Distributed Computing on Oriented Anonymous Hypercubes with Faulty Components*

Evangelos Kranakis† Nicola Santoro†

Abstract

We give efficient algorithms for distributed computation on anonymous, labeled, asynchronous oriented hypercubes with possible faulty components (i.e. processors and links) and deterministic processors. Initially, the processors know only the size of the network and that they are inter-connected in a hypercube topology. Faults may occur only before the start of the computation (and that despite this the hypercube remains a connected network). However, the processors do not know where these faults are located. As a measure of complexity we use the total number of bits transmitted during the execution of the algorithm and we concentrate on giving algorithms that will minimize this number of bits. The main result of this paper is an algorithm for computing Boolean functions on anonymous hypercubes with bit cost $O(N\delta_n(\gamma)^2\lambda^2 \log\log N)$, where γ is the number of faulty components (i.e. links plus processors), λ is the number of links which are either faulty, or non-faulty but adjacent to faulty processors, and $\delta_n(\gamma)$ is the diameter of the hypercube.

*A preliminary version of this paper has appeared in the proceedings of the 6th International Workshop on Distributed Algorithms, held in Haifa, Israel, November 2-4, 1992. Springer Lecture Notes in Computer Science, Vol. 647, A. Segall and S. Z. Saks, editors, pages 253 - 263, 1992.

†School of Computer Science, Carleton University, K1S 5B6, Ottawa, Ontario, Canada. **Email:** {kranakis,santoro}@scs.carleton.ca. Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) grants.

1 Introduction

In this paper we consider algorithms which are appropriate for distributed computation on anonymous, oriented, asynchronous, n-dimensional hypercubes Q_n with faulty components (i.e., processors and links).

The problem arising is to determine the computability and associated bit cost (i.e. total number of bits transmitted) of Boolean functions on faulty hypercubes. In the present paper we give efficient algorithms for computing Boolean functions on such networks and also consider the related problem of computing the automorphism group of the network.

1.1 Assumptions

The network we consider is the anonymous, asynchronous oriented hypercube with possible faulty components. The number of faulty components may be arbitrary as long as the hypercube remains connected. If a processor is faulty then all the links adjacent to it are also interpreted as faulty. Faults may occur only before the start of the computation.

We assume that the network links are FIFO, and that the processors have a sense of direction. By this we mean that the hypercube is canonically labeled (the label of link xy is i if and only if x, y differ at exactly the ith bit) and that these labels are known to the processors concerned. In addition we assume that the following assumptions hold:

- the processors know the network topology (in this instance hypercube), and the size of the network, but they do not necessarily know where the faulty components may be,
- the processors are anonymous (i.e., they do not know either the identities of themselves or of the other processors), they are deterministic (i.e. they all run deterministic algorithms), and they all run the same algorithm given the same data,
- the processors can distinguish the faulty links adjacent to it, as well as non-faulty links adjacent to a faulty processor.

The processors occupy the nodes of a hypercube and want to compute a given Boolean function f on $\leq N = 2^n$ variables. Initially each non-faulty processor p has an input bit b_p. When the computation terminates all

processors must output the same value $f(< b_p : p \text{ non-faulty} >)$. A boolean function is called computable if all the processors of the network compute its value correctly on all inputs. (Our notation b_p for the bit associated with processor p does not mean that we assign names to processors. In addition, the input $< b_p : p \in \text{ non-faulty} >$ represents the assignment of bits to all the non-faulty processors of the network, and it will be computed by all the processors via an "input collection" algorithm.)

The assumptions listed above are meant to take "maximum" advantage of network distributivity.

1.2 Related literature

For a discussion regarding the necessity of some of the above assumptions see [**?**]. Routing algorithms on hypercubes have been studied in [**?**]. Faulty hypercube networks have been examined in several papers under the much stronger assumption of synchronous and/or non-identical processors. In such networks it is possible to apply reconfiguring techniques [**?**] (nodes of an $n-1$-dimensional hypercube are mapped into non-faulty nodes of an n-dimensional hypercube with $O(1)$ dilation) or even non-faulty subcube techniques [**?**] (for a given k determine an $n-k$-dimensional subcube with no faulty links). However such techniques are not applicable in our case since they require the availability of processor identities.

1.3 Notation

Let γ denote the number of faulty components of the network, i.e. faulty links plus faulty processors. Let π be the number of faulty processors and λ the number of links which are either faulty, or non-faulty but adjacent to a faulty processor. Since a hypercube has $\log N$ faulty links per faulty processor we obtain at most $\pi \log N$ faulty links associated with these π faulty processors. In general we have that $\gamma \leq \lambda + \pi$ and it is easy to see that equality may not be true. Notice that our definition of λ suggests that in the complexity results we encounter in Section **??** we interpret as faulty all the links which are adjacent to a faulty processor.

Let Q_n denote the n-dimensional hypercube on $N = 2^n$ nodes. xy is a link of Q_n, where $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$, if $x_i \neq y_i$ for a unique i; in

addition, i is called the label of xy and we write $\ell(xy) = i$.* Let $Q_n[l_1, \ldots, l_\lambda]$ denote the hypercube Q_n with the links $l_1, \ldots, l_\lambda$ faulty. In general, the hypercube always remains a connected graph if the number λ of faulty links is less than $\log N$. However it is important to note that the hypercube may remain connected even if $\lambda \geq \log N$.

We define $\delta_n(\lambda)$ as the maximal possible diameter of a connected hypercube with at most λ faulty links, i.e.

$$\delta_n(\lambda) := \max\{diam(Q_n[l_1, \ldots, l_\rho]) : \rho \leq \lambda \text{ and } Q_n[l_1, \ldots, l_\rho] \text{ is connected}\}.$$

We define similarly $\delta_n(\gamma)$ for the more general case of hypercubes with at most γ faulty components.

1.4 Results of the paper

The following table summarizes previous results on computing Boolean functions on asynchronous, anonymous, labeled networks.

Network	Bit Cost	Paper
Rings	$O(N^2)$	[**?**]
n-Tori, n constant	$O(N^{1+1/n})$	[**?**]
Hypercubes: $\gamma = 0$	$O(N \log^4 N)$	[**?**]
Hypercubes: $\gamma \geq 1$	$O(N\lambda^2\delta_n(\gamma)^2 \log\log N)$	This paper

The result of [**?**] is valid both for oriented as well as unoriented rings. The result of [**?**] is valid for n-dimensional tori, where n is a constant (independent of the number of nodes). Moreover, the constant implicit in the bit cost bound $O(N^{1+1/n})$ depends on n [**?**]. Hence this result cannot apply to the hypercube which has variable dimension n. Bit cost bounds for non-faulty hypercubes are given in [**?**].

In this paper we give an algorithm for computing Boolean functions on asynchronous, anonymous oriented hypercubes having bit cost

$$O(N\lambda^2\delta_n(\gamma)^2 \log\log N).$$

*At this point it is necessary to emphasize that this global labeling is unknown to the processors. The algorithm to be discussed in the sequel uses only the fact that the processors know the labels of their adjacent links.

Here N is the number of nodes and $n = \log N$ is the dimension of the hypercube. Since a connected, n-dimensional hypercube with polylogarithmic (in N) number of faulty components has diameter $O(\log N)$ (see [**?**]) we have an $O(N polylog(N))$ bit cost for n-dimensional hypercubes with $1 \leq \gamma = polylog(N)$ faulty components.

Notice the different estimates on the bit cost implied by the algorithm for hypercubes with exactly one faulty link versus hypercubes with exactly one faulty processor; in the former case the bit cost is $O(N \log^2 N \log\log N)$ while in the latter $O(N \log^4 N \log\log N)$. At first glance it may also come as a surprise that the bit cost in a faulty hypercube can be lower than the bit cost in a non-faulty hypercube (e.g. this can be the case when there are no faulty processors and $\lambda < \log N/\sqrt{\log\log N}$). This however can be explained by the fact that in hypercubes with faulty links we can take advantage of asymmetries in the network topology in order to design algorithms with improved bit cost. Thus our main algorithm takes advantage of "symmetry breaking" by distinguishing faulty links from non-faulty links.

2 Hypercubes with Non-faulty Processors

In this section we give algorithms for computing Boolean functions on a hypercube which does not have any faulty processors, i.e. $\pi = 0$. We indicate later how to extend our results to hypercubes with arbitrary faulty components. Our main theorem is the following.

THEOREM **1** *In a hypercube with at most λ faulty links, $\lambda \geq 1$, every computable Boolean function may be computed in $O(N\lambda^2\delta_n(\lambda)^2 \log\log N)$ bits.*

PROOF. The proof of the theorem is carried out in Subsections **??** and **??**. Before giving a detailed account of the algorithm achieving the desired complexity we present a summary of the main steps of our construction.

Let f be a given Boolean function. Each processor p is given an input bit b_p and the Boolean function f. Let $Input =< b_p : p \in Q_n >$. In outline, and under the assumptions of Subsection **??** each processor p concerned executes the following algorithm:

1. determines whether or not the hypercube has a faulty link,

2. uses a "path-generation" algorithm in order to determine the location of the faulty links relative to itself,

3. uses an input collection mechanism in order to determine the entire input configuration $Input_p$, where $Input_p$ denotes p's view of $Input$, (in executing the algorithm, the processors collect input bits in a manner specified by the protocol thus forming the view $Input_p$ associated with processor p),

4. determines whether or not the given function is computable on the given input (this step is actually performed only locally and hence does not contribute to the overall bit cost) by checking an invariance condition on the given function f,

5. if f is computable then processor p outputs $f(Input_p)$.

Details of the proof will be given in the sequel. ■

2.1 Determining if there are any faulty links

The first step in our algorithm is to determine whether or not the hypercube has any faulty links. This follows from the following lemma.

LEMMA **2** *There is an algorithm with bit cost $O(N \log^2 N)$ which detects whether or not the hypercube has any faulty links.*

PROOF. Let us use the abbreviation **0** = "I have no faulty links" and let **1** = "I have a faulty link". Each processor initializes the variable *value* locally. To determine whether there is a faulty link the processors execute an algorithm for computing the Boolean function OR_N by using the Boolean constants $\mathbf{0}, \mathbf{1}$ previously defined. If the output is **1** then there is a faulty link else there is no faulty link. The algorithm they execute is as follows.

Faultylink
Algorithm for processor p:
Initialize: $value_p$;
for $i := 1, \ldots, \log N$ **do**
    send $value_p$ to all neighbors of p;
    receive $value_q$ from all neighbors q of p;

compute $value_p := \text{OR}(\{value_q : q \text{ is neighbor of } p\}) \vee value_p$;
od;
output $value_p$.

There are $\log N$ iterations of the **for** loop and in each iteration at most $\log N$ bits are transmitted by each processor. Hence the bit cost of the algorithm is $O(N \cdot \log^2 N)$.

It remains to prove the correctness of the algorithm. We show that if there is a faulty link then every processor of the hypercube is at distance at most $-1 + \log N$ from some faulty link. Indeed, let x be an arbitrary node. We show that there is a node y adjacent to a faulty link and such that x is at distance at most $-1 + \log N$ from y. Let y be any node which is adjacent to a faulty link and at a minimal distance to x in the non-faulty hypercube. There is a path $x_0 = x, x_1, \ldots, x_d = y$ of length $d \leq \log N$ connecting x to y in the non-faulty hypercube. Because all nodes of the path, but y, are closer to x than y, it follows from the choice of y that they cannot be adjacent to a faulty link. Since y is adjacent to a faulty link it is clear that x is at distance at most $-1 + \log N$ from a faulty link. This completes the proof of the lemma. ■

If it turns out there is no faulty link then (assuming that the given Boolean function is computable in the network) the processors execute the algorithm of [**?**] which has bit cost $O(N \log^4 N)$. Else they proceed to the next phase of our algorithm.

2.2 Path generation and input collection

The algorithm to be presented in this subsection requires the existence of faulty links. Therefore this phase is executed only if it turns out from the execution of the algorithm in Subsection **??** that $\lambda \geq 1$. Let f be a Boolean function known to all processors of the (faulty) hypercube. We present the algorithm in three steps. The processors execute the following algorithm.

Main Algorithm ($\lambda \geq 1$):

1. PATH-GENERATION:

The processors adjacent to faulty links become leaders and compute the configuration of the hypercube as follows. Let M be the set of faulty links. Let L be a processor adjacent to a faulty link. For each $x \in Q_n$ there are many paths connecting L to x. However L can choose a set of paths (in a canonical

way) $\{p(L,x) : x \in Q_n\}$ such that $p(L,x)$ connects L to x, has length at most $\delta_n(\lambda)$ and avoids the missing link(s). Each processor adjacent to a faulty link generates a set of paths, one path for each processor of the hypercube. In generating paths the processor takes into account its current knowledge of the position of the set of faulty links (which is only a subset of the set of all faulty links). Each such path is transmitted to its destination node along the sequence of links determined by this path. If during transmission of this path a faulty link is encountered then the corresponding processor adjacent to this faulty link sends back (along this same path but in the reverse direction) to the originating processor a complete list of its missing links. Based on this information each processor adjacent to a link in M updates its current list of faulty links and generates a new set of paths which avoid the previously encountered faulty links. Now iteration of this procedure continues as long as new faulty links are found. (Notice that nowhere in this algorithm do the processors need to know an upper bound on the number of faulty links. The iterated procedure terminates execution when no new faulty links are found.) After execution of this algorithm all processors receive a complete path from each processor adjacent to a link in M.

Since each iteration of this algorithm generates a new collection of paths by "eliminating" newly encountered faulty links and since there are at most λ faulty links it is clear that after at most λ iterations all processors will receive paths from all processors adjacent to processors with faulty links. The bit cost of this algorithm depends on the length of the paths which are created during the execution of the λ iterations of this algorithm (in this instance the paths have maximal possible length $\delta_n(\lambda)$) and can be computed as before. There are $\leq 2\lambda$ processors adjacent to the λ faulty links. Paths can be coded with $\delta_n(\lambda) \log\log N$ bits (all that is needed is the sequence of labels traversed by the path). Each path is transmitted at a distance $\leq \delta_n(\lambda)$. Each iteration of the algorithm involves $\leq 2\lambda$ processors adjacent to a faulty link in M. Hence each iteration of the algorithm involves the transmission of at most $O(N\lambda\delta_n(\lambda)^2 \log\log N)$ bits. Since the number of iterations is $\leq \lambda$ the actual bit cost of this step will be $O(N\lambda^2\delta_n(\lambda)^2 \log\log N)$ bits.

2. Input-collection:

For each x, and each L adjacent to a link in M, processor x sends its input bit b_x together with its "identity" $p(L,x)$ to L in the reverse direction along path $p(L,x)$ ($p(L,x)$ is the path computed in step 1). Now L has a view of the entire input configuration of the hypercube, say I_L, and can compute

$f(I_L)$. The bit cost of this step is $O(N\lambda\delta_n(\lambda)\log\log N)$.
3. Computing the Output:
Let F be the set of processors which are adjacent to faulty links. By executing the above algorithm each processor $L \in F$ computes its "view" I_L of the given input configuration. In particular, each $L \in F$ will know the view $I_{L'}$ of all processors $L' \in F$. Hence all processors $L \in F$ may execute the invariance test

$$f(I_L) = f(I_{L'}), \text{ for all } L, L' \in F. \tag{1}$$

If (**??**) is true each processor $L \in F$ computes $f(I_L)$ and transmits it to all processors of the hypercube along the paths previously specified. Finally, $f(I_L)$ is the output bit of each processor of the hypercube. If on the other hand (**??**) is false then the processors $L \in F$ will transmit to all processors of the hypercube that f is not computable on the given input. Clearly, test (**??**) is local to the processors and does not contribute to the overall bit cost of the algorithm. The bit cost of this step is $O(N\lambda\delta_n(\lambda)\log\log N)$.

Notice that nowhere in this algorithm did we have to assume that the processors have identities. All identities used there were generated by the algorithm and were relative to a particular leader. In addition the processors execute identical algorithms given identical input data. This completes the proof of Theorem **??**. ∎

An interesting observation concerns the size of the input data of a processor. In computing Boolean functions the input to a processor was assumed to be a bit. However, if the size of the input data of a processor is $\leq s$ bits then the contribution to the overall bit cost of the input collection step is at most $O(Ns\lambda\delta_n(\lambda)\log\log N)$. In particular, the bit cost stated in Theorem **??** remains valid, even if the size of the input data is up to $O(\lambda\delta_n(\lambda)$ bits.

2.3 Estimates depending on the number of faults

Theorem **??** raises the problem of studying $\delta_n(\lambda)$ as a function of λ. Results of B. Aiello and T. Leighton in [**?**] show that an n-dimensional hypercube with $n^{O(1)}$ worst-case faults can simulate the fault-free n-dimensional hypercube Q_n with only constant slowdown. In particular, this implies that $\delta_n(\lambda) = O(n)$, for $\lambda = n^{O(1)}$. As a consequence we obtain the following result for hypercubes with polylogarithmic number of faulty links.

THEOREM **3** *The bit cost of computing Boolean functions on a hypercube with polylogarithmic number of faulty links (i.e.* $\lambda = (\log N)^{O(1)}$*) is*

$$\begin{cases} O(N \log^4 N) & \text{if } \lambda = 0 \\ O(N\lambda^2 \log^2 N \log\log N) & \text{if } \lambda > 0. \end{cases}$$

PROOF. If $\lambda = 0$ then by [**?**] the bit cost of computing the function f is $O(N \log^4 N)$. If $\lambda \geq 1$ then applying Theorem **??** we see that the bit cost of computing a Boolean function is $O(N\lambda^2 \delta_n(\lambda)^2 \log\log N)$. Since the number of faulty links is $n^{O(1)}$ we have that $\delta_n(\lambda) = O(n)$. Hence the combined bit cost is

$$O(N\lambda^2 \log^2 N \log\log N),$$

as desired. ■

Thus we see that $\log N/\sqrt{\log\log N}$ is the threshold number of faulty links for which the bit cost of computing Boolean functions on an N node hypercube using our algorithm exceeds the bit cost of the algorithm in [**?**] for a non-faulty hypercube.

3 Hypercubes with Faulty Components

So far we have considered the case of hypercubes having only faulty links. However, it is straightforward how to adapt the Path-generation and Input-collection algorithms presented in Section **??** to the case of hypercubes whose faulty components may be links and/or nodes. If a node is faulty then all its adjacent links are interpreted as faulty. The Path-generation algorithm is initiated by non-faulty processors which are adjacent to faulty links (there are $\leq 2\lambda$ such processors) and the iterated procedure is repeated $\leq \lambda$ times. Thus we can prove the following theorem.

THEOREM **4** *In a hypercube with* γ *faulty components exactly* λ *of which are faulty links,* $\lambda \geq 1$*, the bit cost of computing Boolean functions is*

$$O(N\delta_n(\gamma)^2\lambda^2 \log\log N). \blacksquare$$

4 Conclusion and Further Research

We have presented algorithms for distributed computation on oriented anonymous asynchronous hypercubes with faulty components. Our algorithms rely on the possibility of distinguishing faulty links from non-faulty ones and are based on broadcasting and path generation.

Looking at tradeoffs and algorithms with improved performance is an interesting problem for further research. As pointed out by an anonymous referee, potential improvements to the bit cost of computing Boolean functions may be achieved by using sequential probing techniques that reduce the total number of probes used in the algorithm presented in Subsection **??** from λN to N. However, the analysis of the number of steps needed is a more complicated combinatorial problem which is worthy of future investigations.

In our present analysis, the hypercubes may be faulty but the faults can occur only before the start of the computation. An interesting problem would be to design more "adaptive" algorithms that allow for faults to occur at different parts of the computation. In addition, very little is known on the optimality of the algorithms presented.

Acknowledgments

Many thanks to Danny Krizanc and Hisao Tamaki for useful advice and the anonymous referees for useful comments that provided significant improvements to the overall structure of the paper.

References

[1] B. Aiello and T. Leighton, "Coding Theory, Hypercube Embedding and Fault Tolerance", Proceedings of 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, 1991, 125 - 136.

[2] D. Angluin, "Local and Global Properties in Networks of Processors", 12th Annual ACM Symposium on Theory of Computing, 1980, 82 - 93.

[3] H. Attiya and M. Snir and M. Warmuth, "Computing on an Anonymous Ring", Journal of the ACM, 35 (4), 1988. (Short version has

appeared in proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computation, 1985, 845 - 875.)

[4] P. W. Beame and H. L. Bodlaender, "Distributed Computing on Transitive Networks: The Torus", 6th Annual Symposium on Theoretical Aspects of Computer Science, STACS, 1989, B. Monien and R. Cori, editors, Springer Verlag Lecture Notes in Computer Science. 294-303.

[5] B. Becker and H.-U. Simon, "How Robust is the n-Cube?", Information and Computation, 77(2), pp. 162-178, May 1988. (Also in proceedings of IEEE 27th Annual Symposium on Foundations of Computer Science, 1986, 283 - 291.)

[6] M-S. Chen and K. G. Shin, "Adaptive Fault-Tolerant Routing in Hypercube Multicomputers", IEEE Transactions on Computers, 39 (12), December 1990, 1406 - 1416.

[7] J. Hastad and T. Leighton and M. Newmann, "Reconfiguring a Hypercube in the Presence of Faults", Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, 274 - 284.

[8] E. Kranakis and D. Krizanc, "Distributed Computing on Anonymous Hypercube Networks", Journal of Algorithms, 23, 32-50, 1997. (Also in proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing, Dallas, Dec. 2-5, 1991, 722 - 729.)

[9] H. Wielandt, "Finite Permutation Groups", Academic Press, 1964.