Speeding Up Connectivity Analysis of Large Computer Networks by Topology Reformulation and Parallelization[1]

Jerzy A. Barchanski

Brock University
Department of Computer Science
St.Catharines Ontario L2S 3A1
CANADA

Abstract: This paper presents an approach to connectivity analysis of large computer networks and networks interconnection. The essential feature of the approach is the reformulation of a flat network topology into a balanced-tree hierarchical (BH) topology, distribution of this topology onto a tree of parallel processing elements and usage of parallel algorithms for connectivity analysis of the reformulated topology. It is shown that the performance of the parallel algorithms is much higher than that of the corresponding sequential algorithms. The parallel algorithms are implemented in a parallel logic programming language CS-Prolog running on a multitransputer network. As an example, a parallel program for connectivity analysis of a 3 level BH topology network is described.

Keywords: connectivity analysis, parallel algorithms, hierarchical topology, computer networks, parallel logic programming, transputers.

Dr. **Jerzy A. Barchanski** obtained his MSc. and Ph. D degrees in Computer Science from Silesian Technical University in Poland in 1969 and 1977. Upon receiving a scholarship from the Japanese Government he spent 3 years in Japan as a researcher at Osaka University and Kyoto University. After returning to Poland in 1976 he worked as an Assistant Professor at the Silesian Technical University and the Technical University of Wroclaw. In 1982 he was invited to Canada to work as a research associate at the University of Montreal and in 1983 he accepted a position of Associate Professor of Computer Science at Brock University in St.Catharines where he works until now. During his tenure at Brock University he was moreover a Visiting Professor at the University of Ottawa. His main areas of interest are computer networks and their protocols, distributed and object-oriented simulation, parallel logic programming and expert systems. He has published over 40papers in these areas.

1. Introduction

One of the requirements usually imposed on computer networks is that they be reliable, even in the face of unreliable nodes and links. To achieve high reliability with unreliable components, the network must be redundant. A sufficiently redundant network can loose a small number of components and still function properly, albeit with lower performance. To find out whether there is a connection between a specific pair of nodes, how many disjoint paths are between any pair of nodes in the network, what is the maximum possible information flow, what is the probability of partial or complete network disconnection - connectivity analysis is used [4,9]. It provides a number of algorithms to answer such questions. The common characteristic of these algorithms is sequentiality - they were developed for implementation on sequential computers. Due to this, even the largest computers cannot find in reasonable time the answers for a 50 node network, let alone a 1000 node network [9].

This paper presents an approach to speeding-up connectivity analysis of large computer networks or network interconnection by reformulation of network topology and parallelization of the connectivity analysis algorithms.The reformulation of a network topology is made by decomposing a complex network topology with a flat graph structure into a hierarchical tree.This approach has proven its strength in several distinct domains (like parsing of formal languages, automatic test pattern generation for digital circuits,solving constraint satisfaction problems or

1 This paper is a revised and extended version of [2].

complex system simulation [3]).While the hierarchical approach can provide considerable speed-up even on a sequential computer,it can realize its full potential on a parallel computer system only.

2. Reformulation of Network Topology

Many networks possess certain structures in their topology. Especially interesting is a class of hierarchical topologies called balanced-tree hierarchical topology (BH topology)[1].

Definition 1.(Balanced-tree).

$A(b_{lo},b_{up})$ balanced-tree is defined as a tree in which the number of children for each father is lower bounded by b_{lo} and upper bounded by b_{up}, where b_{lo} and b_{up} are independent of n (the total number of nodes in the tree) and $b_{lo} >= 2$.

Definition 2.(BH topology).

If all the nodes in a network graph can be organized into $a(b_{lo}, b_{up})$ balanced-tree such that each leaf on the tree represents a node, and each father on the tree represents a cluster, which contains a group of connected children (where the children may represent either clusters or nodes), then the graph is said to have a (b_{lo}, b_{up}) BH topology.

The actual nodes of the network are referred to as level 0 clusters, and the clusters whose members are nodes are referred to as level 1 clusters. In general, the clusters whose members are level j clusters are referred to as level j+1 clusters (Figure l).

We may identify several subclasses of the BH topologies. In fact, every connected graph (not necessarily fully connected) with 0(1) connectivity (i.e. with several paths between any two nodes) can be characterized as a BH topology. It would be very desirable to construct an algorithm that could transform graphs to BH topologies. The algorithm would have to check many different node set descriptions until one is found that satisfies the definition of a BH topology. It turns out however, that on finding such a node set description, an

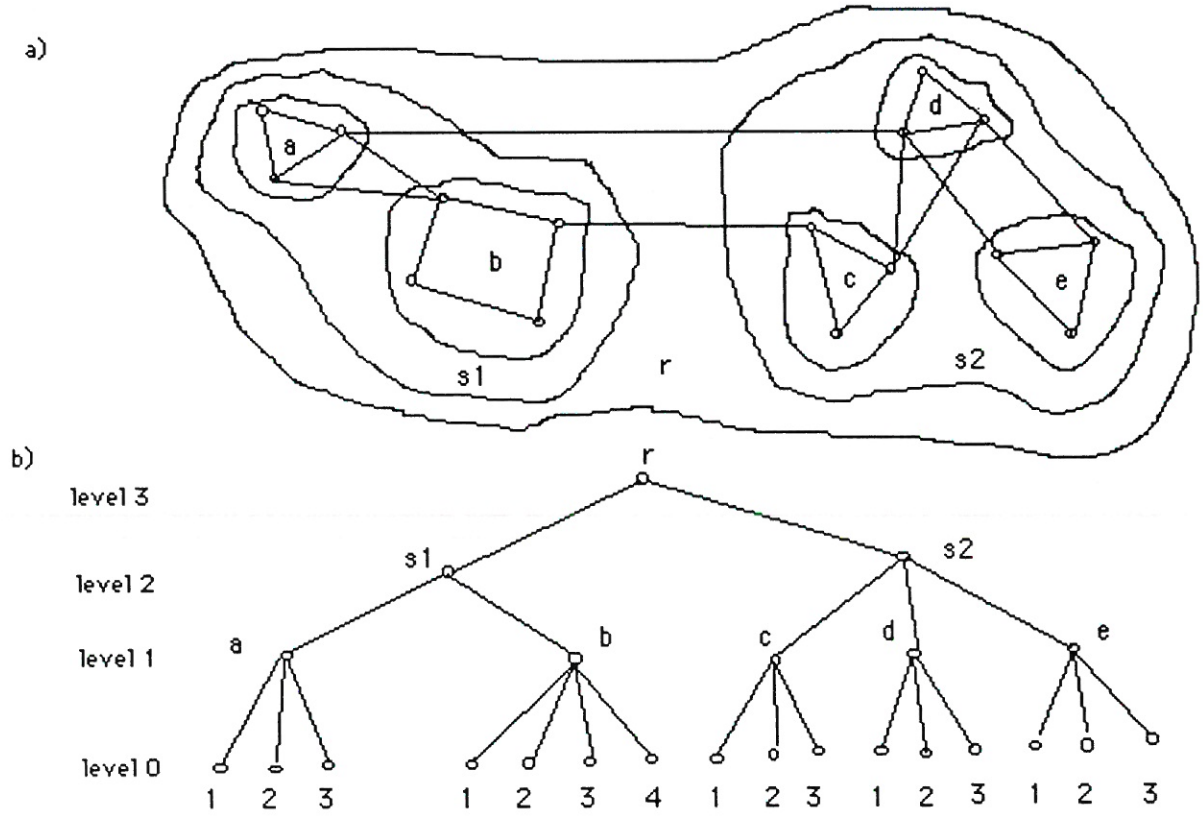

Figure 1. The example network in (a) can be characterized as a 3-level BH topology in (b)

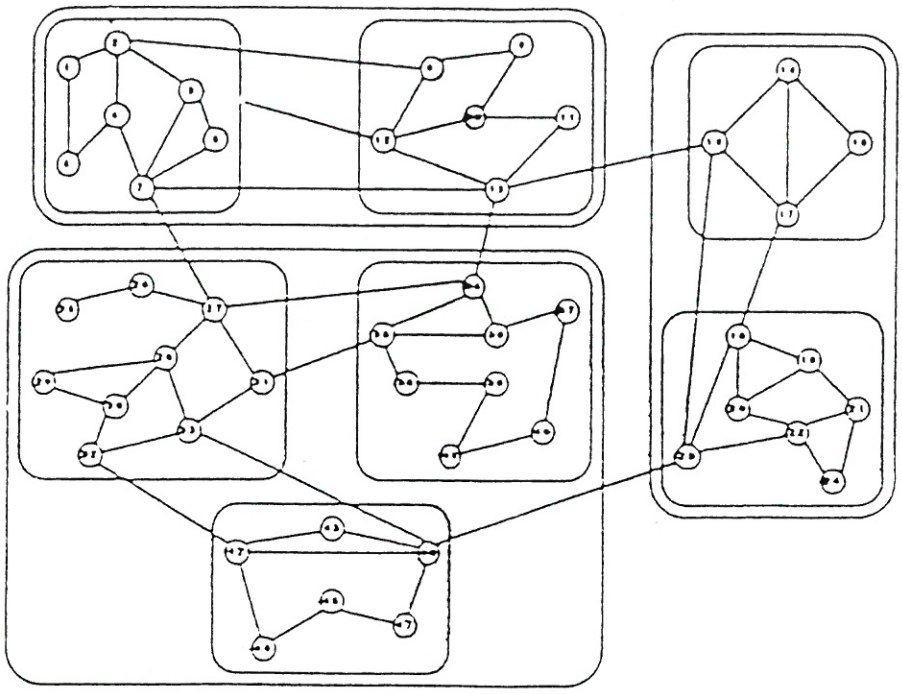

Figure 2. Naturally hierarchical topology

NP-complete problem has to be solved (this conclusion can be derived from a result in [11]).It is known in general that reformulation of NP-complete problems into tree structures has the advantage that polynomial time algorithms can be constructed for solving them.

Instead of trying to solve the NP-complete reformulation problem we will in the following create the BH topology heuristically (what seems to be a viable,though not necessarily an optimal alternative) by decomposing a flat topology into subnetworks and abstracting them into supernodes of a higher level network.

This can be done for instance by taking advantage of the physical characteristics of most large data networks. Quite often, the BH topology is obvious upon inspection. Most often data networks exhibit some sort of hierarchical structure, e.g. nodes belonging to a geographical area, or a particular existing subnetwork form of natural clusters (Figure 2).

There may exist some networks however, in which an underlying hierarchical structure may not be readily visible. For example, Figure 3 shows a uniform mesh network represented as a 3-level BH topology and Figure 4 depicts a simple "linear" (multidrop) network, which can be represented as well as a 3-level BH topology.One should realize moreover that,in general,a given network may have several different BH topology descriptions.

3. Parallel Connectivity Analysis Algorithms

3.1. Node-to-node Connectivity Analysis

By representing a network topology as a BH topology we can carry out the connectivity analysis hierarchically and in parallel. For instance, to find out whether there is a path from node A1 of the internetwork S1 to node E3 of the internetwork S2 (Figure 5.a.) we will check first if the two internetworks are connected. If they are, we will examine their internal topology (Figure 5.b.) and

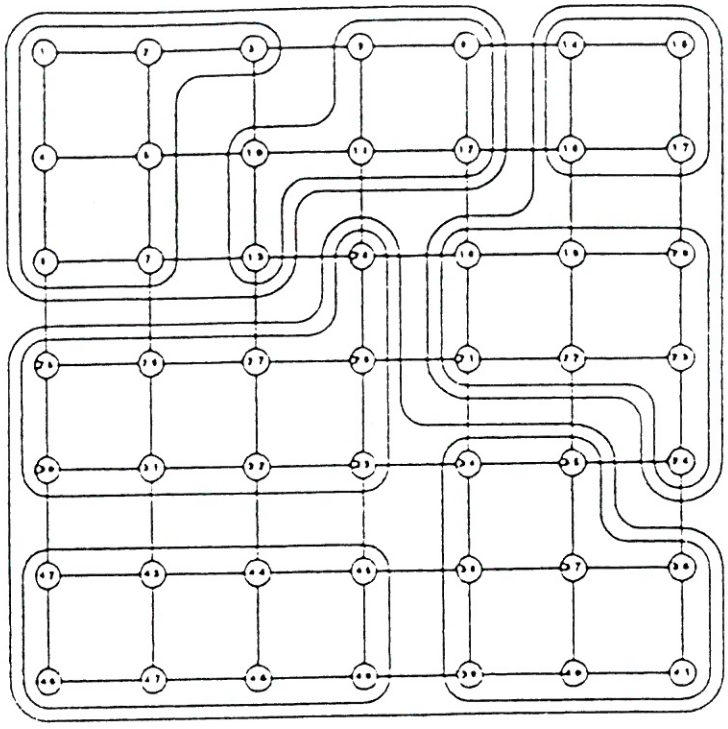

Figure 3. A uniform mesh topology reformulated into 3-level BH topology

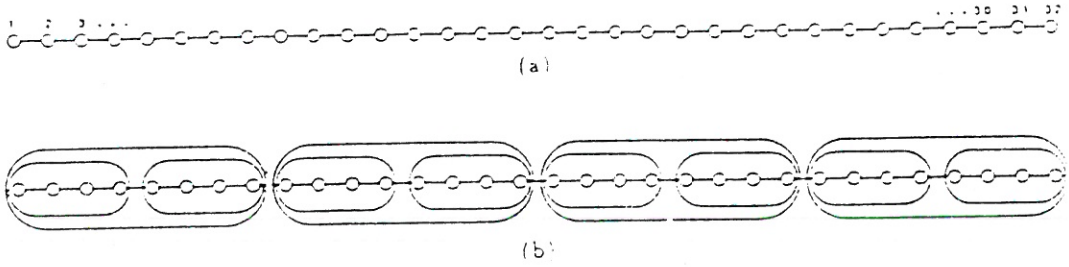

Figure 4. A linear "multidrop" topology reformulated into 3-level BH topology

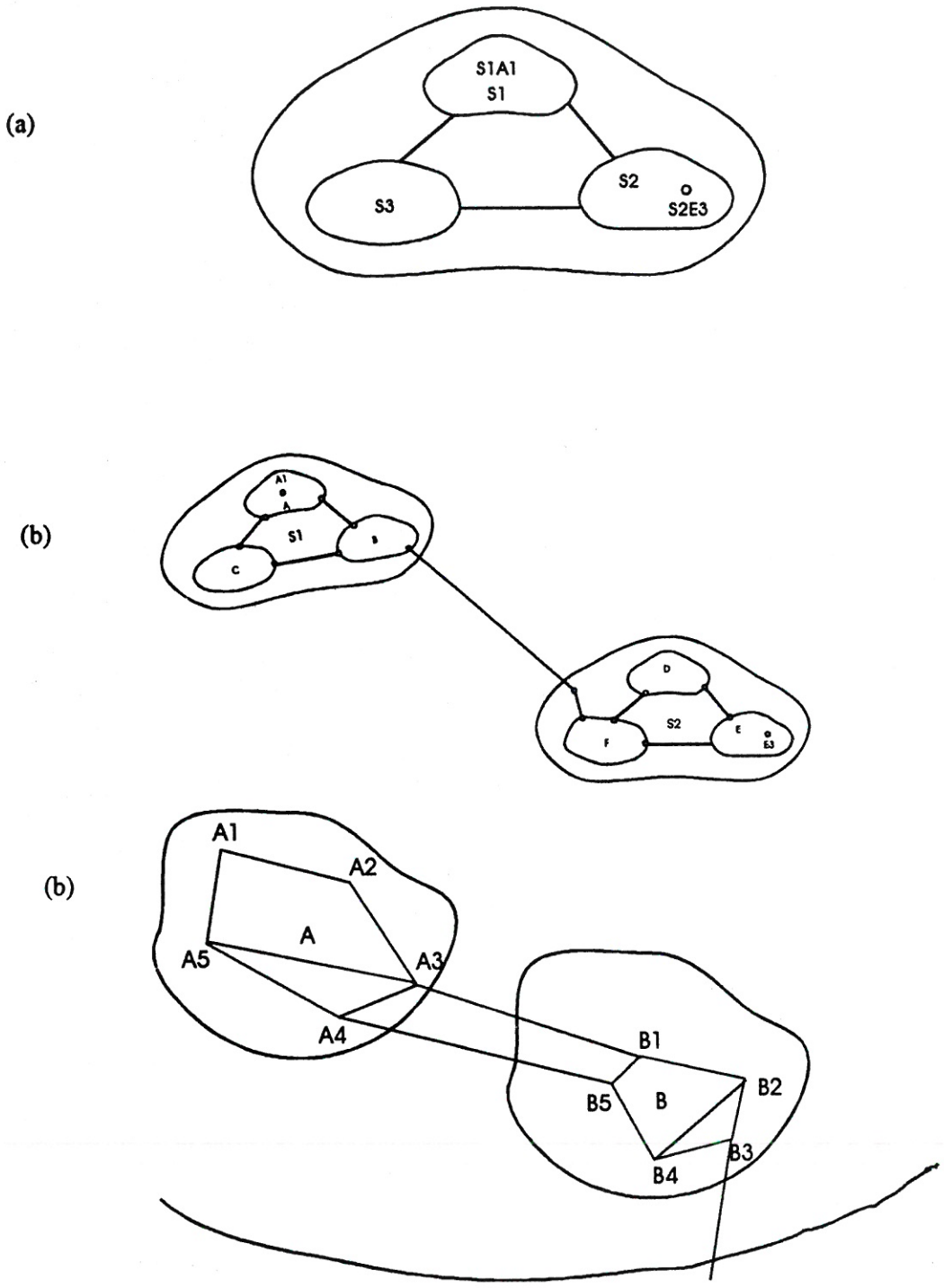

Figure 5. Hierarchical and parallel node-to-node connectivity analysis

check in parallel connections between gates (the nodes connecting the adjacent networks) of their component networks (A,B,C) and (D,E,F). Finally we will search in parallel paths from node A1 to node E3 through all the intermediate networks and gates (Figure 5.c.shows only two adjacent lowest level networks). If there is no connection at a level then the analysis can be stopped. This allows to eliminate the search of lower levels which is bound to fail anyway.

3.2. Shortest Path Search

To find the shortest path between a pair of nodes or between any node and a given destination node, we start the search from the bottom-most level, by finding in parallel the shortest paths between the origin node and its cluster (network) gate, between the destination node and its cluster (network) gate and between the gates of all the other clusters (networks) at this level. We can proceed then to the next level for finding the shortest path between the origin higher-level cluster (network) gate and the destination higher-level cluster (network) gate through the intermediate gates identified at this level. This procedure has to be repeated until the top-most level is reached.

3.3. Monte Carlo Reliability Analysis

Network reliability can be represented by the probability of network disconnection (i.e. a situation,when there is no path for at least one pair of nodes). To determine the reliability of a network with uniform node connectivity (i.e. the same number of links per each node), algorithms proposed by Kleitman or Even [3] can be used. For most large, irregular networks however,the connectivity of the nodes is not the same, so the only recourse is to simulation.

A straightforward but nevertheless useful approach is to assume that at any one time a link (or a node) is either up (working) or down (failing). The probability of a link failing during a time interval (corresponding to a simulation run) can be represented by a variable LF. Network reliability is a function of LF and can be found by extending the algorithm to parallel node-to-node connectivity analysis with the link failure probabilities.

The algorithm will first check network connectivity at the top-most BH level. If the network is disconnected at this level, there is no need to continue. Otherwise, the algorithm will iteratively check the connectivity at the next lower level until it detects disconnection or reaches the bottom level.

4. Implementation Environment

4.1. Software Environment

We have implemented the parallel connectivity analysis algorithms in a parallel logic programming language CS-Prolog [5,6]. There were several reasons underlying the selection of a logic programming language for this project.

A logic program consists of a set of facts and clauses describing declaratively the knowledge required to solve a problem. The clauses are independent; each has its separate logical meaning. Solutions are primarily obtained through a default inference mechanism- a built-in depth-first search with backtracking,which can directly support a general graph search.

Logic programs are very concise - it is common for a program written in a procedural language to require five to ten times more source code than the corresponding logic program.

The declarative style of programming is very natural, making programming easier, and requiring smaller time for program development. Logic programming is very close to writing a program specification, so it is much easier to address the issues of correctness and verification than in procedural languages [7]. Independence of clauses is advantageous for incremental development and program maintenance. It also facilitates rapid prototyping. Many logic programs are invertible, e.g. a sorting program can be run "backwards" to produce permutations.

We have been particularly interested in those features of logic programming languages which

are important for efficient implementation of our parallel algorithms.

Logic programming languages in general and Prolog in particular can naturally express a large number of different types of parallelism [10]. The most renowned types are AND and OR parallelism. In general, AND parallelism is the ability to execute two conjunctive tasks in parallel; OR-parallelism is the ability to execute two disjunctive tasks in parallel. In terms of logic programming, the task has the granularity of a goal execution, i.e. a procedure call and execution.

We have selected CS-Prolog as the most suitable for implementation of our algorithms.

CS-Prolog (Communicating Sequential Prolog) is a parallel logic programming language, based on sequential Prolog extended by concepts of process, communication and time.

CS-Prolog is similar to occam-2 and 3L Parallel-C in a sense that all these languages are based on the Hoare's concept of CSP (Communicating Sequential Processes) and are developed for parallel computer systems with non-shared memory and message passing - such as multitransputer systems.

In CS-Prolog it is possible to assign a process to a goal and to execute the goal as a Prolog program concurrently with other goals (processes). Communication and synchronisation of these concurrent processes are done by messages. The processes can be suspended waiting for messages and they can send messages to activate other waiting processes. Unlike in occam-2, the communication is asynchronous, i.e. the sender process can continue running without waiting for reception of a message. Processes can be created and deleted during program execution. Unlike most other parallel logic programming languages [7], CS-Prolog program can generate alternative solutions by backtracking. The CS-Prolog interpreter is actually a set of independent Prolog interpreters running on different transputers and communicating by messages. Beside the typical built-in predicates of sequential Prolog, CS-Prolog provides a number of special predicates supporting the notions of process, communication and time. The most important ones from the viewpoint of our application are:

new (G,N,S,E,T)

A new process is created with goal G and name N on processor T. The starting local time is S and the resolution of G is terminated by the local time E. The N,S,E,T arguments are optional.

send(M,PL)

The calling process sends message M to the processes being on the process list PL.

wait_for(M)

The caller waits for a message which is unifiable with M.

message_arrived(X)

If there is a message sent to an active process and unifiable with X then it succeeds, otherwise fails.

advance(T)

The local time of the calling process is incremented by T.

4.2. Hardware Environment

We have implemented the parallel connectivity analysis algorithms on a distributed-memory (message-passing) parallel computing system consisting of eight T8oo transputers connected in two stages to a root T8oo transputer linked directly to a host PC (Figure 6). The most important rationales for selecting this hardware architecture were the following:

- Availability of the CS-Prolog interpreter and compiler;
- Computing power: the transputer is one of the most powerful and cost-effective microprocessors that are commercially available;
- Large and fast main memory: with over 4 Gbytes of addressable off-chip memory (100ns) and 4 Kbytes of on-chip memory (50 ns), every transputer node in the distributed transputer system can hold all the modelling code and database representing a network.
- Expandability: a transputer-based system can be expanded by connecting each transputer to four other transputers and by adding more transputers to such an array incrementally;

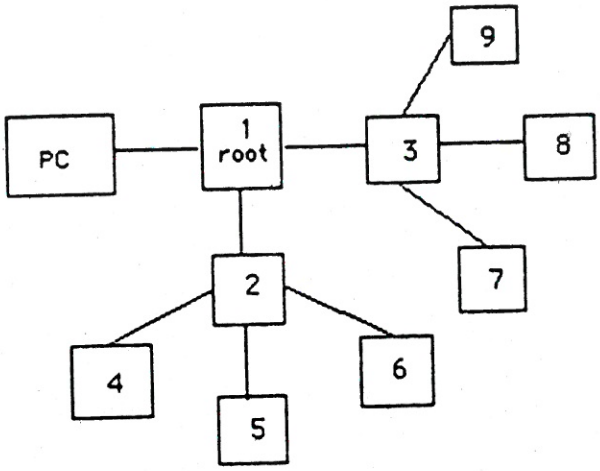

Figure 6. Configuration of the experimental multitransputer system

- Cost: a single transputer equivalent in computing power to a Vax 780 minicomputer costs less than a PC;
- Topology: a multitransputer system consists of a root transputer linked directly to a PC and up to 3 branch transputers connected to the root. Each of the branch transputers may have up to three branch transputers and so on. This topology corresponds directly to the balanced-tree, hierarchical topology.

5. Example of Connectivity Analysis of a BH Topology Network

We will describe now a CS-Prolog program for connectivity analysis of a network with 3 level BH topology (Figure 7). At the top level the network (called supernet) is represented as two interconnected clusters called internet1 and internet2. Each of these clusters is decomposed at the lower level into the interconnection of three clusters, which are in turn shown at the bottom level as networks of several nodes connected by links. They are represented in CS-Prolog declaratively by facts of the form :

node(R,N).	% N is the identifier of a node of network R.
link(R,N1,N2).	% link between nodes N1 and N2 of network R.
interlink (R1,N1,R2,N2).	% a link between node N1 of network R1 and % node N2 of network R2.
superlink (S1,R1,N1,S2,R2,N2).	% a link between node N1 of network R1 of internet1 and % node N2 of network R2 of internet2.

Bidirectional link (or interlink) is represented by a pair of links (or interlinks) in which one link has the node identifiers reversed, e.g.:

```
link(R,N1,N2).
link(R,N2,N1).
```

A cluster at any level is created by the **new** process creating predicate. For example, to create a supernet with the name **supernet** consisting of two internets S1 and S2, which contains the terminal nodes N1 of the network R1 and N2 of the network R2, on transputer 1 we will use :

```
new(supernet(S1,R1,N1,S2,R2,N2), supernet , - , - ,1).
```

The presence of a connection between any two nodes N1 and N2 of a single network R1 can be checked by the first definition of the **internet** goal:

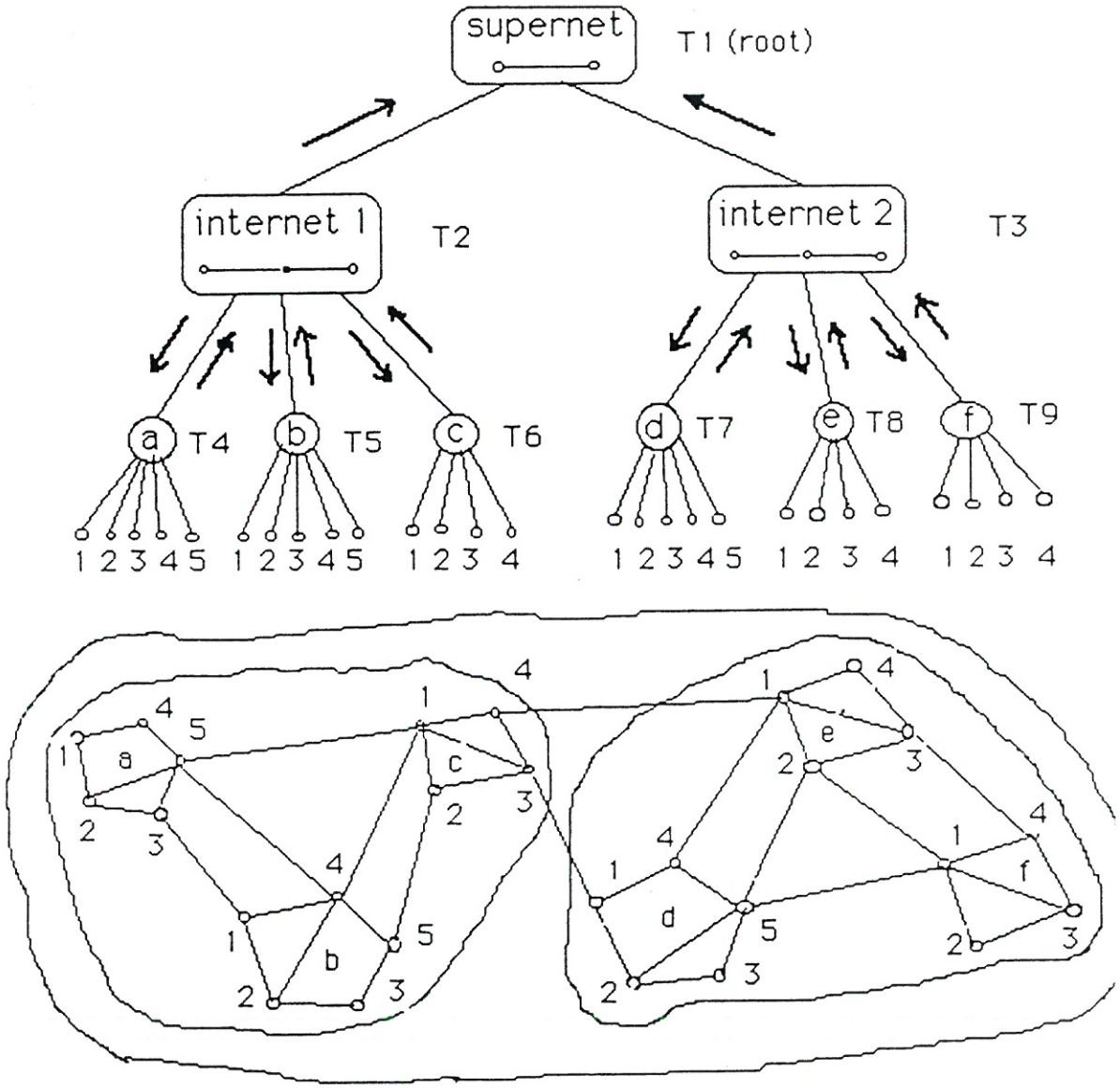

Figure 7. Topology of the example network and its distribution on the multitransputer system

```
internet(R1,N1,R1,N2) : -send(path(R1,N1,N2,P)),
wait_for(MR1), write_inside(["route between",
R1,N1, "and", R1,N2, "is", MR1]), nl.
```

The **send** predicate is used to send a request to the process representing network R1 for checking a connection between nodes N1 and N2. The **internet** process is then suspended by the **wait_for** predicate, waiting for the response MR1 from the process R1. The response is then used to print the list of nodes and links between the nodes N1 and N2. If no connection is possible due to a missing node or link fact (representing that node or link failure) the network process sends to the internet process a "disconnected" message.

Connections between two nodes R1,N1 and R2,N2 through some intermediate nodes R1,N3 and R2,N4 (representing gateways between the networks) of two adjacent networks R1 and R2 connected by an interlink R1,N3,R2,N4 are checked by the second definition of the **internet** predicate :

```
internet (R1,N1,R2,N2) : -interlink(R1,N3,R2,N4),
    send(path(R1,N1,N3,P),[R1])),
    send(path(R2,N4,N2,P),[R2])),
    wait_for(MR1),  write_inside(["route
    between",R1,N1, "and", R1,N3, "is", MR1]), nl,
write_inside([R1, "and",R2,"are connected by
interlink", R1,N3,R2,N4]), nl, wait_for(MR2),
write_inside (["route between", R2,N4," and ",R2,
N2,"is",MR2]), nl.
```

The two **send** predicates are used to send requests to processes representing the networks R1 and R2 for checking their internal connections. The **wait_for** predicates are used to receive their responses which are then used to print the list of nodes, links and interlinks composing the required connection. In case of missing node or link facts (representing node or link failure) the appropriate network process sends a "disconnected" message to the **internet** process. This can be used for location of a faulty node or link in the analysed internet.

Similarly, the third definition of the internet is used to check a connection between two nodes of any two not-adjacent networks R1 and R2 connected through an intermediate network R3 :

```
internet(R1,N1,R2,N2) : -interlink(R1,N3,R3,N5),
send(path(R1,N1,N3,P), [R1])), wait_for (MR1),
write_inside(["route between", R1,N1,"and",
R1,N3,"is",MR1]), nl,
write_inside ([R1,"and",R3, "are connected by
interlink", R1,N3,R3,N5]), nl, internet(R3,N5,
R2,N2).
```

This is a recursive definition, so it may be used for any number of interconnected networks. The worker processes representing particular networks have the following general form :

```
R :- wait_for(path(R,X,Y,P)), path(R,X,Y,P),
     send (P,[internet]).
R :- message_arrived(_), send(disconnected,
     [internet]).
```

The first definition is used to receive a message from the internet process with specification of the path to be checked. If the path exists, a list specification of the path is returned to the internet. Otherwise the message "disconnected" is returned.

The **path** goal checks availability of the requested nodes and links :

```
path(R,X,X,P) :-    !, node(R,X), P = [node(R,X)].
path(R,X,Y,P) :-    node(R,X), link(R,X,Y),
                    node(R,Y), P = [node
                    (R,X), link(R,X,Y),
                    node(R,Y)].
path(R,X,Y,P) :-    node(R,X), !, node(R,Y),
                    link(node(R,X,Z),
                    P1 = [node(r,X),
                    link(R,X,Z)], path(R,Z,Y,P2),
                    append(P1,P2,P).
```

The first definition checks availability of a single node, the second definition checks availability of a direct connection between two adjacent nodes and the last recursive definition is used to check availability of a path between two not-adjacent nodes. It contains an **append** predicate which is used to concatenate a sequence of nodes and links represented as lists.

Availability of a route e.g. between node(a,2) and node(c,3) can be checked by entering the following goal :

check(a,2,c,3).

If the nodes are connected the program will print a message specifying the route:

route between a2 and a1 is [node(a,2),link(a,2,1),node(a,1)]

a and b are connected by interlink a 1 b 1

route between b1 and b2 is [node(b,1),link(b,1,2),node(b,2)]

b and c are connected by interlink b 2 c 2

route between c2 and c3 is [node(c,2),link(c,2,3),node(c,3)]

It is possible to check availability of an alternative route by backtracking and requesting an alternative solution. If a component of a route is not present in the model (e.g. link(a,2,1)) then the program will print instead a message :

route between a2 and a1 is disconnected

and the specification of the remaining, correct part of the route.

The program for parallel Monte Carlo reliability analysis is a version of the above described program extended in the following way. It is assumed that the probability of a link failing during a time interval (represented by a simulation run) is LF. During a simulation run whenever the program checks the availability of a link, a random number F is generated, whose probability density function is uniform between 0.0 and 1.0. If the number is less than LF, the link is down, so an alternative path has to be attempted. Because the path is generated by a recursive procedure it is important to define it in such a way that for every attempted link the random number is generated only once during the simulation run. This is achieved by using the **once** predicate in the **state** subgoal of the modified path definition given below :

```
path(R,X,Y,LF,P): -  node(R,X),link(R,X,Y),
                     !,state(R,X,Y,LF),node
                     (R,Y), P=[node (R,X),
                     link(R,X,Y), node(R,Y)].

path(R,X,Y,LF,P): -  node(R,X),!,node(R,Y),link
                     (R,X,Z),state(R,X,Z,LF),
                     P1 = [node(R,X),
                     link(R,X,Z)], path(R, Z,Y,
                     LF,P2), append(P1,P2,P).

state (R,X,Y,LF) : -  once(random(F)),F>LF.

once(Goal) :      -  Goal,!.
```

The **state** subgoal is used to find the current state of a link (up or down). It is preceded by a **cut**, to ensure program termination if an adjacent link is down.

At the beginning of a simulation run a user can enter the value of LF for this run , and the program generates the available route between any two given nodes.

The user can specify the number of simulation runs for a given value of LF and of increments of the LF value for consecutive run sets to find the probability of network disconnection as a function of LF. Moreover, the implemented program generates the length of the available route in terms of hops or propagation delay, so it is possible to find the shortest available route.

6. Performance of the Connectivity Analysis Algorithms

Connectivity analysis algorithms are actually special cases of graph search procedures, so we can evaluate their performance in a similar way to that used for graph search procedures. We will estimate in the following the performance of the node-to- node connectivity algorithms,which can be easily supported by CS- Prolog (using depth-first search or breadth-first search).

This performance depends on the following factors [12]:

The size of the search space;
The branching factor of the search tree.

The search space is defined to be the set of all possible states of the world under study. Knowing the size of the search space is insufficient however in deciding which search procedure is the most suitable. More information for such a decision provides the branching factor. It is defined as the average number of next states of any problem state. The branching factor can be used to estimate the size of the search tree, which enables us to measure the performance of any search procedure (assuming that no distinction can be made between the next states of the decision tree).

Generally, the performance of a search procedure (or any procedure) can be measured by the following accounts:

The amount of memory space needed in running the procedure (space complexity);
The time consumed in running it (time complexity).

With any search procedure, the main concern in memory space is the size of the stack needed to store the path being developed (in case of our implementation,but it seems to be typical) . On the other hand, the running time is proportional to the number of iterations generated by the search process.

To facilitate the measurement, let us call stack size S the estimated number of states contained in the paths that need be stored, and iteration count I the estimated number of iterations generated by the search process.

Let B be the branching factor of our search problem. Then the search tree has 1 node at level 1, B nodes at level 2, B^2 nodes at level 3,.... B^{K-1} at level K.

Also, let N be the estimated length of a solution-path. Then, the stack size S and iteration count I are calculated as follows:

When considering our experimental network (Figure 7) as a flat topology network with N = 8 and B = 3:

a) Depth-first search stores only one path at any time, so the stack size is N. On the other hand, the iteration count is

$$I = 1+B+B^2+.....+B^{N-1} = \frac{B^N-1}{B-1}$$

For our network

S = 8 I = 3280.

b) Breadth-first search stores all developed paths, but it extends only one path at a time. Thus, after the first iteration the stack contains B paths of length 2; after the (1 + B)-th iteration, B^2 paths of length 3; after the (1 + B + B^2)-th iteration, B^3 paths of length 4, and so on. So to find a solution path of the estimated length N, the stack size is B^{N-1} x N and the iteration count is

$$I = 1+B+B^2+.....+B^{N-2} = \frac{B^{N-1}-1}{B-1}.$$

For our network

S = 17496 I = 1093.

Reformulation of the flat topology into a 3 level BH topology lets us do the search hierarchically. The values of the B and N parameters for our network become:

B = 2 and N = 1 at level 3

B = 2 and N = 2 at level 2

B = 3 and N = 3 at level 1

The hierarchical search may be done either sequentially or in parallel,with depth-first or breadth-first search at every level.

In case of the sequential hierarchical search the performance measures become:

(a) for depth-first search

S = 23 I = 85

(b) for breadth-first search

S = 182 I = 26

Mapping the hierarchical search algorithm onto a tree structured parallel system like the one we have used,gives the following estimates:

(a) for depth-first search

S=23 I=17

(b) for breadth-first search

S=182 I=5

We have implemented the hierarchical,parallel depth-first search because it is much simpler than the breadth-first search (it is possible to use for this the Prolog's built-in inference engine) and requires much less memory.

The above analysis does not take into account the overhead inherent in the actual implementation, the most important component of which in our case is the message passing time. We have measured the actual execution time of the same CS-Prolog implementation of our algorithms in three hardware configurations- with 1,3 and 9 transputers.In the first case all processes were running on the root transputer,in the second case the top level process was created on the root while the first and second level processes were created on the second level of the binary hardware tree and the last case was as described on page 226.

The other possible 3 transputer case with the third and second level processes on the root and the first level processes on the second level of the hardware tree gave virtually identical results as the first 3 transputer case. The appropriate execution times,the corresponding speed-ups and processor efficiencies are shown in Table l. The speed-up is calculated as the ratio of T(l)/T(N),where T(l) is an execution time on a single processor and T(N) is the execution time on N processors.The processor efficiency measures the average contribution of each processor to the parallel solution when N processors are employed and is calculated as the ratio of speed-up(N)/ N.

Table 1. Performance parameters

No. of Processors	1	3	9
Execution time (msec)	1.49	0.61	0.44
Speedup	-	2.44	3.39
Processor efficiency	-	0.81	0.38

7. Conclusion

We have demonstrated that by reformulation of network topology and parallelization, the connectivity analysis of large networks becomes computationally tractable. The hierarchical parallel search used by our algorithms provides considerable improvement over a flat search used by the sequential algorithms - it finds a solution much faster and needs much less memory. The main contribution to this result is made by reformulation of the topology.Further improvement is obtained by executing the hierarchical search on a parallel system.Even better results could probably be obtained by using more efficient implementation language-like parallel C.

The actual speed-up obtained on the parallel system was not as high as its estimation due to communication overhead. Moreover,it turns out that increasing the number of processors beyond some value does not improve much the speed-up, because the processor efficiency drops.This is typical for tree structured distributed systems with message passing and confirms validity of the model and the conclusion reported in [8].

REFERENCES

1. ANTONIO, J.K. et al, **A Fast Distributed Shortest Path Algorithm for a Class of Hierarchically Structured Data Networks**, Proceedings of the IEEE INFOCOM'89, Vol.I,1989, pp. 183-193.
2. BARCHANSKI, J.A., **Parallel Modelling of Computer Networks for Connectivity Analysis**, Proceedings of the International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS'93, La Jolla, CA., January 1993,pp. 287-290.
3. BARCHANSKI,J.A.,**Increasing Flexibility of Simulation by Reformulations**,Studies in Informatics and Control,Vol.l, No.3, September 1992, pp.199-213.
4. BERTSEKAS,D. and GALLAGER,R., **Data Networks**, PRENTICE HALL, 1992.
5. **CS-Prolog, Version 3.25**, Multilogic Computing Ltd, Budapest, 1991.
6. FUTO, I. and KACSUK, P. , **CS-Prolog on Multitransputer Systems**, MICROPROCESSORS AND MICRO- SYSTEMS, Vol. 13, No. 2, March 1989, pp. 103-112.
7. LAZAREV,G.L., **Why Prolog-Justifying Logic Programming for Practical Applications**, PRENTICE HALL,1989.
8. SREEKANTASWAMY,H.V. et al, **Performance Prediction Modelling of Multicomputers**,Technical Report 91-27, November 1991,Dept.of Computer Science,University of British Columbia.
9. TANNENBAUM, A. , **Computer Networks**, PRENTICE HALL, 1981.
10. TICK, E. , **Parallel Logic Programming**, MIT Press, 1991.
11. TSAI,W.T., **Control and Management of Large and Dynamic Networks**,Ph.DThesis, Department of EECS,University of California,Berkeley,CA.,1985.
12. VAN LE, T., **Techniques of Prolog Programming**, JOHN WILEY, 1993.