DIMACS Technical Report 94-49
October 1994

Issues in parallel computing with hypercube multiprocessors

by

Afonso Ferreira[1,2,3]
CNRS - LIP
CNRS-URA n° 1398
ENS Lyon – 69364 Lyon Cédex 07 France
ferreira@lip.ens-lyon.fr

[1]Visiting Member, partially supported by Dimacs NSF grant STC–91–199999 and the NJ Commission on Science and Technology.
[2]Partially supported by the PRC PRS and Math-Info of the French CNRS.
[3]A revised version of this report will appear in the *Handbook of Parallel and Distributed Computing*, to be published by McGraw-Hill.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

The hypercube architecture has been extensively used in the design of parallel algorithms and for the construction of parallel computers. The aim of this report is to serve as a concise introduction to computational issues related to parallel and communication algorithms for hypercube multiprocessors.

Section 2 describes structural characteristics of the hypercube topology, that made it very attractive for the realization of the interconnection network of parallel computers. Graph properties and embeddings are studied.

Section 3 surveys algorithms and techniques for implementing inter-processor communication in hypercubes. On- and off-line permutation routing techniques in store & forward mode are detailed.

Section 4 is devoted to the study of useful algorithmic tools for the fine-grain hypercube. These include semigroup and data movement operations, as well as merging and sorting, that are used for implementing all kinds of synchronous algorithms.

Section 5 presents some important algorithms that were designed for the fine-grain hypercube. Main issues such as the implementation of concurrent data-structures are discussed.

A large number of references to more in-depth research papers are given to those willing to pursue investigation in this challenging domain.

1 Introduction

The most widely accepted model for the design of parallel algorithms is the *Parallel Random Access Machine*, known as $PRAM$. In this model, it is supposed that each processor (denoted PE, for processing element) has a local memory whose reduced capacity is compensated by the concurrent access to a large central memory, shared by all processors. In one cycle, a PE can gain access to any cell of the shared memory or execute a basic operation on its own local registers. Conflicts among processors arise when some or all of them try to access the same cell of the shared memory, and the different rules used to resolve such conflicts define the different models of $PRAM$s [23].

Unfortunately, implementing a shared memory in a parallel computer becomes infeasible when the number of processors grows, due to the complexity of the hardware required to allow all processors to have access to every memory cell. Thus, in large-scale parallel systems, the memory is distributed among the processors. Inter-processor communication is then guaranteed by the exchange of messages through an interconnection network.

For the sake of simplicity, such distributed-memory parallel computers are modeled as graphs, where vertices represent processors and edges represent communication links. Two processors are said to be *neighbors* if they are connected by a direct communication link. A *path* of length λ is a sequence of neighbors $\{PE_{k_0}, PE_{k_1}, \ldots, PE_{k_\lambda}\}$, where $k_i, 0 \leq i \leq \lambda$, is a valid processor label. The *distance* between any two processors is the length of the shortest path connecting them, and the *diameter* of a distributed-memory parallel computer is the maximum distance taken over all pairs of processors. Finally, the *degree* of a processor is the number of neighbors it has, and the degree of such a computer is the maximum of the degrees taken over all processors.

The time for communication between neighbors being proportional to the amount of information exchanged, the time for communication between any pair of processors is therefore proportional to their distance. Thus, it is of primary importance that the interconnection network have a small diameter. Graphs having the topology of a *hypercube* — to be defined in the next Section —, are ideally suited to modeling distributed-memory parallel computers, mainly because of the fact that the diameter of a hypercube is the logarithm of the number of vertices.

Several parallel computers based on such an architecture, and supporting different programming models — synchronous or asynchronous —, were actually built and sold. These include the T-series by FPS, the Connection Machine CM-1 and CM-2 by Thinking Machines, the Intel iPSC, the NCube from NCube, and many others. As far as the choice of processor was concerned, it varied from company to company. Such computers could present, for instance, either 8 transputers on a 3-dimensional hypercube or $65,536$ bit serial processors sold as a 16-dimensional hypercube.

With so many computers based on the hypercube architecture, models and algorithms were proposed in order to allow applications to be efficiently solved in parallel. For instance, it is interesting to note that restrictions on the access to specific memory positions yield a shared memory model for hypercube architectures. This can be seen as follows. Let

$G(V, E)$ be a hypercube with N vertices. Then, we can imagine that the memory positions $m_1, m_2, \ldots, m_n$ are such that PE_i can only access memory positions m_j, where $j = i$, or the vertices v_j and v_i of V are neighbors in G [10].

In the remainder of this report, we shall try to answer most of the questions that a non-specialist reader may have. However, some areas of "hypercube computing" will not be covered here. We shall focus on the topological aspects of the hypercube, covered in Section 2, followed by some topics in communications and routing in Section 3. Sections 4 and 5 describe the main tools and some algorithms for the fine-grain hypercube (to be defined in Section 4). We close the report with some open problems and directions for further research.

2 Topological Aspects

For the sake of completeness, we define the notation to be used in this report. An interconnection network with the topology of a d-dimensional hypercube, denoted $H(d)$, is composed of $N = 2^d$ processors, labeled from 0 to $N - 1$, and $dN/2$ communication links. Let $(i)_2$ be the binary string representing i. Then, the neighbors of PE_i are all PE_j such that $(i)_2$ and $(j)_2$ differ in exactly one bit position, say k, $0 \leq k < d$. In this case, we say that PE_i and PE_j are neighbors along dimension k. For instance, PE_0 and PE_4 are neighbors along dimension 2 in any $d > 2$ dimensional hypercube. The processors are then the vertices of a hypercube of dimension d, each connected to d neighbors (see figure 1). It is not difficult to see that the maximum distance in a hypercube is given by those pairs of processors whose binary string differ in all d positions, implying that its diameter is[1] $d = \log N$. In the following, the terms vertices, nodes and processors all refer to a computing element.

2.1 Graph properties

A *bipartition* of $H(d)$ is a partition of V into two sets, say A and B. Let $[A : B]$ denote the set of all edges of the hypercube with one endpoint in A and another in B. A *bisection* of $H(d)$ is defined as a bipartition where $|A| = |B| = N/2$. The *bisection width* of $H(d)$ is then the minimum cardinality of $[A : B]$ taken over all possible bisections of $H(d)$. Note that this is an important parameter for measuring the quality of an interconnection network, since any communication between the processors in A and B must traverse edges in $[A : B]$. Hence, the higher the bisection width of an interconnection network, the smaller the communication contention. Therefore, apart from having a small diameter, the hypercube is also a good choice for the interconnection network of distributed memory parallel computers because of its high bisection width, which equals $N/2$.

Furthermore, a hypercube $H(d)$ can be decomposed in d different ways into two copies of $H(d - 1)$, with $N/2$ edges connecting them. In order to find one such decomposition, it suffices to fix any one bit position in the processors' addresses, say position k, $1 < k < d - 1$.

[1] All logarithms in this report are to the base 2, unless otherwise specified.

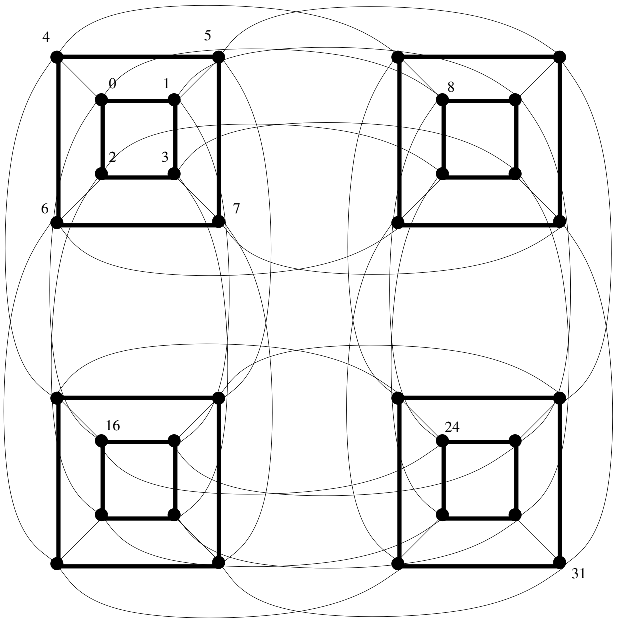


Figure 1: *An $H(5)$ as 4 copies of $H(3)$ (in bold).*

Then, the two copies of $H(d-1)$ are composed of the vertices $i_{d-1}\dots i_{k+1}0i_{k-1}\dots i_0$ and $i_{d-1}\dots i_{k+1}1i_{k-1}\dots i_0$, respectively.

Such a decomposition technique can be extended in such a way that a d-dimensional hypercube can be decomposed as 2^b copies of $H(d-b)$, $0 \leq b \leq d$. This is obtained by fixing an arbitrary number b of positions in the binary string of the processors' addresses, since the remaining non-fixed positions represent a $(d-b)$-dimensional hypercube. In figure 1, an $H(5)$ is depicted as 4 copies of $H(3)$. In that case, bit positions 3 and 4 are considered as fixed in each of the copies.

It is interesting to notice that one can use these decompositions in order to implement divide & conquer algorithms [24] in hypercubes. Moreover, they yield a recursive construction of the hypercube, as follows. A 0-dimensional hypercube is composed of one node only. In order to build $H(1)$, we take two copies of $H(0)$ and join the two nodes by an edge. In general, we can build an $H(d)$ by taking two copies of $H(d-1)$ and connecting the isomorphic nodes by an edge (see figure 2).

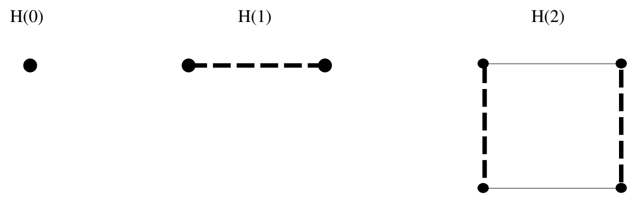


Figure 2: *Recursive construction of H(1) and H(2).*

One further advantage of the hypercube, making it easy to use, is the fact that it is vertex and edge symmetric. This means that for every pair of edges (v_i, v_j) and (u_r, u_s) of $H(d)$, one can find an automorphism[2] f of $H(d)$ such that $(f(v_i), f(v_j)) = (u_r, u_s)$ (see figure 3) [25, 33]. Very informally, we could say that, as far as the neighbors are concerned, the hypercube looks the same from every node.

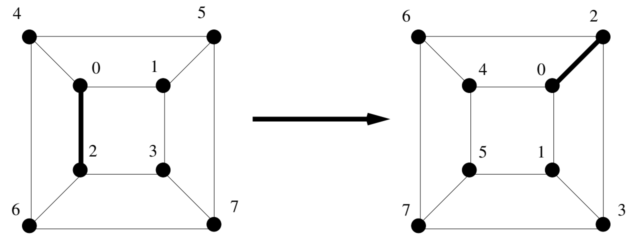


Figure 3: *Example of symmetry in the hypercube.*

2.2 Embedding

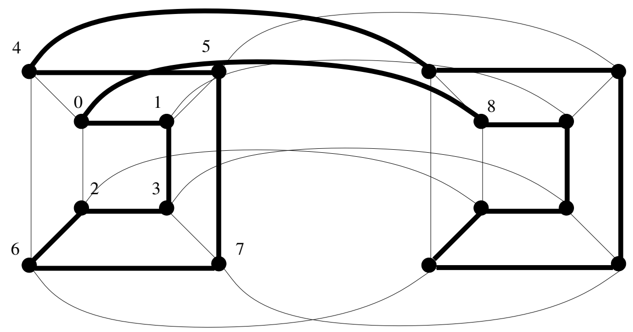
Another attractive characteristic of this architecture is that it has a large number of other architectures as its subgraphs. This implies that algorithms designed for several architectures can be implemented on hypercubes with no increase in the elapsed time. For instance, if $d = p+q$, then the two-dimensional torus with $2^p \times 2^q$ vertices is a subgraph of the hypercube[3]. So, any algorithm running in such torus runs in the hypercube in exactly the same number of steps.

When a given topology $G(V, E)$ is not a subgraph of the hypercube, the goal is to find an embedding[4] that minimizes some parameters, that can be informally described as follows.

- *dilation:* the maximum distance in the hypercube between any two neighbors in G.
- *congestion:* the maximum number of different paths, representing edges of G, through a same edge of the hypercube.
- *expansion:* the ratio $N/|V|$. In case the hypercube is the smallest one such that $N \geq |V|$, it is called *optimal hypercube* with regard to G.

The general case of deciding whether a given graph is a subgraph of the hypercube is NP-Complete [34]. Nevertheless, some results exist concerning specific topologies.

[2]An automorphism of $H(d)$ is a bijection from V onto V which induces a bijection from E onto E.
[3]A $2^p \times 2^q$ torus is a $2^p \times 2^q$ mesh with wraparound connections [32].
[4]An embedding of a graph $G_1(V_1, E_1)$ into a graph $G_2(V_2, E_2)$, $|V_2| \geq |V_1|$, is an injective mapping $f : V_1 \rightarrow V_2$, along with a mapping $g : E_1 \rightarrow \mathcal{P}$, where $\mathcal{P}$ is the set of all simple paths of G_2, such that $g((u, v))$ is a simple path that starts in $f(u)$ and ends in $f(v)$.



Corresponding reflected Gray code:

0000	0110	1100	1010
0001	0111	1101	1011
0011	0101	1111	1001
0010	0100	1110	1000

Figure 4: *A reflected Gray code-based Hamiltonian cycle in $H(4)$.*

Cycles Proving that a cycle of N vertices is a subgraph of $H(d)$ is done with the help of Gray codes. A *Gray code of order d*, $\mathcal{G}_d$, is an ordered sequence of the 2^d binary strings of length d, such that two consecutive strings differ in exactly one position. It is clear that this definition implies that hypercube nodes represented by two consecutive strings in $\mathcal{G}_d$ are therefore neighbors. Moreover, the first and the last strings in the code also differ in only one position, yielding the cycle. The question is then how to build such a sequence.

Definition 1 *A reflected Gray code of order d, $\mathcal{G}_d$, is recursively defined as follows. Let $\mathcal{G}_1 = \{0, 1\}$ and $\mathcal{G}_{d-1}$ be a Gray code of order $d-1$, composed of the strings $\{\alpha_1, \alpha_2, \ldots, \alpha_{2^{d-1}}\}$. Then, $\mathcal{G}_d = \{0\alpha_1, 0\alpha_2, \ldots, 0\alpha_{2^{d-1}}, 1\alpha_{2^{d-1}}, \ldots, 1\alpha_2, 1\alpha_1\}$.*

As an important consequence, this constructively proves that the hypercube is *Hamiltonian*[5] (see figure 4). As a matter of fact, all cycles of even length are its subgraphs.

Binary trees Another well studied embedding problem is the one of complete binary trees, noted $T(h)$, with $2^{h+1} - 1$ nodes. It is known how to embed them in $H(h+2)$ with dilation 1, although it is impossible to find such an embedding in $H(h+1)$, its optimal hypercube [25, 34]. On the other hand, a slightly different tree, namely the *double rooted complete binary tree DRT(h)*, with 2^{h+1} nodes, is a subgraph of $H(h+1)$ (see figure 5).

Several more results can be found in the literature, related to the embedding of different topologies in the hypercube, such as meshes, meshes of trees, pyramids, and others. For a thorough discussion on embedding in general, we refer the reader to [25, 34].

[5] A graph is said to be Hamiltonian if it has a *Hamiltonian cycle*, i.e., there is a path containing all vertices exactly once, such that the last vertex in the path is a neighbor of the first.

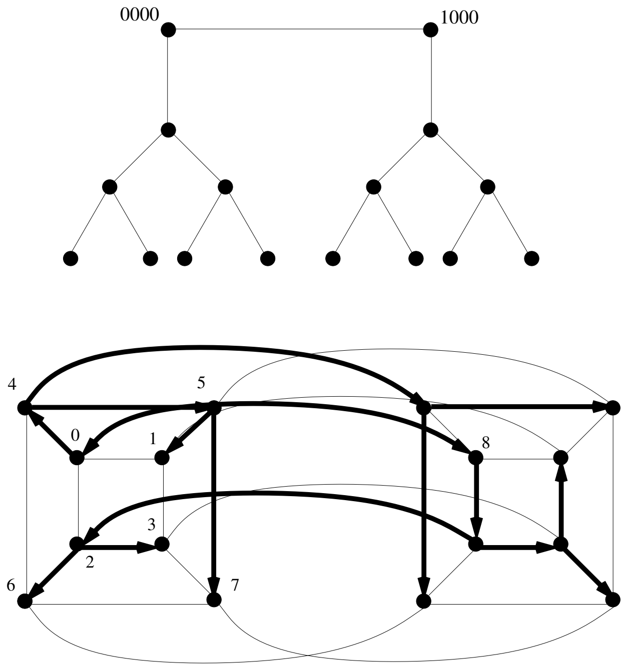


Figure 5: *A DRT(3) and its embedding in H(4).*

3 Communication issues

In parallel applications, a large amount of processing time is spent routing data among the processors. Therefore, routing algorithms, techniques, and models, are of primary importance for an efficient use of parallel computers. In order to design fast routing algorithms, several machine-related features as well as the specific protocol to be used, or even how the destination addresses are set, have to be taken into account.

Since the early days of parallel computing, packet routing has been implemented to support inter-processor communication. Suppose that a processor sends a message to a non-neighbor. Under this protocol, also known as *store&forward*, every processor in the path taken by the message first stores the message, then forwards it to the next processor in the path. Practical experiments showed that the cost of a sending-to-neighbor operation could be modeled as a startup time β (establishing the links, for instance) plus the time spent by the actual sending of the message, which depends on the size of the message (L) and on the inverse of the link's bandwidth (τ). So, the time for such an operation could be expressed as $t = \beta + L\tau$, known as the *linear* model. However, in the case of short messages and large start up times, it is important to understand the behavior of communication algorithms with regard to the number of steps taken only. Then, it could be assumed that a sending-to-neighbor operation takes constant time, regardless of the size of the message to be sent. This is the *constant* model [20].

Another important parameter for routing problems is the concurrency of communication links, i.e., the number of links that can be used concurrently by each processor [18]. Algorithms that, in a single step, allow communication with only one neighbor are said to respect the *1-port* model. If, on the other hand, communication with all the d neighbors is allowed, then they are said to respect the *d-port* model.

When the routing pattern is known in advance, we call it *off-line* routing. This happens to be the case in many scientific applications. Also, for some applications where this is not true, it could be interesting to precompute the communication pattern and store it in the processors (for instance, when the same communication pattern is to be repeated over and over during the execution of the algorithm). On the other hand, it could be crucial to be able to route *on-line* patterns also, i.e., patterns that are run-time dependent and thus, known on the fly.

3.1 Off-line routing

Of particular interest in off-line routing are *global* communication procedures like *one-to-all* or *all-to-all*, and *permutation* routing, where each processor has a piece of information to transmit to one processor, and all destinations form a permutation of the processors' addresses. Some examples are as follows.

Broadcasting: one processor transmits a piece of information to all processors. It is also known as one-to-all.

Gossiping: every processor transmits a piece of information to all processors. It is also known as all-to-all.

Scattering: one processor transmits a different piece of information to each other processor.

Gathering: every processor transmits a piece of information to the same processor.

BPC permutations: Bit Permute Complement permutations are those where the destination for each processor is a permutation of its own bit-string address, and some of the resulting bits may be complemented. Examples of BPC permutations are: matrix transpose, shuffled row major, perfect shuffle, and others.

3.1.1 Constant model

In the constant model, the use of d-ports allows all permutations to be routed off-line within $(2d-1)$ steps. In the one-port, the communication patterns above can be routed optimally, in $\log N$ neighbor-to-neighbor communication steps. In the following we show the interesting example of routing BPC permutations.

Definition 2 *A BPC permutation on $H(d)$ is specified by a permutation vector $p = [p_{d-1}, p_{d-2}, \ldots, p_0]$, where $-d \leq p_i \leq d$, such that $[|p_{d-1}|, |p_{d-2}|, \ldots, |p_0|]$ is a permutation of $[0, 1, \ldots, d-1]$. The destination processor PE_j of data stored in PE_i, $(i)_2 = i_{d-1}i_{d-2}, \ldots, i_0$, is given by $j_{|p_k|} = i_k$, if $p_k \geq 0$, or $\overline{i_k}$ otherwise. We remark that -0 is considered less than 0.*

The routing corrects the dimensions from right to left in the destination address, i.e., from j_0 to j_{d-1}. If $p_k = k$, then the k-th bit of the address is already correct. If $p_k = \overline{k}$, then this register has its k-th dimension corrected. If $|p_k| \neq k$, then a non-trivial cycle starts at bit k. The next position in the cycle is given by $c = |p_k|$, and the position following this one is given by $e = |p_c|$. Since this is a BPC permutation, $c \neq e$. Thus, if $e = k$, then c is the end of the cycle, otherwise e is the next position in the cycle. Repeating this reasoning, the complete cycle starting at position k can be identified.

Therefore, once all cycles have been identified, routing a BPC permutation is implemented so that cycles are routed one after another, with data changing dimension when necessary. When handling a cycle $c_0, c_1, \ldots, c_l$, routing is done, if necessary, with regard to dimension c_1 first, then c_2, eventually c_l, and finally c_0. Notice that we need queues of size two in order to store data that are to be routed along the next routing dimension [33].

3.1.2 Linear model

Under the linear model, the idea is to use pipelining to hide the diameter of the hypercube. For instance, the technique for broadcasting is based on the existence and construction of height h spanning trees[6] of the hypercube (see figure 6).

Assume that a message of length L is to be broadcast from PE_0. Before routing starts, the message is divided into packets of size s, whose optimal value can be analytically computed as $\sqrt{\frac{L\beta}{(h-1)\tau}}$. The basic idea is then to send the packets through the spanning tree in a pipelined way. Hence, the key point is to find small height spanning trees, because the time for broadcasting with this technique is then shown to be $(\sqrt{L\tau} + \sqrt{(h-1)\beta})^2$.

Note that this result holds for the d-port model, since processors send and receive messages onto two links concurrently. However, only two ports out of d are being used. Therefore, broadcasting can be improved if one considers edges as two arcs, one in each direction, and finds d arc-disjoint spanning trees in $H(d)$. By pipelining packets through different arc-disjoint spanning trees, the following can be shown.

- Under the 1-port model, broadcasting completes in $(\sqrt{L\tau} + \sqrt{d\beta})^2$ steps [21].
- Under the d-port model, broadcasting completes in $(\sqrt{\frac{L\tau}{d}} + \sqrt{(d-1)\beta})^2$ steps [36].

The spanning tree used is the *Spanning Binomial Tree* (see figure 6), and the arc-disjoint spanning tree is the *Rotated Spanning Binomial Tree* (see [18, 19, 34, and references therein] for more details). Recall that, because of the vertex symmetry of the hypercube discussed in Section 2.1, one needs to find only one spanning tree, rooted in any node, say i. If the broadcast is to be executed from another node, say j, it suffices to find an automorphism f that maps node i into node j. One spanning tree rooted in node j is then the image by f of the spanning tree rooted in node i.

[6] A *spanning tree* of a graph G is a tree that contains all the vertices of G and is a subgraph of G.

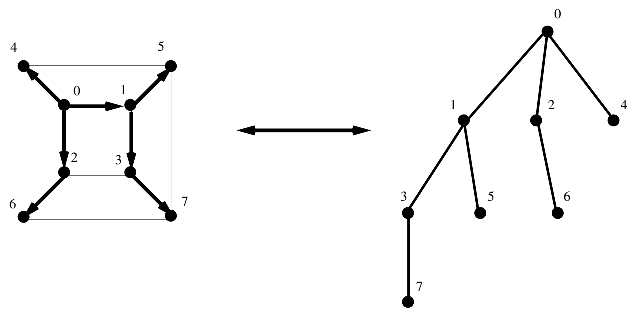


Figure 6: *A spanning binomial tree of $H(3)$, rooted in node 0.*

3.2 On-line permutation routing

Packet routing algorithms for on-line permutation routing on the hypercube may be classified into *oblivious* or *adaptive.*

For *oblivious algorithms* the routing path of a packet is uniquely determined from its source and final destination. Both the commonly used greedy algorithm, which uniformly corrects hypercube dimensions in either increasing or decreasing order, and also a variant which corrects dimensions in random order belong to this class [7, 37]. Although these strategies are simple, their worst-case time complexity is $\Omega(\sqrt{N})$ [39]. An optimal oblivious algorithm, based on many to one and one to many routing in subhypercubes, achieves a worst-case time delay of $\Theta(\sqrt{N}/\log N)$ [22].

Even though optimal deterministic oblivious routing schemes on the N-node binary hypercube, could be constructed for $N \leq 64$ [19], of major importance is the *randomized permutation routing* algorithm, first proposed by Valiant and Brebner [39], and later improved by Valiant [38]. For this algorithm, the probability that all packets have been routed correctly within $c \log N$ time is proved greater than $(1 - e^{-\delta \log N})$, where δ is a constant depending only on c.

The algorithm consists of a *randomization phase*, where packets are deterministically sent to randomly selected nodes through the network, followed by a *deterministic phase*, where packets follow – deterministically again – a shortest-path route to their final destination. Notice that, at the end of the first phase, packets are distributed randomly. The role of randomization is to reduce the gap between the average and the worst-case time complexities by avoiding worst-case congestions.

In the context of generalized hypercubes[7], this result has been slightly improved in [16].

Adaptive routing algorithms can use much more information about the messages and the system congestion, like the destination of many packets for instance. They are usually disguised as sorting algorithms. Therefore, the best known asymptotic result for deterministic routing is the same as for sorting (see Section 4.3).

[7] A b-ary k-dimensional generalized hypercube has $N = b^k$ nodes. A node x can be represented as $x = x_0x_1 \ldots x_{k-1}$, where $x_i \in \{0, 1, \ldots b-1\}$, for $0 \leq i \leq k-1$. Two nodes $x = x_0x_1 \ldots x_{k-1}$ and $y = y_0y_1 \ldots y_{k-1}$ are connected by an edge if and only if there is exactly one j, $0 \leq j \leq k-1$, such that $x_j \neq y_j$.

3.3 Wormhole routing

The *wormhole* protocol was conceived in order to mask distances in the network. In fact, wormhole routing is used in most of the distributed-memory parallel computers built today. It is implemented through the use of special-purpose communication devices — named routers —, incorporated with the computing elements, that solely perform routing operations. Each message is composed of a *header*, coding the destination address; the body of the message, divided into a number of atomic units called *flits*; and a *tail*, indicating the end of the message.

The message is routed through the interconnection network so that intermediate computing elements in the message path do not have access to it. Only their associated routers deal with the message, treating it in a pipeline-like manner, flit after flit. Hence the name *wormhole* routing, since the delivering of messages reminds us of worms crawling in the network. The main feature of this protocol is that as soon as the tail of a message leaves a non-final router, the router is freed to perform other communication tasks.

For the one-port model, most information dissemination problems have $\log N$ as a lower bound on the number of steps required to complete. Thus, under such a model, wormhole routing is of little use in the hypercube. On the other hand, under the d-port model, the lower bound for this kind of problems is usually $\lfloor \log_{d+1} N \rfloor = \Omega(\log N / \log\log N)$. For optimal broadcasting, for instance, a tiling of the hypercube, such as the tiling of the 2-dimensional torus proposed in [32], has to be found.

4 Useful algorithmic tools

This Section and the next focus on synchronous algorithms for *fine-grain* hypercubes. In this model, there are at least as many processors as the problem input size, and each processor has some local memory, of constant size, organized in $O(d)$-bit words. At each time step, a processor can simultaneously send a word of data to one of its neighbors, receive a word of data from one of its neighbors, and perform a local operation on word-sized operands.

Algorithms where at each time step, only the edges associated with a single dimension are used, and consecutive dimensions are used on consecutive steps, are called *normal* algorithms [25, Section 3.1.4].

From the architectural point of view, one of the main drawbacks of the hypercube stems from its logarithmic degree, due to the large number of wires required to interconnect the processors in big systems. However, this problem can be solved through the use of bounded-degree hypercube related networks, better known – along with hypercubes – as *hypercubic* networks. Any normal algorithm designed for the hypercube can be simulated in any of these networks with only a constant slowdown. The most studied members of this class, besides the hypercube itself, are the *butterfly*, the *shuffle-exchange*, and the *de Bruijn* graphs (see [25] for details on such interconnection networks). All the algorithms described henceforth are normal.

The procedures described next are well known and represent powerful tools for the design of hypercubic algorithms, although their most important feature may be the fact that they can be combined to implement, in time proportional to sorting, *random access concurrent read* on hypercubes, as well as *random access write*, and even some forms of *concurrent* write. This implies that PRAM algorithms can be simulated in hypercubes with an overhead equal to the time complexity of the best sorting algorithm. We recall that the diameter of the hypercube yields a trivial $\Omega(\log N)$ time lower bound for solving any global sensitive problem.

For the sake of clarity, we shall show the detailed implementation of some of these procedures. They all use the same strategy, correcting the bits in consecutive dimensions either from right to left, or from left to right. The interested reader is referred to [30, 33] for more details.

4.1 Semigroup operations

A semigroup computation applies an associative operation to all selected data items. The procedures that follow are computed in optimal time, i.e., they complete in $O(\log N)$ time.

Procedure Count.

> **Input:** One flag per processor. **Output:** For all PE_i, the total number of processors PE_j whose flag was *up*.

Let i_b denote the b-th digit, from right to left, in $(i)_2$. Let each processor have variables T() and ST(), denoting total and subtotal respectively, both of them initialized with the value 0. Counting is as follows.

```
Each PE_i:
  if flag is up then T(i):= 1;
  for b := 0 to d − 1 do
    exchange T(i) along dimension b and store it in ST(i);
    T(i) := ST(i) + T(i);
```

Procedure *Count* above can be extended to compute a special case of *parallel prefix*, known as *rank*, described in the following.

Procedure Rank.

> **Input:** One flag per processor. **Output:** For each PE_i whose flag was *up*, the number of processors PE_j, $j < i$, such that PE_j's flag was *up*.

Let i_b denote the b-th digit, from right to left, in $(i)_2$. Let each processor have variables R(), T(), ST(), denoting rank, total, and subtotal respectively, all of them initialized with the value 0. The *Rank* algorithm is as follows.

```
Each PE_i:
  if flag is up then T(i):=R(i):= 1;
  for b := 0 to d − 1 do
    exchange T(i) along dimension b and store it in ST(i);
    if i_b = 1 then R(i) := R(i) + ST(i);
    T(i) := ST(i) + T(i);
  if flag was down then R(i) := ∞.
```

It is not very difficult to modify the above procedures so that they can compute a *segmented parallel prefix*, defined as follows. Let PE_{b_b} and PE_{b_e} be the processors at the beginning and the end of a block respectively. (A block is a set of contiguous processors that have the same value stored in *block(i)*.) We leave as an exercise to the interested reader the description of a procedure SegmentedPrefix that takes two registers per PE_i, *block(i)* and *r(i)*, as input, and that computes $\bigoplus_{j=b_b}^{b_e} r(j)$, where $\bigoplus$ is an associative operation.

4.2 Data-movement operations

The procedures that follow can be seen as special cases of routing, and were studied in [30]. Again, they are all time-optimal.

Procedure Concentrate.

Input: One register per processor, associated to a flag. **Output:** All registers whose flag was *up* are concentrated at the beginning of the hypercube, one per processor. The initial relative order remains unchanged.

Let i_b denote the b-th digit, from right to left, in $(i)_2$. Let each processor have a variable G() to be concentrated. The record R() stores the value given by the rank procedure described previously.

```
Rank
Each PE_i:
  for b := 0 to d − 1 do
    if R(i) ≠ ∞ and (R(i))_b ≠ i_b then
      exchange (G(i),R(i)) along dimension b.
```

Procedure Route.

Input: One register per processor, associated to a destination *dest(i)* and to a flag. The destinations should be *monotone*, i.e., for two processors PE_i and PE_j, it holds that $i < j \Leftrightarrow dest(i) < dest(j)$. **Output:** All registers whose flag was *up* are routed to the processor given by *dest()*. Let i_b denote the b-th digit, from right to left, in $(i)_2$. Let each processor have a variable G() to be concentrated. The record R() stores the value given by the rank procedure described previously.

```
Concentrate
Each PE_i:
  if R(i) = ∞ then
    dest(i) := ∞
  for b := d − 1 to 0 do
    if dest(i) ≠ ∞ and (dest(i))_b ≠ i_b then
      exchange (G(i),dest(i)) along dimension b.
```

With the same input as for the Route procedure, we propose that the reader describe a procedure RouteAndCopy that, for all registers whose flag is *up*, routes a copy of the register of PE_i to processors $PE_{dest(i-1)+1}, \dots, PE_{dest(i)}$.

4.3 Sorting

Below we discuss the main deterministic sorting algorithms for fine-grain hypercubes. We describe the *bitonic* merge and sort, giving a recursive description of the latter, since this is the usual way algorithms are described for hypercubes. Computing the time complexity is then a matter of solving a recurrence relation. In the case of the bitonic sort, the relation to solve is $T(N) := T(N/2) + O(\log N)$, that equals $O(\log^2 N)$.

Procedure BitonicMerge.

Input: Two sorted lists, one in ascending order, the other in descending order. **Output:** The two lists merged into one sorted list. **Time:** $O(\log N)$.

Let i_b denote the b-th digit, from right to left, in $(i)_2$. Let each processor have a variable M(), where its corresponding element in the merged list is to be stored, and an auxiliary variable L'(). The original lists are stored in the record L().

```
Each PE_i:
  for b := d − 1 to 0 do
    exchange L(i) along dimension b and store it in L'(i).
    if (i)_b = 0 then
      M(i) := min{L(i), L'(i)}
     else M(i) := max{L(i), L'(i)}.
```

Procedure BitonicSort.

Input: An unsorted list. **Output:** The sorted list. **Time:** $O(\log^2 N)$.

```
Do in parallel
  BitonicSort{first half of the list in ascending order}
  BitonicSort{second half of the list in descending order}
BitonicMerge{the two lists}
```

For more details on both algorithms, such as proof of correctness and time complexity, we refer the interested reader to [1, 24].

In the case where there are more computing elements than data elements to be sorted, the *sparse enumeration* sort of [31] is able to sort N elements in a hypercube with $N^{1+1/k}$ processors, $1 \leq k \leq \log N$, in time $O(k \log N)$.

It is worth mentioning that the bitonic sort remained the best sorting algorithm for hypercubes for more than 20 years. Moreover, it is very simple to describe and implement, with a fairly small constant (approximately 1/2) hidden in its time complexity of $O(\log^2 N)$. Nevertheless, the asymptotical time complexity of the bitonic sort was finally improved by the *share* sort algorithm proposed in [11], that has another version described in [25]. To the best of our knowledge, the share sort is the asymptotically fastest existing sorting algorithm for sorting N data elements with N hypercube processors, in $O(\log N (\log\log N)^2)$ time[8]. We note that the share sort has a so-called *non-uniform* version that runs in $O(\log N \log\log N)$ time, where non-uniformity means that a routing table for each processor has to be computed beforehand, once and for all runs of the share sort on a given machine. A routing phase of the algorithm is then accelerated by table lookups. The problem is the exponential complexity of the algorithm building such tables. This is the reason why we preferred to list here the time complexity of the uniform version.

Finally, the sparse version of share sort was studied in [6]. It is therefore possible to sort N elements in a hypercube with $N^{1+1/k}$ processors, $2 \leq k \leq \log N$, in time $O(\log N \log^2 k)$.

4.4 Random Access Read

Suppose that each PE_i stores in a variable $A(i)$ the address from where data have to be read, and that processors that are not reading in this step have $A() = \infty$. Let *ID(i)=i* be used to keep track of the reading processors. Suppose further that each processor shall work with tuples containing four variables, namely, *A(), ID(), D()*, and *S()*. The algorithm implementing random access read is as follows [30].

1. Sort tuples by $A()$.
   All valid addresses are now concentrated at the beginning of the hypercube. Let $S()$ take the value of the current address.

2. IdentifyEndOfBlock.
   Since we are implementing concurrent read, several processors may have the same value in $A()$. Then we avoid routing conflicts by selecting end-of-block processors to be in charge of getting the data and distributing it to the others in its block. (Recall that a block is a set of contiguous processors that have the same value stored in *block(i)*.) This procedure is as follows.

[8] Unfortunately, the constant hidden in the *big Oh* notation is huge.

Procedure IdentifyEndOfBlock.

Input: One register per PE_i, called *block(i)*. **Output:** The last processor of each block is identified.

```
All PE_i, i > 0:
  Route{block(i) into aux(i-1)};
Each PE_i:
  if block(i) ≠ aux(i) then flag:=up
```

3. Route tuples of end-of-block processors by $A()$.
   The required data are then copied into $D()$.

4. RouteAndCopy tuples by $S()$.
   Every processor in each block receives a copy of the information required. The data now have to be sent to the original processors requiring them.

5. Sort tuples by *ID()*.
   In order to make the destinations monotone.

6. Route tuples by *ID()*.
   The reading step finishes.

It is easy to see that the time complexity of a random access read is dominated by the two calls to sorting algorithms in steps 1 and 5, since all the routing procedures are optimal because of the monotonicity of the addresses. Then, *PRAM* algorithms can be implemented in the hypercube with a time overhead proportional to the time complexity of sorting in the hypercube.

5 Solving Problems

In this Section, we review some algorithms used for solving different problems in Computational Geometry, since they are good representatives of the most difficult problems arising in the design of hypercube algorithms.

5.1 Classic problems

When it comes to discussing optimality of hypercube algorithms, one has to be somewhat careful. We can say that an algorithm is *time-optimal* if its number of steps is $O(\log N)$. However, this definition is not complete, since sparseness is not taken into account (note that the sparse enumeration sort, from Section 4, is time-optimal provided N^2 processors are available). Then, we define *dense* algorithms as those that solve problems of size N with

N processors, as opposed to *sparse* algorithms, that use more processors than the size of the input. For instance, in Section 4 we saw several time-optimal dense algorithms.

The classic problems we are going to address in this Section have serial time complexity less or equal to that of sorting. For further algorithms see, e.g., [2, 28].

Efficient $O(\log^2 N)$ time dense algorithms were introduced in [35] for several geometric problems like convex hull, ECDF search, and all points closest neighbors. Notice that, before the advent of the share sort [11], such algorithms matched the best known bound for sorting on a hypercube. However, since the serial time complexity of those problems is the same as for sorting, better hypercube algorithms were likely to exist.

5.1.1 Optimal dense algorithms for sorted data

Indeed, in the literature we can find some algorithms for geometric problems that take $O(T_s + \log N)$ time, where T_s is the time complexity of the best sorting algorithm to date. Then, these are efficient dense algorithms, which are optimal in the case where the input is given in a sorted order. On the other hand, and in view of the time/processor tradeoff obtained by the sparse share sort, these are also time-optimal sparse algorithms. This is the case of the solutions for the convex hull algorithm from [29], triangulation and visibility from a point in [26], and the algorithms for maximal points and smallest enclosing box in [9], that we describe in the following.

Convex hull The algorithm for convex hull starts by sorting the points by x coordinates. The points are then divided into $N^{1/4}$ groups of size $N^{3/4}$, to be stored in consecutive subhypercubes. For each group the solution is computed recursively. The partial solutions are then merged by routing, back and forth a number of times, $N^{1/4}$ slopes from each group to *all* other groups in order to compute the supporting lines between any two partial solutions. Thus, there are only $N^{3/4}$ messages being routed, and this can be accomplished within $O(\log N)$ steps if one uses the extra dimensions of the hypercube to avoid routing conflicts. Once the points are sorted, the time complexity is given by $T(N) = T(N^{3/4}) + O(\log N) = O(\log N)$.

Triangulation The solution for triangulation is based on the PRAM algorithm from [40]. First the points are sorted by x coordinates, and the partial solutions, of size $N^{3/4}$ each, are found recursively. The triangulation of the partial solutions uses the supporting lines between the convex hulls of each set. These can be computed with the method used for computing convex hull, as described above. This gives a time complexity as $T(N) = T(N^{3/4}) + O(T_s) = O(T_s)$.

Smallest enclosing box Finally, the smallest enclosing box of a set of points is a box of minimum area of any of the smallest enclosing boxes that contains an edge collinear with a hull edge. Then, once the convex hull of the set has been computed, for each hull edge the support lines parallel and perpendicular to the edge are computed through search techniques where each point of the convex hull has an interval of support angles, and the edges are located in the interval containing their support angle. Such search operations can be accomplished in logarithmic time using the standard operations described in Section 4.

5.1.2 Time-optimal sparse algorithms

A different approach for designing time-optimal sparse algorithms can be found in [5], where ECDF search, all points closest neighbors, and convex hull are solved in time $O(k \log N)$ with $N^{1+1/k}$ processors, $1 \leq k \leq \log N$. In the ECDF search, one computes for each point the number of points it dominates (i.e., the number of points with both x and y coordinates less than those of the point). The algorithm is based on the observation that, if subsets of points are sorted by x coordinates, then the number of points dominated by a given point can be computed as the sum of the points it dominates in each subset to its left, including its own subset. A divide & conquer strategy is used, where up to $2^{\log N/k}$ partial solutions are merged together at each step. Partial solutions are computed through the bitonic merge, and the extra dimensions are used to avoid collisions during the routing steps, ensuring that no more than $O(\log N)$ time is required in each one of the k steps.

5.1.3 Suboptimal dense algorithms

ECDF search Other dense algorithms were proposed in [26]. They solve ECDF search, all points closest neighbors, and 3-dimensional maxima in time $O(T_s \log\log N)$. The idea is again to recursively solve problems whose size is a root function of the original one. For the ECDF search the recursion is called twice, leading to a time complexity given by $T(N) = 2T(\sqrt{N}) + O(T_s) = O(T_s \log\log N)$. In the algorithm, data are first sorted by x coordinates, then by y coordinates. Then, $\sqrt{N}$ groups of $\sqrt{N}$ consecutive points are formed according to each direction. Let X_i and Y_j, $1 \leq i, j \leq \sqrt{N}$, denote such groups. This defines virtual cells $C_{i,j}$ composed of the points in the intersection of X_i and Y_j (see figure 7). The number of points dominated by a point in $C_{i,j}$ is then the sum of three values, namely, the number of points it dominates in its vertical region (X_i), the number of points it dominates in its horizontal region (Y_j), and the number of points completely to the left and completely below its cell. The first two values are computed recursively.

Selection A sorting based $O(\log N \log^* N)$ time dense algorithm[9] for selecting the k-th smallest element out of N unsorted items was given in [6]. The algorithm is a succession of phases where finer approximations for the k-th smallest element are obtained, so that the correct element is eventually computed. Each such approximate selection phase is identical to the others: sort sublists, then choose splitters as the new sublist. This is iterated until good bounds for the k-th smallest element can be calculated. From one phase to the next,

[9] The function $\log^* N$ gives the number of iterations of the *logarithm* function in order to get the value 1 from N.

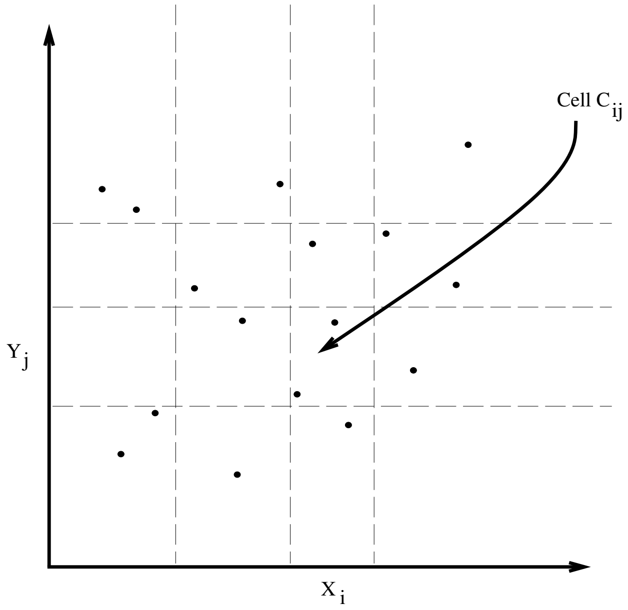


Figure 7: *Groups and cells for solving ECDF search.*

elements that are not between the bounds are definitely discarded and the whole remaining list concentrated. As far as the time complexity is concerned, each phase is tuned so that it costs $O(\log N)$ time, for a total of h phases. We observe that the only costly operation used is sorting. At the beginning of the algorithm sublists are sorted with the share sort, since the hypercube is full. However, in all remaining sort operations, it can be shown that the hypercube has enough free dimensions to allow the use of sparse sorts, that can be used in different places during the algorithm. The best combination of the sorting algorithms yields $h = O(\log^* N)$.

Polygons Finding shortest paths in rectilinear polygons was solved in time $O(T_t + \log N)$ with P_t processors in [17], where T_t and P_t are, respectively, the time and the number of processors required for solving trapezoidal decomposition (see Section 5.2). The algorithm is a combination of finding the visible pairs from special points (where the trapezoidal decomposition is used), and a technique for nested parenthesis elimination, along with the traversal of monotone rectilinear staircases.

Very recently, time-optimal dense algorithms were given for simple-polygon problems, like visibility, convex hull, monotonicity, and others ([3]), and for tree contraction, a difficult problem arising in the solution of many graph algorithms ([27]).

5.2 Data structures

Besides the diameter, one of the main differences between the *PRAM* and the hypercube is that the *PRAM* has a large shared memory similar to that of a sequential computer, while in the hypercube the memory is divided into blocks of constant size that are distributed among the processors. The concurrent access to the central memory has been extensively used in the design of efficient parallel search algorithms for Computational Geometry in the $CREW$[10] $PRAM$, where each processor is in charge of answering a different query. The shared memory paradigm allows the implementation of very sophisticated data structures, and once the data structure has been collectively built by all the processors, each one can independently gain access to it and execute a standard sequential search algorithm.

In hypercubes, concurrently answering several independent queries is not as straightforward. Direct approaches either use very simple data structures and concentrate on routing and collision avoidance issues, or simulate $PRAM$ algorithms that use complex data structures, yielding more elegant, though less efficient, parallel algorithms for the hypercube.

One of the first tools for elegant and theoretically efficient use of data structures in hypercubes was the *Multi-Way Search* (M-way search, for short) proposed in [15]. With the M-way search, several queries — whose maximum search path size is h —, can traverse tree-like data structures simultaneously, with an overhead of only $O(\log N)$ with respect to $CREW$ $PRAM$ algorithms.

In the M-way search the queries advance synchronously through the data structure, one level per phase. It is clear that if during a phase the queries were to be routed to the nodes they have to visit, then we would have severe congestion problems. Therefore, the basic idea is to create as many copies of each node of the data structure as necessary, and route them to the queries that requested them. If a certain order is kept among the queries, guaranteeing address monotonicity for the routing operations, then each phase can be implemented in $O(\log N)$ steps, by the use of a constant number of calls to the procedures previously described in Section 4.

A powerful extension to this technique was proposed in [14]. *Hypercube cascading* is based on the *dynamic* M-way search in [12], which is in turn an extension of the M-way search that copes with dynamic structures. With hypercube cascading, most data structures represented by directed acyclic graphs can be traversed by concurrent queries that synchronously advance in their paths, step by step. Moreover, it keeps an important feature of the dynamic M-way search: it takes into account dynamic data structures, where at most $O(N)$ insertions and deletions can take place at each step, with structure update, with the same overhead as before. Finally, hypercube cascading allows more complex operations concerning the computation of the queries' paths.

Such a multi-iterative search paradigm consists of h phases such that, at the end of a phase x, $(1 \leq x \leq h)$, all queries are sorted with regard to the index of the x-th node of the search path. Furthermore, each PE_i has a copy of the x-th node of the search path of the i-th query, as well as the information concerning the next node in the query's search path.

Phase 1 is implemented through a sorting. In case the directed acyclic graph representing the data structure is monotone (see definition below), then sorting can be avoided in the re-

[10] In the $CREW$ (Concurrent Read Exclusive Write) model, all processors are allowed to read from the same memory cell in a given step.

mainder of the algorithm and all the procedures used in a phase x, $(2 \leq x \leq h)$, take $O(\log N)$ time, implying that the time complexity of one such phase is also $O(\log N)$. Therefore, if we have a bounded degree monotone data structure of size s, then we can answer m iterative search queries whose search paths are no longer than h, in time $O((h + (\log N \log N)^2) \log N)$ in hypercubes with $N = \max\{s, m\}$ processors [14].

The only condition imposed on the data structure so that this technique applies is that it should be *monotone*, as follows (see figure 8).

Definition 3 *A directed acyclic graph $G(V, E)$ is monotone if there exists a numbering of the nodes $f : V \Rightarrow \{1, \ldots, n\}$ such that if (v,v') and (w,w') are two edges of G, then $f(v) < f(w) \Leftrightarrow f(v') < f(w')$.*

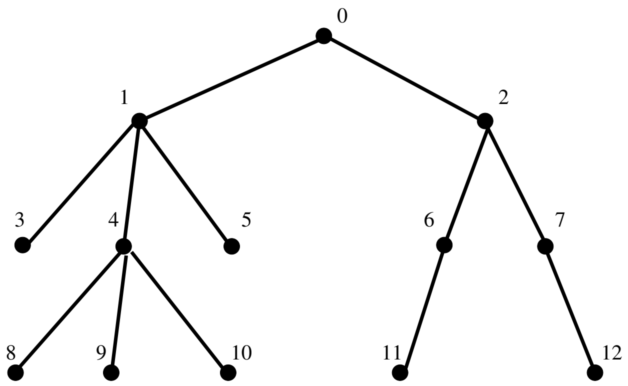


Figure 8: *Example of a monotone numbering of a tree.*

In the following we briefly describe an application of hypercube cascading.

5.2.1 Solving the multiple stabbing problem

This problem consists of determining all intersections of m lines with a simple polygonal path of size s (see figure 9). The solution shown in [14] presents a slowdown factor of only $O(\log N)$ with respect to the sequential version, where just one query line is answered.

With fractional cascading [8], the serial solution of the multiple stabbing problem is based on the following observation: a line l intersects a simple polygonal path P if and only if l intersects the convex hull, $CH(P)$, of P. Then, a decomposition tree of the convex hulls for a given P is built. Such a tree is a binary tree, T, with $CH(P)$ assigned to its root. The two children are the roots of the subtrees representing the first half and the second half of P respectively. Hence, all intersections of P with a query line l can be found through the following traversal of T. At each node, if l intersects the convex hull assigned to the node, then the two subtrees are traversed recursively, otherwise the traversal finishes at this point.

In order to solve the multiple stabbing problem on a hypercube, T has to be built and a strategy for its concurrent traversal defined. Hypercube cascading is used to implement both

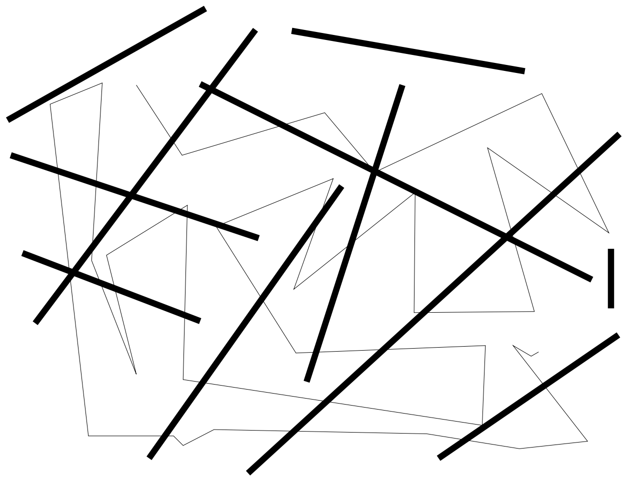


Figure 9: *Example of a multiple stabbing problem.*

phases. Below we give a brief description of the traversal of T. Notice that T is monotone and has bounded degree.

For traversing T, each query line could simply start at the root and be duplicated at each branching in order to simulate the sequential traversal. Unfortunately, this could lead to as many as (sm) queries by the end of the algorithm. The solution stems from a strategy where each line is represented by a single query that answers all intersections with P, traversing T in inorder.

Let $T'(l)$ be the subtree of T that shall be traversed in the sequential solution with fractional cascading for answering the query associated with l. In [8] it was shown that $|T'(l)| = O(k \log \frac{s}{k})$, where k is the number of intersections between l and P. Therefore, the length of the path (of an inorder traversal of T) with hypercube cascading shall also be $O(k \log \frac{N}{k})$, where $N = \max\{s, m\}$, yielding an overall time complexity of $O((k \log \frac{N}{k} + (\log \log N)^2) \log N)$ to answer m queries simultaneously.

5.2.2 Summary

The problems that could be solved through the use of the paradigms cited above are as follows. (For more details on the algorithms, see [2, 28].)

- trapezoidal decomposition and triangulation of polygons of size N in time $O(\log^2 N)$ with a hypercube of size $N \log N$ [15].
- parallel branch and bound on hypercube multiprocessors [12].
- multiple stabbing of a simple polygonal path, i.e., determining all k intersections of m lines with a simple polygonal path of length l, in time $O(k \log N/k + (\log \log N)^2) \log N)$ with a hypercube of size N, where $N = \max\{l, m\}$ [14].

- multiple slanted range search. Let S be a set of l points in the plane. An *aligned trapezoid* is a trapezoid with one side on the x-axis and the two adjacent sides parallel to the y-axis. The *slanted range search problem* consists of reporting all k points contained in the aligned trapezoid. The multiple slanted range search problem consists of answering in parallel, m slanted range search queries on the set S. This can be done in time $O(k \log N/k + (\log\log N)^2) \log N)$ with a hypercube of size N, where $N = \max\{l, m\}$ [14].

- parallel processing of pointer-based quadtrees of height h in hypercube multiprocessors of size N [13]. Namely,

  – convert image or boundary code to quadtree in time $O(\log^2 N)$ or $O((h + (\log\log N)^2) \log N)$, respectively.

  – determine neighbors of all leaf nodes and compute perimeter in time $O(h \log N)$.

  – compute union and intersection of two quadtrees in time $O((h+(\log\log N)^2) \log N)$.

Recently, it was shown that the M-way search paradigm can be used to solve some query problems even faster. In the conference version of [4] an $O(\log N(\log\log N)^3)$ time algorithm is described to solve the multiple point location in a slab, and in the journal version of this paper, it is shown how to apply the same techniques to solve the trapezoidal decomposition in time $O(\log N(\log\log N)^3)$ with $N \log N$ processors in a hypercube.

6 Conclusions and Future Directions

6.1 Some open problems

Even though the design and analysis of hypercube algorithms have been extensively studied, there are several problems still open that would be worth studying. The main one concerns the complexity of sorting, and its implications on the complexity of many other problems in a number of other areas, such as convex hull in computational geometry ([2]) and on-line permutation adaptive routing ([25]), for instance. A related problem with open complexity is the problem of selecting the k-th smallest element out of a non-sorted list. Its current upper bound ([6]) is better than the one of sorting ([11]), but no tight lower bounds exist for either problem for the fine-grain model. Finally, new techniques have recently emerged for optimally solving problems in this model ([3, 27]). It would be very interesting to study their impact on the solution of graph and polygon related problems.

6.2 Summary

In this report, we first described some topological aspects of the hypercube. This study naturally directed our attention to communication issues in parallel machines based on these topologies. Then, we described some of the main tools for algorithm design in the fine-grain hypercube, whose use was exemplified with the solution of problems in computational geometry and with the implementation of powerful and complex parallel data-structures.

In fact, the existing research connected with hypercubes would constitute enough material for several books. Hence, in this report we covered only part of it and we tried to give the reader some flavor of the main issues arising from the use of such an architecture in the design of parallel algorithms. On the other hand, extensive references to text books and to recent research have been provided, even when the specific subject was not discussed here because of space considerations.

Acknowledgments

I am indebted to Pascal Berthomé, Frank Dehne, Apostolos Gerasoulis, Pascal Koiran, Andrew Rau-Chaplin, Ivan Stojmenović, and A. Zomaya, whose invaluable help was most important for the completion of this work. I would like also to thank Olivier de Vel and Kevin O'Brien for carefully reading previous versions of the manuscript.

References

[1] S. G. Akl. *Parallel Sorting Algorithms.* Academic Press, Orlando, FL, 1985.

[2] S.G. Akl and K. Lyons. *Parallel Computational Geometry.* Prentice Hall, 1993.

[3] M. J. Atallah and D. Z. Chen. Optimal parallel hypercube algorithms for polygon problems. *IEEE Transaction on Computers*, to appear.

[4] M. J. Atallah and A. Fabri. On the multisearching problem for hypercubes. In *Proc. 6th Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science. Springer Verlag, 1994. To appear in *Computational Geometry - Theory and Applications.*

[5] P. Berthomé and A. Ferreira. Efficiently solving geometric problems on large hypercube multiprocessors. In S. Tzafestas, P. Borne, and L. Grandinetti, editors, *Parallel and Distributed Computing in Engineering Systems*, pages 123–128. IMACS - North Holland, 1992. To appear in *Parallel Algorithms and Applications.*

[6] P. Berthomé, A. Ferreira, S. Perennes, G. Plaxton, and B. Maggs. Sorting-based selection algorithms on hypercubic networks. In *Proceedings of the 7th IEEE International Parallel Processing Symposium*, pages 89–95. IEEE Press, 1993.

[7] L. N. Bhuyan and D. P. Agrawal. Generalized hypercube and hyperbus structures for a computer network. *IEEE Transactions on Computers*, 4(C-33):323–334, 1984.

[8] B. Chaselle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[9] E. Cohen, R. Miller, E.M. Sarraf, and Q. Stout. Efficient convexity and domination algorithms for fine- and medium-grain hypercube computers. *Algorithmica*, 7:51–75, 1992.

[10] M. Cosnard and A. Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1(2):103–111, 1991.

[11] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 193–203, May 1990.

[12] F. Dehne, A. Ferreira, and A. Rau-Chaplin. Parallel branch and bound on fine grained hypercube multiprocessors. *Parallel Computing*, 15(1-3):201–209, 1990.

[13] F. Dehne, A. Ferreira, and A. Rau-Chaplin. Efficient parallel construction and manipulation of pointer based quadtrees. In *Proceedings of the International Conference on Parallel Processing*, pages 255–262, St. Charles (Ill), 1991.

[14] F. Dehne, A. Ferreira, and A. Rau-Chaplin. Parallel fractional cascading on hypercube multiprocessors. *Computational Geometry - Theory and Applications*, 2:141–167, 1992.

[15] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing*, 8(4):367–375, 1990.

[16] A. Ferreira and M. Grammatikakis. Improved probabilistic routing in generalized hypercubes. In *Proc. 6th Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science. Springer Verlag, 1994.

[17] A. Ferreira and J. Peters. Finding the smallest path in a rectilinear polygon on a hypercube multiprocessor. In *Proceedings of the Third Canadian Conference on Computational Geometry*, pages 162–165, Vancouver (Ca), 1991.

[18] P. Fraigniaud and E. Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics (special issue on broadcasting)*, (to appear).

[19] M. D. Grammatikakis, D. F. Hsu, and F. K. Hwang. Adaptive and oblivious routing on the d-cube. In *Proceedings of the 4th Annual International Symposium on Algorithms and Computation*, to appear in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[20] M. D. Grammatikakis, D. F. Hsu, and M. Kraetzl. Multicomputer routing. Technical Report 94-5, School of Mathematics and Statistics, Curtin University of Technology, Perth, Australia, 1994.

[21] C.-T. Ho and S. L. Johnson. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[22] C. Kaklamanis, D. Krizanc, and T. Tsantilas. Tight bounds for oblivious routing on the hypercube. In *Proceedings of the 2nd Annual ACM Symposium of Parallel Algorithms and Architectures*, pages 31–36, 1990.

[23] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 869–941. Elsevier/MIT Press, 1990.

[24] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.

[25] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1991.

[26] P.D. MacKenzie and Q.F. Stout. Asymptotically efficient hypercube algorithms for computational geometry. In *Proc. 3rd Symposium Frontiers of Massively Parallel Computation*, pages 8–11, 1990.

[27] E. W. Mayr and R. Werchner. Optimal tree contraction on the hypercube and related networks. In J. Flanagan, Y. Matias, and V. Ramachandran, editors, *DIMACS Workshop on Parallel Algorithms: from solving combinatorial problems to solving grand challenge problems*, page 36, November 1993.

[28] R. Miller, A. Rau-Chaplin, and Q. Stout. *Parallel Algorithms for Regular Architectures*. MIT Press, Cambridge, MA, to appear.

[29] R. Miller and Q. Stout. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, 37(12):1605–1618, 1988.

[30] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C–30:101–107, 1981.

[31] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *J. of the ACM*, 29:642–667, 1982.

[32] J.G. Peters and M. Syska. Circuit-switched broadcasting in torus networks. Tech. Rep. TR 93-4, School of Computing Science, Simon Fraser Univ., 1993.

[33] S. Ranka and S. Sahni. *Hypercube Algorithms, with applications to image processing and pattern recognition*. Springer-Verlag, 1990.

[34] J. Rumeur. *Communications dans les Réseaux d'Interconnexions*. Masson, Paris, 1994.

[35] I. Stojmenović. Computational geometry on a hypercube. In *Proceedings of the International Conference on Parallel Processing, Vol. III Algorithms and Applications*, pages 100–103, St. Charles (Ill), 1988.

[36] Q. F. Stout and B. Wagar. Intensive hypercube communication: prearranged communication in link-bound machines. *Journal of Parallel and Distributed Computing*, (10):167–181, 1990.

[37] T. Szymanski. On the permutation capability of a circuit-switched hypercube. In *Proceedings of the IEEE International Conference on Parallel Processing, Vol. I*, pages 103–110, 1989.

[38] L. G. Valiant. A scheme for fast parallel communication. *SIAM J. of Computing*, 2(11):350–361, 1985.

[39] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, May 1981.

[40] C. A. Wang and Y. H. Tsin. An O(log n) time parallel algorithm for triangulating a set of points in the plane. *Information Processing Letters*, 25(1):55–60, 1987.