An Algorithm for Dynamic Reconfiguration of a Multicomputer Network

J. M. García
Departamento de Informática
Universidad de Castilla-La Mancha
Albacete, SPAIN 02071

J. Duato
Facultad de Informática
Universidad Politécnica de Valencia
Valencia, SPAIN 46071

Abstract

The dynamic reconfiguration of the interconnection network is an advanced feature of some multicomputers to reduce the communication overhead. In this paper we present an algorithm for the dynamic reconfiguration of the network. Reconfiguration is limited, preserving the original topology. Long distance message passing is minimized by positioning communication partners close to each other. That algorithm is transparent to the application programmer and is not restricted to a particular class of applications, being very well suited for parallel applications whose communication pattern varies over time. The paper also presents some simulation results, showing the benefits from the new reconfiguration algorithm.

1. Introduction

The increasing demand of larger processing power in computers has led to the development of several parallel architectures, usually belonging to the SIMD or MIMD classes according to the classification proposed by Flynn [1]. Among the broad spectrum of parallel architectures, MIMD systems with distributed memory are especially promising. Mostly based on standard VLSI components they offer a very good price/performance ratio. Furthermore, the lack of global resources, which usually are a potential bottleneck, makes them easily expandable up to a large number (hundreds or thousands) of nodes.

Usually, a MIMD computer with distributed memory is called a multicomputer. A multicomputer is formed by several processing nodes, where each processing node consists of a (standard) microprocessor (plus eventually a numerical coprocessor), local memory and communication links to other nodes in the system. The interconnection network between nodes and the internode communication strategies play a major role because the internode communication is the bottleneck for this architecture. Therefore, the ratio computation/ communication must be high. As a full interconnection is impractical for a large number of processors, topologies with a restricted number of neighbours like rings, trees, meshes, pyramids or hypercubes have been proposed. Hypercubes have become very popular and are used for several commercial machines, i.e., Intel iPSC, Ncube or Ametek [2].

In current machines the communication between neighbours is rather fast while communications between non-neighbours has to be implemented by some routing scheme via intermediate nodes. The two most important techniques are store-and-forward and wormhole routing [3]. In store-and-forward routing a message is completely buffered in an intermediate node before it is forwarded to the next one. This causes a considerable message latency, proportional to internode distance. In wormhole routing, especial routing hardware forwards a message as soon as the head of the message containing the routing information has arrived, i.e. a message is spread over several nodes, leading to a very short latency even for long communication paths. Unfortunately, the link protocol of some wide spread processors like current transputers [4] does not support wormhole routing. However, this has been

announced to be changed for the next generation, the T9000 [5].

Especially if store-and-forward routing has to be used, mapping schemes must be employed which try to map application tasks to nodes in such a way that not only load balancing is achieved but also the overhead due to non-neighbour communication is minimized [6]. For wormhole routing this point is no longer of that importance, but minimization of network traffic is still desirable to keep the blocking probability of messages small in the intermediate nodes, thus increasing throughput and reducing message delay for heavily loaded networks. Of course, it is possible to use adaptive algorithms to avoid congested regions of the network. In [7] we have proposed a very powerful theory for the design of adaptive routing algorithms for both store-and-forward and wormhole routing.

An alternative approach to support the mapping of different application problems on a distributed memory machine is dynamic remapping, i.e., moving the processes from a processor to another in order to minimize the cost of the communications. References to this solution can be found in [8].

Another possible solution is to make the interconnection topology reconfigurable. Under this assumption, the architecture of these machines can be classified as follows:

! *Static topologies*: Fixed point-to-point links between neighbour nodes which are either hard-wired or at best can be replugged by hand. Examples for this kind of systems are simple transputer systems or most hypercube machines.

! *Quasy-static topologies*: A switching network allows to establish a topology specified by the application program *before* its execution starts. During program execution the topology remains unchanged.

! *Dynamic topologies*: The network topology can change almost arbitrarily at runtime, i.e. it can be easily matched to the communication requirements of the application program. Another advantage is the potential for fault-tolerance: faulty nodes or links can easily be bypassed and spare nodes be switched in. The Esprit P1085 Supernode [9], the DAMP System [10] and multistage networks [11] belong to this category.

Basically, the main difference between quasy-static and dynamic topologies is the time needed to reconfigurate the network. Also, dynamic topologies usually offer the possibility to perform a partial network reconfiguration, this operation being faster than overall reconfiguration. However the dynamic reconfiguration of the network implies a cost, not being possible to reconfigurate each time a message [10] is sent, since then we would obtain no improvement.

In this paper we present an algorithm for the dynamic reconfiguration of the network. The algorithm decides when the reconfiguration will take place. Reconfiguration is limited, allowing processors to swap places with their neighbours and preserving the original topology. In this way, routing algorithms remain unaltered. A simple node renumbering is required. The algorithm evaluates the communication cost and decides when the reconfiguration is more favourable. This algorithm is based on a cost function and only requires local information. It can be applied to both store-and-forward and wormhole networks, being more interesting in the first case. We also show some numerical results obtained by simulation.

The rest of the paper is organized as follows. In section 2 we describe the machine model and what the implementation of dynamic configuration control is like. In section 3 we present the algorithm for dynamic reconfiguration that we have developed, and in section 4 we evaluate this algorithm offering numerical results. Finally, in section 5 we give some conclusions and ways for future work.

2. Machine Model and Dynamic Reconfiguration Control

This work was originated while trying to increase the communication performance of a transputer-based machine, the PARSYS SN 1000. Then, we have mainly focused on store-and-forward networks.

All the work has been carried out on the FDP environment [12], which permits the simulation of the behaviour of a multicomputer with a particular topology. FDP also offers a simple and efficient programming environment with a friendly user interface.

This environment allows the dynamic reconfiguration of the network, using the *algorithm* which will be presented in the following section. This *algorithm* uses a centralized control scheme to govern the reconfiguration of the network, as is available on the SN 1000. Likewise, we only allow

processors to swap places with neighbours, matching the Supernode capabilities.

2.1 The machine model

Our machine model consists of a main node or host, and a series of nodes over which the different processes executed in parallel have been distributed. FDP allows us to set the machine parameters, including the number of nodes. Most simulations have been carried out for a number of nodes equal to 16, because this is the size of the SN 1000 we have at the laboratory.

However, most algorithms have a number of processes greater than the number of nodes, and thus various processes will be executed in each node. The operating system kernel we have simulated schedules the different processes in a round-robin fashion.

The mapping of processes to processors is carried out in a simple way, without taking into account requisites such as balanced load or small communication cost. However, network reconfiguration is exploited. A process *i* is assigned to a node *j* according to the following function:

$$j = i \ \textbf{mod} \ node\_count$$

where *node_count* indicates the total number of nodes selected for that simulation. Applying the previous function we obtain a good load balancing for many numerical algorithms.

The number of links each processor has to communicate with other processors is limited to 4, as in the transputer [4]. This still allows us the simulation of the best known topologies for multicomputers. To be precise, the FDP permits the simulation of 2-D meshes, rings and hypercubes, the last one only up to 16 nodes.

As indicated above, store-and-forward routing is assumed. Because the mapping is static and known throughout the entire system, the run-time kernel of a node can determine if the source and destination processes of a message are in the same node, or must execute the routing algorithm to the destination process.

2.2 Dynamic reconfiguration control

As mentioned before we have chosen a centralized switch control for the dynamic network reconfiguration. In this model, a master node (the system controller) is responsible for controlling the reconfiguration by means of a control bus.

The network reconfiguration protocol works in the following way:

1. When a node decides that it is necessary to reconfigurate the network, it sends a signal to the system controller through the control bus.
2. Next, the system controller informs all the nodes that it is going to make a reconfiguration and therefore they should stop sending messages to each other. To minimize the reconfiguration time and relieve the cost that it implies, nodes stop transmitting even the messages that are in intermediate nodes.
3. The node which made the request sends the reconfiguration data to the system controller to carry out the reconfiguration.
4. The system controller modifies the interconnection network topology, adapting it to the new circumstances.
5. Once the new configuration has been established, the system controller sends this configuration to all the nodes and permits node communication again.

This protocol is easily implementable using the control bus available in the Supernode architecture, which does not add message traffic to the network. Moreover, the use of a bus allows to perform efficiently the broadcast operations required in steps 2 and 5.

The centralized control could be a bottleneck for the whole system, since all the nodes must send reconfiguration requests to the system controller. This situation is true in the event of frequent reconfiguration, e.g. for each message. But because there exists a considerable overhead, the network will only be reconfigurated when a large amount of data has to be transferred. Thus, for moderate systems, e.g. up to 64 nodes, the centralized control will not represent a bottleneck for the system. For larger systems, the Supernode architecture requires a two-level switch and more than a single control bus.

3. Algorithm for Dynamic Network Reconfiguration

In this section we present the algorithm we have developed for the dynamic reconfiguration of the interconnection network. We can see from the reconfiguration protocol presented in the previous section that this algorithm has two

parts: one for the system controller and another one for the remaining nodes. These algorithms are included in the run-time kernel, whether of the system controller or the nodes.

Firstly, we shall develop the algorithm to be executed in each node. This is the main and most difficult part, the system controller algorithm being very simple.

Before detailing the algorithm, we shall take into account some preliminary considerations.

3.1 Definitions and tradeoffs

Now we are going to define some concepts and consider some tradeoffs for the design of the reconfiguration algorithm:

1. **Local versus global reconfiguration**. The question here is to determine which of the two is more suitable. By *local reconfiguration* we mean that reconfiguration only affects the requesting node and possibly the nodes directly connected to it. *Global reconfiguration*, on the other hand, can bring about a modification in the links of multiple nodes at the same time. As the system controller needs to receive information from all the nodes in the network, *global reconfiguration* implies a greater cost. The advantage is that several changes can be carried out in one step. The algorithm we have developed performs a *local reconfiguration* of the network.

2. **Preserving a given topology**. Reconfiguration can be made making sure that the new network has the same topology (only some nodes have changed their places) or allowing any arbitrary topology. In the algorithm we propose the network topology will be maintained in such a way that if we have a hypercube topology, after any reconfiguration the network still keeps the hypercube topology, but there are a few nodes (*local reconfiguration*) which have changed their place in the network. This reconfiguration is equivalent, although much faster, to renumbering a pair of nodes, also exchanging all the processes they are executing.

3. **Reconfiguration triggering**. Somehow the algorithm has to decide when the reconfiguration of the network is suitable. There are basically two possibilities:

a) *Measuring message traffic*. A node can reconfigurate the network taking into account the number of messages that the node has sent, or has received or have passed through it.

b) *By means of a cost function*. Every node determines the convenience of producing a network reconfiguration based on a function cost.

It must be noticed that in both cases we are speaking about nodes and not about processes, as what matters are the node communications and not the process communications. Besides, internal communications are not taken into account (between processes executed in the same node) to trigger the network reconfiguration.

In our algorithm both possibilities have been tried, better results being obtained when the change is triggered by means of a cost function. This *cost function* should take into account two important aspects: the *weight* of a communication, and the *distance* between nodes. By *weight* of a communication we mean the number of messages which have been sent in a particular direction. The *distance* between nodes measures the number of intermediate nodes that a message has to cross to go from node A to node B; if A and B are physically connected, the distance is null. This distance receives the name of *nominal distance* because it is directly obtained from the network topology. In [13] the aim is to use the *actual distance*, which is a modification of the nominal distance taking into account the traffic in the intermediate nodes. In our case, and due to the fact the reconfiguration is aimed to be local, using only local information, we have chosen the *nominal distance*, because the computation of the actual distance needs information from the rest of the nodes and therefore would require communications between them.

4. **Type of change**. Once the algorithm has decided to make a change, there are several possibilities. Thus, we are going to explain how the possible changes are tested and carried out. When a node, by means of the cost function, decides that it is convenient to make a reconfiguration, it computes the effect of possible changes to see if the traffic conditions improve. There are two strategies to choose possible changes:

a) *Small alteration in the network*. A node can only make an exchange with one of its neighbour nodes. For example, in a ring topology a node only has two possibilities of exchange, with the node on the right, or the node on the left.

b) *Large alteration in the network*. A node can carry out a change to any other place in the

network. To keep the same topology easily, it is convenient for the changed node to take the position left by the node which requested the change.

We are going to detail with an example the difference between both strategies. Let the network have a four-dimensional hypercube topology and let us imagine that the only communications in the network are due to the node 0 sending messages to the node 15 (see figure 1).

With a strategy of *small alteration*, at a given moment (when the cost of the communications is greater than a threshold) the node 0 decides that it is necessary to make a change and, after trying the possible changes, it decides to exchange its position with node 1. After node 15 decides that it is necessary to make a change, it chooses an exchange with node 7. Finally, node 0 (in position 1) requests a change once more, choosing the exchange with node 3. Now the source and the destination nodes are neighbours, without traffic through intermediate nodes, and without producing more changes in the topology.

On the other hand, if we take a *large alteration* strategy, the behaviour is different. When the node 0 decides that it must carry out an exchange, it does so directly with node 7. There are no more changes as the traffic in the network has disappeared (except communication between neighbour nodes).

Although it seems better to use a *large alteration* strategy, when all the nodes are communicating with each other this is not true. Better results are achieved by a *small alteration*, which produces changes that are more uniform and less brusque.

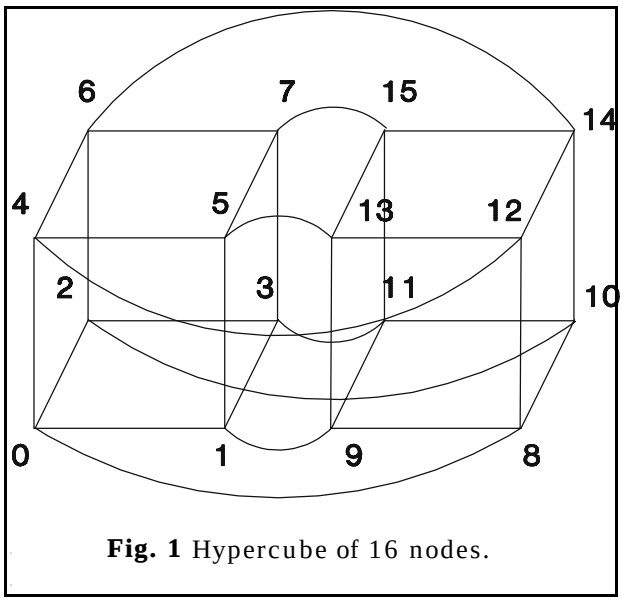

Fig. 1 Hypercube of 16 nodes.

A last consideration. Sometimes, there are several possible changes that minimize the cost function. If the algorithm always chooses the same change, it can lead to endless cycles, which could produce a practically infinite number of changes. This situation usually appears when several nodes exchange the same amount of information in a cyclic way. The solution adopted is to choose each time one of the possible changes according to a *round-robin* strategy. In this way, endless cycles are avoided as is shown in the following section.

5. **Thresholds in the reconfiguration**. As stated above, a large number of reconfigurations reduces the performance of the multicomputer due to the cost implied by a reconfiguration. On the other hand, a small number of reconfigurations leads to slow systems, almost insensitive to changes in message traffic. This is a problem of dynamic optimization, being necessary to determine the threshold range for which network reconfiguration is convenient. Also, the evaluation of the cost function implies an overhead. Then, we must determine how often the *cost function* is going to be evaluated. That evaluation will take place every certain number of messages sent or received. This leads us to handle another threshold. Thus, our algorithm will be evaluated for several pairs of these thresholds.

3.2 The cost function for the dynamic reconfiguration

Before detailing the reconfiguration algorithm, we are going to propose the *cost function* on which it is based. Our goal is to minimize the traffic of messages in the network. Therefore, the cost function must evaluate the network traffic under the above considerations. Then, for a given node *i*:

$$CF_i = \sum_{j=1}^{nodes} C_{ij} * d_{ij}$$

where C_{ij} is the total message traffic between nodes *i* and *j* (i.e., the messages sent by the node *i* to the node *j* and the messages received by the node *i* from the node *j*) and d_{ij} is the *nominal distance*. With this cost function we

take into account the real communications between *i* and *j* weighted with their *nominal distance*, which give us a measure of the traffic those communications suppose. This function needs to record information for every message in a suitable data structure. For each node, an array with as many elements as nodes will be sufficient.

3.3 The reconfiguration algorithm

The algorithm we have developed for the dynamic network reconfiguration has the following properties: uses **local reconfiguration, preserves the topology, is based on a cost function, produces a small alteration in the network and uses two thresholds for network reconfiguration.** In this paper, we do not take into account the operational cost of the reconfiguration. In the simulations shown, we suppose a null cost for reconfiguration.

The basic idea is the following: When the traffic between a pair of nodes is intense, the algorithm will try to put the source node close to the destination node by exchanging the positions of either the source node and a neighbour of it or the destination node and one of its neighbours. However, the implementation of this algorithm is distributed, each node taking into account the information recorded locally.

Under all those considerations, firstly we are going to present the reconfiguration algorithm for each node *i*. It is executed each time a message is sent to or received from a given node *j*:

```
begin
 record_message_information (node_j);
 message_count:= message_count + 1;
 if (message_count mod threshold_2) = 0 then
begin
   cost:= cost_function(node_i);
   if cost>threshold_1 then begin
     change:=
check_possible_changes(selected_change);
    if change then reconfig(node_i,
selected_change);
   end;
 end;
 end;
```

where

Message_count: Measures the total number of messages sent or received by that node.
Threshold_1: Minimum cost for reconfiguration. Takes into account that the reconfiguration requires some time.
Threshold_2: Time interval between evaluation of the cost function. Takes into account that this evaluation requires some time.
Check_possible_changes: It evaluates if a new configuration would reduce the cost function. If it does, the change is made. In the case that there are several new configurations with a smaller cost, the one with the minimum cost is chosen. When several configurations have a cost equal to the minimum, a round-robin strategy is used.
Change: Boolean variable which indicates if a change will be made.
Reconfig: Procedure initiating the reconfiguration protocol with the system controller.

Now, we are going to present the algorithm for each node *i* that is executed each time it receives a message containing a new configuration from the system controller:

```
begin
  update_node_mapping;
end;
```

where

Update_node_mapping: Procedure that updates the mapping of physical nodes to positions in the network topology. It must be executed by all the nodes, not only the requesting node. This update is necessary to correctly compute the cost function and the routing algorithm.

Finally we will detail the reconfiguration algorithm for the system controller. It is executed each time a reconfiguration request is received:

```
begin
  broadcast(suspend_message_traffic);
  update_switches;
  broadcast(new_configuration);
  broadcast(continue_message_traffic);
end;
```

where

Broadcast: Send a message to all the nodes. After sending it, the system controller waits until all the nodes acknowledge the reception. The acknowledgement implies that the requested operation has been performed.
Update_switches: Switch the topology of the inter-connection network.

4. Evaluation of the Algorithm

In this section we are going to evaluate our network reconfiguration algorithm for several examples. The results illustrate the improvement that can be obtained when the network is reconfigurated. This improvement will be measured by means of three parameters:

1. **Total message traffic in the network**: We shall only count those messages crossing intermediate nodes. Each time a message crosses an intermediate node, the total traffic is increased.

2. **Maximum node traffic**: It tells us how many messages have crossed the most saturated node. This is an important parameter since a highly saturated node reduces the performance of the whole system.

3. **Number of changes**. It is important to know how many changes have been carried out to obtain a given result.

These results have been obtained with the FDP environment. First of all, we present the results for some simple cases, then we show the results for a complex numerical algorithm.

4.1 Simple cases

We have chosen two simple cases. They are the following:

- Case 1: Two source nodes and one destination node. Node 0 and node 8 send messages to node 15.

- Case 2: A cyclic case. Node 0 sends to node 12, node 12 sends to node 3 and node 3 sends again to node 0. This case will serve to explain the concept of a *cyclic change* and the necessity of using a round-robin strategy, at the moment of choosing one change or another.

The conditions that we have chosen for these cases are the following: the first threshold is equal to 10, and the second threshold is equal to 5. The number of messages that a node sends is 100. A four-dimensional hypercube has been chosen in the case of dynamic reconfiguration.

Figure 2 shows the results obtained in the case 1. For comparison purposes we also show the results obtained with the static topologies supported by the FDP environment. As we can

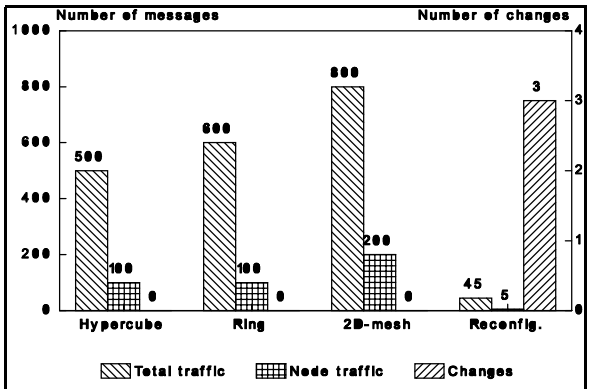

Fig 2. Dynamic reconfiguration for simple case 1.

see, the results are very good, existing a very drastic reduction in message traffic, and a reduction in the traffic through the most saturated node too. Moreover, once the reconfiguration algorithm has carried out three changes, the source and the destination nodes are neighbours. Although this case is very simple, these results are similar for more complex algorithms. A wider study of several test cases as well as different thresholds can be found in [14].

The other test case will serve to explain the *cyclic change*. Let us imagine the following situation (see figure 1). Node 0 sends to node 12, node 12 sends to node 3 and node 3 sends again to node 0. If each node always chooses the same change among various alternatives, the produced changes are: node 12 to node 8 (trying to approach node 3 and node 0), node 3 to node 2 (trying to approach node 0 and node 12), node 3 (position 2) to node 0 (trying to approach node 12), node 12 (position 8) to node 10 (trying to approach node 0 in position 2), and again node 3 is moved from position 0 to position 2. We have a cycle since the node 3 is always changed from position 0 to position 2 of the hypercube and viceversa, and the node 12 is always changed from position 8 to position 10 and viceversa. Then we must avoid *cyclic changes*, because they lead to a high number of changes and a slight reduction in message traffic. When we use the *round-robin* strategy we break the cyclic changes.

4.2. A complex case: the sparse matrix triangularization.

To evaluate the reconfiguration algorithm in a complex case, we have chosen the triangularization of a sparse matrix based on the rotations of Givens, as this algorithm is very suitable for parallel machines because of its inherent parallelism. A deep study of a parallel version of this algorithm can be found in [15,16]. A detailed explanation of the algorithm as it has been simulated in the FDP can be found in [12].

We will briefly describe the algorithm. It requires as many processes as columns the sparse matrix has. If we define the type of a row as the column position occupied by its leftmost non-zero element, then it is well known that only rows of the same type can be rotated together. Then we distribute the rows among processes in such a way that each process stores all the rows of the same type. After a pair of rows has been rotated, one of them increases its type, being sent to the corresponding process to be processed again. Of course, empty rows are discarded. The algorithm finishes when there is at most a single row in each process. As the rotation of a pair of rows cannot produce a row of a lower type, a token is passed through all the processes to determine when the triangularization program has finished.

This algorithm has been chosen because we cannot know a priori the communication pattern between nodes, because it depends on the structure of the sparse matrix, and therefore a suitable process mapping cannot be chosen. Moreover, the communication pattern will vary over time, making it very suitable for dynamic reconfiguration.

For dense matrices, the rotation of two rows of type *t* produces two rows of types *t* and *t+1* respectively. Then, this algorithm performs very well on a ring. For sparse matrices, as processing advances, matrix fill-in increases, approaching the behaviour of dense matrices. In fact, when this algorithm is executed statically on a ring, a 2D-mesh and a hypercube, the best results as far as traffic is concerned are obtained for the ring. Therefore we will take the ring as a suitable point of reference, which will serve to measure how much we gain by reconfigurating the network dynamically.

Figure 3 shows the total message traffic and the message traffic through the most saturated node for the above described algorithm on different topologies supported by FDP, including

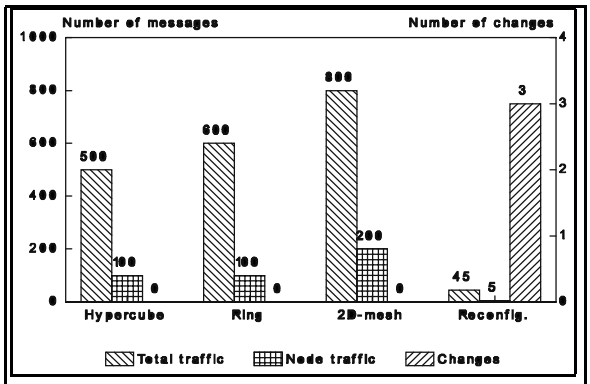

Fig. 3. Dynamic reconfiguration in a 150x75 matrix.

dynamic reconfiguration. To obtain these values we have used a 150x75 matrix, with an average of two non-zero elements per row randomly placed. Before applying Givens rotations to triangularize the matrix, its columns have been

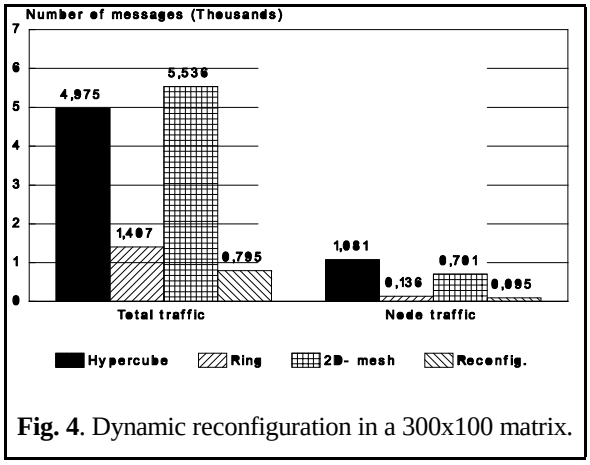

Fig. 4. Dynamic reconfiguration in a 300x100 matrix.

reordered to reduce the number of messages and increase the efficiency of the parallel algorithm. In the case of dynamic reconfiguration, we have chosen the hypercube topology as before.

The number of changes to obtain this result has been 14, threshold_1 having a value of 16, and threshold_2 having a value of 64. As can be seen, message traffic has been drastically reduced, also existing a significant reduction in the traffic through the most loaded node. With respect to the best static topology (the ring), a

reduction of 35% is obtained in the total message traffic.

To confirm these results, we have carried out several tests with other matrices, usually with an average of two non-zero elements per row [14]. The results obtained are very good, showing us that the larger the matrix, the better results are achieved. As an example, the results obtained for a 300x100 matrix with static topologies and the best dynamic reconfiguration are shown in figure 4. As we can see, the traffic reduction is significant for a relatively small number of changes. In the best case, the traffic is a sixth of the one achieved with a static hypercube, and about 40% less than the traffic obtained with a ring.

Depending on the threshold values, the results obtained are different. Low thresholds seem more adequate; however they cause many changes, many times without an important traffic reduction.

This situation will change after introducing the operational cost of the reconfiguration in the cost function. Then, it will be easier to determine the optimal values for the thresholds.

These results confirm the suitability of the dynamic reconfiguration as a means of achieving a reduction in message traffic.

5. Conclusions and Future Work

In this paper we have presented an algorithm to reconfigurate dynamically the interconnection network for multicomputers with store-and-forward routing. We have detailed the reconfiguration protocol and we have evaluated our algorithm both for simple cases and for a complex mathematical problem. In all the cases the behaviour of the algorithm has been good, leading to a reduction both in the total message traffic in the network and in the message traffic through the most saturated nodes. This improves the performance of multicomputers as their main bottleneck is the communication cost between nodes.

In relation with the dynamic reconfiguration of the interconnection network, we have analysed several important tradeoffs. The proposed algorithm features are *local* reconfiguration, *preserves the topology*, produces a *small alteration*, and is based on a *cost function*.

As future work we want to test several things. Firstly, we want to include the operational cost of the reconfiguration in the cost function. Secondly, we want to evaluate the reconfiguration algorithm for other parallel programs, with larger matrices and larger networks showing the speed-up obtained with dynamic reconfiguration. Thirdly, we would like to find a relationship -if there is one- between the most suitable values for the thresholds and other network parameters. Finally, we plan to extend the proposed algorithm for large Supernode networks, requiring a two-level switch and several control buses.

References

[1] Flynn, M.J. "Some computer organizations and their efectiveness". *IEEE Trans. on Computers*, Vol. C-21, pp. 948-960, 1972.

[2] Almasi, G.S. and Gottlieb, A. "*Highly Parallel Computing*". The Benjamin/Cummings Publishing Company, 1989.

[3] Dally, W.J. and Seitz, C.L. "Deadlock-free message-routing in multiprocessor interconnection networks". *IEEE Trans. on Computers*, Vol. C-36, No.5, pp. 547-553, May 1987.

[4] Inmos Corporation. *"The Transputer Databook"*. Inmos Ltd., England, 1989.

[5] May, D. "The next generation transputers and beyond". *2nd European Distributed Memory Computing Conference*, Munich, April 1991.

[6] Bokhari, S. H. "On the mapping problem". *IEEE Trans. on Computers*, Vol. C-30, No.3, pp. 207-214, March 1981.

[7] Duato, J. "On the design of deadlock-free adaptive routing algorithms for multicomputers: design methodologies". *Parallel Architectures and Languages Europe 91*, Eindhoven, June 1991.

[8] Nicol, D.M. and Reynolds, P.F.Jr. "Optimal dynamic remapping of data parallel computations". *IEEE Trans. on Computers*, Vol. C-39, No.2, pp. 206-219, Feb. 1990.

[9] Nicol, D.A. "Reconfigurate transputer processor architectures", in T.J. Fountain and M.J. Shute (Ed), *Mul-tiprocessor Computer Architectures*, North-Holland, 1990.

[10] Bauch, A.; Braam, R. and Maehle, E. "DAMP: A dynamic reconfigurate multiprocessor system with a distributed switching network". *2nd European Distributed Memory Computing Conference*, Munich, April 1991.

[11] Hofestädt, H.; Klein, A. and Reyzl, E. "Performance benefits from locally adaptive interval routing in dynamically switched

interconnection networks". *2nd European Distributed Memory Computing Conference*, Munich, April 1991.

[12] García, J.M. and Duato, J. "FDP: An environment for a MIMD programming with message-passing". Technical Report GCP #1/91, Departamento de Ingeniería de Sistemas, Computadores y Automática. Univ. Politecnica de Valencia, 1991.

[13] Lee, S.Y. and Aggarwal, J.K. "A mapping for parallel processing". *IEEE Trans. on Computers*, Vol. C-36, No.4, pp. 433-442, April 1987.

[14] GarcRa Carrasco, JosJ M. "*Desarrollo de Herra-mientas para una Programación Eficiente de las Redes de Transputers: Estudio de la Reconfiguración Dinámica de la Red de Interconexión*". PhD thesis. Universidad PolitJcnica de Valencia. November, 1991.

[15] Duato, J. and Pons, J. "Parallel triangularization of sparse matrices on distributed memory multiprocessors", in F. AndrJ and J.P. Verjus (Ed), *Hypercube and Distributed Computers*, North-Holland, 1989.

[16] Duato, J. "Parallel triangularization of a sparse matrix on a distributed-memory multiprocessor using fast Givens rotations". *Linear Algebra and its Applications*, No.121, pp. 582-592, 1989.