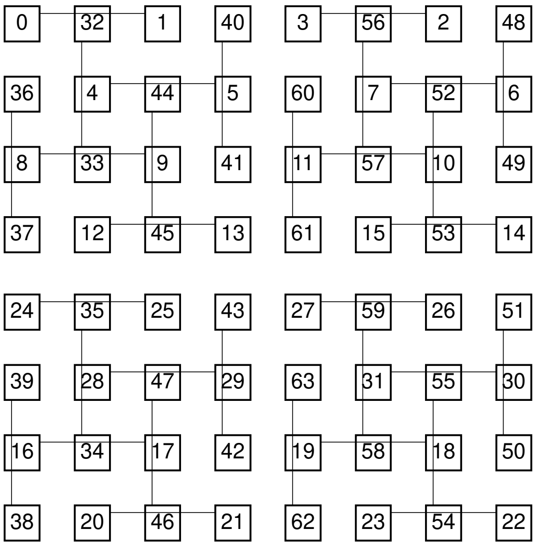

Figure 2: The non-separable, mapping based on two folded grids ($2^6 \rightarrow 64$). The drawing includes the routing for logical links of dimension 0.

We leave as an open question to what extent the congestion can be reduced for small meshes and how to find such embeddings.

4 A communication architecture for mapped hypercubes

The architecture of the iWarp system has been described in detail in [BCC+88] and [BCC+90]. Since our method of emulation and our model for the analysis are strongly motivated by architectural features of the iWarp component, we present briefly the architectural elements relevant to mapped hypercubes and to the sorter.

4.1 High speed processor interconnects

All processor interconnects in the 2 dimensional torus of the iWarp machine are realized as parallel buses with eight data lines. Every processor has four incoming and four outgoing ports, connected to its mesh neighbors. We call the hardware of such an interconnect a physical link. The maximum aggregate speed of a physical link is 40 MBytes/sec in each direction. A processor can therefore communicate over all of its eight links together at rates up to 320 MBytes/sec.

4.2 Logical channels and iWarp pathways

The concept most relevant to our emulated hypercube model is the iWarp pathway [BCC+88] [Gro89]. Pathways allow permanent logical connections between cells that are not physical neighbors in the mesh. Logical channels are used to share physical links and to build pathways. Every logical channel has its own buffer, and therefore a pathway includes its own buffers at every intermediate node. The bandwidth of a shared physical link is allocated by a round robin scheduler among the logical channels. Routing, flow control and exceptions are also handled separately for each logical channel.

The hardware unit that performs this switching and scheduling is called the *communication agent*. The agent handles the links to the four neighbors and further interfaces to the *computation agent*. A third unit, the *memory agent* provides direct access to and from the communication channels. The communication agent is implemented with a fixed number of buffer resources. The current implementation of iWarp allows 20 logical connections at one time. This adds an interesting constraint to the networks that can be embedded and set up statically with long lived connection. However, networks are also dynamically reconfigurable with a reasonable overhead.

4.3 Asynchronous memory communications (iWarp spools)

Many algorithms use block transfers to move data from pathways to memory or vice versa. To overlap communication and computation this must be done without affecting the computation in the processor. In iWarp, the interface between communication and memory is done by eight independent agents, called *spools* [BCC+90]. Similar to a DMA controller, an iWarp spool copies data from memory to a pathway or from a pathway to memory. The data transfers happen in the background asynchronously with little influence on a computation going on at the same time.

4.4 Processing power and memory bandwidth

The iWarp instruction set includes a long instruction word operation that can execute a branch, a load, a store, an integer or two address computations and two 64 bit IEEE floating point operations in as little as 4 cycles at 50 ns/cycle. At peak rate one processor can execute 10 million of these instructions per second. The maximal bandwidth of load- and store- operations to memory is approximately 80 MBytes/sec.

4.5 Processing data directly from communication channels (systolic gates)

A simple throughput (piping) analysis of instruction rates, operands and memory bandwidth shows that the full processing power of the iWarp communication can not be used unless streams of incoming data are processed by the computation agent without going first to memory. Therefore, some operands of the computation can be brought in from the communication unit through systolic gates. Gates are used in the instructions like a register [BCC+90].

5 A fast parallel sorter for iWarp

In this chapter we define the problem of parallel sorting more precisely and give a detailed description of the algorithm that resulted in the fastest sorter for iWarp systems. We address the following sorting problem:

Parallel machine: p processors, organized in an $n \times n$ mesh with wrap- around links.

Input to be sorted: m keys in 64 - bit IEEE floating point number representation. The keys are evenly partitioned among the p processors, i.e. m/p keys per processor.

Output: m keys, sorted in ascending order. More precisely the m keys are evenly partitioned among p processors and each processor holds a sorted sub-sequence of length m/p. The p sorted sequences are positioned in the processors in sorted order, i.e. the do not overlap and their ranges are ordered according to the Gray coded processor indices.

Workspace: A workspace of the size of the input data plus a space of constant size for counters are granted.

The sorting algorithm we used to evaluate our emulated hypercube model for meshes is based on the following two steps:

- The keys are sorted by a *radix sort* locally within all processors, resulting in p monotonic sequences. Algorithm 5.1.
- The global sorting is done with *sequence merges* on the bitonic sorting graph. This requires $O(\log^2 p)$ stages of merges. Algorithm 5.2.

The idea of sorting multiple elements per processor by a local sort followed by a series of sequence merges was discussed in [BS78] within the context of parallel machines.

5.1 The local sort

The local (intra-processor) sort is a radix sort exactly as described in e.g. [CLR90]. In a pre- and a post- processor phase the sign bits are translated between the IEEE floating point representation and a plain 64 bit integer representation.

In our description of the algorithms, we use the following variable names for the arrays to hold the data, the workspace and the counters of the radix - counting sort.

data: The input/output array of m 64-bit keys.

tmp: An array of m storage locations for intermediate results.

cnt: An array of fixed size for the counters of the radix sort. In the current implementation there are 2^{10} counters, which allow a radix sort of 64 bit keys in 6 passes.

Algorithm 5.1 Local Radix Sort

mask(*data, bit_field*) returns a set of key bits determined by the mask *bit_field*.

for all processors q **do** *:*
    bit_fields $= \{0..10, 11..21, 22..31, 32..42, 43..53, 54..63\}$
    for $k = 0$ **to** 5 **do**
        $cnt[\,] = 0$
        for $i = 1$ **to** m/p **do**
            count[mask($data[q][i]$*,* bit_field*[k])]++;*
        for $i = 2$ **to** max_cnt **do**
            $cnt[i] = cnt[i] + cnt[i-1]$
        for $i = m/p$ **to** 1 **do**
            $q = mask(data[q][i],$ bit_field$[k])$
            $cnt[q] = cnt[q] - 1$
            $tmp[cnt[q]] = data[q][i]$
        $data[q][\,] = tmp[\,]$

5.2 The sequence merge

The global (inter-processor) sort is performed with a series of sequence merge steps. A sequence merge is an extension of the compare and swap concept to monotonic sequences of elements. A merge can be used in place of a compare and swap step in any sorting network. For the merge step, a pair of processors (we call them neighbors) must work together to merge the monotonic sequences stored in those processors. After a merge, one processor holds the lower half of the merged sequence and the other processor holds the upper half.

The *spooling agent* and the *computation agent* are both involved in the merge. They are processing two streams of data asynchronously. The *spooling agent* sends part of the sequence over a pathway to its neighbor while the *computation agent* receives some elements from its neighbor, compares them to the elements of the local sequence and stores the results as a merged sequence to memory. Figure 3 explains this process in more detail.

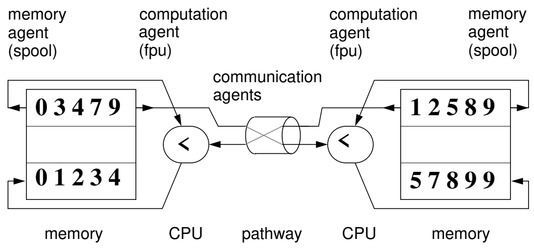

Figure 3: Fine grain parallelism is used to overlap communication and computation in the sequence merge step.

The algorithm for the inter-processor sorting step can be stated as follows:

Algorithm 5.2 Sequence Merge

send($neighbor[i]$*,* $data$*)*	*sends* $data$ *to* $neighbor$ i*.*
recv(neighbor[i])	*returns the* $data$ *received from* $neighbor$ i*.*
do *in parallel* $\{\} \parallel \{\}$	*In parallel within each processor. (Fine grain parallelism)*
select *(*$\ldots \diamond \ldots$*)*	*Select proper choice according to corresponding stages in bitonic graph.*

for all *processors* q **do** *:*
    for $k = 1$ **to** $\log p$ **do**
        for $l = 1$ **to** k **do**
            do *in parallel* $\{$
                for $i = 1$ **to** m/p **do** *(memory agent)*
                    send$(neighbor[l], data[q][i])\ \}$
            $\parallel \{$
                j=1; rcv=recv$(neighbor[l])$ *(computation agent)*
                for select *(*$i = 1$ **to** $m/p \diamond i = m/p$ **to** 1*)*
                    if select *(*$rcv < data[q][i] \diamond rcv > data[q][i]$*)*
                    then $tmp[i] = data[q][j]$*;* $j = j + 1$*;*
                    else $tmp[i] = rcv$*;* $rcv =$ *recv*$(neighbor[l])$
                discard rest of $data[q]\ \}$
            $tmp[\,] = data[q][\,]$

The bitonic network is structured in $\frac{k^2+k}{2}$ stages and some of the stages are grouped together. Figure 4 shows a bitonic network for 8 processors. The $\diamond$-selectors are computed by a series of xor operations on selected bits of the processor number, the stage and the group number. The direction of the merge step is determined by this predicate. The selector for the direction of the index in store loop is determined by the merge direction of the *next* merge step. This assures that the keys are in the right order for transmission through spooling in the next stage.

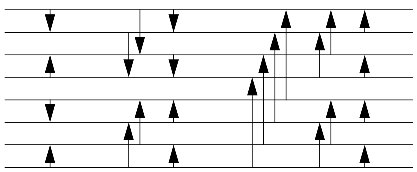

Figure 4: The bitonic merge graph according to [Bat68] and [Knuth73].

The *systolic implementation* of the sequence merge step in Algorithm 5.2 is crucial to the success of our algorithm design for iWarp parallel systems. The algorithm for the sequence merge was carefully chosen and has maximal *fine grain parallelism*.

Our implementation does merges with in 0.5 μs per element, which is only three times slower than a simple compare operation of two floating point numbers between registers. The execution time includes the load from memory, and the store to memory as well as the sends and receives over the network and the branches for conditionals and loop processing. The merge is pipelined and properly scheduled for the long instruction word operations in iWarp.

Interestingly, merging two sequences with a pair of processors is more than two times faster than merging two sequences on an iWarp uni-processor. Designed for systolic algorithms, the processors provide direct access to the communication channels through systolic gates in the register file. In the parallel version of the merge the operands of the comparisons can be received and processed at the same time. This saves one third of the memory accesses and removes a bottleneck in the single processor implementation.

Given the architectural model of private memory mesh computers with high speed processor interconnects, we found that the bitonic sorting network executed on an emulated hypercube is the best method possible for parallel sorting on these kind of mesh machines for *all practical machine sizes*. We will support this statement by a comparative analysis of hypercube- and mesh- sorting algorithms in the next chapter.

6 Analysis of the mapped bitonic sorter

We present the analysis of our algorithm for the case of an r-dimensional torus, assuming that all r-dimensions have the same size and are powers of two. Since we use separable mappings, we can generalize the analysis to different array sizes by using different mappings for torus dimensions of different size.

p	the number of processors in the hypercube or the mesh.
r	the dimension of the mesh - ($r = 2$ for iWarp tori.)
$n = \sqrt[r]{p}$	the number of processors in each dimension.
m	the problem size (total number of elements to sort).
m/p	the number of elements per processor.
t	the number of passes of the radix sort.
u	the length of the key in bits.
$c(i)$	the congestion factor along dimension i of the hypercube
$s(m)$	the sorting rate for problem size m (in elements/sec)

Lemma 6.1 *The time complexity of radix – counting sort is* $O(t * m/p) + O(2^{u/t})$. *The space complexity of radix/counting sort is* $2 * \frac{m}{p}\, u$ *bits for the elements and* $2^{u/t} * \log \frac{m}{p}$ *bits for the counters.*

For the global sorting phase the time of a single sequence merge step is the basic unit of interest:

Lemma 6.2 *The* time complexity *of a single sequence merge step is* m/p *comparisons,* m/p *stores and* m/p *loads. The* space complexity *of a merge is* $2 * m/p$ *elements.*

Corollary 6.1 *In the asymptotic case the running time of the radix sort (local sort) and a single merge stage (part of the global sort) scale linearly with the problem size, i.e. are* $O(m/p)$.

The primitives for local sorting and global merging both scale very well over a large range of problem sizes. In our implementation we achieve 50% of the maximum performance for problem sizes as small as $m/p \approx 512$ elements per cell in the radix sorter. In the two cell sequence merge step 50% of performance can be reached with as little as $m/p \approx 64$ elements per cell. The low startup overhead is due to a statically embedded network with long lived connections. Dynamic connections with message passing would result in larger overheads to establish and conclude the short term connections after each message.

For the analysis of the global sort the number or merge stages for bitonic sort must be determined. In [Bat68] and [Knu73] the depth of the network is derived for the number of processors p.

Lemma 6.3 *For* p *processors the bitonic sorting network contains* $\frac{1}{2}\,(log^2 p + log p)$ *stages.*

The bitonic sorting network does all communication along hypercube connections. The network can therefore be embedded onto a physical hypercube network with load 1 and congestion 1. Furthermore every stage of the bitonic sorting network uses the links of exactly one hypercube dimension. The number of parallel merge steps required for a bitonic sort on a hypercube- connected parallel computers with p processor equals the number of stages in the bitonic sorting network with p inputs.

The performance of the mapped algorithm is affected by the characteristics of the mapping, when executing the stages of a bitonic network on a physical mesh. The computation costs do not differ from the logical hypercube, since all mappings considered have load $l = 1$. The merge step handles m/p elements and is pipelined. The effect of the dilation e on the communication latency can be neglected as long $e \ll \frac{m}{p}$.

The principle parameter affecting the efficiency of the algorithm is the *congestion* of the embedding. Since the bitonic sorting network operates on one hypercube dimension at a time, the congestion is bounded by the maximal congestion of the links emulating that specific dimension rather than the overall congestion of the total mapping.

In the mapped hypercube model a permanent connection, i.e. a pathway is set up in advance for every hypercube connection. The bandwidth is shared evenly among multiple pathways if they are routed over a single physical link. Congestion never blocks the algorithm but slows down the communication of certain merge phases. The total running time of the global sort is determined by the sum of the merge stages of the algorithm, weighted by their specific congestion factors.

To carry out the congestion analysis we need to determine the ratio v between communication and computation and incorporate it into the calculation. This ratio limits the amount of congestion that can be handled without slow-down.

Based on the architectural specification of iWarp we predict the following running times for the computation and communication operations: Computation takes 14 clocks per element, while the communication only takes 4 clocks per element. The ratio is therefore $\frac{2}{7}$. Without congestion a sequence merge step are clearly computation bounded. In the calculations we assume the next entire fraction that is an upper bound, for $\frac{2}{7}$ this is a rate of $\frac{1}{2}$. The ratio of $\frac{1}{2}$ is close to the actual measurement since the hardware schedulers tend to allocate even fractions of bandwidth.

When multiple merge steps share a physical link, the slow down depends on the number of merges in excess of $\frac{1}{v}$ running over that link. The slow down observed by the algorithms on a specific link is is not the embedding congestion but rather the congestion adjusted by a factor v. The minimal adjusted congestion is one since a single connection can not gain a speed-up from extra bandwidth.

Definition 6.1 *The* effective *number of steps in the mapped algorithm is the sum of the steps weighted by their corresponding adjusted congestion. (In our case, a merge stage over links with congestion 2 counts as 2* effective *merge stages).*

The time complexity of our algorithm is linear in the number effective number of merge steps and the number of elements to sort. The effective number of merges is a good measure to evaluate the mapped hypercube algorithm and compare it to direct mesh-sorting algorithms.

To compute the number of effective merges for our Gray code mapped bitonic sort algorithm, we derive the adjusted congestion from the congestion of the embedding given in Theorem 3.1 and the communication – computation ratio:

$$\widehat{c} = \max\{1, c(i)v\}$$

For the different stages of the bitonic network the adjusted congestion is determined by the hypercube dimension i associated to that stage. The number of mesh dimensions is r and the size of the binary hypercube is $k = \log p$. For simplicity we assume the same width of the mesh $n = \sqrt[r]{p}$ in all mesh dimensions and that the hypercube dimensions can be partitioned evenly into mesh dimensions $r \mid k$. Note that for a torus the wrap-around links reduce the congestion for the highest dimensions.

$$\widehat{c}^{\,v}_{p}(i) = \begin{cases} \max(1, 2^{\lceil \frac{i}{r} \rceil - 1} * v) & 2^{\lceil \frac{i}{r} \rceil} < n \\ \max(1, 2^{\lceil \frac{i}{r} \rceil - 2} * v) & 2^{\lceil \frac{i}{r} \rceil} = n \end{cases}$$

To compute the number of effective merge stages q_{merge} we carry out the weighted sum over all the stages of the bitonic sorting network:

$$q_{merge} = \sum_{i=1}^{k} (k - i + 1) * \widehat{c}^{\,v}_{p}(i)$$

For the symbolic summation the sum is rewritten as:

$$\begin{aligned} q_{merge} = & \textstyle\sum_{i=1}^{r\,(1-\log v)} (1 - i + k) + \\ & \textstyle\sum_{i=r\,(1-\log v)+1}^{k-r} \frac{2^{\lceil \frac{i}{r} \rceil}}{2}\ (1 - i + k)\ v + \\ & \textstyle\sum_{i=k-r+1}^{k} \frac{2^{\frac{k}{r}}}{4}\ (1 - i + k)\ v \end{aligned}$$

The running time of the global sort, t_{merge} is the number of effective merge stages times the time complexity of a single merge step (Corollary 6.1).

$$t_{merge} = q_{merge} \times \frac{m}{p}$$

Theorem 6.1 *Let r be the number of mesh dimensions and v the communication – computation ratio. The running time of the global sort in the mapped bitonic sorter is given by*

$$t_{\text{merge}} = \left(\begin{array}{l} \left(\frac{11\ v}{2} + \frac{r\ v}{8} + \frac{r^2\ v}{8} + \left(v - \frac{2\ v}{r}\ k \right) \right)\ 2^{\frac{k}{r}} + \\ (r\ (1 - \log v)\ r - 4)\ k \\ \left(-\frac{1}{2} + \log v - \frac{\log^2 v}{2} \right)\ r^2 \\ \left(\frac{1}{2} - \frac{\log v}{2} \right)\ r - 8\ \log v + -6 \end{array} \right) \times \frac{m}{p}$$

Proof 6.1 *Symbolic summation of the adjusted congestion over the stages of a bitonic sorting network and Corollary 6.1 on the time complexity of a merge step.*

The time complexity is linear in the number of hypercube dimensions k and quadratic in the number of mesh dimensions. The quadratic term is small for any mesh dimensions of our interest i.e. $r = 1, 2, 3$.

Corollary 6.2 *The mapped bitonic sorter scales linearly in the number of elements per processor and with $\sqrt[r]{p}$ in the number of processors.*

Corollary 6.3 *For two dimensional tori $r = 2$, $p = n \times n$, $k = 2 * \log n$ and $2^{\frac{k}{2}} = n$ and the time complexity for the mapped bitonic sorter is:*

$$t_{\text{merge}} = \left(\begin{array}{l} \frac{25}{4}\ v\ n + (-4\ \log v - 4)\ \log n \\ -5\ \log v - 2\ \log^2 v - 7 \end{array} \right) \times \frac{m}{p}$$

Finally with communication - computation ratio of iWarp derived earlier the complexities can be determined:

Corollary 6.4 *For iWarp $v = 0.5$ and the complexity of the mapped bitonic sorter is:*

$$t_{merge} = (3.125\, n - 4)\, \frac{m}{p}$$

For other values of v, the time complexities are ($v = 1/4$: $1.563\, n + 4\ \log n - 5$ and $v = 1$: $6.200\, n - 4\ \log n - 7$. A value of $v = 1$ corresponds to perfectly balanced architectures with a communication – computation ratio of 1.

The asymptotic result of the analysis is slightly worse than the $3n + o(n)$ lower bound for 1-1 sorting given in [SS88], but in our mapped algorithm we have small low order terms. This is the reason why this method resulted in the *fastest* sorter on our 64 cell iWarp machines.

Current iWarp systems, which run at $20 * 10^6$ cycles per second, require 14 cycles to process a single 64 bit key in a merge step and a total of 230 cycles to process a 64 bit key during the local radix sort. Together with the the time complexities we are able to estimate the execution times of the whole sorter. The estimate match the measured execution times listed in Chapter 8 very well.

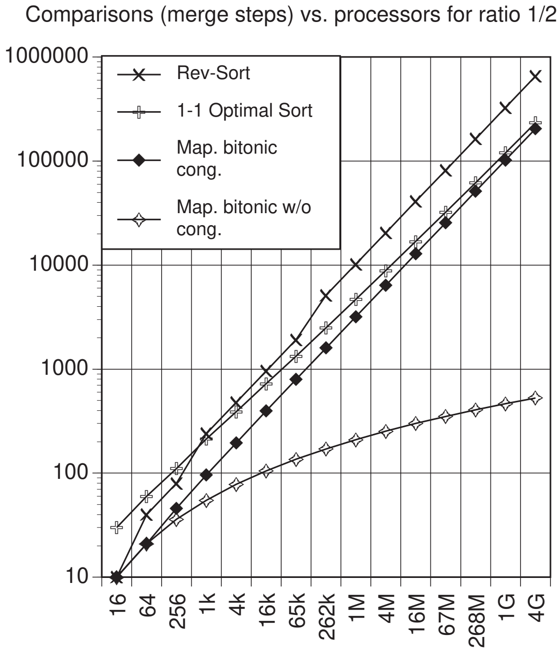

Figure 5: Number of effective merge steps vs. number of processors for RevSort, optimal sort and mapped bitonic sort. Assumed is communication – computation ratio $v = \frac{1}{2}$.

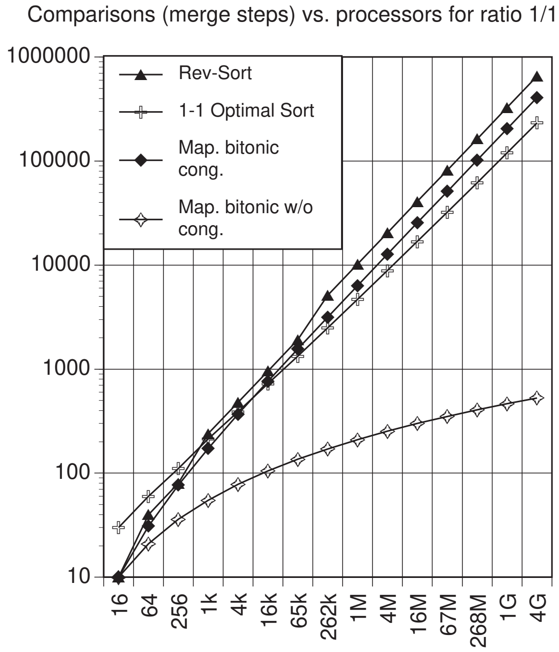

Figure 6: Number of effective merge steps vs. number of processors for RevSort, optimal sort and mapped bitonic sort. Assumed is perfectly balanced communication – computation ratio $v = 1$.

7 Comparison of the mapped hypercube sorter with mesh sorters

In this chapter we use results of our analysis in Chapter 6 to compare the mapped bitonic sorter to known mesh sorting algorithms, including the asymptotically optimal sorter in the 1-1 mesh sorting model.

In mesh algorithms the actual number of merge steps always equals the number of effective merge steps since communication is only with the nearest mesh neighbor and there is no congestion. The measure of merge steps replaces the swap and compare in the 1-1 sorting model when more than one element is sorted per processor.

7.1 The asymptotically optimal 1-1 sorter

The asymptotically optimal 1-1 sorter relies on a partitioning of the processors into rows, columns and sub-blocks of size $n^{3/4}$. It sorts these sub-blocks and puts the results together in $3n$ compare-and-swap steps.

As pointed out in [SS88] the complexity of the sorter is $3n + O(n^{3/4})$, with a factor of at least 9 in front of the low order term $n^{3/4}$.

7.2 RevSort, a row- column- sorter

In [SS88] *RevSort* is described as a simple, almost optimal sorter with no low order terms. It can sort in $\frac{5}{2} * n \log\log n$ steps. The algorithm is simple and therefore a good candidate for the fastest sorter on practical machine sizes, since $\log\log n \leq 5$ in practice.

As seen in Figure 5, *RevSort* it is better than the optimal mesh sorter for small cases but can not outperform the mapped bitonic sorter. For large n the optimal 1-1 sorter is best.

The key to the high efficiency of the sorter lies in the good behavior of smaller cases and in the fact that all "practical" machines sizes are well within that range optimality. At this time iWarp is commercially available up to sizes of 1024 cells. Larger iWarp tori are hypothetical.

7.3 Using the wrap around links in a mesh sorter

A very interesting aspect of mesh sorting is that the row and column sorting steps can not make use of the torus unless they use extra elements of buffer-space. A result with six extra buffers is mentioned in [Kun91a] and attributed to [MS89].

7.4 The h-h mesh sorting models

Some recent literature points out the importance of the $h - h$ sorting models [Kun91b], [Kun91a] and gives a $2.5n$ algorithm for a slightly modified 1-1 sorting model. We found most of the methods impractical because of their large low order terms, but much more work is needed to come to a conclusion about all methods for mesh sorting proposed in the literature.

7.5 Comparison between mesh and mapped hypercube sorters

The Figures 5 and 6 show that the complexities for the mapped algorithm are better than for the two direct mesh sorters sorters for all practical machine sizes.

To break even with our mapped sorter in the weighted number of merge steps, the optimal algorithm needs to run on a two dimensional torus of size $n > 65536$, with $p > 4 * 10^9$ processors; see Figure 5.

Even under the assumption of an $1/1$ ratio between communication and computation, a machine that is $3 - 4$ times less efficient in communication than iWarp, the mapped bitonic merge algorithm can outperform the optimal mesh sort, unless more than $n > 128$ or $p > 16 * 10^3$ processors are used.

8 Comparison to other parallel sorter programs

To evaluate the speed of the mapped bitonic merge sorter, we give an overview of some published execution times for sorting **one million 64-bit keys** on different machines:

- A 1024 Sprint node Connection Machine (CM-2) sorts one million elements in 660 ms at a rate of $1.5 * 10^6$ keys/sec. using **sample sort** [BLM$^+$91]. The Connection Machine is based on physical hypercube interconnections between the Sprint nodes. [1]
- A 64 processor iWarp sorts one million elements in 442 ms and at a rate of $2.3 * 10^6$ keys/sec. using the **mapped bitonic merge sorter** discussed in this paper. The machine is based on a two dimensional torus.
- A 4096 processor MasPar MP-1 sorts sort one million elements in 1446 ms at a rate of $0.72 * 10^6 keys/sec.$ using **bitonic sort** on xnet [2]. This machine is based on two dimensional grid, a diagonal grid (xnet) and a general purpose router [Pri91].
- A single processor Cray YMP [3] sorts one million element in 330 ms at a rate of $3.0 * 10^6$ keys/sec using a parallel version of radixsort. An 8 processor Cray YMP can achieve a 5-6 fold speedup resulting in a rate of $15 - 16 * 10^6$ keys/sec. [ZB91].

The measured execution times and the sorting rates for different size sorting tasks are shown in Figures 7 and 8. The measurements on the CM-2 show the performance for parallel sorting on a real hypercube, while the running times on iWarp show the performance of sorting on a logical hypercube mapped to a physical torus.

The implementation of sampling allows fast sorters on large problems on the CM-2 since the overhead of the router can be compensated. In the Figures 8 and 7 it can be noted that iWarp delivers high sorting performance also for small problem sizes. This is due a good match of the algorithm and the balanced architecture with high speed low latency communication.

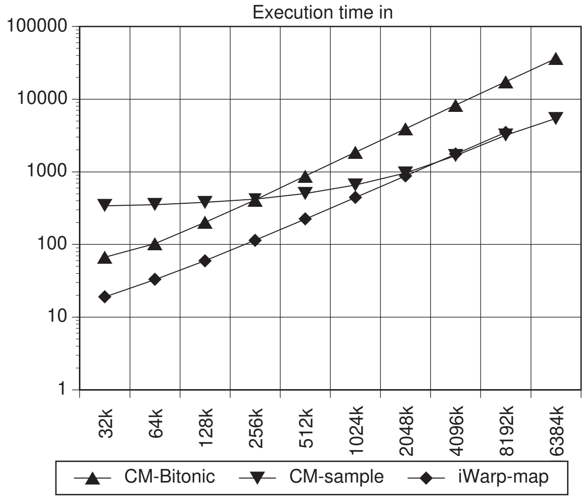

Figure 7: Execution times for sorting different problem sizes, with different algorithms on the CM-2 and the mapped hypercube sorter on a 64 processor iWarp

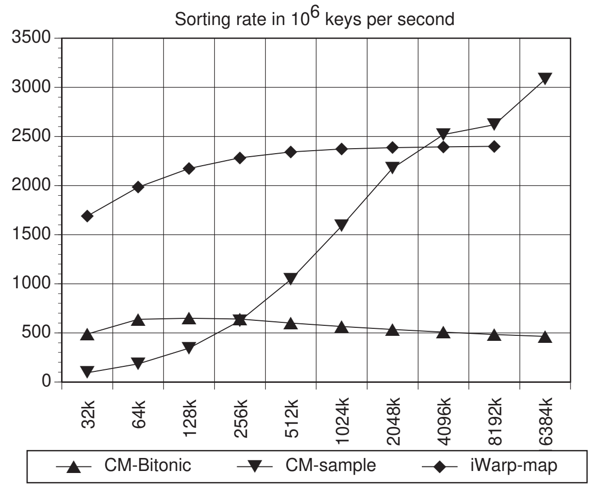

Figure 8: Sorting rates (performance) for sorting different problem sizes, with different algorithms on the CM-2 and the mapped hypercube sorter on a 64 processor iWarp

9 Conclusion

We have established a method and a model to map hypercube computations efficiently to lower dimensional tori. The method is based on past-neighbor communication over fast processor interconnections with logical channels. We expect these features to be included in many mesh machines of the future. The mapped hypercube model provides the programmer with the view of the machine as a hypercube and lets him work with the simpler hypercube algorithms.

The hypercube network is embedded onto the torus using a simple separable Gray code mapping. The Gray code mapping is general and results acceptable congestion, also slightly better mappings were found for special cases. An detailed analysis of the time complexity and an implementation was done for a mapped bitonic merge sorter in our mapped hypercube model. This allowed us to compare the approach to several widely known sorting algorithms for conventional 1-1 mesh model. For all practical machine sizes the asymptotically optimal mesh algorithm as well as a simple, nearly optimal algorithm are inferior to the mapped bitonic merge sorter because of their low order terms or high constant factors.

The viability of our approach was shown with an implementation of the mapped hypercube model on the iWarp mesh machines. As a first application program we implemented a bitonic merge sorter. The mapped bitonic sorter became a highly efficient and scalable program that can sort one million 64-bit keys in less than half a second at rates of 2.3 million sorted elements per second.

Acknowledgments

Numerous people were involved to get our fast iWarp sorter going. I am grateful to all of them particularly to Thomas Gross, H.T. Kung, Guy Blelloch and Dave O'Hallaron. Thanks also for the many

[1] The authors estimate their numbers on the more recent CM-200 to be 369 ms execution time for one million elements and $2.892 * 10^6$ keys/sec sorting rate.

[2] The published times are for 32 bit key and we leave it to the reader to scale them to 64 bit keys.

[3] The Cray YMP is a shared memory vector machine with a fast clock rate of 6 ns.

helpful discussion with our theory community at CMU, particularly with Anja Feldmann, Gary Miller and Eric Schwabe.

References

[AKS83] M. Ajtai, J. Komlos, and E. Szemeredi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pages 1–9, April 1983.

[Bat68] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[BCC+88] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, FL, USA, November 1988. IEEE Computer Society and ACM SIGARCH.

[BCC+90] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proceedings of 17th Intl' Symposium on Computer Architecture*, May 1990.

[Bla90] T. Blank. The MasPar MP-1 architecture. In *35th IEEE Computer Society International Conference*, pages 20–40, Spring 1990.

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Planxtonand S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *1991 ACM Symposium on Parallel Algorithms and Architectures*, June 1991.

[BS78] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, C–27:84–87, 1978.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1990.

[CTH89] S. L. Johnson C. T. Ho. Embedding meshes in boolean cubes by graph decomposition. Computer science technical report, Yale University, New Haven, CT 06520, 1989.

[Gro89] Thomas Gross. Communication in iWarp systems. In *in Proceedings of Supercomputing '89*, pages 436–445, Reno, NV, USA, November 1989. IEEE Computer Society and ACM SIGARCH.

[Hoa61] C. A. R. Hoare. Quicksort. *Computing Journal*, 5, 1961.

[KH88] B. Kahle and D. Hillis. The connection machine model cm-1 architecture. *IEEE Systems, Man, and Cybernetics Special Issue*, March 1988.

[Knu73] D. E. Knuth. *Sorting and Searching. The Art of Computer Programming. Vol 3.* Wesley, Reading Massachusetts, 1973.

[Kun91a] M. Kunde. Routing and sorting on grids. Habilitationsschrift, Fakultaet fuer Mathematik und Informatik der TU Muenchen, 1991.

[Kun91b] M. Kunde. Sorting on meshes. In *Proceedings of the 32st Annual Symposium on Foundations of Computer Science*, October 1991.

[LSBD88] S. LakshmivarahanL. S. Barasch and S. K. Dhall. Generalized gray codes and their properties. In *Third International Conference on Supercomputing*, May 1988.

[MS89] Y. Mansour and L. Schulman. Sorting on a ring of processors. Technical report, Laboratory of Computer Science, Massachusetts Inst. of Technology, Cambridge MA, 1989.

[Pat90] M. S. Paterson. Imporoved sorting networks with $O(\log n)$ depth. *Algorithmica*, 5:75–92, 1990.

[Pri91] J. Prins. Efficient bitonic sorting of large arrays on the MasPar MP-1. Technical Report TR91-041, University of North Carolina, Chapel Hill NC, 1991.

[RR89] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM Journal on Computing*, 18(3):594–607, June 1989.

[Sch92] Eric Schwabe. Personal Communication, January 1992.

[She91] Jonathan Shewchuk. Personal Communication, August 1991.

[SS88] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh-connected computers. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 255–263, 1988.

[TK77] C. D. Thompson and H. T. Kung. Sorting on a mesh connected parallel computer. *Communications of the ACM*, 20:263–271, 1977.

[ZB91] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, November 1991.