Mapping Programs on a Torus–like Transputer Network

Peter Kolbezen and Peter Zaveršek
University in Ljubljana, Ljubljana, Slovenia

This paper proposes a torus based network architecture which consists of multiple, unidirectional rings. Such a system can be made very flexible by combining the proposed architecture, adequate program graph presentation, and a suitable allocation algorithm. The task allocation and reconfiguration is carried out dynamically. Some possible process allocation algorithms are considered and an experimental verification of these algorithms is performed according to the proposed multitransputer network.

Keywords: Transputer-based systems, mapping and allocation algorithm, interconnection networks

1. Introduction

The importance of parallel computing hardly needs emphasis. It is widely believed that the only feasible path towards higher performance is to consider radically different computer organizations, in particular ones exploiting parallelism. One of the simplest and most promising types of parallel machines is the well known multiprocessor architecture. Of all the techniques used to improve processing throughput, parallel processing must be one of the simplest to design in hardware but the most challenging in software. Parallelism can be fine-grained as in array processors, coarse-grained as in machine networks, or medium-grained as in the transputer systems.

A lot of problems associated with multiprocessor systems design using conventional microprocessors center on the use of centrally shared resources through which communications among processors and processes take place. Hardware provisions must be incorporated in such systems to resolve multiple simultaneous requests for accessing the shared resources and the software must guarantee mutual exclusion in executing critical regions of programs where shared variables are being used (HWANG and BRIGGS 1984). As number of processors and processes increases, the management overhead increases as well and bottleneck situations become increasingly more prominent. The combined effect of all these is to limit the system's performance so that the incremental performance improvement achieved with every additional processor decreases fairly sharply. Transputer-based multiprocessor systems eliminate most of these problems by not having shared resources at all.

Many multiprocessor systems are based on the idea of static allocation of processors, which means the algorithm should be analyzed very carefully and a decision must be made which processor to place a certain process onto and at which time instance to do it. As we see we must have a complete knowledge of the algorithm (exact process execution times and relations between processes) on one hand and of the process network on the other hand as well.

Our contribution to solving the problem stated above is based on a multiprocessor system which could allocate processes dynamically having poor knowledge about the algorithm and knowing even less about the processor network itself.

The accent of our work was to exam a possibility of efficiently executing irregular algorithms on a regular architecture. Efficiency of a processor network is superior if the system and the program algorithm are matched. Matching can be achieved either by careful analysis of the program and/or configuring multiprocessor network to suit the program algorithm. We considered achieving efficient mapping of different algorithms onto a given fixed structure network.

2. Algorithm Presentation

We assume that algorithms are presented in traditional way (sequential programs). An algorithm (program) which requires optimal execution on a multiprocessor system must be carefully analyzed and decomposed into a set of processes to exploit the parallelism. One of the necessary conditions of a program for parallel processing is that the program possesses many parallel paths.

Methods to detect the parallelism in Algol-like languages are known. There is no limit to the amount of parallelism detected, hence the need to harness it. A tool using a specification language (SL) is introduced which does it (KATTI and MANWARING 1989). This tool outlines a procedure to detect parallelism in sequential programs (SP), expresses this parallelism and achieves efficient parallel execution of the sequential program on a multiprocessor architecture.

Data flow analysis is used to detect parallelism in Algol-like language SL, where basic units are variables (or processors). As with each program so also with SP there is associated some flow graph G(SP) that carries the information of control flow.

As a rule in our examples all program statements in SL are defined on the medium-grained level. Any maximal group of consecutive program statements is called *a block* if entry to this portion of the program is possible only through the first statement (called *the block head*), and if exit is possible only through the last statement (called *the block tail*), and there is at most one control statement present in this group. It is clear that all control statements represent block tails and any statement following a control statement must be a block head. All assignment statements in the block represent *the interior of a block*.

In the block diagrams the nodes may be atomic, such as adders or multipliers, or non atomic, such as digital filters, FFT units, multipliers, modulators, phase locked loops, etc. The complexity of the function (the granularity) will determine the amount of parallelism available. The block function may also be a matrix multiplication. If so, the systolic array of processors can be realized with the association of more local processor sets in the architecture (SZTURMOWICZ and TUDRUJ 1989) (reconfigurable architecture). This kind of architecture also permits concurrent processing of the blocks.

The data flow graph of a block is an acyclic directed graph. Such blocks can be efficiently executed on the proposed transputer network.

An algorithm can generally be split into processes. How many processes we shall split the algorithm into, which we call a granulation of an algorithm, depends greatly on the number and capabilities of the employed processors. We define T_{exe} as execution time of a graph node, and T_{trans} as the time required for interprocessor data and program code transmision. The granulation degree influences the ratio T_{exe}/T_{trans} which we call the transmission time ratio. We wish to keep this ratio as large as possible. We can define a minimal the transmission time ratio. value which can give acceptable results. We can, thus, granulate the algorithm properly to match the prescribed ratio as much as possible.

We intended to develop a mechanism for run-time allocation of processors. The algorithms which will be executed on a processor network should be presented in a proper form to enable efficient allocation of processors. We selected the presentation form of data driven graphs which must not be cyclic, must be *simple* and no communication may take place between two concurrently executed processes. Algorithms are widely presented in a form of data flow graphs (DFG) which are cyclic in general. A conversion of a cyclic DFG into a convenient form which we named a *Hierarchical Acyclic Data Flow Graph* (HADFG) may be performed by following stages:

1. Remove cycles from the DFG. Cycles must be decomposed into a sequence of acyclic blocks or should be performed on a higher level (e.g. as proposed in (SZTURMOWICZ and TUDRUJ 1989)).

2. Simplify the blocks. Acyclic DFG obtained by stage 1 should be simplified. Simple graphs expose a property of stating completely defined time relations between processes (i.e. between nodes in the graph) (HWANG 1984). There are many possible approaches to graph simplification problem depending on time characteristics of certain processes within the acyclic block. Process time characteristics may be completely known, may vary within certain bounds, or may be unknown. It is possible to simplify the graph in either of the three cases, although we must emphasize that the output graphs tend to be least complex when time characteristics are completely defined. We may notice that the knowledge about some graphs can be loose due to unknown node execution time characteristics despite completely stated time relations between nodes. The simplification procedure for acyclic DF graphs has already been automated by recent work of our group.

3. Convert acyclic and simple blocks into HADF graph. Blocks of simple DF graphs are ready to be converted into a HADF graph form. HADF graph is a graph which consists of alternating sequential and parallel levels. Parallel HADF graph levels consist of branching nodes with branches that can be executed concurrently. Sequential levels include the nodes with outgoing branches which must be executed sequentially. HADF graph exactly matches the mathematical description of simple, acyclic data flow graph.

We can compare DFG and HADF graph in fig. 1. Fig. 1a shows the original ADFG. An equivalent HADF graph is shown by fig. 1b, where thick lines represent sequential and thin lines parallel levels. To clear up the example we also expose mathematical presentation according to (HWANG and BRIGGS 1984) of the same graph:

$$x = x_1 \times [(x_4 + x_5) \times (x_7 + x_8) + (x_3 + x_2) \times x_6] \times x_9$$

It can clearly be seen that the mathematical formula perfectly matches the HADF graph, where x_i represents process node i.

Now we will use to introduce a notion of *sorted graphs*. We must notice that all branches leaving a single concurrent node are equivalent as they represent concurrent processes. Consequently, their position within a node may be changed and the nodes may be sorted according to the execution time of the outgoing branches. The node is qualified to be sorted if the branch exposing the longest execution time is placed to the leftmost and the branch stating the shortest

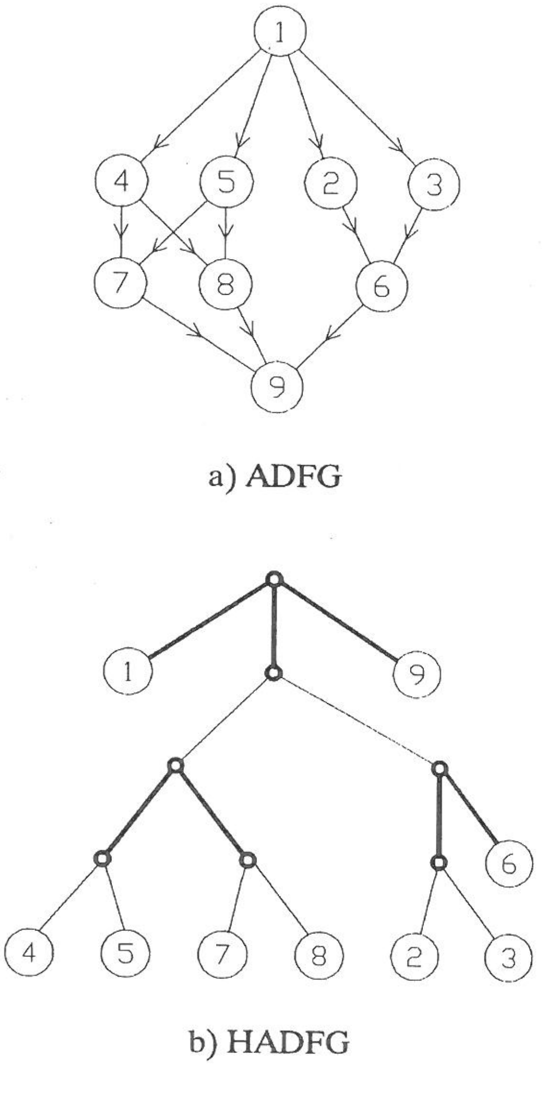


Fig. 1. DFG and equivalent HADF graph

execution time to the rightmost position within the node. If this procedure is performed on the entire HADF graph from its bottom to the top, then we say that the whole HADF graph is sorted (KOLBEZEN and ZAVERŠEK 1991).

2. Physical Configuration

The basic topological element of the proposed multi-processor network is a *ring* of processors. Two or more equal rings may be wrapped together by additional rings which connect equally positioned processors in adjacent rings. Such a form of multiple equal cross-linked rings is widely called a *torus*. Processing elements can be transputers which perfectly fit this type of topology. Four connections needed to ensure paths to each of the four neighbours are obtained by four link-pairs on each transputer. Transputer boards are commercially available containing four transputer devices connected into a ring. The rest of the links are available for custom connection to other devices (INMOS 1988).

Each physical ring we consider as two unidirectional communication rings which do not have to be of the same size. We only demand that each processor be connected to two physical rings, which may also be obtained by short connecting two links of a single processor. This fact may be useful e.g. when a processing device breaks down. Damaged devices can simply be removed from the network and links should be short connected to establish closed ring communication paths for the non-damaged devices (KOLBEZEN and ZAVERŠEK 1993). Fig. 2 shows some valid network examples.

In the considered networks (Figure 2) all 4 link-pairs for each transputer are employed. Such networks could be conected to the "outside word" (i.e. to a front-end computer) by one of two possible ways:

– by adding a new "input-output transputer" in one of links, or

– by adding a new medium to the manufacturer-provided transputer links — shared common memory for one or more transputers within the network and the front-end (host) computer. Access to this shared memory, by each of the input-output transputers in a network and the host computer, is controlled by specially developed arbitration logic.

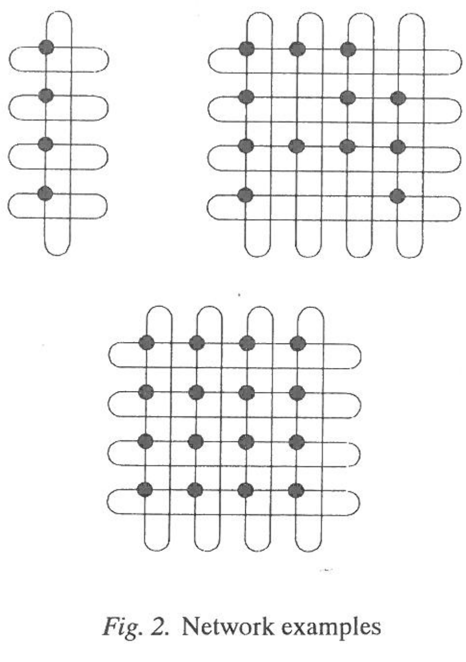


Fig. 2. Network examples

Message passing

Message passing mechanism is based on token passing principle. As we consider each physical ring as two unidirectional communication rings, the employed tokens are also divided into *control and data tokens*. Control tokens pass the control commands and system status words, and data tokens pass the data packets. Control tokens transfer should be fast to reduce latency and improve the array response time. Data transfers take longer and, thus, transmitting both groups of tokens using the same paths could degrade system's performance. There are many possible approaches for balancing the two factors, e.g. using separate communication paths, priority, multiplexing messages, use of virtual channels, etc. We used the simplest possible method by applying a separate communication ring for either group of messages.

Traffic effect

Transputers are devices which can communicate over the links at several different speeds from 5 up to 20 Mbits^{-1} over each of four available links in either of two possible directions. Peak message transfer rate depends on the transputer used, e.g. comparing T800 and T414 we can notice that T800 provides an improved communication protocol and, thus, enables higher data transfer rates. In order to utilize full bandwidth of the links, almost 50% of the external bandwidth is used in DMA transfers by the link controller. If data is buffered into internal memory the percentage is somewhat smaller (15%) (JESSHOPE 1988).

A practical message traffic effect investigation based on T414 transputer was presented in reference (WARING 1990). The results show that a single, unbroken stream of data packets causes no more than 10% reduction in computational capacity and that the reduction does not expose a high dependence to message packets' length. When having more streams of data, reduction in computational capacity rises rapidly as message length is shortened. For 1-byte messages and more than two streams the reduction tends to be almost 100%. However, if the message length is about 1024 bytes the reduction is not worse than 7% for single and not worse than 12% for multiple data streams.

3. An Allocation Example

The most suitable network for execution of HADF graph is a treeshaped network. Considering the graph in fig. 1, we can construct a network of processing elements which fully exploits the stated parallelism (see also ZAVERŠEK and KOLBEZEN 1992). A possible example of an adequate network is shown in fig. 3a. This is neither the only possible, nor the optimal network. The network can be reduced as shown in fig. 3b without harming the network performance. Time analysis may show that the execution of process pairs (4,5) and (7,8) takes much longer than the sequence (2,3) and 6. In this case the network may be reduced further (fig. 3c). If the exact time characteristics are not completely known it is hard to compose the optimal network and the network can easily exeed the required size.

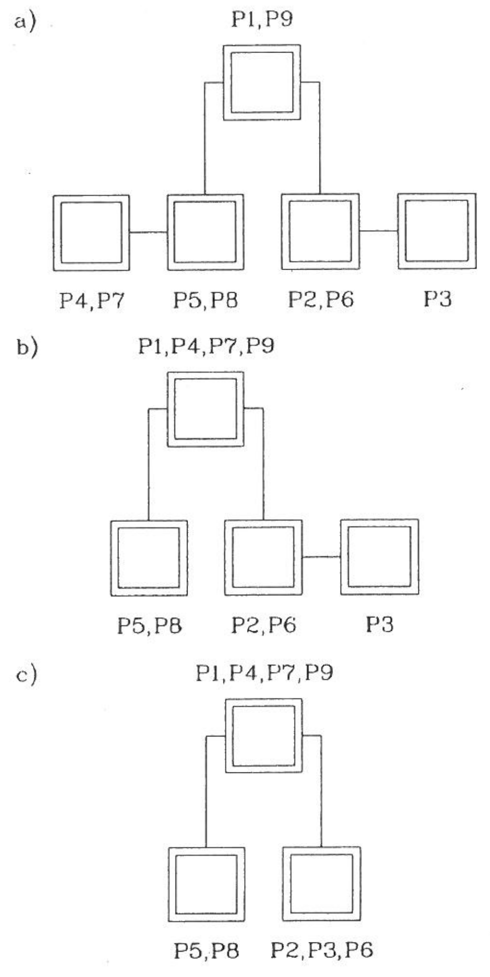


Fig. 3. Process alloocation example

Our idea was to perform a tree-like allocation employing a ringbased processor network to override the above stated drawback. Processors in such a ring-network always have a successor, which is not the case when observing the tree-network. Ring-based architecture does not have to be constructed for a particular graph of explicitly stated depth and width but is virtually scalable according to current graph dimensions. Process mapping can be automated enabling efficient run-time allocation of processes with unknown time characteristics. One can notice that it would be easier to execute HADF graphs on a reconfigurable architecture where the network could be dynamically rearranged. We believe, however, that our system could be competitive because of its hardware simplicity. The delays due to ring control communications in the proposed system may well be comparable to time delays which a matrix switch would need to establish the demanded connection. Delays caused by more dense traffic between processors may be reduced by prescribing acceptable transmission time ratio (algorithm granulation).

4. Proposed Allocation Algorithm

The proposed allocation algorithm is based on HADF graph representation of program algorithms. A HADF graph instructs the allocation routine exactly which processes to execute in parallel and which in sequential manner.

The allocation procedure is started at the topmost node of the program graph, at the root node. Consequently, the processor which performs the allocation is called a *root processor*. When a new graph or a sub-graph is encountered on a processor, it must be analyzed first to determine which nodes can be allocated to processors concurrently and which ones sequentially. The allocation procedure proceeds down the HADF graph tree taking the leftmost branch in each sequential node. All the nodes finding in branches on a parallel level are allocated to queues of waiting processes except the leftmost branch. If the leftmost outgoing branch terminates at a node that is not a process, then the above procedure is repeated. However, if the leftmost branch ends at a process then the process is executed on the current processor and nodes waiting in the queues are allocated to available resources.

We have already mentioned the ring as the basic topological element of our network. All further actions and notions are subordinate to this point of view. Processors may be allocated onto at most two processor levels relative to the observed processor which we call a root processor. *The first processor level* is composed by processors which fit into the two rings that cross the root processor. *The second processor level* includes all the rest of the processors in the network which are, as we must notice, not directly reachable from the position of the root processor. The two levels are illustrated in fig. 4.

We must notice that notion of the root is relative and, hence, we may find more than one root processor in the network at the same time. When a node from the process waiting queue is allocated to an idle processor and a node turns out to be a root point of a subgraph (the node is not a process), then a part of the original HADF graph, i.e. the subgraph in question, is transferred to the newly allocated processor. The allocated processor becomes a root processor of a subgraph. Consequently, more than one root processor may appear in the network. We can conclude that the original program graph is being partitioned. Graph partitions, i.e. subgraphs and processes, are being allocated to remote processors causing program code and process data to be distributed over the network.

Allocation of processors is performed on every root processor as follows. Let's presume that a processor received the program HADF graph and that the concurrent nodes were allocated to waiting queues already. The first step is then to locate and seize an idle processor. This is done by sending appropriate control token via control ring. We allocate processors in both available directions, horizontal and vertical, simultaneously. As soon as the resource is seized, program code and data are transferred to the allocated processor. Search for idle processors primarily takes part on the first processor level. If no idle processor is found on the first processor level then the second processor level is examined. To do so, the root processor instructs the first level processors, one after another, to search the first processor level excluding processors from the root's first level for unused processors. The search process is successful if an idle processor is found and unsuccessful if all the first and the second level are examined and the result is negative. Upon successful termination the search procedure is repeated, otherwise the search procedure is stopped, and we have to wait for some processors to be set free. When an arbitrary processor terminates execution of its process or subgraph respectively, it sends a control token to announce to all other members that it is free to receive another request. Concurrently it starts returning execution results, while program code and data are rejected as they are not needed any more. Processors with non-empty waiting queues can then repeat their demands and the allocation procedure is repeated. Such an algorithm is able to employ all the processors in the network if only there were enough processes waiting in processors' queues.

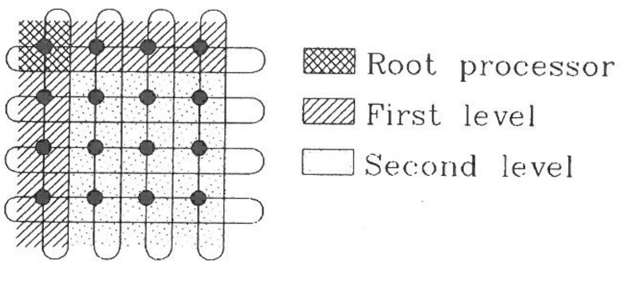


Fig. 4. First and second processor level

5. Simulation

We have observed the behaviour of the allocation procedure on a simulator that was built for our particular case. When building the simulator the increase of the computional time caused by communication was taken into account. We presumed 10% reduction of computational power when passing messages in only one direction and 15% reduction when passing message streams in two directions. The two selected values are slightly higher than those stated in (WARING 1990) to catch the allocation algorithm overhead. Our interest concerned the acceptable granulation of he algorithm (i.e.

transmission time ratio) and the most appropriate network configuration.

When preparing for simulation at the first step a set G_n of 20 random HADF graphs was generated. The graphs were sorted and an additional set G_s of 20 sorted HADFG's was obtained. The simulation set $G = G_n \cup G_s$ contained the total of 40 graphs in two functionally equivalent subsets of 20 graphs each.

Execution of the two subsets was simulated on several different configurations of processor network $a \times b$; $a, b \in \{1, 2, 3, 4\}$. We have split the simulation into three phases.

The first phase of simulation was to establish the most suitable processor configuration which we would decide to take into account for further simulation.

The second phase was to state the acceptable transmission time ratio by varying the ratio in steps 1,2,5,10,...,1000 and observing the final execution time. The process transmission time was properly modified for the whole set of test graphs after establishing acceptable transmission time ratio.

The emphasis of the third phase was in observing the efficiency of execution of sorted and unsorted graphs at selected processor configurations and transmission time ratio according to

– execution time

– efficiency of processor network.

Definitions

Minimal transmission time ratio

The time taken for executing a certain HADF graph on a certain network greatly depends on the transmission time ratio. In general, a higher ratio provides a shorter execution time and vice versa. We define the minimal transmission time ratio as the nearest value of ratio T_{exe}/T_{trans} where the time taken for executing the HADF graph is not more than twice as long as the best time (at ratio 1000) when varying the ratio.

Graph execution time

Graph execution time is the first measure of the allocation algorithm success. Besides network configuration it depends on algorithmic properties of the graph as well. To avoid the affecting the algorithmic properties of particular test graphs, we define a sum Sk(Gj) of all observed HADF graphs' execution times as a measure:

$$S_k(G_j) = \sum_i T_i$$
$$k \in \{2 \times 2, 3 \times 3, 4 \times 4\}$$
$$G_j \in \{G_n, G_s\}$$

$T_i \ldots$ execution time of i-th HADF graph from subset G_j ($i = 1, 2, ..., 20$).

Efficiency of the processor network

Efficiency E defines the execution cost ratio when comparing a single- and multi-processor system where a higher efficiency signifies a lower execution cost.

$$E = \frac{T_s}{n \cdot T_a}$$

$n \ldots$ number of processors in the network
$T_a \ldots$ average processor busy time
$T_s \ldots$ sequential execution time of a HADF graph

$$T_s = \sum_i T_{p_i}$$

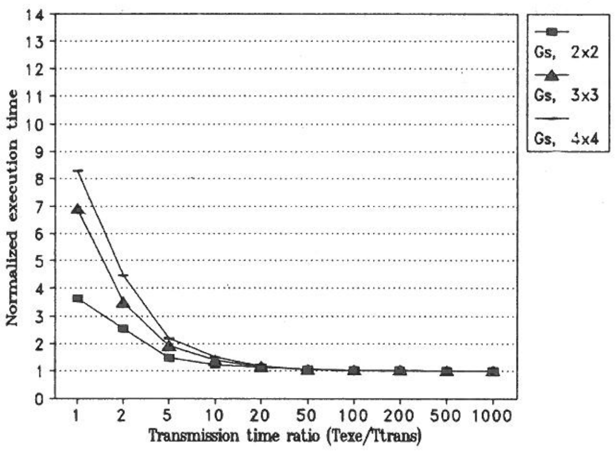
$T_{p_i} \ldots$ execution time of i-th process in a graph

Simulation results

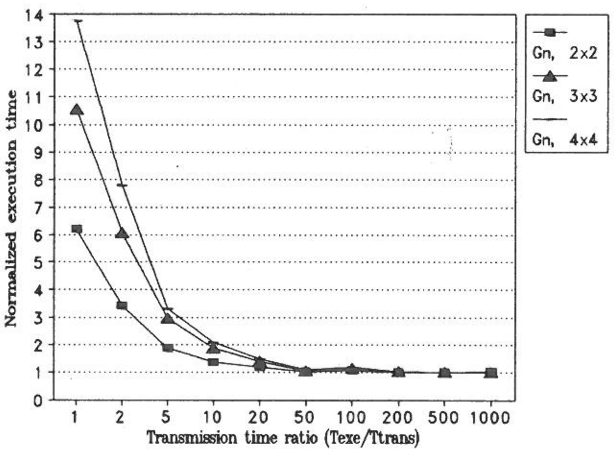
Simulation has shown that the best results can be achieved on a square processor network configuration. Such a configuration provides equal interprocessor distances and the smallest maximum traffic density. Thus, for further research we adopted square configuration networks 2×2, 3×3 and 4×4.

Fig. 5 shows the transmission time ratio and how it influences the execution time. We must notice that the minimal transmission ratio, varies in effect of examined configuration and graph set. The value of the ratio is lower on smaller networks and sorted graphs and higher when considering larger networks and unsorted graphs. Our further simulation was based on the T_{exe}/T_{trans} ratio value of 10.

Fig. 6 shows the execution time sum $S_k(G_j)$ vs. network size. Execution time is much shorter for the sorted graph set. This is due to communication overhead when allocating processes or



a) sorted graphs



b) unsorted graphs

Fig. 5. Normalized execution time vs. ratio

subtrees of greater size to remote resources. We conclude that sorted graphs provide significant improvement in contrast to the unsorted ones. Consequently the network efficiency (fig. 7) is much better in the case of sorted graphs as well.

Unsorted graphs may represent algorithms with completely unknown process execution time characteristics. In practise it is often possible to estimate approximate execution time length, so we believe that typical results for our set of graphs should find place somewhere between the upper and lower simulated bounds. Somewhat small efficiency can be explained by two facts. The first is that some of the test graphs could not possibly exploit all the processors in the array because they do not contain enough parallelism. The second is the fact that the graphs were generated on a random basis and that some of them could be dominated by a sequential component.

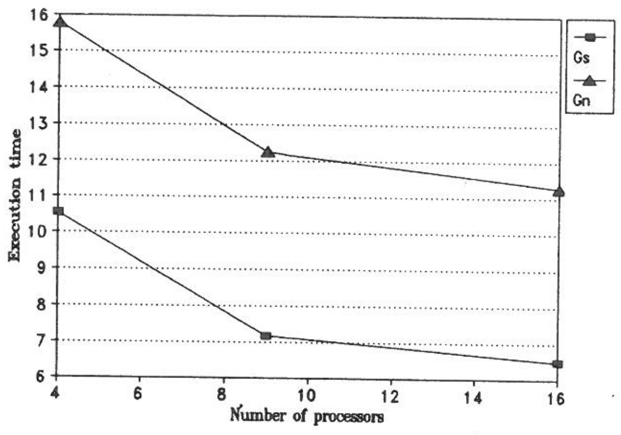


Fig. 6. Absolute execution time

5. Conclusion

The proposed multiprocessor system described a possible approach for executing irregular program algorithms employing a regular network architecture. Program algorithm should be presented in a tree-like HADF graph form. Further the proposed run-time allocation algorithm enables efficient mapping of tree-like HADF graphs onto a torus-like network providing efficient process allocation. When comparing tree- and torus-like networks, we conclude that the torus can better serve the processor utilization due to virtual network scalability. A simulation

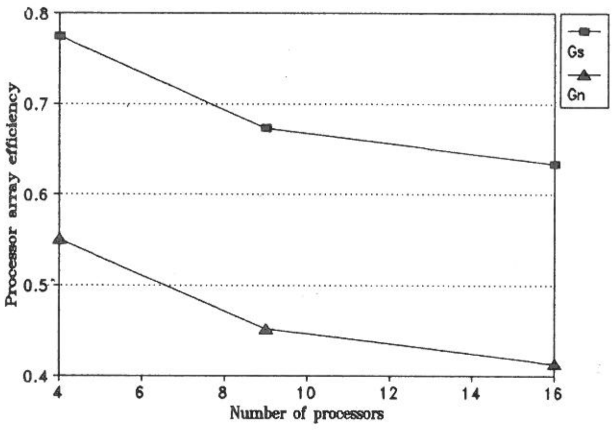


Fig. 7. Network efficiency

based network analysis assuming transputers as processing elements was performed. Some factors such as processor network form, array dimensions, transmission time ratio and HADF graph subset were compared vs. graph execution time and efficiency of the processor network. The square network form was generally found to be the most appropriate although some specific configurations may achieve better performance for a particular HADF graph.

The proposed data driven processor network system may be found useful in applications where process execution times are not explicitly stated (static), but can vary within certain bounds. The system does not employ resources by creating waiting queues on each one of them, but rather seizes a resource after it is set free. This may be a limitation of the system. Thus, to avoid unwanted delays, we demand the execution time to be much greater than the processor seizing and data transmission time. The value of transmission time ratio should be held as high as possible; it is not advisable to be less than 10 for a 4×4 network.

Although transputers were adopted as processing elements this is not necessary, neither is significant the type of employed transputer. The results are applicable to any kind of system resources. We must bear in mind, however, that transmission time depends on system's transmission bandwidth and that execution time is dominated by processing capabilities of employed resources. Transmission ratio will change depending on the system characteristics, so we must partition the graph properly to match our system.

In case of need for executing more independent tasks we can establish appropriate numbers of network entry nodes.

References

K. H. HWANG, F. A. BRIGGS (1984) *Computer architecture and parallel processing*, McGraw-Hill Book Company, Chaps. 7 & 8.

INMOS SPECTRUM (1988) Product information, *The Transputer Family*.

C. JESSHOPE, Transputers and switches as objects in OCCAM (1988) North-Holland, *Parallel Computing* 8 , 19–30.

R. S. KATTI, M. L. MANWARING (1989) Executing sequential programs in parallel on a multiprocessor architecture, Presented at the *Proceedings of the 3rd Annual Parallel Processing Symposium*, March 29–31, pp. 385–400, California State University, Fullerton, California.

P. KOLBEZEN, P. ZAVERŠEK (1991) A hierarchical multimicroprocessor system, *Informatica*, A Journal of Computing and Informatics, The Slovene Society Informatika, Vol. 15, Nr. 1, 65–76

P. KOLBEZEN, P. ZAVERŠEK (1993) A Fault-Tolerant Torus-Based Transputer Network, Presented at the *Proceedings of the Second Electrotechnical and Computer Science Conference* ERK'93, Vol. b, pp. 139–142, Portorož, Slovenia.

M. SZTURMOWICZ, M. TUDRUJ (1989) A multi-layer transputer network for efficient execution of OCCAM programs, North-Holland, *Microprocessing and Micro-programming*, Vol. 28 , 133–138.

L. C. WARING (1990) A general purpose communications shell for a network of transputers, North-Holland, *Micro-processing and Microprogramming*, Vol. 29, 107–119.

P. ZAVERŠEK, P. KOLBEZEN (1992) Dynamic Allocation on the Transputer Networks, *Parallel Processing*: CONPAR 92-VAPP V (L. BOUG et al., Eds.) Lyon, France, *Lecture Notes in Computer Science* 634, 825–826

Received: November, 1993
Accepted: January, 1994

Contact address:
Peter Kolbezen, Peter Zaveršek
University in Ljubljana, "Jožef Stefan" Institute
Department of Computer Science and Informatics
Jamova 39, 61111 Ljubljana, Slovenia
Phone: +386 61 12–59–199
Fax: +386 61 219–385, 273–677
Tlx: 31 296 jostin si
E-mail: peter.kolbezen@ijs.si

PETER KOLBEZEN received B.Sc., M.Sc. and D.Sc. degrees in electrical engeneering from the University in Ljubljana, Slovenia, in 1957, 1968 and 1974, respectively. Since 1957 he joined the "Jožef Stefan" Institute, Ljubljana, where he is a researcher and since 1956 the head of research grupe at the Department for Computer Science and Informatics. Since 1989 he has been assistant professor and since 1989 full profesor at the Faculty of Electrotechnics and Computer Science from University in Ljubljana. His main interests are digital networks, computer architectures, parallel processing and computer control.

PETER ZAVERŠEK received B.Sc. and M.Sc. in the area of computer control of industrial processes and computer science from the University in Ljubljana, Slovenia, in 1989 and 1993, respectively. In 1989 he joined the Industrial concern "Gorenje", Velenje, and simultaneously he obtained the post of a research assistent at the Department of Computer Scence and Informatics at the "Jožef Stefan" Institute. His research and development interest have been in the areas of parallel and distributed computing, communications systems and computer control.