Self-adjusting Mapping: A Heuristic Mapping Algorithm for Mapping Parallel Programs on to Transputer Networks*

H. SHEN
Department of Computer Science, Åbo Akademi University, Lemminkäisenkatu 14, SF-20520 Tuku, Finland

The problem of mapping parallel programs on to multiprocessor systems is a fundamental problem of great significance in parallel processing. In this paper we propose a fast heuristic algorithm to solve this problem on transputer networks. Our mapping algorithm mainly contains three modules: grouping, placement and routeing, where grouping puts processes in the program into tasks which can be one-to-one placed on to processors in the transputer network, placement sets the grouped tasks on to the processors and routeing produces edge-disjoint physical communication paths for logical communication requirements. The algorithm works by combining three modules under a self-adjusting scheme towards a successful mapping result. For mapping n processes in an arbitrary parallel program on to m processors in a transputer network of grid structure, our algorithm has a worst-case time complexity $O(\max\{n^2, m^5\})$ under full adjusting, $O(\max\{n^2, m^4\})$ under semi-adjusting and $O(\max\{n^2, m^2\})$ under no adjusting, where the last holds only for the transputer networks providing message routeing and multiplexing. The algorithm has been implemented in Occam on the Hathi-2 transputer system.

Received September 1989, revised August 1991

1. INTRODUCTION

A fundamental problem of great significance in parallel processing is mapping parallel programs on to parallel computers. When the parallel programs are written in Occam[9] and the parallel computers are transputer-based networks,[8] this mapping problem becomes extremely important because of some properties of the Occam language which require that the programmer should have knowledge of the detailed network configuration before he can run an Occam program on a transputer networks. A mapping algorithm for transputer networks can take the duty of mapping parallel programs on to the transputer networks automatically, thereby hiding the network hardware configuration from the programmer.

Although a variety of approaches to solving the mapping problem have been made in the literature,[3-7, 11-16] no fast deterministic solution has been found yet because this problem has been shown to be NP-hard in general.[7] In order to have a fast but sub-optimal solution to this problem, we must take into account necessary heuristic strategies when we develop a mapping algorithm. Literally, heuristic solutions are mainly based on the techniques of local (neighbourhood) search[1] and simulated annealing[10] that require carrying out a series of batched data-swapping (enormous data migration), and therefore usually cannot be realised efficiently in practice. Moreover, without the requirement of edge-disjointness of the physical paths to be routed, most of the literature approaches, unlike ours, care mainly for the quality of task scheduling (grouping and placement) but not the realisation of the final routeing.[7, 11, 14, 15] Some of them are only available for task graphs and processor graphs with specific topologies.[4, 7]

In this paper we will propose a novel heuristic approach, *self-adjusting mapping*, for solving the mapping problem on transputer networks. Our mapping algorithm, which has been successfully implemented in Occam on the Hathi-2 transputer system,[20] can be easily and efficiently realised in practice. We shall begin by introducing the mathematical models for the mapping problem and different strategies to approach the solution to this problem, then describe our self-adjusting mapping algorithm to solve this problem on transputer networks, and finally show the time complexity, implementation result and performance evaluation of the algorithm. The work reported here is a part of the MILLIPEDE project, aimed at developing a programming environment for the transputer system. An earlier version of this paper appeared in Shen.[17]

* This work was supported by the FINSOFT III Research Programme.

2. THE MAPPING PROBLEM AND MAPPING STRATEGIES

The problem of mapping a parallel program on to a parallel computer is that of allocating processes and communication channels in the program on to processors and physical communication links in the machine such that the execution time of the program is minimised. This problem is also called the *process-to-processor mapping problem.*

2.1 Mathematical models for the mapping problem

A parallel program and parallel computer can be described as a task graph $G_t(T, E_t)$ and processor graph $G_p(P, E_p)$ respectively,[7, 10] where a 'task' is a set of processes of the program and originally is a single process. For simplicity and without loss of generality, we assume that both G_t and G_p are undirected and without self-loops. In G_t, node set T and edge set E_t respectively represent tasks and communication channels between the tasks, while node weight at node t_i, denoted by w_{t_i}, and edge weight between adjacent nodes t_i and t_j, denoted by $e_{t_i t_j}$, respectively represent known or estimated computation amount of t_i and communication amount between t_i and t_j. We can form different tasks and change the structure of G_t by grouping processes under different strategies. In G_p, node set P and edge set E_p respectively

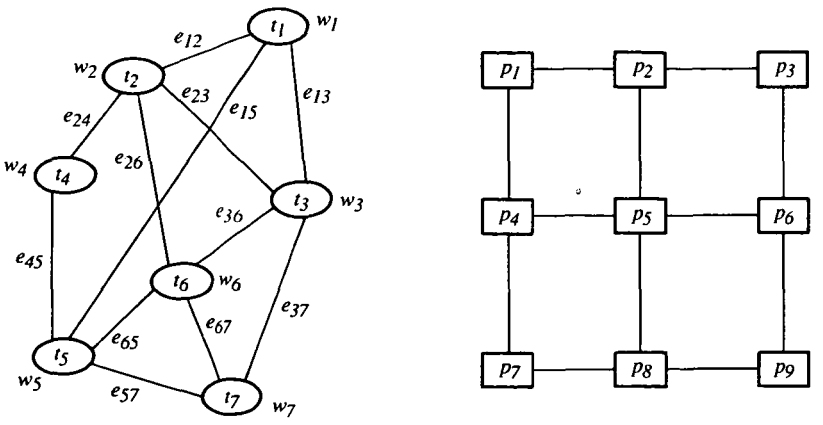


Figure 1. Task graph (*a*) and processor graph (*b*).

represent processors and physical communication links between the processors (we assume that all processors have the same computational power). Figure 1 gives an example of a G_t and G_p.

Now the mapping of the parallel program on to the parallel computer becomes the problem of task-to-processor mapping from G_t to G_p. This can be expressed as a function $\pi: T \rightarrow P$. The mapping π is said to be optimal if it minimises the completion (total execution) time of the parallel program. The mapping problem is therefore concluded to find such an optimal mapping π.

Mathematically, the parallel program completion time can be characterized by a cost function. Thus the mapping problem is to find a mapping π which minimises the cost function.[11] Normally there are two different approaches to defining the cost function, as follows.[14]

2.1.1 *Maximum cost approach*

There are two kinds of load carried by all processors in G_p under mapping π: computation load and communication load. During the execution of the program, each processor at any time is in one of the following states: computation, communication and idle waiting (due to the communication delay). Let the number of processors be m, the time for processor p_i spent in computation, communication and idle waiting be $\Gamma_{\text{comp}}(p_i)$, $\Gamma_{\text{comm}}(p_i)$ and $\Gamma_{\text{idle}}(p_i)$ respectively, $1 \leqslant i \leqslant m$. The completion time for processor p_i is

$$\Gamma(p_i) = \Gamma_{\text{comp}}(p_i) + \Gamma_{\text{comm}}(p_i) + \Gamma_{\text{idle}}(p_i) \quad 1 \leqslant i \leqslant m. \tag{1}$$

$\Gamma_{\text{comp}}(p_i)$ can be measured accurately by pre-calculating the computation load for p_i. The computation load for p_i, denoted by $L_{\text{comp}}(p_i)$, is the total computation amount of all tasks mapped on to p_i:

$$L_{\text{comp}}(p_i) = \sum_j \{w_j \mid (\pi(t_j) = p_i) \wedge (1 \leqslant j \leqslant n)\}, \quad 1 \leqslant i \leqslant m. \tag{2}$$

Assume that the time for unit computation is δ_{comp}, then we have

$$\Gamma_{\text{comp}}(p_i) = \delta_{\text{comp}} * L_{\text{comp}}(p_i), \quad 1 \leqslant i \leqslant m. \tag{3}$$

$\Gamma_{\text{comm}}(p_i)$ can be measured approximately by pre-calculating the communication load for p_i. The communication load for p_i, denoted by $L_{\text{comm}}(p_i)$, is the total weighted communication amount of all communication channels (edge weights) that go through p_i under mapping π. Let d_{ij} be the distance between p_i and p_j, i.e. the shortest length of the physical path between p_i and p_j, under mapping π. Each edge with an edge weight e_{kl} in G_t is re-weighted by the physical path length it traverses in G_p under mapping π. Therefore we have

$$L_{\text{comm}}(p_i) = \sum_{j=1}^{m} \sum_{k,l} \{e_{kl} * d_{ij} \mid (\pi(t_k) = p_i) \wedge (\pi(t_l) = p_j) \wedge (1 \leqslant k, l \leqslant m)\}, \quad 1 \leqslant i \leqslant m. \tag{4}$$

Assume that the time for unit communication is δ_{comm}, then

$$\Gamma_{\text{comm}}(p_i) = \delta_{\text{comm}} * \Gamma_{\text{comm}}(p_i), \quad 1 \leqslant i \leqslant m. \tag{5}$$

Formulation (5) is approximate since it does not consider the communication cost for intermediate processors to transmit messages.

$\Gamma_{\text{idle}}(p_i)$ is determined by synchronisation delay during program execution, therefore it can only be measured by simulating the execution of the program under mapping π, which seems too difficult to be done in practice. A simple but inaccurate way is to substitute $\Gamma_{\text{idle}}(p_i)$ with some fixed value, for instance the maximum possible idle time.

The maximum cost approach takes the maximum completion time of all processors as the program completion time. In this approach, the cost function is expressed as follows:

$$\begin{aligned} \text{Cost} &= \max_{1 \leqslant i \leqslant m} \{\Gamma(p_i)\} \\ &= \max_{1 \leqslant i \leqslant m} \{\Gamma_{\text{comp}}(p_i) + \Gamma_{\text{comm}}(p_i) + \Gamma_{\text{idle}}(p_i)\}. \end{aligned} \tag{6}$$

2.1.2 *Summed cost approach*

Another alternative approach to defining the cost function is *summed cost approach*. Let the ideally average load state be such an ideal state that each processor has the same computation load as all others, no communication load and no idle waiting. The summed cost approach takes the cost for all processors at the ideally average load state as the 'zero cost', therefore the cost

for all processors at the actually allocated load state under mapping π can be measured in the sum of the cost deviation among all processors from the ideal state.

The sum of the cost deviation includes three parts: computation load deviation D_{comp}, communication deviation D_{comm} and idle waiting deviation D_{idle}. Assume that the computation load for each processor at the ideal state is $\bar{L}$, then

$$\bar{L} = \frac{\sum_{i=1}^{m} L_{comp}(p_i)}{m}. \quad (7)$$

Since the communication load and idle waiting are zero for each processor at the ideal state, the formulation of the cost function here becomes

$$\text{Cost} = \delta_{comp} * D_{comp} + \delta_{comm} * D_{comm} + D_{idle}, \quad (8)$$

where

$$D_{comp} = \sum_{i=1}^{m} |L_{comp}(p_i) - \bar{L}|, \quad (9)$$

$$(\text{or} \quad D_{comp} = \frac{1}{m} * \sum_{i=1}^{m} (L_{comp}(p_i) - \bar{L})^2)$$

$$D_{comm} = \sum_{i=1}^{m} L_{comm}(p_i), \quad (10)$$

$$D_{idle} = \sum_{i=1}^{m} \Gamma_{idle}(p_i). \quad (11)$$

2.2 Strategies to solve the mapping problem

Strategies suggested in the literature to solve the mapping problem can be roughly classified into three categories: topological mapping, cost optimisation mapping and heuristic mapping.

2.2.1 Topological mapping

Mapping from task graph $G_t(T, E_t)$ on to processor graph $G_p(P, E_p)$ is called topological mapping, if neighbouring tasks in G_t get assigned to neighbouring processors in G_p under the mapping. Clearly, the mapping problem can be solved optimally by using topological mapping, if such a mapping exists. The topological mapping problem is known to be equivalent to the graph isomorphism problem,[7] that is, NP-hard in the general case.

Topological mapping can be used in the specific case that both G_t and G_p are very regular graphs,[7] but it cannot be expected to be applied to the general case that G_t and G_p are arbitrary graphs.

2.2.2 Cost optimisation mapping

Since the mapping problem can be formulated mathematically as a cost optimisation problem to minimise a cost function as discussed in Section 2, this problem can be solved also by applying mathematical programming techniques, a process called cost optimisation mapping. Like topological mapping, cost optimisation mapping can reach an optimal solution, but usually it is NP-hard.

There are two keys that decide the quality of the result for cost optimisation mapping, one is a group of cost function formulae which gives a measurement for the quality of the result and the other is a group of optimisation criteria which describes the search space and decides the search speed for the optimisation. The cost function formulae, as we discussed before, should be chosen to be able to approach the theoretical measurement for the mapping quality as accurately as possible and to be expressed in a form as simple as possible. The optimisation criteria should be chosen to be able to reduce the search space and increase the search speed as much as possible without loss of correctness in the searching.

2.2.3 Heuristic mapping

Since topological mapping and cost optimisation are usually NP-hard, though they can lead to an optimal solution, they are therefore not practically applicable in most cases. In order to obtain a fast but sub-optimal solution, we have to turn to heuristics. Mapping under heuristic strategies works much faster but less accurately than the previous two approaches. In practice, heuristic mapping is often based on one or both of the previous two. For instance, one can apply the cost optimisation mapping with some heuristic optimisation criteria, which may produce a satisfactory result.

Mapping can be performed either statically before a program is executed if the processes and communication channels of the program are static, i.e. they are known before the program run-time and never changed during the run-time, or dynamically in an adaptive manner during the program run-time if the processes and communication channels are dynamic, i.e. they can be changed during the run-time. In the second case, the mapping is called adaptive mapping, which can be realised in many ways with different strategies.[11] This paper only considers the first case, under the assumption that both the task graph and processor graph are static.

3. A MAPPING ALGORITHM FOR TRANSPUTER NETWORKS

We now turn to seeking a practical solution for the process-to-processor mapping problem in the case that the parallel computer is a transputer-based network such as the Hathi-2 parallel computer. We propose here a fast heuristic mapping algorithm which can be easily applied to transputer networks.

3.1 The Hathi-2 transputer system

Hathi-2 is a general-purpose transputer-based MIMD multiprocessor system built by the Department of Computer Science at Åbo Akademi University and the Technical Research Centre of Finland at Oulu (VTT/TKO).[2] The system consists of 25 identical boards, each with four 32-bit Inmos T800 transputers, one 16-bit Inmos T212 control transputer and one Inmos C004 crossbar switch.[8] The connection network of Hathi-2 is a combination of a fixed, two-dimensional torus configuration and a distributed switching network formed by C004 switches. The intra-board connection of the system is shown in Fig. 2, which leads the C004 switches to form a distributed switching network connecting the communication links of the T800 transputers. This

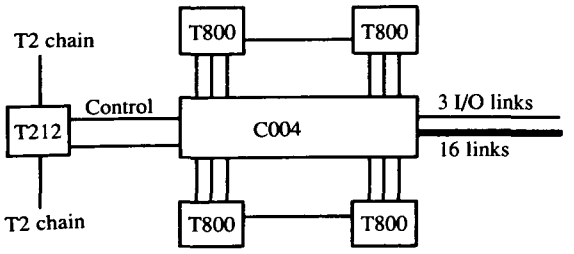


Figure 2. The intra-board connection in Haithi-2.

allows the system topology to be changeable (reconfigurable) by software. The inter-board connection of the system is shown in Fig. 3: 24 boards are statically connected in a configuration of 4×6 torus (another board is used as a separate partition of the system), where each board is connected to its four neighbours via the C004 switch communication links. The T212 control processors are connected to each other in a ring, where each T212 is also connected to one C004 switch, thus forming a separate control system which controls setting of the switches. The Hathi-2 system provides a multi-user environment by partitioning the system into several independent subsystems each of which can be used by one user via a host (computer system). The host provides I/O to the multiprocessor and interaction with the user. The user's program is edited, compiled, linked on the host and run on the transputer network. Hathi-2 has a performance of 150 Mflops or 1000 Mips.

For the problem of mapping parallel programs on to transputer networks, taking some properties of the transputer networks into consideration is helpful. One common property for transputer networks at present is that a transputer network normally provides a user with only a single I/O port via the host transputer, thus the task containing the maximum I/O should be mapped on to the I/O port in the network in order to reduce the communication delay caused by transmitting I/O messages.

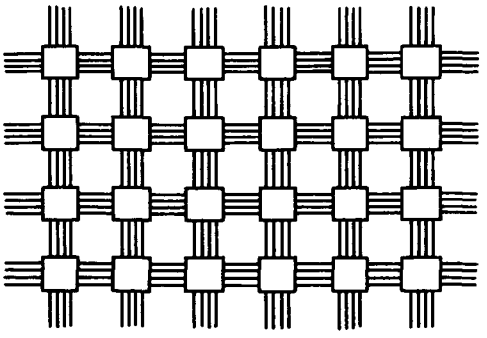


Figure 3. The inter-board connection in Haithi-2.

Our mapping algorithm for transputer networks, namely self-adjusting mapping, involves three modules: grouping, placement and routeing. They work co-operatively by a self-adjusting strategy. These modules and the self-adjusting strategy are described individually in the following subsections.

In order to avoid the need for message routeing and multiplexing, we require that all paths to be routed by the routeing module are edge-disjoint paths, which means that each physical link can belong to only one communication path. This will, however, make our mapping algorithm more complex than those without this requirement, because path-disjoint routeing[18] is much more difficult to realise than conventional (shortest) path routeing.

3.2 The grouping module

Assume that a parallel program is represented in a task graph by a special analyser, where each task is initially a static process of the program. A task containing the maximum I/O is marked as the host task. The number of links of each processor is d, the number of static processes is n and the number of processors is m. The job for the grouping module is to group the initial tasks (static processes) in the task graph into target tasks, so that after grouping the tasks in the task graph can be placed on to the processors in the processor graph one to one.

Obviously, in order to minimise the program completion time, the parallelism for program execution should reach maximum while the inter-processor communication, which is much more expensive than intra-processor communication, should be kept to the minimum. Note that for transputer networks the communication time includes link setup time and message transfer time, where message transfer can be executed concurrently with processor operation.[8] Thus we have the following criteria for grouping.

(1) The number of tasks is not greater than and as close as possible to the number of processors.

(2) The number of communication channels in the task graph is as small as possible.

(3) The load for all tasks is well balanced.

The algorithm of the grouping module, load-balanced grouping, is briefly described as follows.

(1) If the number of tasks in the task graph is greater than m, do grouping in the task graph such that after grouping the number of tasks is not greater than and close to m.

(2) Check all tasks in the task graph. If the number of edges connecting task $t_i (1 \leqslant i \leqslant m)$ is greater than d, do grouping within the subgraph including t_i itself and all its neighbours such that after grouping the number of neighbours of t_i is not greater than d. The above procedure is continued until no task in the task graph has more edges than d.

(3) For all tasks in the task graph, check the amount of computation and the amount of communication for each task, do necessary grouping such that after grouping the amount of computation and the amount of communication both for each task and for all tasks are well balanced.

Figure 4 provides an example of grouping when $n = 10$, $m = 9$ and $d = 4$, where (a) is the task graph and (b) and (c) are different grouping schemes which depend on the amount of computation and the amount of communication of the tasks.

3.3 The placement module

The job of the placement module is to place the tasks in the task graph on to processors in the processor graph one-to-one, such that after placement the inter-task

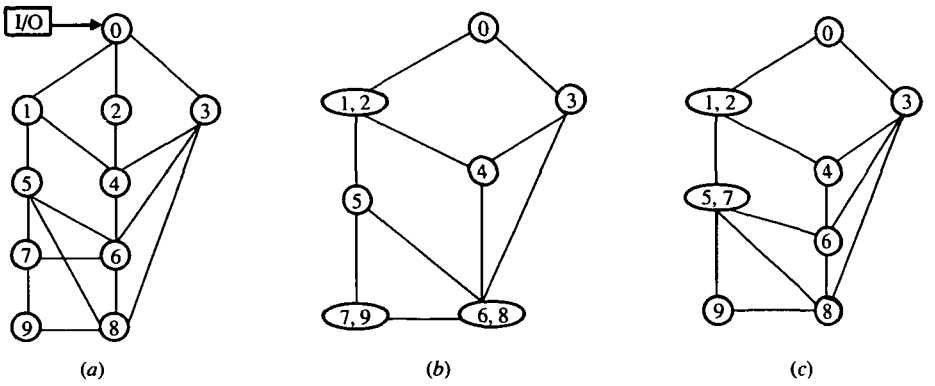


Figure 4. An example of grouping. (*a*) Task graph; (*b*) first grouping; (*c*) second grouping.

communication requirements in the task graph can be efficiently realised by physical communication paths among processors in the processor graph, where a path is a chain of communication links.

The criteria for placement are the following.

(1) If there exists an embedding from the task graph to a subgraph of the processor graph, the placement can lead to it.

(2) The total length of the physical communication paths to fulfil the inter-task communication requirements is minimum.

An intuitive idea for this module is to place tasks in the task graph on to processors in the processor graph according to the neighbour-first method, that is, if task t_i is placed on to processor $p_{\pi(i)}$, then the neighbours of t_i in the task graph should be placed on to the processors that are closest to $p_{\pi(i)}$ in the processor graph.

Our placement algorithm, neighbour-first placement, is described as follows.

(1) Place the host task on to the I/O port processor (the host/master transputer).

(2) For each task t_i which is placed on to processor $p_{\pi(i)}$, do the following neighbour-first placement until all tasks have been placed:

(*a*) If the number of neighbours of task t_i is one, place the neighbour on to the processor closest to processor $p_{\pi(i)}$.

(*b*) If the number of neighbours of task t_i is greater than one, placement should be done in such a way that the total length of physical paths used for the communication among t_i and its neighbours is minimum.

Figure 5 shows an example of placement from Fig. 4(*c*) to a 3×3 torus.

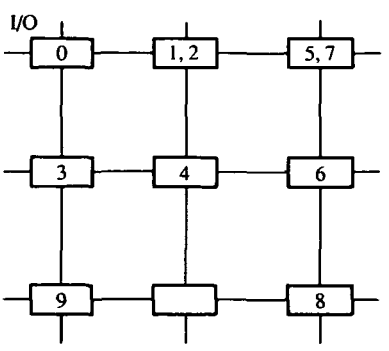


Figure 5. Placement from Fig. 4(*c*) to a 3×3 torus.

3.4 The routeing module

The job of the routeing module is to allocate the physical links and to build the edge-disjoint physical communication paths for the inter-task communication requirements. It produces a layout of the embedded subgroup of the processor graph from the task graph.

Clearly, in order to achieve the minimum communication delay, routeing should produce a layout of the embedded subgraph whose total edge length is minimum.

The routeing algorithm we apply here is a fast heuristic algorithm for path-disjoint routeing in a grid structure which can produce a routeing layout with minimum total path length and fewest total path bends, where a bend on a path is a 90° angle on the path in the grid structure.[18,19]

Our routeing algorithm is based on two heuristic criteria, routeing order criterion and route selection criterion, which are motivated by the observation that for path-disjoint routeing the routeing order for all paths and the route selection for each path are two key issues to the routeing result, and they should therefore meet certain criteria. For a given problem, routeing paths in a wrong order may result in routeing failure, as may choosing a route for a path in a wrong way.[18]

Figures 6 and 7 show the importance of the routeing criteria. For the routeing requirements in Fig. 6(*a*), routeing order (r_1, r_2, r_3, r_4) affects the routeing result as described in (*b*) and (*c*), where touteing is assumed to proceed in the shortest way.

For the routeing requirements in Fig. 7(*a*) under the routeing order r_1, r_2, the route selection for each $r_i, r = 1, 2$ also affects the routeing results, as described in (*b*) and (*c*).

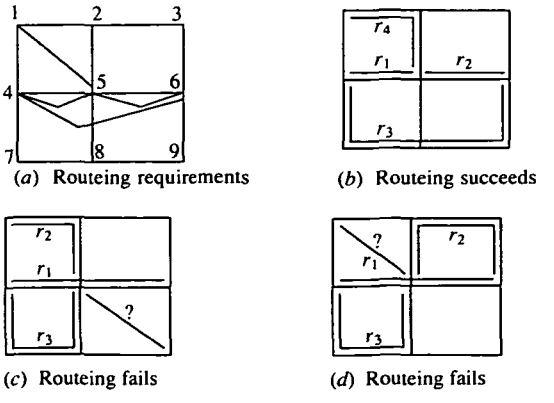
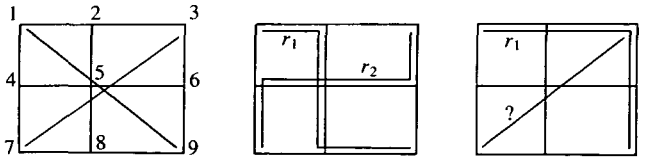


Figure 6. The importance of the routeing order criterion.



(*a*) Routeing requirements (*b*) Routeing succeeds (*c*) Routeing fails

Figure 7. The importance of the route selection criterion.

Assume that we want to build k communication paths $r_1, r_2, \ldots, r_k$ in the processor graph. The basic idea of our routeing algorithm, path-disjoint routeing, is described as follows.

(1) Give the local heuristic information for each path, for instance the weight assignments to different routes and different bends in the route for each path.

(2) Give the global heuristic information for all paths, namely the routeing order for all paths that is assumed to be $\phi(i)$ for r_i, i.e. $r_{\varphi(i)}$ is routed before $r_{\varphi(i+1)}$, $1 \leqslant i < k$.

(3) Find the possible route for path $r_{\varphi(1)}, r_{\varphi(2)}, \ldots, r_{\varphi(k)}$ based on the local heuristic information such that the final layout has minimum length and fewest bends.

The sketch of the algorithm, path-disjoint routeing, is briefly described as follows.

(1) *Initialisation.*

(2) *Routeing order calculation*

  (*a*) Calculate heuristic information.

  (*b*) Calculate the routeing order according to the routeing order criterion.

(3) Do route finding and route collection for each of the paths in the routeing order.

  (*a*) Route finding: find a physical route for the path according to the route selection criterion by using the minimum cost and fewest bends search.

  (*b*) Route collection: take the built route away from the processor graph, update heuristic information.

3.5 Self-adjusting mapping

A possible mapping algorithm consists of three modules, grouping, placement and routeing, as described above. However, even if grouping and placement are very powerful in function, it is still difficult to guarantee that they can lead to a successful embedding on the processor graph. Therefore, a self-adjusting idea is proposed such that if the routeing module cannot find a successful embedding, the placement module should be able to adjust itself and give a new adjusted scheme according to the heuristic information provided by the routeing module. Further, if all the necessary adjusted schemes of placement can still not lead to a successful embedding, the grouping module should also be able to adjust itself according to the heuristic information and produce a new adjusted scheme. One obvious criterion for self-adjusting is that the new schemes after each self-adjusting step both for placement and for grouping should increase the possibilities for a successful embedding to be produced by the routeing module.

A failure path is one that fails to be routed by the routeing module. A failure node is a node in G_p that has to be reached or passed by more paths embedded from G_t than the edges physically connected to the node. Here a node is said to be either reached by a path if it is an end-node of the path, or passed by a path if it is an intermediate node on the path. Routeing therefore fails on each failure node. Let P_f and N_f stand for failure path set and failure node set, whose elements are all failure paths and failure nodes produced by the routeing module, respectively. Assume that P_i is the path set including all paths that have to reach or pass failure node $f_i \in F$, $1 \leqslant i \leqslant \xi$. We call a node in G_p occupied node (unoccupied node) if a (no) task has been placed on it after placement. N_o and N_u stand for occupied node set and unoccupied node set, which contain all occupied nodes and unoccupied nodes in G_p after placement, respectively.

A path p is said to be terminal exchangeable if exchanging its terminals (end-nodes) with unoccupied nodes and/or terminals of another path, q, where p and q belong to the same P_i w.r.t. failure node f_i, will reduce the number of failure paths in P_f.

Our self-adjusting strategy for placement and grouping is a strategy to reduce $|P_f|$, and finally to remove all paths from P_f and thus to make the routeing succeed. Self-adjusting for placement is realised by path-terminal exchanging and for grouping by path-terminal merging. They are described as follows.

(i) *Path-terminal exchanging*

(1) If $|N_u| > 0$, do exchanging with unoccupied nodes until no terminal-exchangeable path w.r.t. unoccupied nodes can be found.

For each failure path in P_f, select a constant number (c_1) of unoccupied nodes in N_o and check whether the path is terminal-exchangeable or not w.r.t. the selected nodes by comparing the routeing results before and after each exchange. Do the exchange if terminal-exchangeable.

(2) Do exchanging with occupied nodes until no terminal-exchangeable path w.r.t. occupied nodes can be found.

For each failure node, $f_i, 1 \leqslant i \leqslant \xi$, in N_f, take a constant number (c_1') of paths in P_i and check whether there is any pair of terminal-exchangeable paths by comparing the routeing results before and after each exchange. Do the exchange if terminal-exchangeable.

(ii) *Path-terminal merging*

Assume that the number of target tasks in G_t after grouping is n_0. For $1 \leqslant i \leqslant n_0 - 1$, do the following until the number of failure paths in P_f after regrouping does not increase by a constant (c_2).

(1) Regroup the n_{i-1} tasks in G_t into $n_i (n_i \leqslant n_{i-1} - 1)$ tasks by merging some path terminals together.

(2) Place the target tasks in the regrouped G_t on to the processors in G_p.

(3) Route the paths among the placed tasks and produce the new P_f.

Obviously if the algorithm works in such a progressively self-adjusting manner, a successful solution will certainly be found for the mapping problem eventually. In the worst case, all processes may for example be grouped into one task and placed on to one processor.

By combining the self-adjusting strategy and the three modules for a mapping, we finally obtain the self-adjusting mapping algorithm, which can be sketched as follows.

(1) Do initialisation: *routed* = *FALSE*.

(2) Do load-balanced grouping in G_t to form n_0 ($n_0 \leqslant m$) target tasks, where each task has a degree not greater than the degree of G_p and all tasks are cost-balancing.

(3) Do neighbour-first placement to place the tasks in G_t on to processors in G_p, where neighbouring tasks in G_t are placed on to possibly neighbouring processors in G_p so as to keep the total length of the shortest physical communication paths among the placed tasks in G_p as small as possible.

(4) Do path-disjoint routeing for placed tasks in G_p. If routeing is successful, *routed* = *TRUE*, otherwise output the failure path set P_f and failure node set N_f.

(5) If *routed* = *FALSE* do the following self-adjusting until *routed* = *TRUE*: (*a*) Do path-terminal exchanging until there is no terminal exchangeable path any more in P_f. (*b*) Do path-terminal merging (task regrouping) until the number of failure paths in P_f after regrouping does not increase by a constant.

4. COMPLEXITY AND PERFORMANCE OF THE ALGORITHM

4.1 Time complexity

Assume that the grouping, placement and routeing modules need time T_{group}, T_{place} and T_{route} respectively. From Refs 19 and 20 we know that

$$T_{\text{group}} = O(n^2) \tag{12}$$

$$T_{\text{place}} = O(m^2) \tag{13}$$

$$T_{\text{route}} = O(k^2 + km^2), \tag{14}$$

where $n = |G_t|$, $m = |G_p|$ and k is the number of paths to be routed among the placed tasks in G_p.

Since both G_t and G_p have a constant degree, the maximum number of total disjoint paths among the n_0 grouped tasks after grouping is $O(n_0) \leqslant O(m)$. For P_f and N_f, clearly $|P_f| \leqslant O(m)$ and $|N_f| = \xi \leqslant m$. Let s be the number of phases of self-adjusting. Let k_i and n_i be the number of paths to be routed and the number of target tasks after the ith phase adjusting respectively, k_i^{fo} and k_i^{fu} the number of failure paths deleted after exchanging with unoccupied processors and with occupied processors of the ith phase adjusting respectively, $0 \leqslant i \leqslant s$. Since each phase of path-terminal merging will reduce at least one task (one path) and increase at most c_2 failure paths, we have

$$s \leqslant n_0 - 1 \tag{15}$$

$$k_i \leqslant k_0 - i \tag{16}$$

$$k_0 \geqslant \sum_{i=1}^{s} (k_i^{fu} + k_i^{fo}) - c_s s. \tag{17}$$

Assume that time needed for block 'Exchanging with unoccupied processor', 'Exchanging with occupied processor' and 'Path-terminal merging' are respectively T_e^u, T_e^v and T_m. The ith phase adjusting needs time, $T_{\text{adjust}}^{(i)}$, at most

$$T_{\text{adjust}}^{(i)} = \sum_{j=1}^{k_i^{fu}} (k_0 - i)\, T_e^u + mk_i^{fo} T_e^v + T_m. \tag{18}$$

By equations (15)–(18), and with $n_0 \leqslant m$ and $k_0 = O(n_0)$, all s phases adjusting need time at most

$$\begin{aligned} T_{\text{adjust}} &= \sum_{i=1}^{s} \sum_{j=1}^{k_i^{fu}} (k_0 - i - j)\, T_e^u + m \sum_{i=1}^{s} k_i^{fo} T_e^v + sT_m \\ &\leqslant O((k_0 + c_2 s) k_0)\, T_e^u + O(m(k_0 + c_2 s))\, T_e^v + sT_m \\ &\leqslant O(m^2)(T_e^u + T_e^v) + O(m)\, T_m. \end{aligned} \tag{19}$$

Clearly, for each phase adjusting T_e^u and T_e^v are dominated by T_{route}, and T_m by T_{group}, T_{place} and T_{route}. Noticing that during the ith phase adjusting, $1 \leqslant i \leqslant s$, $n_i \leqslant m$ and $k_i \leqslant O(m)$, by equations (12)–(14) we have

$$\begin{aligned} T_{\text{adjust}} &= O(m^2)\, O(k_i^2 + k_i m^2) \\ &\quad + O(m)(O(n_i^2) + O(m^2) + O(k_i^2 + k_i m^2)) \\ &\leqslant O(m^2)\, O(m^3) + O(m)\, O(m^3) = O(m^5) \end{aligned} \tag{20}$$

Hence

$$T_{\text{samap}} = T_{\text{group}} + T_{\text{place}} + T_{\text{route}} + T_{\text{adjust}} \leqslant O(\max\{n^2, m^5\}). \tag{21}$$

Note that here the time complexity of our algorithm is for that under 'full adjusting' as described in the previous section. It can be reduced by degrading the adjusting heuristic. Instead of full adjusting we can also take 'semi-adjusting' by replacing the strategy in the above path-terminal exchanging with one such that during each phase adjusting only a constant number of end-nodes of failure paths in P_f and a constant number of failure nodes in N_f can be used to check with unoccupied and occupied nodes in G_p for exchangeability respectively. Thus we can reduce the worst-case time complexity of the program to $O(\max\{n^2, m^4\})$.

Furthermore, if we take away the restriction of edge-disjointness of the paths to be routed, the whole self-adjusting part in our algorithm is not needed any more, since path routeing will certainly succeed on any placement scheme provided by the placement module then. By using a usual algorithm of shortest path routeing for all pairs of processors that take time $O(m^2)$,[1] our algorithm thus has a worst-case time complexity $O(\max\{n^2, m^2\})$ for the transputer networks that can provide message routeing and multiplexing.

4.2 Implementation result

Our algorithm has been implemented in Occam on the Haithi-2 transputer system.[20] We have tested the algorithm with various problem instances. For any (regular or irregular) input user-defined task graph and a processor graph of torus of any size, the algorithm will produce a satisfactory process-to-processor mapping. It seems that in many cases the algorithm can find an embedded layout that is even difficult to be achieved by hand drawing. We show two examples of the implementation result of the algorithm in Fig. 8(*a*) and (*b*), where for task graph in (*a*), all tasks have the same computation weight and communication weight, for task graph in (*b*) the underlined numbers are communication weights and computation weights are implicated by node indices: node i has computation weight $i+10$; for routeing layout both in (*a*) and (*b*) bold lines indicate the occupied links by physical paths and plain lines the unoccupied links.

4.3 Performance evaluation

The performance of our algorithm has been measured on

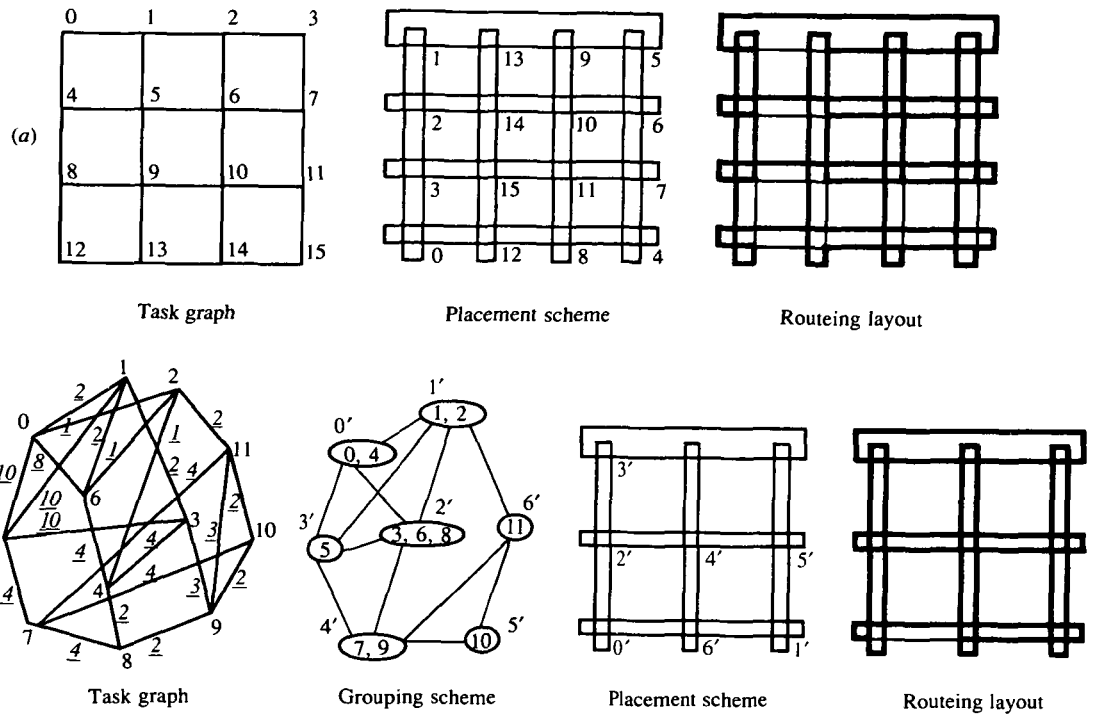


Figure 8. Two examples of implementation result: (*a*) mapping 4 × 4 mesh on to 4 × 4 torus; (*b*) mapping an arbitrary task graph on to 3 × 3 torus.

the Hathi-2 system. For mapping arbitrary task graphs on to a processor torus of arbitrary size, we measure the time elapsed during the whole mapping procedure as well as its different subprocedures of grouping, placement, routeing and adjusting individually. Measuring the time elapsed for mapping a series of arbitrary task graphs of different sizes on to a processor torus of fixed size, we know how the elapsed time varies when the number of tasks of an arbitrary task graph varies, thus get a figure of the algorithm performance w.r.t. the number of tasks. Likewise, measuring the time elapsed when the number of tasks is fixed but the size of the processor torus is varying, we obtain a figure of the algorithm performance w.r.t. the number of processors. The combination of these two measurements will generate an overall evaluation to the performance of our algorithm.

The measured performances of the algorithm under full adjusting are described as in Fig. 9(*a*)–(*e*), where (*a*)–(*d*) respectively show the individual performances of the procedures of grouping, placement, routeing and adjusting in the algorithm, and (*e*) presents the overall performance of the algorithm. In each of these figures, curve $t(n, 100)$ represents the relation between time and n, the number of tasks, when the size of the processor torus is fixed to 100 (10 × 10), while curve $t(100, m)$ shows the relation between time and m, the size of the processor torus, when the number of tasks of an arbitrary task graph is fixed at 100 (the topology of the task graph is not fixed). The column axis with a scaling unit of 10 s is the axis of time. The row axis with a scaling unit of 10 is the axis of task number (n) for curve $t(n, 100)$ and of processor number (m) for curve $t(100, m)$, respectively. All task graphs are generated over a set of random data, therefore their topologies are random. From Fig. 9 it is obvious that the execution time of the algorithm is mainly dominated by the time elapsed in the procedure of adjusting. Therefore to a given processor torus, mapping often takes more time for task graphs with a complex topology than with a simple one, since the former normally requires more work of adjusting.

As samples, in Table 1 we illustrate the performances of the algorithm for some typical categories of task graphs.

5. CONCLUDING REMARKS

The problem of mapping parallel programs on to parallel computers is a well-known fundamental problem in parallel processing and it is of great significance both theoretically and practically. This problem has been known to be NP-hard in general. Therefore it seems impossible to have a fast deterministic algorithm to solve the mapping problem in the general case. In this paper we have presented a fast heuristic algorithm, the self-adjusting mapping algorithm, to solve this problem on transputer networks. The algorithm mainly contains grouping, placement and routeing modules, and they work co-operatively under a series of progressive self-adjustments until a successful solution to the mapping problem is produced. For mapping n tasks in an arbitrary task graph on to m processors in a processor graph of torus, the algorithm has a worst-case time complexity $O(\max\{n^2, m^5\})$ under full adjusting, $O(\max\{n^2, m^4\})$ under semi-adjusting and $O(\max\{n^2, m^2\})$ under no adjusting, where the last holds only for the transputer networks providing message routeing and multiplexing. Our algorithm has been implemented on the Hathi-2 transputer network. The implementation result and demonstrated performances show that the algorithm works well for both regular and irregular task graphs.

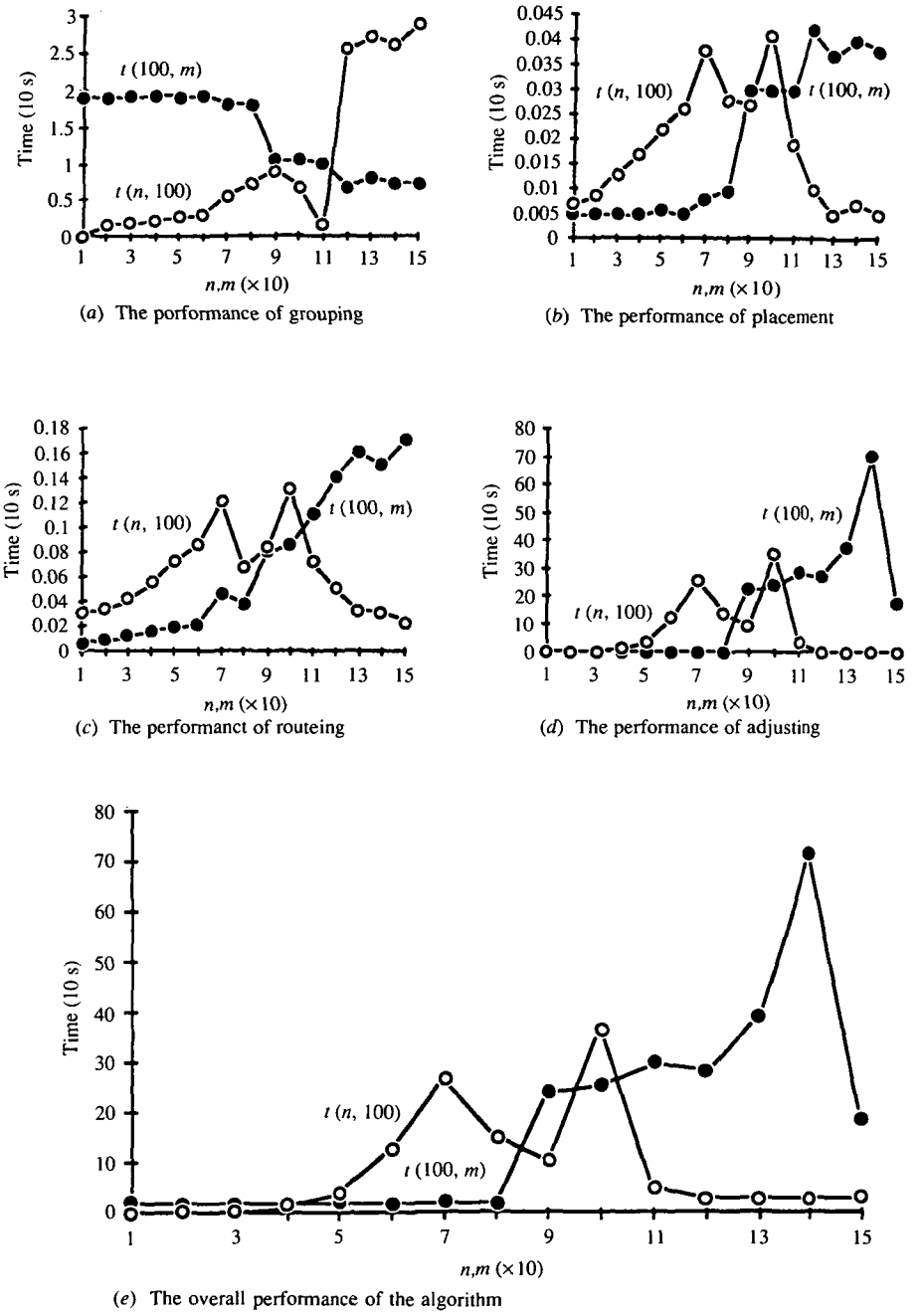


Figure 9. The measured performances of the algorithm.

Table 1. Samples of performances for different categories of task graphs

Task graph		Proc. torus	Measured performance (time in seconds)					
Topology	n	m	Initial	Grouping	Placement	Routeing	Adjusting	Total
Mesh	12×12	12×12	0.62	1.25	0.90	5.22	0	7.99
	5×30	5×6	0.68	24.68	0.16	0.32	0	25.84
Binary tree	31	5×6	0.03	0.27	0.17	0.20	2.43	3.10
	127	5×10	0.48	21.84	0.10	0.31	2.06	24.79
Hypercube	16	4×4	0.01	0.02	0.10	0.12	0	0.25
	128	10×13	0.52	24.87	0.10	0.53	0	26.02
Random	100	10×10	0.30	6.74	0.41	1.27	356.35	365.07
	100	5×10	0.31	19.09	0.06	0.19	0.57	20.22

Acknowledgement

The author wishes to thank Ralph-Johan Back and Tor-Erik Malén for their careful reading of the manuscript of an earlier version. The author also wishes to thank an anonymous referee for his valuable comments and suggestions on this paper.

REFERENCES

1. A. V. Aho, J. E. Hopcraft and J. D. Ullman, *Data Structures and Algorithms*. Addison-Wesley, London.
2. M. Aspnäs and R. J. R. Back, A programming environment for a transputer-based multiprocessor system. Symposium on Programming Languages and software Tools. *Proc. First Finnish–Hungarian Workshop*, edited T. Gyimóthy, pp. 94–103 (1989).
3. F. Berman, Experience with an automatic solution to the mapping problem. In *The Characteristics of Parallel Algorithms*, pp. 307–334. MIT Press, Cambridge, MA (1987).
4. F. Berman, On mapping parallel algorithms into parallel architectures. *J. of Parallel and Distributed Computing* **4**, 439–458 (1987).
5. J. E. Boillat, P. G. Kropf, D. C. Meier and A. Wespi, An analysis and reconfiguration tool for mapping parallel programs onto transputer networks. *Proc. OUG-7*, edited T. Muntean. OPPT (1987).
6. J. E. Boillat and P. G. Kropf, A fast distributed mapping algorithm. *Proc. Conpar90-VAPP IV*, Lectures Notes in Computer Science 457, edited H. Burkhart, pp. 405–416. Springer, Berlin (1990).
7. S. H. Bokhari, On the mapping problem. *IEEE Trans. Comput.* **C-30** (3), 207–214 (1981).
8. Inmos Ltd, *Transputer Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ (1988).
9. Inmos Ltd, *Occam 2 Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ (1988).
10. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, Optimization by simulated annealing. *Science* **220** (4598), 671–680 (1983).
11. O. Krämer and H. Muhlenbein, Mapping strategies in message based multiprocessor systems. *Proc. PARLE '87*, Lecture Notes in Computer Science 258, pp. 213–225. Springer, Berlin (1987).
12. New solutions to the mapping problem of parallel systems: the evolution approach. *Parallel Comput.* **4** (3), 269–279 (1987).
13. H. Ohara and H. Iizuka, A preprocessor to augment the description of Occam processes for multiprocessor machines. *Proc. 9th Occam User Group Technical Meeting*, pp. 71–80 (1988).
14 P. Sadayappan and F. Ercal, Cluster-partitioning approaches to mapping parallel programs onto a hypercube. *Proc. Supercomputing '87*, Lecture Notes in Computer Science 297, pp. 475–497. Springer, Berlin (1987).
15. P. Sadayappan and F. Ercal, Nearest-neighbour mapping of finite element graphs onto processor meshes. *IEEE Trans. Comput.* **C-36** (12) (1987).
16. H. Shen, Mapping parallel programs on to transputer networks. *Proc. 1989 Australian Transputer and Occam User Group Conference*, edited J. Hulskamp, pp. 85–94. RMIT (1989).
17. H. Shen, Self-adjusting mapping: a heuristic mapping algorithm for mapping parallel programs on to transputer networks. *Developing Transputer Applications* (*Prog. OUG-11*), edited J. Wexler, pp. 89–98. IOS (1989).
18. H. Shen, Fast path-disjoint routeing in transputer networks. *Proc. First Finnish–Hungarian Workshop, Symposium on Programming Languages and Software Tools*, edited T. Gyimóthy, pp. 157–167 (1989).
19. H. Shen, Occam implementation of path-disjoint routeing on the Hathi-2 transputer system. *Microprocessing and Microprogramming* **30**, 93–100 (1990).
20. H. Shen, *Occam Implementation of Process-to-processor Mapping on the Hathi-2 Transputer System*. Åbo Akademi, Department of Computer Science, Research Report A (114) (1990).

Book Review

BERNARD DE NEUMANN, DAN SIMPSON AND GIL SLATER (Eds)
Mathematical Structures for Software Engineering
Oxford University Press, 1991
£45.00. ISBN 0-19-853627-5

Software engineering is becoming increasingly mathematical, and at a research level it presents many interesting mathematical challenges. Nevertheless, the software engineering field is viewed apathetically by many professional mathematicians. In response to such concerns, the Institute of Mathematics and its Applications formed its Systems and Software Engineering Specialist Group. The present book presents the proceedings of the first three-day conference of this specialist group at Manchester Polytechnic in 1988.

The papers illustrate the very wide range of different techniques of discrete mathematics that have been found to be valuable in various areas of software engineering. The overall tone is well set by the leading paper by Tim Denvir, who addresses the role of mathematics in software engineering. A natural starting-point is the notion that programming is itself a mathematical activity subject to rigorous mathematical rules, which has fired the thriving formal methods community. However, almost every branch of discrete mathematics has found some kind of software engineering application. This application of mathematics to the field may be interpreted in the very widest sense, for example the mathematical theory of computability provides a 'reductionist' basis for the entire computing discipline, standing in much the same relationship to software engineering as electromagnetic theory does to practical electrical engineering.

Taken as a whole, the remaining papers in the volume bear out the astonishingly wide range of different ways in which mathematics has found application to software engineering problems. Individual papers naturally tend to reflect the application of mathematical techniques in one particular software engineering context. Examples of topics treated include the modelling of software testing, maintenance and reliability; consideration of notations and standards; algebraic approaches to program development; practical problems of safety-critical software and the use of special notations for the study of parallel systems. Relevant mathematical techniques include topics as diverse as statistics, category theory and modal logic.

Overall, this is a remarkably heterogeneous collection of stimulating papers. The diversity of topics, both from a software engineering and from a mathematical point of view, leaves little room for doubt that almost any area of discrete mathematics can illuminate some area of software engineering interst. It is somewhat harder to deduce either that software engineering forms a distinctive discipline worthy of study from a mathematician's point of view, or that the field has the potential ability to stimulate significant new mathematics. Perhaps future conferences of this worthwhile kind will clarify such issues.

PETER WALLIS
Bath