7-1990

Load Redistribution on Hypercubes in the Presence of Faults

Sanjay Ranka
Syracuse University

Jhy-Chun Wang
Syracuse University, School of Computer and Information Science

SU-CIS-90-23

Load Redistribution on Hypercubes in the Presence of Faults

Sanjay Ranka and Jhychun Wang

July 1990

School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, NY 13244-4100

(315) 443-2368

Load Redistribution on Hypercubes in the Presence of Faults

Sanjay Ranka and Jhychun Wang
Syracuse University

July 31, 1990

Abstract

In this paper, we present load redistribution algorithms for hypercubes in the presence of faults. Our algorithms complete in low-order polynomial of the number of faulty nodes and exhibit excellent experimental performance. These algorithms are topology independent and can be applied to a wide variety of networks.

Keywords and Phrases

max-flow, fault tolerant, hypercube, reconfiguration.

1 Introduction

Massively parallel systems are becoming popular for addressing the processing needs of computationally intensive areas like image processing, computer vision, computational geometry, robotics, VLSI *etc.* However with the increase the number of processors, the probability of at least one processor being faulty increases dramatically. Most algorithms are designed assuming no faults are present. In this paper we present strategies to reconfigure the algorithm (via software) in presence of faults.

Conventional fault tolerant schemes concentrate on the hardware approach. Spare processors and communication linkages are embedded in the architecture. Whenever a faulty node is detected and isolated, spare processors take over faulty node's computation role in order to decrease the impact of the faulty node in the whole system. In these approaches, extra cost is associated with the spare hardware.

If we assume that no spare processors are available and no hardware modifications are allowed, a software approach must be employed to achieve fault tolerance. Dutt and Hayes [DUTT88] have proposed a software approach under such a constraint. Their strategy is to identify the maximum size of fault-free subcube in the hypercube and rearrange computation on that smaller subcube. However, this might result in a large performance degradation with a lot of wasted resources. In the presence of even one fault the performance will degrade by up to 50%.

Under the assumption that the original workload is equally balanced among all processors, if the whole workload of the faulty processor is distributed to one of its neighbors, the workload on the neighboring processor will be double. For most hypercube applications that are synchronous, the other processors would have to wait for the slowest processor to finish its computations. Thus, the overall system performance will be degraded by 50%. Clearly it would be desirable to split up the workload of the faulty nodes into several parts and allocate each part to a set of fault-free processors. Our algorithm is based on the notion of multiple virtual processes on a single physical processor and performing load balancing using these virtual processes. Hence, the workload of every node can be distributed among many processors. The reconfiguration algorithm is centralized and static in nature and

requires the knowledge of all the faulty nodes in the system. After distribution of faulty nodes' workload to other processors, there is a nominal computational overhead on each node. This causes small performance degradation.

A similar reconfiguration strategy was presented by Banerjee [BANE89]. Assuming a $N = 2^p$ system with f faulty nodes, each faulty node's workload is reallocated to n of fault-free nodes and each fault-free node only allowed to receive extra workload from two of faulty nodes. This algorithm uses an integer programming approach to find an optimal solution. The time complexity of the algorithm is $O(2^{f \cdot (N-f)})$. It is computationally infeasible even for small values of f (say 2) and $N = 64$.

In this paper, we present an efficient algorithm to dispatch faulty nodes' workload to other fault free nodes. Let n be the number of nodes, f be the number of faults and w be the fraction of extra workload that can be assigned to each processor. Our experimental results suggest that our algorithm completes in $O(\frac{f \log n}{w})$ amount of time. These results are applicable for large value of n (512 nodes and higher), practical value of w (2% to 20%), and reasonable value of f (up to 10% faulty nodes). Though we present our result only for hypercubes, our algorithm is general and can be applied to a wide variety of interconnection networks.

The rest of this paper is organized as following. In section 2, we present the hypercube architecture and max flow algorithm. The max flow algorithm forms a backbone for our later algorithms. In section 3, we describe the reconfiguration algorithm. In section 4, we analyze the time complexity of our algorithm. In section 5, we present simulation results and compare them with the theoretical complexity.

2 Preliminaries

2.1 Hypercube Multicomputer

The topology of a 16-node hypercube interconnection network is shown in Figure 1. A p-dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}\ldots i_0$ be the binary representation of the PE index i. Let $\bar{i}_k$ be the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit;

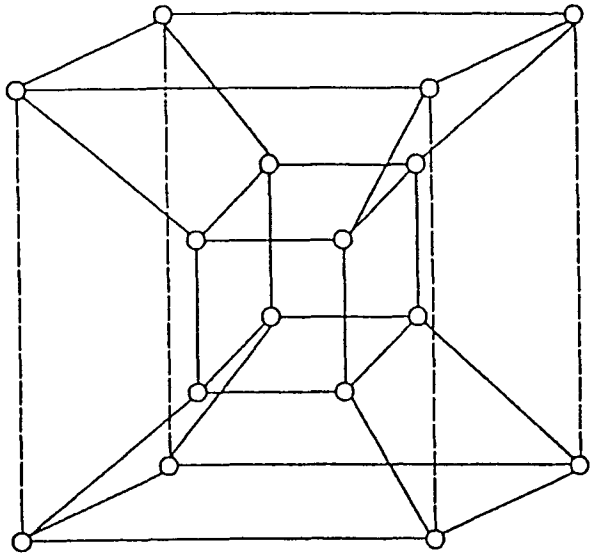


Figure 1: A 16-Node Hypercube (Dimension =4)

i.e., processor $i_{p-1}i_{p-2}\ldots i_0$ is connected to processors $i_{p-1}\ldots\bar{i}_k\ldots i_0, 0 \leq k \leq p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

2.2 Max-flow Algorithm

The maximum flow problem involves a commodity network graph. Figure 2 shows the graph of such a network. There is a source node (S) and a sink node (T). There are several interior nodes linked by weighted branches. Source node represents a production center that is theoretically capable of producing an infinite amount of commodity. Sink node represents a demand center which can absorb an infinite amount of commodity. The branches represent commodity transport linkages, with the weight of a branch indicating the capacity of the corresponding link.

A commodity flow in this network is represented by weighted directed arrows along the branches of the network, with the weight of the arrow indicating the amount of the flow on that branch and the direction of the arrow indicating the direction of

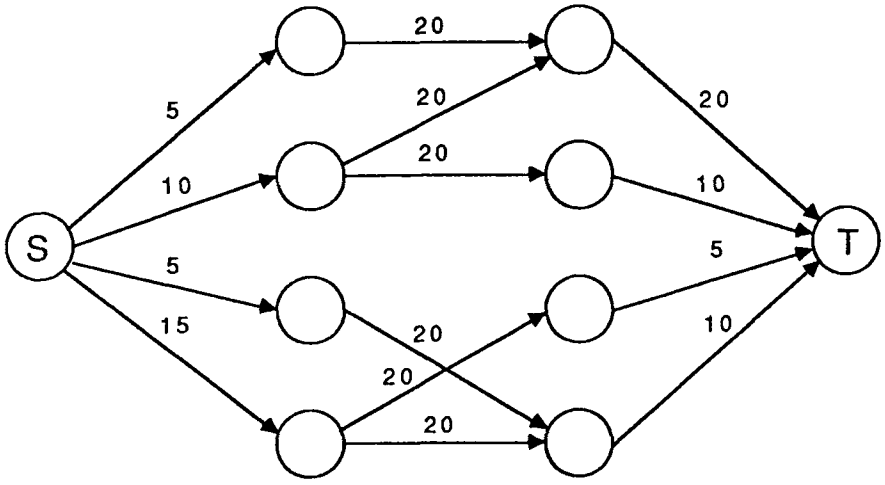


Figure 2: Original Commodity Flow Network

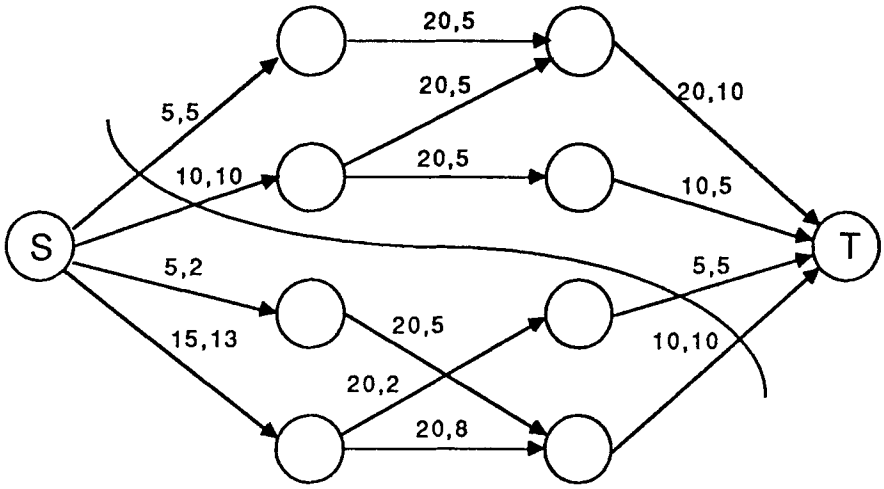


Figure 3: Feasible Commodity Flow Network

the commodity flow. Figure 3 shows a commodity flow for the graph in Figure 2. Each arrow in Figure 3 carries a pair of numbers, the first of which is the capacity of that branch and the second of which is the actual flow on that branch. A feasible commodity flow in this network is a commodity flow originating from the source node and ending at the sink node such that: 1) at each intermediate node, the sum of the flows into the node is equal to the sum of the flows out of the node; 2) at each sink node the net flow into the node is nonnegative, and at the source node the net flow directed out of the node is nonnegative; and 3) the net flow in any branch in the network does not exceed the capacity of that branch. Note that the flow in Figure 3 is a feasible flow according to the above definition.

The value of a commodity flow is the sum of the net flows out of the source node of the network and it is equal to the sum of the net flows into the sink node. A maximum flow is a feasible flow whose value is maximum among all feasible flows.

The maximum flow in commodity network is related to a cutset of the network. A cutset of a commodity network is a set of edges which when removed disconnects the source nodes from the sink nodes. The weight of a cutset is equal to the sum of the capacity of the branches in the cutset. The following theorem categorizes the relationship between max-flow and min-cut.

Max-flow Min-cut Theorem[FORD62]: The value of a maximum flow in a commodity network is equal to the weight of a minimum weight cutset of that network.

3 Reconfiguration Algorithm

Our algorithm is designed to reconfigure the task allocation when faulty node(s) occur in a hypercube multiprocessor system. We assume that the task graph has initially been mapped into system graph by the algorithm presented by [KERN70], [SADA88] or [STON77]. It is clear that every mapping strategy will try to keep each processor's workload balanced and minimize the system's interprocessor communication cost. When a processor fails, its workload must be dispatched to other processors. According to our assumption, the workload of every node can be distributed among many

processors. Distributed memory machines assume that only processors that are connected by links can communicate directly with each other. Processors without direct links communicate via intermediate processors using store-and-forward techniques. Thus, the cost of sending a message from one processor to another is proportional to the product of the size of the message and number of links the message has to travel. The following are the parameters for the time spent in communication from one node to the other.

ξ is the start up time,

τ is the cost of setting up a circuit between two adjacent processors,

i is number of links traveled,

k is length of the message.

The time needed to send a message from a node to another node is $\xi + i \cdot (\tau + k)$. Recently, novel techniques have been proposed for distributed memory machines with circuit switched communication [BOKH90], [DALL87], [DECE89]. To send a message in these architectures, first a physical path between source and destination is established and then the complete message is transmitted. Thus the message transmission time is linearly proportional on the number of links traveled and the size of the message. The time required to send a message from one node to another node (via a path A of length i) is $\xi + i \cdot \tau + i + k - 1$. For the case of Intel iPSC-860, a circuit switched hypercube, this equation is true only if no other path is sharing any edges with path A. In case of congestion (*i.e.* other path are sharing edges with path A) the communication time may increase considerably [BOKH90].

Thus we assume that task pairs with heavy communication were allocated to same processor or near neighbors to minimize on the total interprocessor communication time. This kind of mapping is preferable even in models with circuit switched communication. In order to keep this property after reconfiguration, we will reallocate faulty nodes' tasks to their neighbors (of a faulty node) as close as possible. In case the adjacent neighbors can't absorb the whole workload, we will look for their 2-distance neighbors and so on until we complete the reallocation.

Many algorithms are designed with perfect hypercubes as the underlying architecture. In the presence of faults, these algorithms have to be reconfigured. During

the reconfiguration process, how far each task being moved to other node has a major effect on communication time (and hence the total execution time). In order to measure the efficiency of our reconfiguration algorithm, we define a performance variable, average distance D, which is used to represent the average passes we need to complete the reconfiguration. This variable D is calculated by the following expression:

$$D = (w_1 * 1 + w_2 * 2 + w_3 * 3 + \cdots + w_p * p)/W$$

where w_i represents the total workload being dispatched to nodes at a distance i from the original node and W represents the total workload of faulty nodes. The basic principle in our algorithm is to try to minimize D such that the load is balanced among the nodes.

In order to use the maximum flow algorithm in our problem, we map the system graph into the commodity network by following steps:

1. Separate the processes into two subsets, U and V, representing the faulty nodes and fault-free nodes, respectively.

2. Add nodes labeled S and T that represent unique source node and unique sink node, respectively.

3. For each node in U, add a branch from that node to S. The weight of the branch carries the workload of that node.

4. From each node in V, add a branch from that node to T. The weight of the branch carries the maximum extra workload can be accepted by that node.

5. From each pair of nodes (u, v), add a branch between them if there is a path in processors graph. The weight of the branch must be much larger than the weight assigned in step 3 and 4 in order to prevent it from being part of min-cut set. We set this weight to MAX which is larger than the total weight assigned in step 3 and 4.

After constructing the commodity network, we use max-flow min-cut theorem to decide the min-cut set. There are three possible ways which the min-cut set may present:

1. In Figure 4, the min-cut set goes through branches which connect source node S and nodes u_i. The weight carried by these branches represent the faulty nodes' workload, which in turn equal to the maximum flow in this network. According to Ford-Fulkerson algorithm we can carry out the actual flow in each branch, then calculate the extra workload accepted by each fault-free node.

2. In Figure 5, the min-cut set goes through the branches between nodes v_j and sink node T. The weight carried by these branches represent the maximum extra workload can be accepted by fault-free nodes. In this case, the maximum flow in the network will be equal to the sum of the maximum extra workload received by fault-free nodes.

3. In Figure 6, the min-cut set goes through part of branches $< S, u_i >$ and $< v_j, T >$. In this case, these nodes u_i in the cutset can transfer all of their workload to fault-free nodes and these nodes v_j in the cutset will receive the maximum amount of extra workload which they are allowed to accepted.

The following defines the variables used in the reconfiguration algorithm.
U: represent the set of faulty nodes.
V: represent the set of fault-free nodes.
$workload[u_i]$: represent the amount of workload at node u_i.
$capacity[e_{ij}]$: represent the maximum capacity of branch e_{ij}.
$weight[e_{ij}]$: represent the amount of actual flow in branch e_{ij}.
$extraw[v_j]$: represent the extra workload received by node v_j.
K: represent the value of maximum extra workload can be accepted by each fault-free node.
$DIST$: represent the distance between nodes in U and V.
MAX: value which is much larger than K and $workload[u_i]$.

Our reconfiguration algorithm assumes that the load at every node is initially the same (*i.e.* it is load balanced). Hence the maximum load accepted by any fault free node is the same. This algorithm can be easily transformed to be applicable to case when the nodes are not exactly balances and value of K is different for different

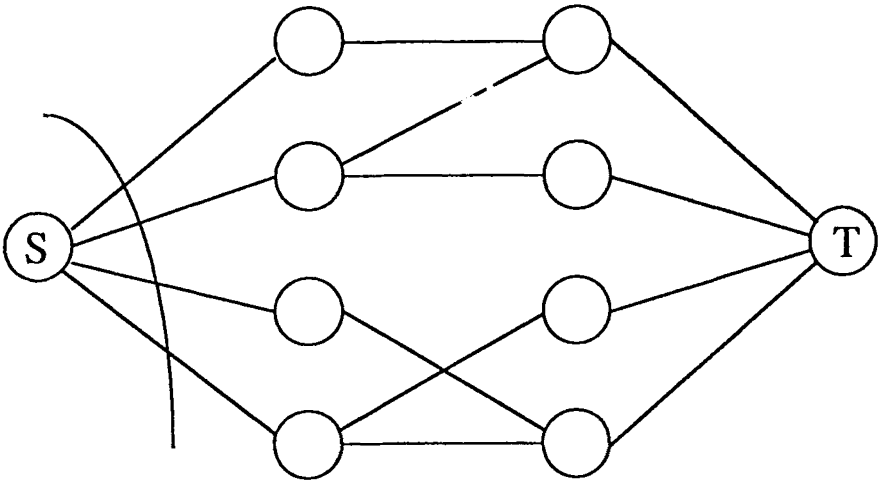


Figure 4: Min-cut set (1)

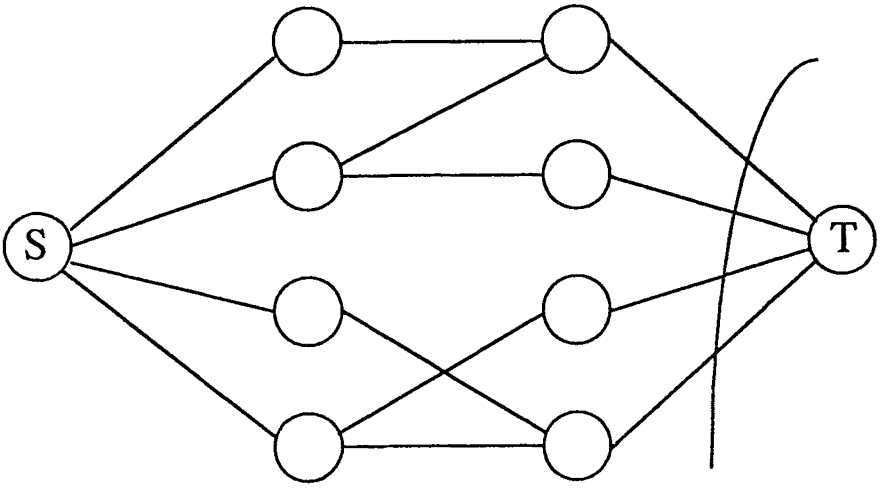


Figure 5: Min-cut set (2)

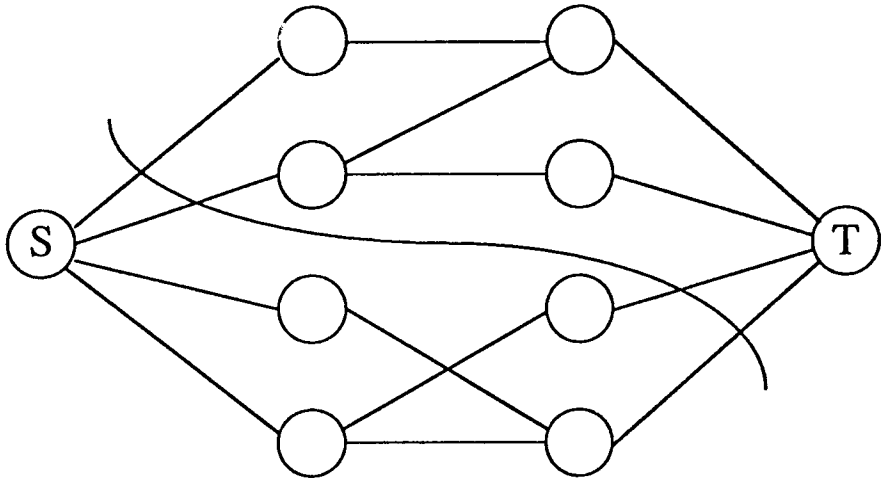


Figure 6: Min-cut set (3)

nodes. A high level description of the reconfiguration algorithm is given in Figure 7.

In this algorithm, value K will dominate the execution time of the reconfiguration algorithm. If K is large enough, we can always complete the reconfiguration job in the first pass ($DIST = 1$). Thus the workload will be dispatched to faulty nodes' adjacent nodes. However, the new workload balance may be poor. When the value K goes down, more steps are needed and the reconfiguration execution time will grow up.

The value of K also determines the load balancing performed by the reconfiguration algorithm. If we assume that the initial configuration is perfectly load balanced with a load of L on each node. The *maximum load / minimum load* after the reconfiguration is

$$\frac{K+L}{L}(1-f) = (1+\frac{K}{L})(1-f)$$

where f represents the fraction of faulty nodes. Thus smaller value of K/L will lead to better load balancing. However $DIST$ will be higher and hence communication

Reconfiguration Algorithm:

1. $DIST \leftarrow 1$;
2. for each node u_i in U, create a branch from u_i to node v_j in V which is $DIST$-distance away from node u_i in processors graph., create a branch from u_i.
3. for each branch e_{ij} created in step 2, $capacity[e_{ij}] \leftarrow MAX$.
4. add two nodes S and T. S is the unique source node and T is the unique sink node.
5. for each node u_i in U, add a branch from u_i to S. The weight of this branch is equal to $workload[u_i]$.
6. for each node v_j in V, add a branch from v_j to T. The weight of this branch is equal to K - $extraw[v_j]$.
7. use Max-flow Min-cut algorithm to calculate the values of $weight[e_{ij}]$.
8. for each i, reduces $weight[e_{ij}]$s' from $workload[u_i]$. If $workload[u_i]$ is equal to 0, remove u_i from set U.
9. for each j, add $weight[e_{ij}]$s' to $extraw[v_j]$.
10. If $U = \{\}$, then EXIT. The values $extraw[v_j]$ in each fault-free node is the amount of extra workload received by that node.
11. If $U \neq \{\}$, then $DIST \leftarrow DIST + 1$; goto step 2.

Figure 7: Reconfiguration algorithm

time may increase and may lead to higher execution time. Thus the value of K should be chosen appropriately.

4 Theoretical time complexity

The most expensive step in our reconfiguration algorithm is step 7. In this paper, we employ the max-flow algorithm developed by Malhotra *et al.* [MALH78]. Several other algorithms for the max-flow problem are available [SLEA80] [TARJ86]. However we decided to choose the MPM algorithm due to ease of implementation and good performance. The time complexity of MPM algorithm is $O(n^3)$, where $n = |V|$. Our processors graph is a p-cube, Let n_u and n_v present the number of faulty nodes and the number of fault free nodes at a distance of d respectively. The time complexity for $d = DIST = 1$ is analyzed as follows.

$$\begin{aligned}
\text{number of vertices } n &= |S| + |U| + |V| + |T| \\
&= 1 + n_u + n_v + 1, \text{ where } n_v \leq n_u * \binom{p}{d} \\
&\leq 2 + n_u + n_u * \binom{p}{d} \\
&= 2 + n_u * (1 + p)
\end{aligned}$$

The time complexity of the first pass of the algorithm is $O(n_u^3 p^3)$. If we apply the $O(nm \log n)$ algorithm of [SLEA80], the time complexity will be $O(n_u^2 p^2 \log(n_u p))$.

The complexity of the k^{th} pass ($2 \leq k \leq \lfloor \frac{p}{2} \rfloor$, we will assume this restriction for our analysis) is $O(n_u^3(1 + n_k)^3)$ using the MPM algorithm, and $O(n_u^2(1 + n_k)^2 \log(n_u(1 + n_k)))$ using the algorithm by Sleator and Tarjan, where $n_k = \frac{n_v}{n_u} \leq \binom{p}{d}$. Better bounds on n_k can be achieved, but is beyond the discussion of this paper.

The number of passes can be approximated by the following:

$$\text{Smallest } k \text{ such that } \sum_{i=1}^{k} \binom{p}{i} \geq \frac{1}{w}$$

where w is the fraction of maximum permissible load. This inequality assumes that the workload of every faulty node can be distributed independently of the other faulty nodes. Further the number of faults are small compared to the total number of nodes.

Thus assuming that $w \leq \frac{1}{p}$, one pass will be sufficient most of the times. If we assume that $w \leq \frac{1}{p^2}$, two passes will be sufficient most of the times. The worst complexity of the Malhotra algorithm for the first two passes are $O(n_u^3 p^3)$ and $O(n_u^3 p^6)$, respectively.

If the reconfiguration algorithm goes through k passes, then the worst case time complexity can be represented as:

$$O(n_u^3 p^3) + O(n_u^3 p^6) + \cdots = \sum_{i=1}^{k} O(n_u^3 (p^i)^3)$$

According to the above analysis, the reconfiguration algorithm may see unpractical. In the next section we demonstrate that our algorithm performs extremely well in practice.

5 Experimental Results

In the case the number of faulty nodes are much less than the total number of nodes, the neighbors of the faulty nodes will be disjoint for most cases. Thus the max-flow algorithm will be applied (in the first pass) to a graph similar to the one shown in Figure 8. In this case the application of max-flow algorithm to the whole graph is similar to the application of max-flow algorithm to each subgraph. For each subgraph $|U| = 1$ and $|V| = \log n$ (for the first pass). In the worst case, time complexity for the max-flow algorithm for such a subgraph is $O(\log^3 n)$. However, we expect it to complete in $O(\log n)$ time in most cases. Thus the expected complexity of the first pass (in practical cases) is $\approx O(f \log n)$.

The assumption that the graph will be decomposed for the later passes may not hold true. Further, many of the source nodes (faulty nodes) get deleted in every pass. Thus it is hard to predict the complexity of future passes. However, our experimental

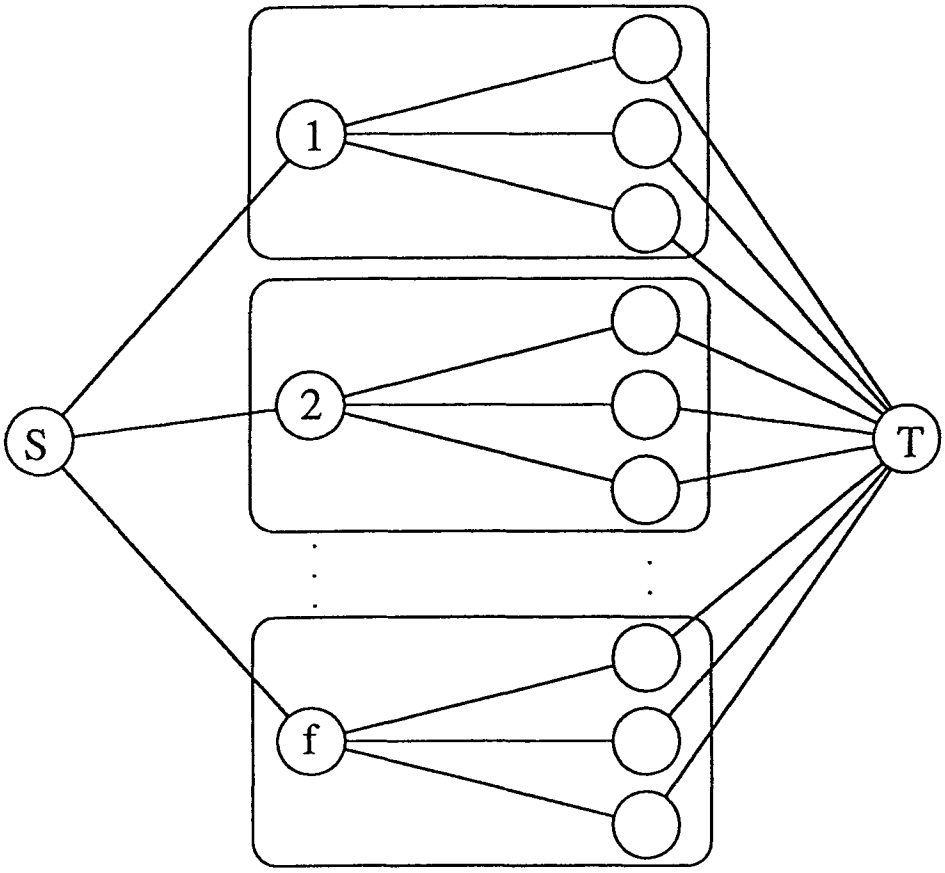


Figure 8: Graph configuration for the max-flow algorithm for small number of faults (as compared to the total number of nodes).

results suggest that for practical values of n, w, and f, the number of passes are close to 1. Thus the complexity in these cases will be determined by the first pass.

There is one key factor which will affect the execution time: the extra load K that each fault free node can accept. If we set a small value for K, we will need more nonfaulty nodes to absorb whole faulty nodes' workload. This will cause the reconfiguration process to go through several passes before it completes, and the time required by the algorithm will increase substantially. On the other hand, if we select a large value K, we may be able to complete the redispatching process at first pass, so the execution time will be much less than the previous one. We present our experimental results in Table 1 to 5. They represent average distance D and execution time T under different number of faulty nodes and different values of extra load K. For each case, we took 50 random instances of the problem. Figure 9 and 10 give the values of D, Figure 11 and 12 give the values of T.

While achieving a K/L within 5%, the average distance the load at a particular node was moved to a distance of 1.5 to 1.6 for a hypercube of size 2^{10} (diameter = 10). The same amount was reduced to 1.3 and 1.03 for balance fraction with 10% and 20%, respectively. Further this distance (for the same number of faulty nodes) seems to decrease as the number of nodes in graph increases (Figure 9 and 10). Thus with larger number of nodes and the same number of faulty nodes, the average distance moved is going to be quite small. We also note that a 5% increase in the load of every node represent that final load balancing is within $1.05 \cdot (1 - f)$, where f is the fraction of faulty nodes. Thus a 5% increase in load represents a load balancing within 2% in case of 32 faulty nodes in 1024 nodes system.

From Figure 11 and 12, we claim that for a fixed number of nodes and extra load, the algorithm has a execution time linear or near linear in the number of faults. The rest of analysis is based most on our experimental results for $n = 512$ and $n = 1024$. For a fixed percentage of faulty nodes and extra load, the algorithm seems to behave asymptotically along $O(n \log n)$. This is shown in Figure 13 to 16. Thus the complexity of our algorithm is proportional to $f \log n$ for a fixed reasonable value of K.

By Figure 17 and 18, we conclude that for a fixed value of $f \log n$ the algorithm

completes in time proportional to $\frac{1}{w}$. Thus we conjecture that our algorithm performs in $O(\frac{f \log n}{w})$ for large value of n, practical values of w (2% to 20%) and reasonable value of f (up to 10% faulty nodes).

6 Conclusion

In this paper, we presented a new load redistribution algorithm on hypercube architecture. The experimental complexity of our algorithm is $O(\frac{f \log n}{w})$. Although we use hypercube as our system architecture, we can use this algorithm to solve redistribution problem in any system architecture without much modification. One needs to construct the commodity flow network G by faulty nodes and their i-distance neighbors ($i = 1, 2, \ldots$), then calculate the actual flow in this graph, continuing this process until the redispatching is completed. However, the time required by the reconfiguration algorithm will depend on the architecture.

Our experimental results are based on the max-flow algorithm of Malhotra. Tarjan has developed a more efficient algorithm [TARJ86] which presented a $O(nm \log(n^2/m))$ time complexity on an n-vertex m-edge graph. Implementing Tarjan's approach in our reconfiguration algorithm will potentially improve the total execution time.

Software reconfiguration strategy does have some potential disadvantages. The strategy assumes division of workload (and hence virtual processes). Thus to get an improved load balancing the granularity of the problem should be small. This will lead to increased cost of context switching. Further the complexity of communication routines become more complex as the number of virtual processes increases. We have assumed that the max-flow algorithm always give integer solutions. In case the flows are not integers, solutions can always be truncated to integers. This may lead to more passes as the flow from faulty nodes to other nodes in a particular pass may not be maximized.

It may so happen that the topology may get disconnected due to the presence of faulty nodes. These checks need to be performed and appropriate actions need to be taken. We have not addressed this problem in this paper. The algorithm discussed

in this paper is centralized. We are currently investigating parallel and distributed approaches to solve this problem.

n_u	1		2		4		8	
K	*dist*	*time*[1]	*dist*	*time*	*dist*	*time*	*dist*	*time*
2	2.64	31	-	-	-	-	-	-
5	1.70	12	1.92	62	-	-	-	-
10	1.40	9	1.42	19	1.48	55	-	-
15	1.10	8	1.16	38	1.23	12	1.38	79
20	1.00	4	1.00	17	1.04	9	1.12	60

Table 1: Simulation results for 64 nodes system

n_u	1		2		4		8	
K	*dist*	*time*	*dist*	*time*	*dist*	*time*	*dist*	*time*
2	2.30	34	2.37	168	-	-	-	-
5	1.65	14	1.66	32	1.70	170	-	-
10	1.30	7	1.30	50	1.35	122	1.41	93
15	1.00	6	1.00	4	1.03	78	1.13	45
20	1.00	5	1.00	56	1.00	32	1.00	18

Table 2: Simulation results for 128 nodes system

n_u	2		4		8		16	
K	*dist*	*time*	*dist*	*time*	*dist*	*time*	*dist*	*time*
2	2.16	82	2.26	357	-	-	-	-
5	1.62	85	1.62	133	1.63	191	-	-
10	1.22	67	1.20	178	1.26	183	1.34	552
15	1.00	35	1.00	66	1.01	120	1.06	360
20	1.00	19	1.00	29	1.00	97	1.00	168

Table 3: Simulation results for 256 nodes system

1: time is in milliseconds

n_u	4		8		16		32	
K	*dist*	*time*	*dist*	*time*	*dist*	*time*	*dist*	*time*
2	2.00	440	2.13	1255	-	-	-	-
5	1.56	157	1.56	500	1.59	1074	-	-
10	1.10	286	1.13	499	1.18	873	1.29	1382
15	1.00	86	1.00	167	1.00	258	1.03	1045
20	1.00	170	1.00	187	1.00	249	1.00	468

Table 4: Simulation results for 512 nodes system

n_u	8		16		32		64	
K	*dist*	*time*	*dist*	*time*	*dist*	*time*	*dist*	*time*
2	1.87	1459	2.04	3272	-	-	-	-
5	1.51	707	1.54	1654	1.55	3246	-	-
10	1.02	377	1.06	927	1.10	2252	1.24	4911
15	1.00	360	1.00	763	1.00	1788	1.01	3252
20	1.00	380	1.00	516	1.00	1084	1.00	2651

Table 5: Simulation results for 1024 nodes system

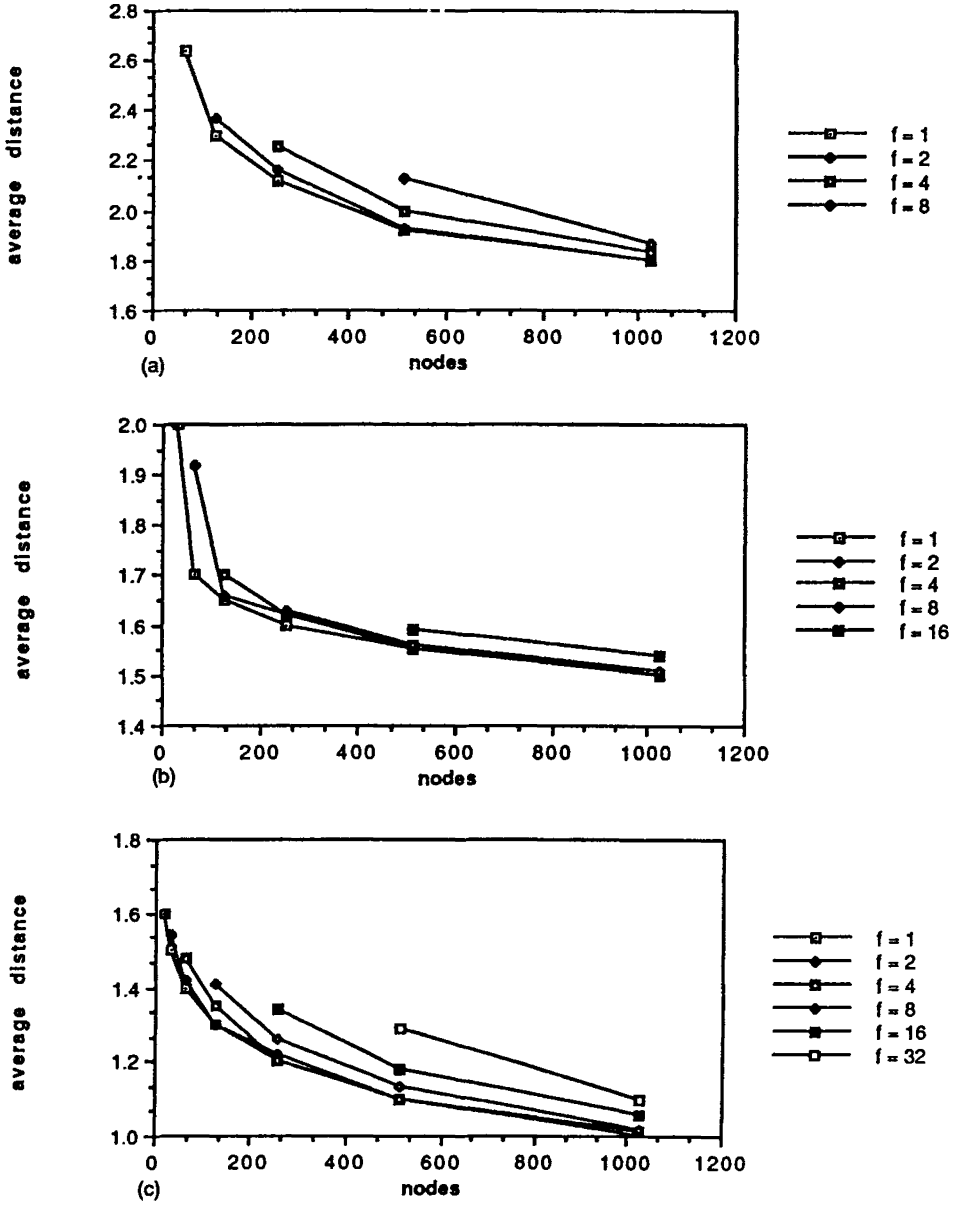


Figure 9: Average distance D in reconfiguration algorithm vs number nodes. (a) Extra load $K = 2$. (b) $K = 5$. (c) $K = 10$.

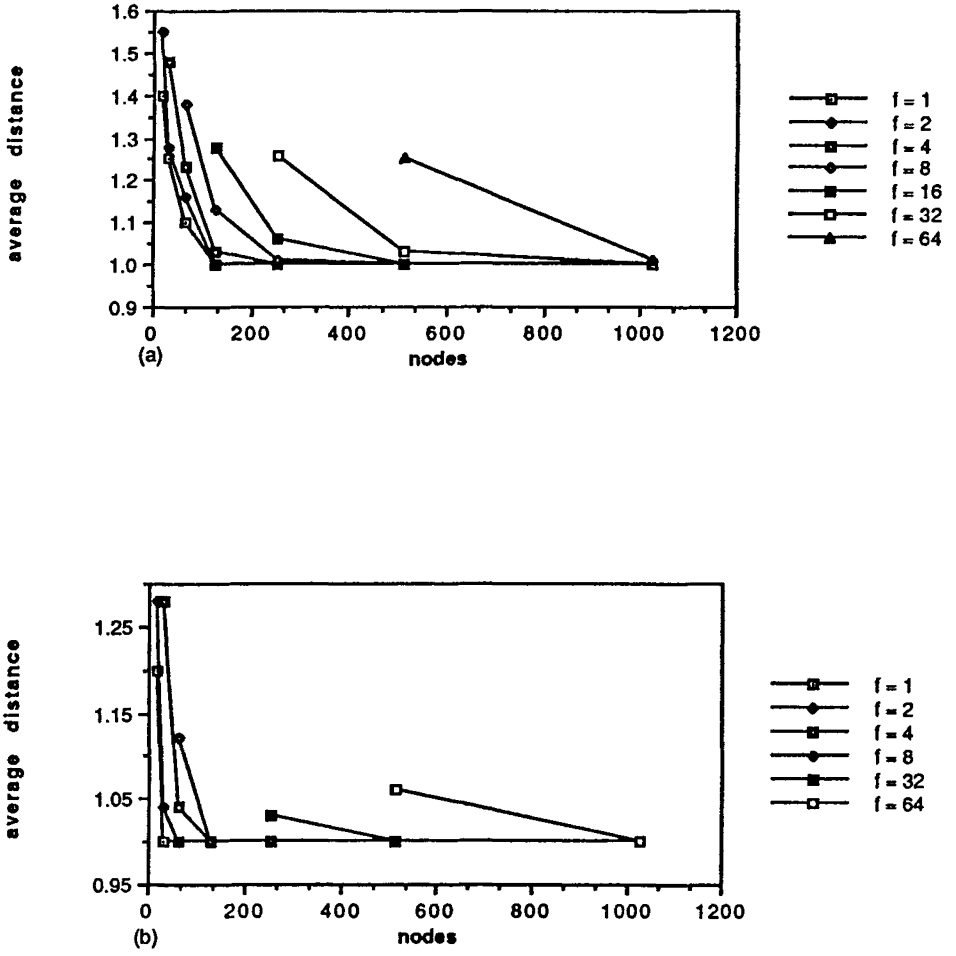


Figure 10: Average distance D in reconfiguration algorithm vs number of nodes. (a) Extra load $K = 15$. (b) $K = 20$.

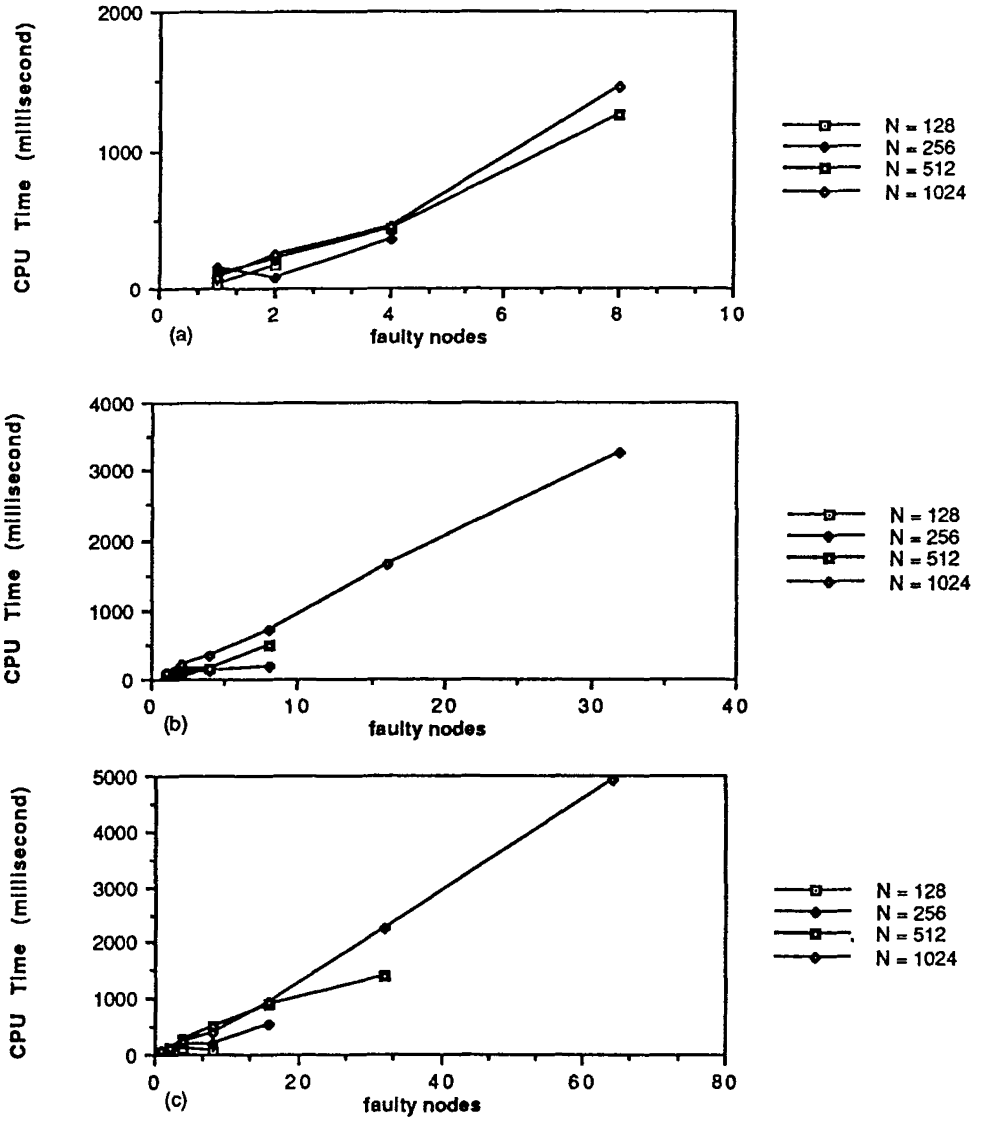


Figure 11: Execution time in reconfiguration algorithm vs number of faulty nodes. (a) Extra load $K = 2$. (b) $K = 5$. (c) $K = 10$.

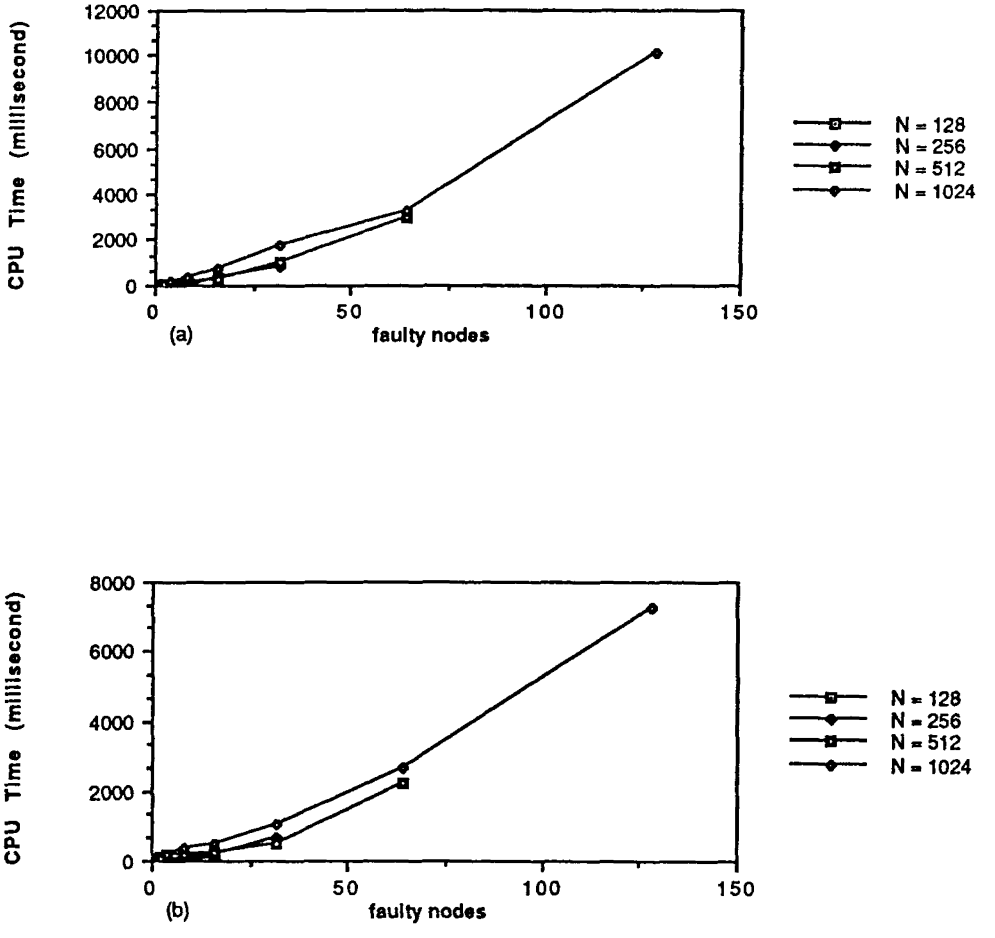
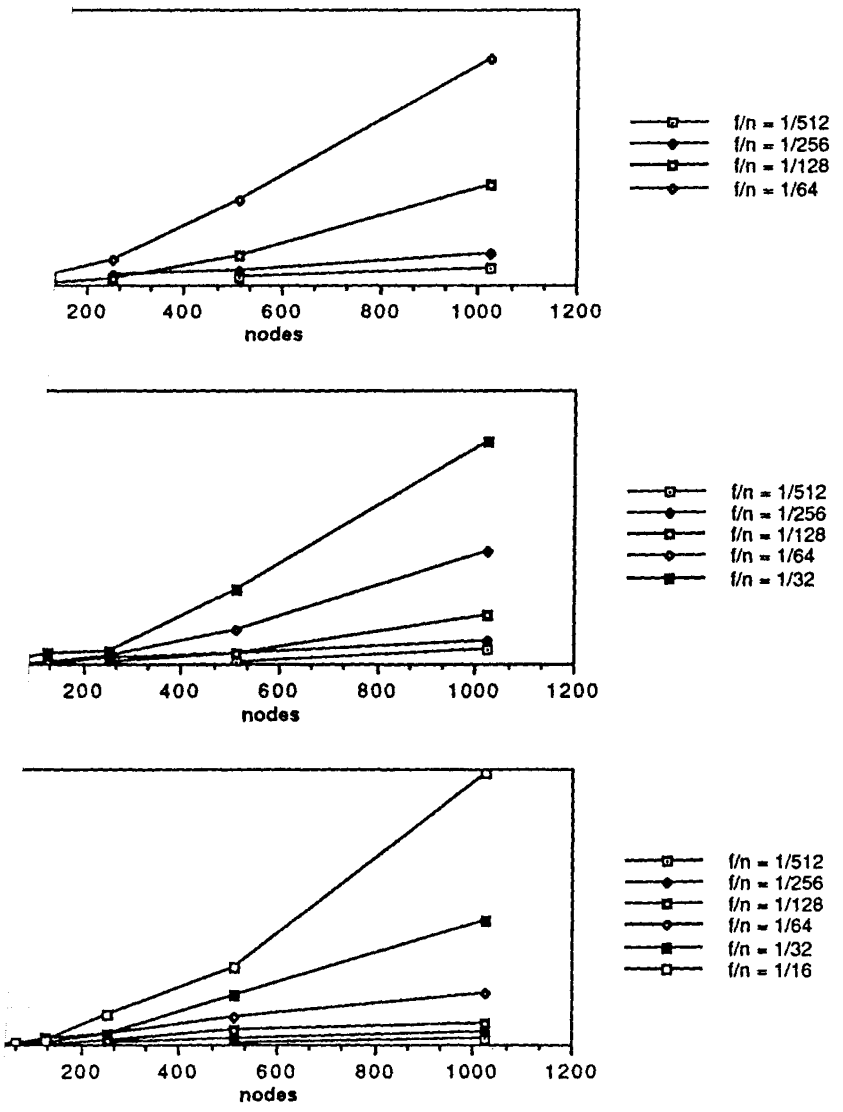


Figure 12: Execution time in reconfiguration algorithm vs number of faulty nodes. (a) Extra load $K = 15$. (b) $K = 20$.



3: Execution time in reconfiguration algorithm vs number of fixed value of f/n. (a) Extra load $K = 2$. (b) $K = 5$. (c)

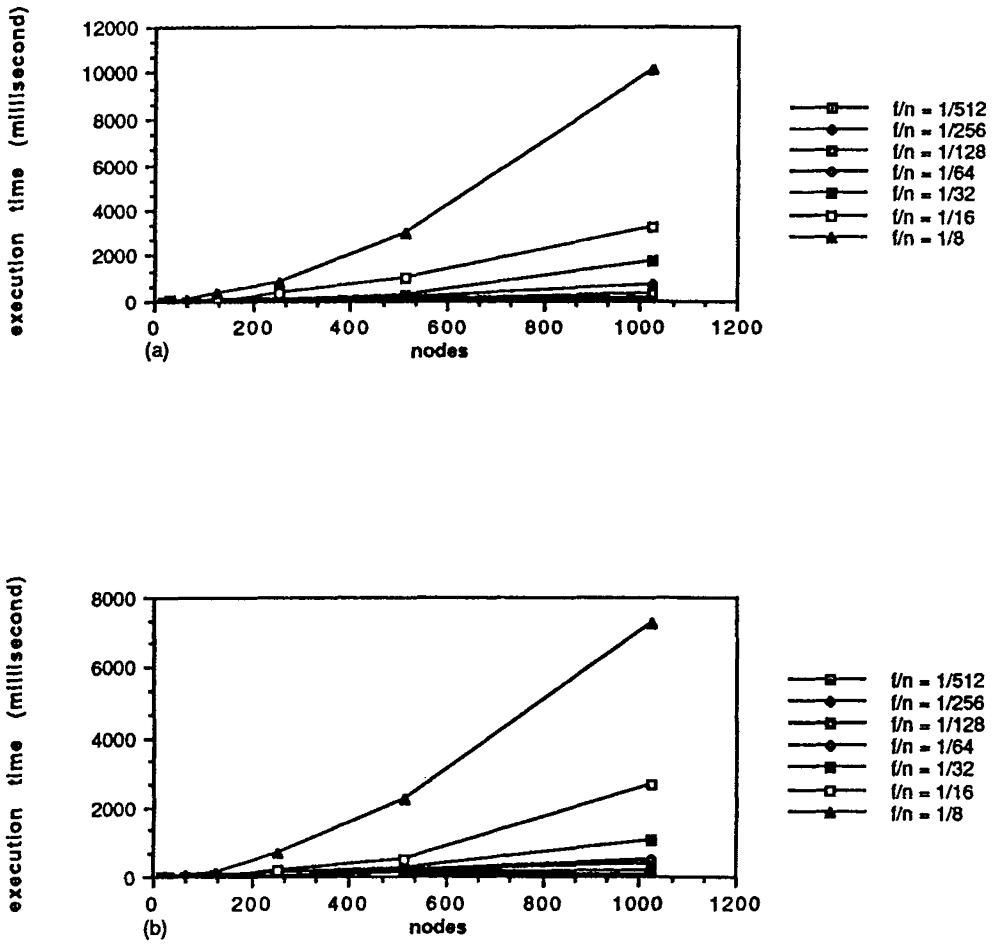


Figure 14: Execution time in reconfiguration algorithm vs number of nodes with fixed value of f/n. (a) Extra load $K = 15$. (b) $K = 20$.

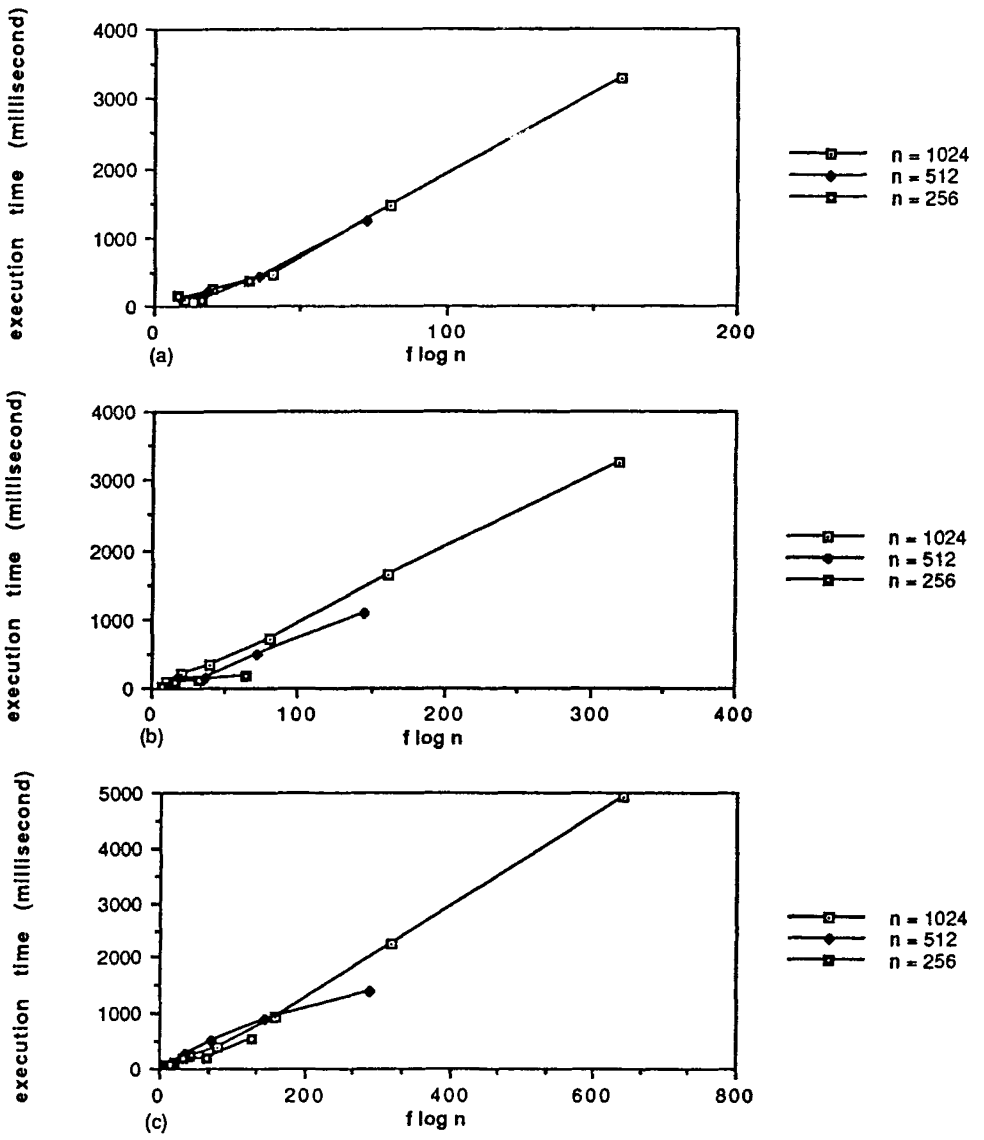


Figure 15: Execution time in reconfiguration algorithm vs value of $f \log n$ with fixed value of n. (a) Extra load $K = 2$. (b) $K = 5$. (c) $K = 10$.

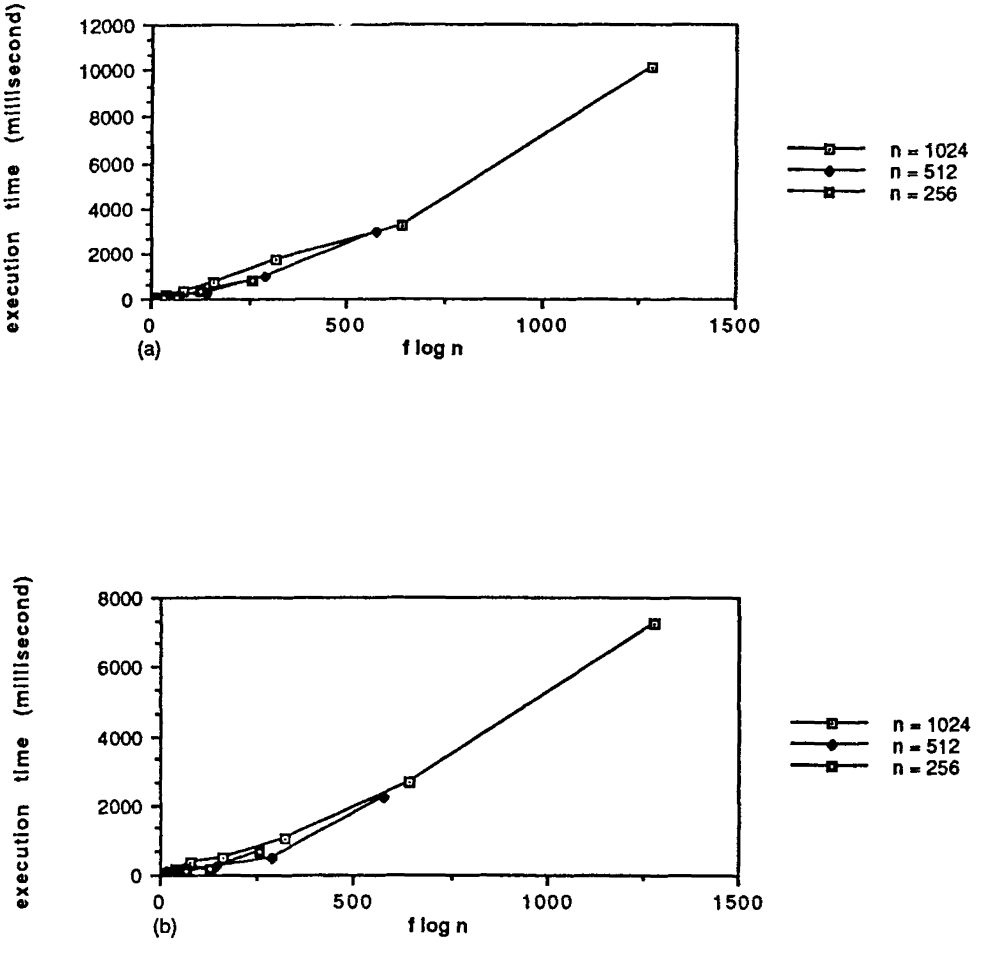


Figure 16: Execution time in reconfiguration algorithm vs value of $f \log n$ with fixed value of n. (a) Extra load $K = 15$. (b) $K = 20$.

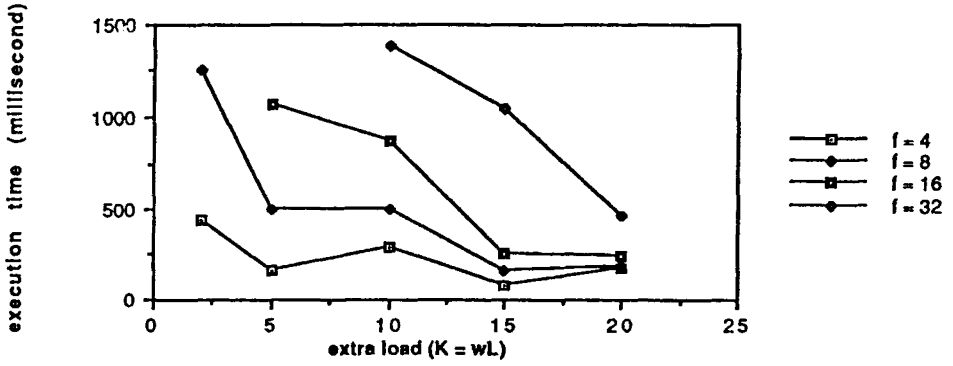


Figure 17: Execution time in reconfiguration algorithm vs w with fixed value of f and $n = 512$.

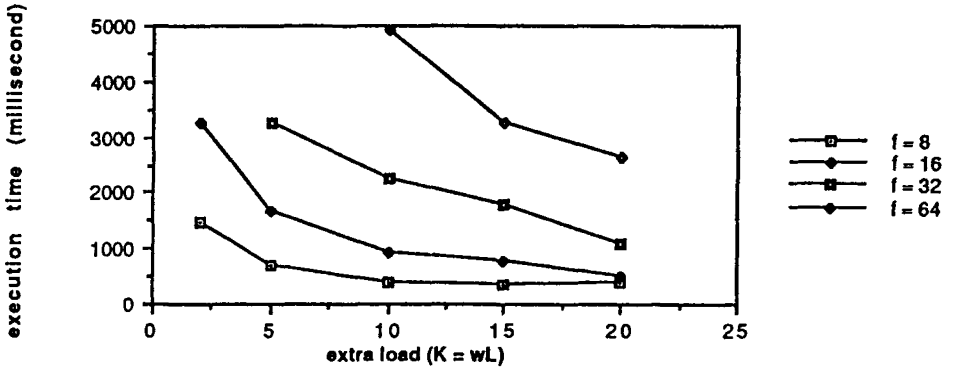


Figure 18: Execution time in reconfiguration algorithm vs w with fixed value of f and $n = 1024$.

7 Bibliography

[**AGGA87**] S.Y. Lee and J.K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Trans. on Computers*, vol. C-36, April 1987, pp.433-441.

[**BANE89**] Prithviraj Banerjee "Reconfiguration Strategies for Hypercube Multicomputers," manuscript, 1989, University of Illinois at Urbana-Champaign.

[**BOKH88**] S.H. Bokhari, "Partitioning Problems in Parallel, Pipelined, and Distributed Computing," *IEEE Trans. on Computers*, Vol. 37, Jan. 1988, pp.48-57.

[**BOKH90**] S.H. Bokhari, "Communication overhead on the Intel iPSC-860 Hypercube," *Technical Report 10*, ICASE, NASA Langley Research Center, May 1990.

[**DALL87**] W.J. Dally and C.L. Seitz, "Deadlock-free Message Routing in Multiprocessor Interconnection Networks," *IEEE Trans. on Computers*, Vol. 36, May. 1987, pp.547-553.

[**DECE89**] A.L. DeCegama, *The Technology of Parallel Processing, Volume 1 (Parallel Processing Architectures and VLSI Hardware)*, Prentice-Hall Inc., 1989.

[**DUTT88**] S. Dutt and J.P. Hayes, "On Allocating Subcubes in a Hypercube Multiprocessor," *Proc. 3rd Conf. Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan. 1988.

[**FORD62**] L.R. Ford, Jr., and D.R. Fulkerson, *Flows in Networks*, Princeton, NJ: Princeton Univ. Press, 1962.

[**KATS88**] H.P. Katseff, "Incomplete Hypercube," *IEEE Trans. on Computers*, Vol. 37, No. 5, May 1988.

[**KERN70**] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, Vol. 49, No. 2, 1970, pp.291-308.

[**MALH78**] V.M. Malhotra, M. Pramodh Kumar, and S.N. Mahcshwari, "An O($|V^3|$) Algorithm for Finding Maximum Flows in Networks," *Inform. Process. Lett.*, 7, 1978, pp.277-278.

[**SADA87**] P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes," *IEEE Trans. on Computers*, Vol. C-36, Dec. 1987, pp.1408-1424.

[**SADA88**] P. Sadayappan, J. Ramanujam and F. Ercal, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning," Dept. of Computer and Information Science, The Ohio State University, Columbus, Ohio 43210.

[**SLEA80**] D.D.Sleator, "An O($nm \log n$) algorithm for Maximum Network Flow," *Tech. Rep.*, Computer Science Dept., Stanford University, Stanford, CA, 1980.

[**STON77**] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp.85-93.

[**TARJ86**] R.E. Tarjan, "A New Approach to The Maximum Flow Problem," *Proc. 18th ACM Symposium on Theory of Computing*, 1986, pp.136-146.