Appl. Math. and Comp. Sci., 1992, vol.2, No.1, 21–47

LOAD CONTROL IN DISTRIBUTED DATABASE BASED MULTISTAGE DECISION–MAKING SYSTEM*

LESZEK BORZEMSKI**

In shared database management systems there is a need for load control to improve distributed system utilization. It is shown how this aim can be achieved in the case of a distributed database developed for supporting a decision–making distributed system where decisions are based on a common decision tree. The load balancing approach is investigated and several general and application specific task load balancing algorithms are proposed and tested. The results show the relative merits of different approaches to the load balancing problem. A pre–scheduling policy is proposed which is superior to other investigated algorithms in the real–time environments.

1. Introduction

Considering distributed decision–making applications, one can find the environment where a set of geographically distributed, local decision makers (DMs) make a global decision in the context of some common tree–like decision making skeleton whose vertices refer to local DMs and edges describe relationships between local DMs. Local decisions are made on the basis of both some locally gathered data and data stored in a shared database. Each global decision process always starts at the same local DM which is called the root DM. Next admissible stages of the global decision are made by local DMs along the paths starting from the root DM and ending at the terminal DM connected with the last decision stage where the final decision is made. Such specific characteristics of the decision making or very similar ones can be found e.g., in the multistage pattern recognition (Kurzyński and Puchała, 1990), hierarchical classification (Wang and Suen, 1987), computer–aided testing and diagnostics (Camurati *et al.*, 1988), with all activities beging performed within spatially distributed systems, as well as in human organizations where local DMs can be humans from some personal hierarchy (Levis, 1988). Here, the specific multistage decision making system is not considered in detail, and the reader is referred to the literature.

We define a distributed system as any configuration of two or more processors with their own internal and external memories. Private external memories are for

* This work was supported by the State Committee for Scientific Research under Grant No.3 0762 91 01.

** Institute of Control and Systems Engineering, Technical University of Wrocław, Wybrzeże Wyspiańskiego 27, 50–370 Wrocław, Poland

storing local databases which form a distributed database used in the multistage decision–making. A system–wide operating sytem provides a mechanism for data access and task distribution. Each task occurrence generates a single complex data access transaction of one particular type and the workload of distributed database consists of a set of transaction types defined according to the rules of the multistage decision making whereas the workload of system processors consists of a set of corresponding task types occurrences, each with a particular set of simple transactions which form that set together. A simple transaction deals with a single file and the relevance of corresponding task type to the individual transtions is given. When designing a distributed computer system it is advantageous to consider the applications for which the architecture will be used. This paper discusses a dedicated system architecture in the context of transaction allocation, tailored to fit the multistage decision making problems in distributed environment. Because all simple transactions are pre–compiled, thus all of them from the same complex transaction are allocated to the same processor as its source, that is, the specific task type occurrence. Therefore, we definitely deal with the task assignment problem for the needs of transaction processing in distributed database system.

The load balancing is considered as a key load control approach in maintaining efficient distributed database management systems operation. The load balancing algorithms can distribute computational tasks to processors such that the processor utilization and the throughput of the system are maximized. In our distributed database system the tasks which generate transactions are not fully independent. In each set of tasks under consideration which we shall refer to as a task force, the tasks are grouped into jobs. A job consists of a set of tasks to be performed for the needs of a common goal. These tasks are not known *a priori* but they are dynamically created as the computation proceeds. A task occurrence (in short, a task) belongs to some prescribed task type. The exact task type specification is given but the exact task specification in particular jobs is not known *a priori*, we only know some task type occurrence statistics. Each member of a job, beside of the last one, creates a new task according to the results of processing of the global decision–making concerning a given decision–making object. The task creation process is based on a given precedence graph which is a particular decision tree. The transitions are governed by the results of local decisions at every decision–making stage. The system supports simultaneous distributed processing of several jobs defined with regards to the same decision tree. The jobs are independent whereas each job has precedence–constrained tasks in the from of a chain of tasks. In our model the jobs arrive in the batches and all jobs may start at the same time moment. The computation always starts from the same task type for all jobs which is the root node. The task force is dynamically changed when particular tasks have been completed. We propose an approach to load balancing which reflects the characteristics of the multistage decision problem to give better performance in terms of load balancing as well as total execution speed of a batch of global decision to be processed. This approach is based on the pre–allocation calculated on the basis of the given *a priori* knowledge. In the simulation study, the multiple

decision making system and appropriate LB algorithms were implemented. The results from simulation experiments showed that the proposed LB algorithms are much superior to the other allocation policies investigated.

2. Task Allocation Strategies

In order to be able to take full advantage of a distributed computer system it is important to distribute the system workload among processors. This problem of the control of the system load is studied from different points of view and applying different allocation models (Chou and Abraham, 1982, 1986; Efe, 1982; Hac and Jin, 1988; Johnson and Harget, 1989; Ni and Hwang, 1981; Ni *et al.*, 1985; Price and Salama, 1990). These works can be classified in several ways due to a number of differences in the analysis and solution of the general workload allocation problem in distributed systems. There is no obvious taxonomy of the existing approaches. Therefore, it is rather difficult to locate among other researches, the specific problem under consideration, in order to compare the approaches and results. This dilemma also holds in our case. Some taxonomies exist (Casavant and Kuhl, 1988; Wang and Morris, 1985) with the basic classification schemes and terminology. Nevertheless, recent research results create a need for the current update of them, to include new cases and for better understanding of the issues in this area. The question of how to prepare the taxonomy that is consistent and foreseeing the future is open and we do not address this problem here. But due to this we use our own terminology which may be slightly different from other proposals.

We consider a dedicated distributed system where the workload consists of a set of tasks. We shall refer to such a set of tasks as a task force. The specific assumptions concerning the task force used in our study will be presented after introducing the multistage decision making application. A distributed system is defined as any configuration of two or more processors with internal and external memories. All processors are homogeneous and uniprogrammed. Private external memories are used for storing local database which from a distributed database accessed by running tasks. The workload of the system is due to task execution on the processors.

We study here the problem of load balancing which is the task allocation category viewed from the system's point of view. The goal of load balancing can be generally stated as follows. Given a global workload L submitted to a set of p processors, the load balancing (LB) problem is to find a feasible partition $(L_1, L_2, ..., L_p)$ of L $(L = L_1 + L_2 + ... + L_p)$ such that the individual processor performances are equal i.e., if P_k is the performance index value for the k–th processor then $P_1(L_1) = P_2(L_2) = ... = P_p(L_p)$. This idea assumes that the load on all processors can be balanced by some given task allocation mechanism. Then LB strategy makes it possible to progress the computation for a given workload by all processors at approximately the same rate. This, of course, is done for the reasons of maximal processor utilization, and not for the needs of minimization of the response time for an individual user or minimization of the completion time for

the task force. But generally, in several cases, the LB task allocation strategy minimizes the total execution time of the workload when this workload is introduced just as a given task force with the finite number of tasks and not as a stream of arriving tasks. The obvious advantage of LB policy is that by applying rather simple mechanisms, comparing to the optimization of some objective function approaches, we are able to obtain very effective solutions which are often good from both the user's and the system's points of view. The general LB approach is most effective when the system is homogeneous, fully connected and the workload consists of a set of independent tasks. It is very important for this task allocation policy to be easily implemented in the particular distributed systems. But then demand for an equal load is practically unrealistic. Moreover, the optimal load balancing, in cases where this requirement can be defined, is computationally very expensive and not needed. Thus we put the LB requirements as follows:

$$\forall_k \; 1 - \underline{d} \leq P_k \leq 1 + \overline{d}, \quad P_k = L_k / \widetilde{L}, \quad \widetilde{L} = 1/p \sum_k L_k, \quad k = 1, 2, ..., p \qquad (1)$$

where $\underline{d}$, $\overline{d}$ are given admissible imbalance parameters. For comparison reasons we use here normalized values of the performance index, so that $P_k = 1.00$ is equivalent to the best performance. If this inequality is satisfied for a processor, then the load assigned to it is acceptable, otherwise, if $P_k > 1 + d$, processor k is overloaded, or if $P_k < 1 - d$, processor is underloaded. Analyzing this criteria the load can be balanced by allocation or reallocation of tasks and in the case of full knowledge about the load, it can be even optimally distributed. The load balancing optimality criterion can be formulated as maximum distance between values of P_k for all processors and the requirement is to minimize this imbalance distance. But in most practical cases only the heuristics can be accepted because optimal algorithms are usually complex and time consuming. Moreover, in dynamic task creation environments the approaches with optimization of LB may not be applied.

There are many approaches to load balancing. Most of them can be classified as static or dynamic. In static strategies the complete task allocation is prepared before the task force execution and the scheduling scheme is ready for implementation at load time. The allocation is based on foreknowledge of global task force. When the load is exactly known *a priori*, the optimal allocation can be determined at the system design phase. Nevertheless, the calculations can be time ineffective. The main approaches are graph theoretic (Bokhari, 1988) and integer programming (Ma, Lee and Tsuchiya, 1982). The advantage of static load balancing is that run–time overhead with respect to task allocation is minimal because all allocations are still known. The conclusion could be that these approaches should be used whenever possible.

In dynamic LB schemes, task allocation decisions are made at run–time of the task force on the basis of current system state. Then a considerable scheduling overhead is usually introduced. Thus the requirements are stated to consider rather simple and not too sophisticated policies. Because the problem of the task

allocation overhead is especially related to LB dynamic approaches, thus both load balancing performance and run–time overhead are equally important and neither can be optimized without considering the other. Considering this, a better algorithm would be the one that incurs little overhead and achieves a better load balance. The effectiveness of dynamic LB scheme depends on the chosen performance index and the way how it is calculated or estimated. The overhead depends on several factors, also on distributed system organization. In this study we take into account the time needed for task transferring between the network nodes, assuming on the other hand that each node possesses complete and up–to–date knowledge of the system state at every time moment when scheduling decisions are to be made.

In static load balancing, tasks are assigned to processors just once. In dynamic schemes we can consider one–time assignment versus multiple–time reassignment as the workload fluctuates due to task creation and completion. Most research is on one–time assignments. The latter scheme is an adaptive approach that may result in heavy task migration, but may also be advantageous, for instance in the cases where a load balancing algorithm assumes that the characteristics describing the task force do not remain the same over considered period of time but may change. When the algorithm uses this adaptive option in making task allocation, it would be beneficial to the system as a whole to adapt to the changes even with the increase of the overhead due to task migration.

Finally, in some applications, we can make use of certain allocation suggestions based on our partial knowledge about the characteristics of the task force. This knowledge could be used dynamically, but due to the need of system overhead reduction, it would be advantageous to include these suggestions in the form of a pre–allocation scheme. The pre–allocation may concern all of the tasks or some subsets of tasks. We believe that this proposal, known in the form of pre–scheduling in parallel compilers, can be applicable in some distributed applications. An example of such application is shown in this paper. We propose the task allocation approach which consists of two phases. The first concerning the determination of the pre–allocation based on some *a priori* information about the specific application under consideration. In this phase, initial allocation of tasks which are ready to run in the task force is prepared and tasks are sent to individual nodes of the distributed system under control of the network–wide dispatcher. Next, the task force execution starts at the same moment on every processor. As new tasks are created they are allocated according to the scheme determined in the pre–allocation phase, as well as taking into account current system state. The pre–allocation is designed in such a way as to foresee the load balancing to be achieved after task force completion. But because of dynamic fluctuations of the system load, the dynamic option must be included in order to improve the final load distribution.

The use of the pre–scheduling is especially motivated in the cases where we consider not only task allocation problem but some problems which are closely related. In our case we included task – distributed database requirements. The access to files stored in the distributed database is formulated via given precompiled database transactions issued from the tasks. It is assumed that the files have been previously

optimally allocated, minimizing the total data transmission objective function for a given task force with expected characteristic and with the assumptions about the task allocation (Borzemski, 1992). This assumed static task allocation generally does not meet the load balancing requirements because it was chosen without taking into account processor utilization. It was motivated by user's requirement of locality, stating that the i–th task type should be preferably allocated where the i–th user is located, that is at the i–th processor, for u users on p–computer system, for $p = u$. Then, at the run–time the load balancing problem is considered in order to improve system performance in the context of processor utilization. Changing task allocation yields the total data transmission but it was observed in several cases that the increase in the transmission cost is small but the load is well balanced. In the next part of this text we will show formally how the distributed database allocation problem can be considered jointly with the load balancing problem. A solution to distributed database file allocation problem obtained with some assumptions concerning task allocation at the run–time, forms a basis for the formulation of pre–scheduling.

The pre–scheduling task allocation policy seems to be a good proposal especially in the real–time environments. Such environments usually limit their applications in available size of processor time slices for a given job. Having smaller time slices we can better react to the changes (in so called real–time). From the system utilization point of view we want to use processors as effective as possible. Suppose that for an application we can have a number of w independent jobs clustered into the n batches of the same size w. The pre–scheduling approach seems to be superior to the other investigated algorithms in the real–time environment because its performance is especially good for small job batches which can be completely processed having small processor time slices and whose LB performance level is sufficient. Sometimes we may also need to determine the value of w in order to meet the deadline time constraints on the average job completion time introduced in the hard real–time environments (Stankovic, 1989). Then applying the pre–scheduling option, for not too big values of w (they are dependent on a given application), the average job completion time is usually shorter than after applying other task allocation policies among those which were investigated here.

3. Application Environment

We define our application environment in the language of OR graphs (Wah and Li, 1989). Depending on a specific need in the analysis, each node of an OR graph represents either a local decision problem, the algorithm for solving it, the local DM or a computational task performed for the needs of local decision–making. Considering a decision–making philosophy, a special node root $(\mathcal{G}^{\psi})$ called *root* of $\mathcal{G}^{\psi}$ represents the whole decision space considered for the decision problem ψ which is called *global*, for the graph representation of the problem ψ given by the graph $\mathcal{G}^{\psi}$. A node in such a graph having successors is called *nonterminal*. Nodes with no successors are called *terminal*, and each terminal node represents a final decision of the multistage decision–making problem ψ .

Given an OR graph representation of a problem ψ, we can identify its different solutions, each being represented by a ***solution chain***. A solution chain $C(\omega)$ for a given object ω obtained using a given OR graph $\mathcal{G}^{\psi}$ has the following properties: (i) $root(C(\omega)) = root(\mathcal{G}^{\psi})$ for all ω's ; (ii) terminal $(C(\omega)) \in \{t_1, t_2, ..., t_M\}$, where $t_1, t_2, ..., t_M$ are all possible final decisions in the original problem ψ, (iii) exactly only one terminal node is selected. An object is an entity (Question, etc.) the global decision– making considers to determine its class (state, an answer). The features of an object are locally sensored (tested, measured, checked) and taken into account in local decision, together with some knowledge stored in a shared distributed database.

Basing on the tree $\mathcal{G}^{\psi}$ we can define the ***task types tree*** $\mathcal{T}^{\psi}$ for the problem ψ which represents a "problem decomposition scheme" at the task grain level for solving the problem modeled by the graph $\mathcal{G}^{\psi}$. Nodes of $\mathcal{T}^{\psi}$ represent task types defined for given local decision– making algorithms $\psi_i, i = 1, ..., u$, where u is the number of nonterminal nodes in $\mathcal{G}^{\psi}$ (or in $\mathcal{T}^{\psi}$). The structure of $\mathcal{T}^{\psi}$ is the same as for $\mathcal{G}^{\psi}$, except for the terminals of $\mathcal{G}^{\psi}$, which together with their incoming arcs connecting them to appropriate parent nodes were eliminated. Based on the tree $\mathcal{T}^{\psi}$ each solution chain produces o chain of tasks with the heading being of the root $(\mathcal{T}^{\psi})$ type task occurrence, and with the last task being the terminal $(\mathcal{T}^{\psi}|C(\omega))$ for a given ω.

In several cases, for example in a character recognition system with the ability of matching multiple characters at once (a page of characters), the global decision requests arrive to the system as a whole in the batches of requests Ω, each of the size w. The objects are here the characters to be recognized using the multistage pattern recognition scheme. All requests arrive at the same time to a p–processor system which is idle at that moment. Then the parallelism includes simultaneous execution of local decisions for the needs of the set of mutually independent global decisions. Such a system could be made of several transputers (Luo *et al.*, 1989) for computer recognition of Chinese characters (Gu *et al.*, 1983; Wang and Suen, 1987).

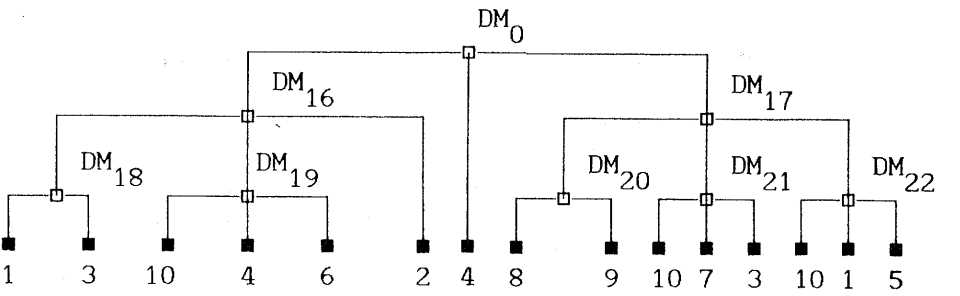


Fig. 1. Example of the N–level decision making tree ($N = 3$).

Example of a decision tree is shown in Figure 1. Two kinds of nodes are distinguished, namely the terminal and non–terminal ones. Each terminal node represents the final decisions which can be reached after execution of the local decision algorithms along the paths starting at the root node and ending at that terminal node. Which of the branches are used in the 'if–then–else' logic is of concern of the local algorithms and specific input data. Non–terminal nodes represent local algorithms ψ_i , DM_i or task types $\mathcal{T}_i$, $i = 1, 2, ..., u$. As it is shown in the figure the decision regions may overlap each other. The tree $\mathcal{T}^{\psi}$ has the nodes with the numbers 0, 16, 17, 18, 19, 20, 21, 22.

As a particular case of the multistage decision making we may refer to the multistage pattern recognition (Kurzyński, 1988). Then the terminal nodes represent pattern classes and every non–terminal node is connected with an appropriate set of pattern classes accessible from that node. In particular, the root node represents the entire decision space. The tree classifier makes use of different pattern features at different nodes. The procedure is initialized for the whole spectrum of possible classes and ends at the N–th stage, where the decision made indicates a single class, which is the final result of multistage classification. Moreover, in the case of recognition with training, the local DM requests for its training set which is some subset of the global training set prepared for the whole system. Then the problem of data partitioning and allocation must be solved, both in the parallel and in distributed systems. In this paper we deal with the problem of how to control the load of a p–processor distributed system to balance the load and obtain good processor utilization level. We assume that there is an application–driven mechanism which generates the task chains and from this point of view a common requirement studied in this paper is the consideration of chain–like task processing where we have w chains to be processed in a distributed system, whereas tasks in these chains are created at run–time on the basis of given tree skeleton of task types and possible 'if–then–else' dependencies. All tasks start with the root task type, and independently proceed to the next level task type at every stage, till the task has no successor. Paths from the root node to terminal nodes can have different lengths, so the appropriate task chains can have different number of tasks.

4. File Allocation and Load Balancing Joint Problem

Let us introduce the following notations: u is the number of task types, $i = 1, 2, ..., u$ is the (task) type index, r is the number of files, $j = 1, 2, ..., r$ is the file index, p is the number of nodes of the computer network, $k = 1, 2, ..., p$ is the node index, f_i is the average frequency of task type i occurrence, expected in the processing of arriving batches of decision requests, L_j is the size of file j, b_k is the storage capacity of node k, w is the size of decision request batch, v is the average data transmission rate, and

$$u_{ij} = \begin{cases} 1, & \text{if task type } i \text{ accesses file } j \\ 0, & \text{otherwise} \end{cases}$$

$$y_{kj} = \begin{cases} 1, \text{ if file } j \text{ is allocated on node } k \\ 0, \text{ otherwise} \end{cases}$$

$$z_{ki} = \begin{cases} \text{portion of incoming tasks of type } i \text{ allocated to} \\ \text{processor } k, \qquad 0 \leq z_{ki} \leq 1 \end{cases}$$

Then the minimization criterion for the total expected file transmission is expressed as follows

$$Q = \sum_{k=1}^{p} \sum_{i=1}^{u} \sum_{j=1}^{r} L_j \, f_i \, z_{ki} \, u_{ij}(1 - y_{kj}) \tag{2}$$

The optimization constraints are

$$\forall_j \sum_k y_{kj} = 1, \qquad \text{non} - \text{redundant case} \tag{3}$$

$$\forall_k \sum_j y_{kj} L_j \leq b_k, \qquad \text{storage limitation} \tag{4}$$

$$\forall_k \sum_k z_{ki} = 1, \tag{5}$$

$$\forall_k 1 - \underline{d} \leq \sigma_k \leq 1 + \overline{d}, \qquad \text{load balancing requirements} \tag{6}$$

where $\sigma_k = q_k/\tilde{q}$, $\tilde{q} = 1/p \sum_k q_k$, $k = 1, 2, ..., p$, $\underline{d}$ and $\overline{d}$ are given admissible imbalance parameters – maximum deviations from equal distributions allowed in the system, respectively the lower and upper deviations, q_k is the load of k–th processor, $\tilde{q}$ is the average load. For comparison purposes we use here normalized values of the performance index σ_k, so that $\sigma_k = 1.00$ is equivalent to the best performance. If this inequality is satisfied for a processor, then the load assigned to it is acceptable, otherwise, if $\sigma_k > 1 + \overline{d}$, processor k is overloaded, or if $\sigma_k < 1 - \underline{d}$, processor is underloaded.

Throughout the paper, we assume that there are u task types and the batches of w decision requests are processed in p–processor system, $w >> p$, $u = p$, $f_i's$, $i = 1, 2, ..., u$ are estimated according to the analysis which is partially application oriented. Generally, it is possible to iterate several times the execution of the application in some initial runs before its normal execution is started. It can also be done in the simulation experiment. Let us denote by $M(N)$ the set of numbers of nodes at the n–th level of the tree, $n = 0, 1, ..., N - 1$. The root node has assigned #0. Then it is clear that $f_0 = 1$ and

$$\forall_n \sum_{i \in M(n)} f_i \leq 1 \tag{7}$$

Assuming temporarily that the index i also depicts all terminal nodes we have

$$f_i = \sum_{j \in M^i} f_j = 1 \tag{8}$$

where M^i is the set of numbers of immediate successor nodes of the i–th node, $i \in \overline{M} = \bigcup_{n=0}^{N-1} M(n)$, f_i for $i \in M(n)$ can be interpreted as the frequency of the global decision in underlying conditions.

We assume that in our homogeneous distributed system made of uniprogrammed local systems, tasks being initiated to run cannot be preempted until they finish their execution on given processor. Furthermore, there are no overlaps among time windows of task execution and its communication (time for database access). Hence, the task is waiting for the access result (i.e. local or remote transfer of appropriate files). Local transfers are not time consuming. We also consider the case where the distributed database access is done concurrently for the needs of all working tasks. The processing time of the i–th task type allocated to the k–th processor is composed of two times as $\tau_{ki} = \tau_i + \tau'_{ki}$, where τ_i – execution time of the i–th type task (only "pure" computation), τ'_{ki} – time delay due to database access,

$$\tau'_{ki} = \frac{1}{v} \sum_{j=1}^{r} u_{ij} L_j (1 - y_{kj})$$

It is interesting to note that, although not given much consideration here, in several cases, the task type execution times are naturally partially ordered in such a way that the time for the root–node task type is maximum, and further ordering of times matches the decision tree construction.

We use the total accumulated processing time of the k–th processor to measure the load q_k . The load of processor k consists of the load due to task execution and database communication and the idealized individual processor loads q_k, $k = 1, ..., p$ are

$$q_k = \sum_i f_i z_{ki} \tau_{ki}$$

These loads are calculated as well as the idealized average load $\tilde{q}$, on the assumption that the load balancing algorithm avoids the occurrence of the idle–while–waiting conditions on processors. In a batch oriented system, idle–while–waiting will not occur when there is at least one task at each processor at any time and the task allocation policy preserves against the idle processor system state.

Here we must check whether it is possible to satisfy the average load conditions for normalized load σ_k at all. This can be done by looking at the total expected load which is estimated as $\underline{q} = \sum_k q_k$, assumming for z_{ki} that $k = i$.

Then for given values of load deviations we can now estimate the conditions of feasible solutions but even then it is possible that there is none feasible solution at the run–time.

Note that the solution of the above combined formulation of file allocation and task allocation problems, that is the solution minimizing total expected data transmission under load balancing requirements, can only be well–behaved in terms of the average conditions with stable f_i values over considered set of batches of decision requests. Due to the inherent characteristic of our system, it can be easily shown that the order of selecting the requests for processing, influences the final result in load balancing. Unfortunately, this cannot be taken into account when respecting the average situation.

We propose to solve the problem (2)–(6) in two phases. First phase is strictly a design phase, whereas the second phase is an execution phase. In the design phase we deal with the file allocation problem based on the average expected values of $f_i's$ and with the static, *a priori* assumed task allocation. Then we must consider the well–founded task allocation scheme. Keeping in mind our objective in processing multistage decisions, for the case $u = p$, we may conclude that assigning all task occurrences of the i–th type to processor i where the i–th local decisionmaker is located, is a good and reasonable choice. Such a choice is motivated by "locality" feature of each task which is associated with the need of some local data access which has not been included in the model. Thus assuming such task allocation we can meet this requirement. So, at the design phase, given the multistage decision making application, estimated values of the frequencies of the task types occurrences and a p–processor distributed system, find the optimal file allocation that minimizes total expected file transmission, subject to the sizes of local databases and non–redundancy requirement, assuming a given stable task allocation. At the execution phase, given the multistage decision making application, estimated values of the frequencies of the task types occurrences, the workload of decision requests, distributed database allocation, allocate tasks from the task force under constraints of load balancing.

Such decomposition of the problem is natural but note that this approach does not ensure the same solution as in the join design problem. To solve the file allocation problem as formulated above we may use the algorithm published in (Borzemski, 1992). Now we need the load balancing algorithm to solve task assignment problem at the execution phase. A brief description of simple heuristics is given next. The simulation mechanism for the evaluation of the load balancing algorithms has also been implemented and the computational experiments have been conducted. The results are presented below.

5. Load Balancing Algorithms Considered

Our loosely coupled multicomputer system consists of multiple independent homogeneous computers (Figure 2). The computers have their own local internal and external memories and can communicate by message passing. The system is uni–programmed (from the user's point of view) and dedicated to the application. Here we consider the communication overhead caused by message transfer which is incurred when the control is passed from task i to task

$i+1, i, i+1 \in \overline{M}$, $i+1 \in M^i$, $i+1 \notin M(N)$. Here, the aspect of processor utilization is mainly analyzed, assuming that the communication overhead depends only on the distance between both nodes and amount of data sent. The message size is assumed to be the same for all transmissions whereas the distance is calculated as the shortest path defined by the number of intermediate computer nodes in a computer network structure increased by one.

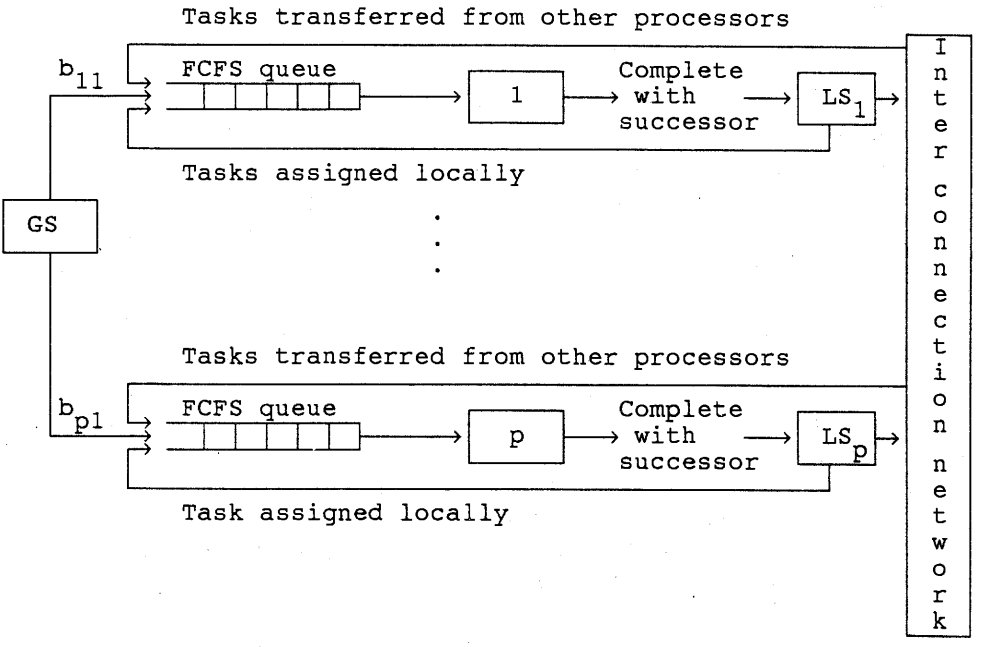


Fig. 2. System model.

Each processor has its own task queue residing in its local memory to hold ready–to–run tasks. There are two kinds of task schedulers in the system. The first is the global scheduler (GS) responsible for dispatching incoming external tasks among all processors. In the system operation we distinguish two phases. The first phase, called the initialization phase is concerned with servicing the incoming task which is defined on the basis of the batch of global decision requests. Once this batch is determined, the task force is composed of w occurrences of the root task type, where each task should be executed using different input data. The global scheduler, depending on the strategy used, assigns the tasks to the processors. This action is done before the start time of the request batch, just in the initialization phase. After the local queues are filled with the tasks, the processors start at the same time.

The first–come first–served (FCFS) queuing discipline is assumed. During the operation of the system, all further scheduling actions are performed in a distri-

buted manner by local task schedulers (LS). Every task can be processed on every processor with the same execution time. Once started, the execution of a task cannot be preempted. As soon as its execution is completed, the control from task can be passed to its immediate successor , if such exists.

A scheduling strategy can be classified as either sender–initiative (SI) or receiver–initiative (RI), depending on whether scheduling decisions are activated in the node where a task incomes (or is created) or in the node which is idle at the moment. In the paper both approaches are represented.

In SI strategy, whenever a task is completed, the local scheduler is invoked if there is a succeeding task in currently created task chain. Otherwise, the next task from local queue is selected for execution. The local scheduler sends this just created task using one of the SI allocation algorithms which are described in the next part of the paper, and selects for execution the first task in its local queue. Newly arriving tasks are queued and served basing on FCFS discipline. When a task arrives at a node, as a result of scheduling action, it is accepted without any objection. Task cannot be rejected or sent back. Execution of any task cannot be preempted and it is also assumed that all needed system actions can be performed simultaneously.

The RI approach requires much more overhead because the nodes which are underloaded must search for the load. Here we assume slightly less rigid requirement and only idle nodes ask for the load from the least loaded processors.

Both static and dynamic SI task assignment algorithms are proposed and evaluated in our application. There is also considered an approach which is called pre–scheduling. The RI policies are dynamic approaches.

The task force creation is governed by prescribed application. For the batch of global decision requests we get w independent jobs to be executed $J = \{J^{(1)}, J^{(2)}, ..., J^{(w)}\}$, where $J^{(1)}, l = 1, ..., w$ is the job required by l–th global decision requests. The task force contents changes every time whenever a job goes one step further, that is, to the next decision stage until successful completion of the whole job. Each time a job is completed, the number of task in the task force decreases by one. The task force state is specified by the numbers of tasks of each type. At the start moment we have w tasks of the root type and task force execution is completed after completion of its last task.

The proposed algorithms consists of two phases:

- phase 1: Initialization,
- phase 2: Execution.

In phase 1, the assignment of tasks included in the initial task force is made. This is done before the execution of the global decision batch starts. Two options are available: R – random, and P – pre–scheduling. In phase 2 which is concerned with the execution of jobs from the set J six basic SI options of task allocation are proposed: D1 – "least loaded processor" dynamic SI policy, D2 – "minimum expected completion time processor" dynamic SI policy, P – pre–scheduling SI policy, and R1, R2, R3 – random SI policies. Moreover two RI policies were

investigated: I1 – "a single call" RI policy, I2 - "constant call" RI policy. They can only be initiated when either P or R1 (described below) policy is chosen.

The random algorithm in the initialization phase is simple. Tasks from the initial task force are assigned at random to processor queues without checking any condition. In the execution phase, random policy has three variants: R1, R2 and R3. The first is a simple random assignment as above. Like in phase 1 all processors are taken into account as a possible task location. Policy R2 requires calculation of the threshold TH. TH is defined as the sum of the CPU time needed for execution of all ready–to–run tasks in particular processor queue and CPU time needed to complete current active task on that processor. There is also calculated the value of global threshold GTH which is the sum of individual TH's averaged over the number of processors. Then, for the R2 algorithm, when a new task is created, a remote processor other than local is chosen randomly and if its TH is less than or equal to GTH, then it takes that task. In R2 policy only one probe is permitted, whereas in the R3 policy probing is performed until suitable processor is found, but no more than three probes are allowed. In both algorithms, the task is queued locally when none remote processor is selected. The random policies R2 and R3 have the threshold options which consider the cases where a remote processor is to be found only when the local load exceeds TH. So we obtain policies R2T and R3T, respectively.

For the dynamic policy D1 each time the least loaded (in the TH sense) processor is searched among all remote processors in the case where local TH is greater or equal to GTH. If its TH is lower than local TH, then task is transferred to that processor, otherwise is passed to the end of local queue. In D2 case the task is transferred to chosen remote processor if the remote completion time is better then a local completion time for a particular task. The pre–scheduling approach is based on the values of f_i, $i = 1, ..., u$ and w which are used to calculate the pre–scheduling assignment $B = [b_{ki}]$, $k = 1, ..., p$, $i = 1, ..., u$, where b_{ki} is the number of tasks of specified type i, which are assigned to the k–th processor. This assignment is determined through exchange heuristics. There may be many feasible solutions or any assignment may satisfy the balancing constraint (6) for the given $\underline{d}$ and $\overline{d}$. The determination of B assumes some initial assignment which is important in further processing if we decide to use option P in Phase 2 or if there are some suggestions stemming from the application (Borzemski, 1992). Experiments were conducted for the initial assignment which assumed that all task type occurrences of type i are assigned to processor k, and $k = i$. From $f_i's$ the integer values of the number of task occurrences $h_i's$ are determined provided that the batch of w requests is to be processed. After that the values of processor loads q_k, $k = 1, .., p$ are calculated and constraint (6) is checked. The loads are determined in the same way as values of TH. If condition (6) is fulfilled for all $k's$, then stop, otherwise we iteratively apply a pairwise exchange scheme to the current assignment in order to obtain a feasible solution. We make an exchange of load gained by the tasks of the root node type. Load exchange is made each time between least loaded and most loaded processors. Iterative improvements are achieved by

partitioning the load difference between both processors, so that a feasible solution is obtained and partition of that load difference is made approximately evenly. The load q_k of k–th processor may be gained by tasks of different types but each time we may exchange only the load due to the tasks of the root type. The load exchanges are constantly tuned to match the value of h_1. Load exchanges are performed until success in fulfilling (6) or until no further advantageous exchanges are possible. After following such a procedure, the first column of B contains the numbers of tasks from the initial task force which will be assigned to particular processors. This is the pre–scheduling assignment used in the initialization phase. If the same option is chosen in the execution phase then the allocation scheme of tasks is the same as that assumed in the initial solution in determining B.

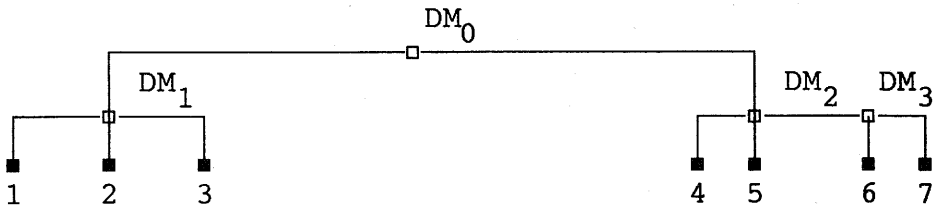
In Phase 2, it is possible to make use of the "idle processor" (I) SI option for algorithms P or R1. Such option combination results in assignment of a particular task to an idle processor if such exits, otherwise the policy P or R1 is used, respectively. Idle processor has no active task and empty queue.

The RI policies are as follows. "A single call" policy is a simple algorithm in which a processor completing a task without a successor and having no more tasks in its local queue, asks for a task from the least loaded processor. "Constant call" policy is more complicated because each time all idle processors ask for load. In case of issuing this request at the same time by more than one processors, the processors are loaded in order of its logical addresses in the network. The last task from queue can be taken each time but to preserve heavy task migration a task cannot be taken from a queue with only one task.

Each LB algorithm, for brevity is also referred to by an ordered list of attributes, namely option in initialization phase, option in execution phase with/without description of I version. So, in order to refer to an algorithm we list in order the acronyms corresponding to the attributes in the order as listed above – e.g. P/P/I algorithm is the pre–scheduling algorithm in both phases with the option "idle processor", whereas P/P is without that option.

6. Simulation Results

Several sets of experiments were conducted using a simulator designed for these reasons. Here we shall present results of some chosen experiments for the following application environment. A decision tree:



Distributed system parameters: $L = [L_j] = [5440, 3584, 6656]$, $u_{1j} = 1$ for

$j = 1,2,3$, $u_{2j} = 1$ for $j = 1$, $u_{3j} = 1$ for $j = 2,3$ and $u_{4j} = 1$ for $j = 3$. The optimum file allocation was determined as $y_{13} = y_{21} = y_{32} = 1$ for $b_k = 8000$, $k = 1,2,3,4$. The average data transmission rate was $v = 9600$ and LB constraints were $\underline{d} = \overline{d} = 0.15$. The average frequencies of task types occurrences, expected in the processing of arriving batches of decision requests, were assumed as $f = [f_i] = [1, 0.31, 0.69, 0.46]$, whereas the particular average execution times were $\tau = [\tau_i] = [1.32, 0.44, 0.5, 0.28]$. The system of $p = u = 4$ computers with full connectivity was considered. Three ranges of the load size have been considered: a load is L(ight) – load if $w \leq 26$, M(oderate) – load if $26 < w \leq 52$, and H(eavy) – load if $52 < w < 91$.

A simulator was implemented to study the performance of scheduling algorithms. The simulation was written in the Turbo Pascal language. We investigated several performance indices for each application environment but here only the completion time τ_{cp}, the speedup S_p, the average imbalance d_{ave} and maximum imbalance $d_{\max}$ for the given example are discussed. For a given set of jobs J, τ_c denotes its serial execution time without any processor idle time. τ_{cp} denotes the parallel execution time of that set of jobs in a p–processor system. For the given p, the speedup S_p is then defined as $S_p = \tau_c/\tau_{cp}$. d_{ave} and $d_{\max}$ are calculated for each schedule over p processors where local imbalance is $|\sigma_k - 1|$. All indices are averaged over several experiments. Experiments runs were conducted until the convergence of the indices values had attained about 2%. In Figures 3, 4, 5 and 6 the results of simulation experiments are plotted for selected allocation algorithms in the case of pure CPU–intensive computations where I/O actions can be neglected. This could be considered as a parallel system where no data transmission is taken into account. The comparison is made for the SI load balancing approaches. Next, in Figures 7–17 the results are shown for the prescribed distributed system where the task–distributed database dependence has been included. In this comparison we consider the RI approach, as well.

Taking into account the speedups in CPU–intensive cases, it appears that the P/P algorithm is superior to P/R1, P/R1/I and R/R1 algorithms. Inclusion of I option to P/P algorithm causes that P/P/I policy is better than P/P and also the R/R1/I one versus all loads. The P/P/I strategy is better than P/R2, comparable with P/R3 for all loads and with P/D for light load, and better than R/D, R/R2 and R/R3 for light load. The R/D, R/R2 and R/R3 algorithms have better performance indices than the other ones for moderate and heavy loads. In dynamic task allocation strategies we should take into account not only the load balancing itself, but also run–time scheduling overhead. The trade–off between load balancing and overhead incurred by the allocation scheme is the key factor in choosing appropriate dynamic allocation scheme. Thus in our case the P/P/I algorithm is an attractive proposal.

For I/O–intensive case we can compare the completion times of computation with and without load balancing in a distributed system. Then, for instance, the N/P algorithm can be considered as the basis in our comparison, on the assumption that N stands for batch loading of all root node type tasks to the root node processor

at the start moment and in the execution phase all the next task assignments are made according to the P rule which in this case is the same as in the initial solution given above for the matrix B. Then all tasks are assigned to their "own" processors. Now we can notice that the N option used in the initialization phase together with any SI algorithm used here can improve the performance no more than shown in Figure 7 due to the assumed values of τ_i. So,we can obtain different task assignments e.g. for N/P and N/R3 algorithms but the total performance depends mainly on the situation that happens when the last job is computed.

In Figure 8 it is shown that the P/R3T policy is better for the light load, whereas the R/R3T policy is better for the heavy load. Comparing the random threshold policies versus non–threshold policies we can conclude that in all cases, that is P/R2T v. P/R2, P/R3T v. P/R3, R/R2T v. R/R2 and R/R3T v. R/R3, the non–threshold policies were better. For example, considering the completion time, decrease in completion time was about 5%, for all pairs of policies compared and for all loads. This is a result of the tendency in R2 and R3 non–threshold policies for spreading out the tasks among all remote processors only, whereas with the threshold constraint the policies tend to process the tasks locally as long as possible. The latter approach appears to be not a good solution in the cases of jobs with chained tasks. Similar conclusions can be drown from the results presented in Figure 9 and Figure 10. Figure 11 shows that the P/P/I2 approach has the best performance for all loads. The general dynamic task allocation SI algorithms R/D1 and R/D2 (both versions have similar performances in this example – the D2 option is recognized to be better when differences between the execution times τ_{ki} are bigger), as well as dynamic allocations with pre–scheduling P in the initialization phase, are comparable to the P/P/I2 policy only for the heavy loads. This is caused by the fact that these algorithms may consider only newly coming tasks to achieve load balancing, and the light and moderate loads generated in the execution phase may not be enough to establish the balance, in comparison to the load assigned statically in the phase 1. In general, the dynamic options D1 and D2, the combinations with pre–scheduling in phase 1 have better performances. Figures 11, 12 and 13 confirm this conclusion.

Transmissions of tasks in the execution phase are plotted in Figure 14. Here we can observe that the increase is approximately linear for all algorithms but the most noticable increase is for I2 options used whereas the dynamic approaches, either D1 or D2, are least exhaustive. The tasks transmissions can be considered as an index in the evaluation of scheduling overhead. The trade–off percent decrease in the completion time and percent increase in tasks transmissions are shown in Figures 15 and 16, for P/P against P/P/I and P/P/I2 methods. The observation is that percent decrease in completion time is in all cases for all loads but for P/P/I2 policy we need more tasks transmissions than for P/P, whereas for P/P/I policy we may obtain even a decrease in tasks transmissions, compared to the P/P policy. This advantageous feature of the P/P/I approach should be taken into account when choosing the allocation policy.

In the real–time environments the obvious requirement is to minimize the average job completion time which can be the performance index from the user's point of view who is interested in expected value of the completion time of the global decision. It was verified in the simulation that for all allocation methods, this time increases linearly with the load. Similar mutual inter relationships are maintained between the task allocation algorithms as for their speedups. The obtained results are shown in Figure 17. It means that applying the P/P/I2 policy we may assure the minimum average job completion time for the given load but degradation in performance when applying the P/P policy, which is a simpler one, is less than 5% only. Together with a possible minimization of the average job completion time we have to take into consideration the problems connected with resources (i.e., processors) utilization. It is easier to meet both requirements using the P/P/I2 algorithm than the P/P algorithm. Such conditions have not been observed between P/D1 and P/D2, as well as between R/D1 and R/D2 policies. Choosing the load size for required average job completion time, which usually is required to be as small as possible, we may obtain better processor utilization, applying one of the P/P policies with the I, I1 or I2 option, than using the dynamic approach, either D1 or D2. This is the next substantiation to use the pre–scheduling approach in multistage decision–making real–time environment.

We discuss only one example. In the simulation, several examples of the multistage decision–making have been tested and similar conclusions could be drawn.

7. Conclusions

Simple and efficient load balancing algorithms for the multistage decision–making parallel system have been presented. The aim was to compare the general dynamic SI approach to a policy termed pre–scheduling. The example shown in this paper and other examples considered by the author have shown that the P/P policy together with one of the I, I1 or I2 options is the task allocation policy well adapted for the application type such as discussed here.

References

Bokhari S. (1988): *Partitioning problems in parallel, pipelined and distributed computing.*– IEEE Trans. Comput., v.37, No.1, pp.48–57.

Brzemski L. (1992): *File allocation in a distributed knowledge based multistage decisionmaking system.*– Systems Science, v.18, No.1, (in press).

Bourbakis N. (1988): *ANAGNOSTIS – An automatic text reading system.*– Microprocessing and Microprogramming v.23, No.1–5, pp.103–114.

Camurati P., Mezzalama M. and Prinetto P. (1988): *Application of AI Techniques in CAR environments.*– Proc. IFAC/IMACS Symposium on Distributed Systems: Methods and Applications, Oxford: Pergamon Press, pp.251–255.

Casavant T.L. and Kuhl J.G. (1988): *A taxonomy of scheduling in general–purpose distributed computing systems.*– IEEE Trans. Soft. Eng., v.14, No.2, pp.141–154.

Chou T.C.K. and Abraham J.A. (1982): *Load balancing in distributed systems.*– IEEE Trans. Software Eng., v.SE–8, No.4, pp.401–412.

Chou T.C.K. and Abraham J.A. (1986): *Distributed control in computer systems.*– IEEE Trans. Comput., v.C–35, No.6, pp.564–567.

Efe K. (1982): *Heuristic models of task assignment scheduling in distributed systems.*– Computer, v.15, No.6, pp.50–56.

Gu Y.X., Wang Q.R. and Suen C.Y. (1983): *Application of a multilayer decision tree in computer recognition of Chinese characters.*– IEEE Trans. Pattern Anal., Machine Intell., v.PAMI-5, No.1, pp.83–89.

Hać A. and Jin X. (1988): *Dynamic load balancing in a distributed system using a sender–initiated algorithm.*– Proc. 13th Conf. on Local Computer Networks, pp.172–180

Johnson I.D. and Harget A.J. (1989): *On the performance of load balancing algorithms in distributed systems.*– Information Processing 89, G.X. Ritter (ed.) Elsevier Science Pub. B.V., (North–Holland), pp.175–180.

Kasahara H. and Narita S. (1984): *Practical multiprocessor scheduling algorithms for efficient parallel processing.*– IEEE Trans. Comput., v.C-33, No.11, pp.1023–1029.

Kurzyński M. (1988): *On the multistage Bayes classifier.*– Pattern Recognition, v.21, No.4, pp.355–365.

Kurzyński M. and Puchala E. (1990): *Computer aided control of the peritoneal dialysis process in acute renal failure in children.*– 11th IFAC World Congress, Tallin, Estonia, Preprints, v.4, pp.33–38.

Levis A.H. (1988): *Human organizations as distributed intelligence systems.*– Distributed Intelligence Systems: Methods and Applications, D. Mladenov (ed.), Oxford: Pergamon Press., pp.5–11.

Luo J., Bruggenan F. and Reijns G.L. (1989): *A flexible transputer network for numerical applications.*– Microprocessing and Microprogramming, v.27, No.1–5, pp.405–412.

Ma P.Y.R., Lee E.Y.S. and Tsuchiya M. (1982): *A task allocation model for distributed computing systems.*– IEEE Trans. Comput., v.C-31, No.1, pp.41–47.

Ni L.M. and Hwang K. (1985): *Optimal load balancing in a multiple processor system with many job classes.*– IEEE Trans. Software Eng., v.SE–11, No.5, pp.491–495.

Ni L.M., Xu C.W. and Gendrau T.B. (1985): *A distributed drafting algorithm for load balancing.*– IEEE Trans. Software Eng., v.SE-11, No.10, pp.1153–1161.

Price C.C. and Salama M.A. (1990): *Scheduling of precedence-constrained tasks on multiprocessors.*– The Computer Journal, v.33, No.3, pp.219–229.

Stankovic J.A. (1989):*Decentralized decision making for task reallocation in a hard real–time system.*– IEEE Trans. Comput., v.38, No.3, pp.341–355.

Wah B.W. and Li G.J. (1989): *A survey on the design of multiprocessing systems for artificial intelligence applications.*– IEEE Trans. Syst. Man and Cybern., v.19, No.4, pp.667–692.

Wang Y.T and Morris R.J.T. (1985): *Load sharing in distributed systems.*– IEEE Trans. Comput., v.C–34, No.3, pp.204–217.

Wang Q.R. and Suen CH.Y. (1987): *Large tree classifier with heuristic search and global training.*– IEEE Trans. Pattern Anal. Machine Intell., v.PAMI-9, No.1, pp.91–102.

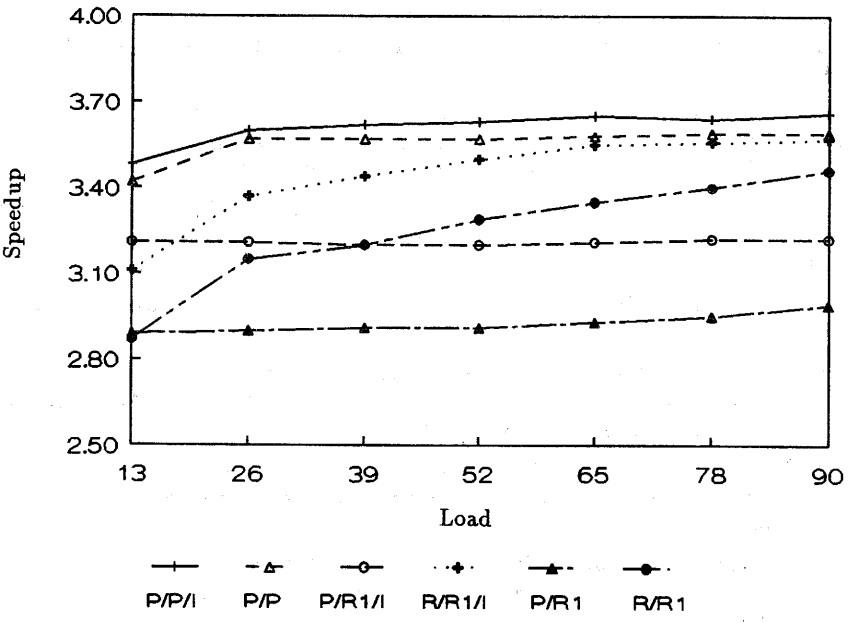


Fig. 3. Comparison of speedups.

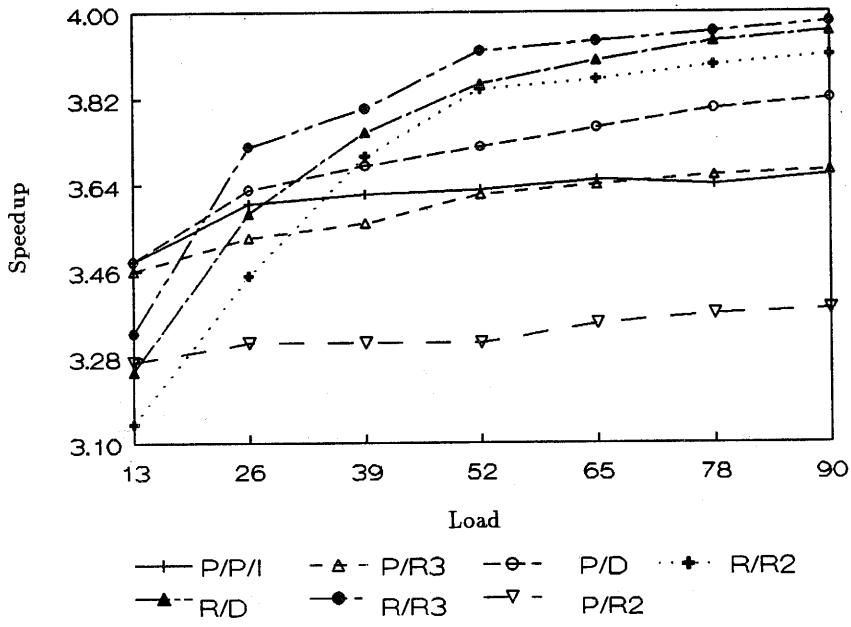


Fig. 4. Comparison of speedups.

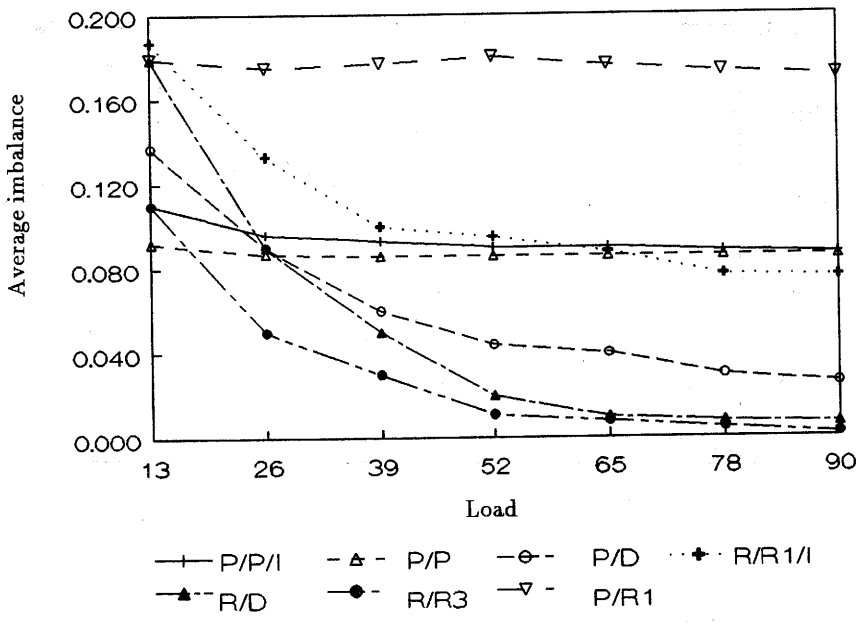


Fig. 5. Load imbalance comparison.

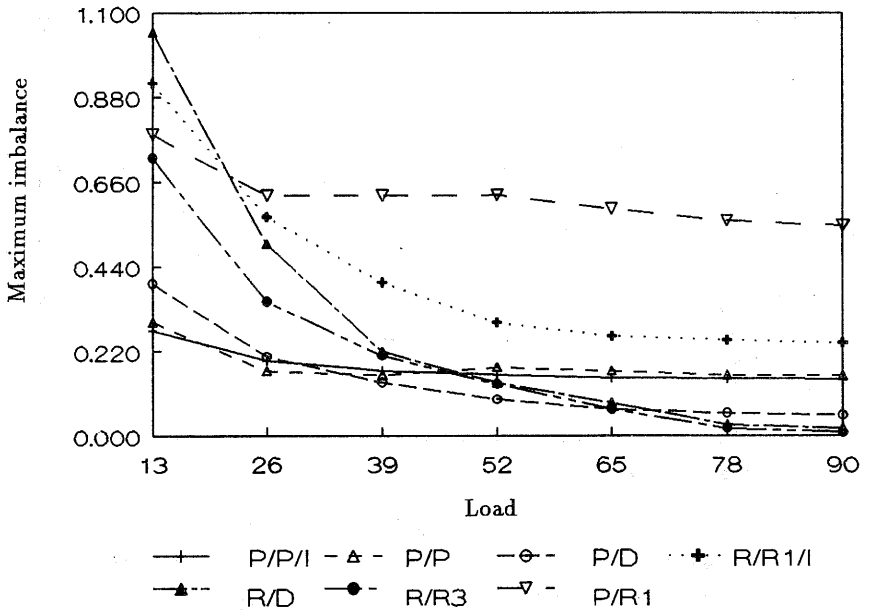


Fig. 6. Load imbalance comparison.

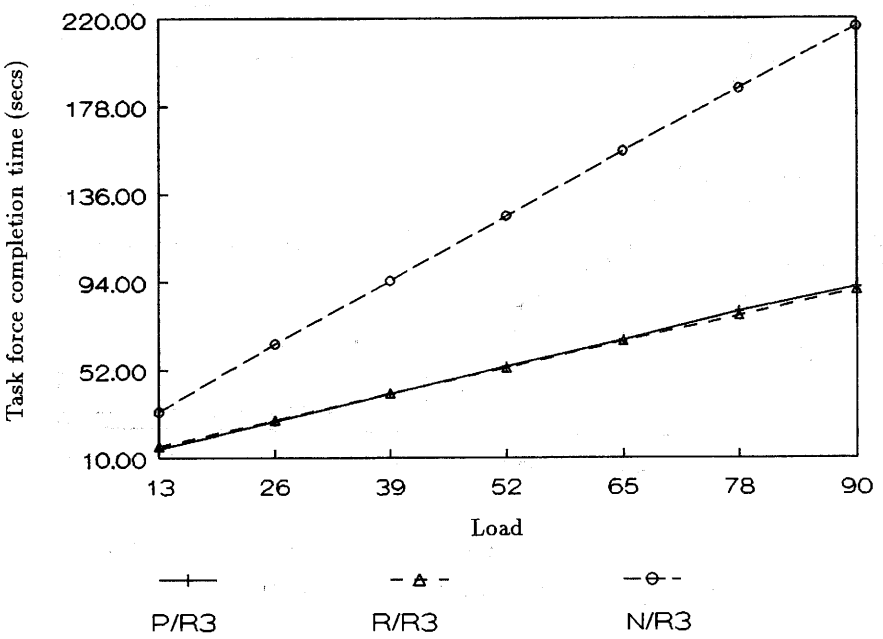


Fig. 7. Comparison of task force completion time.

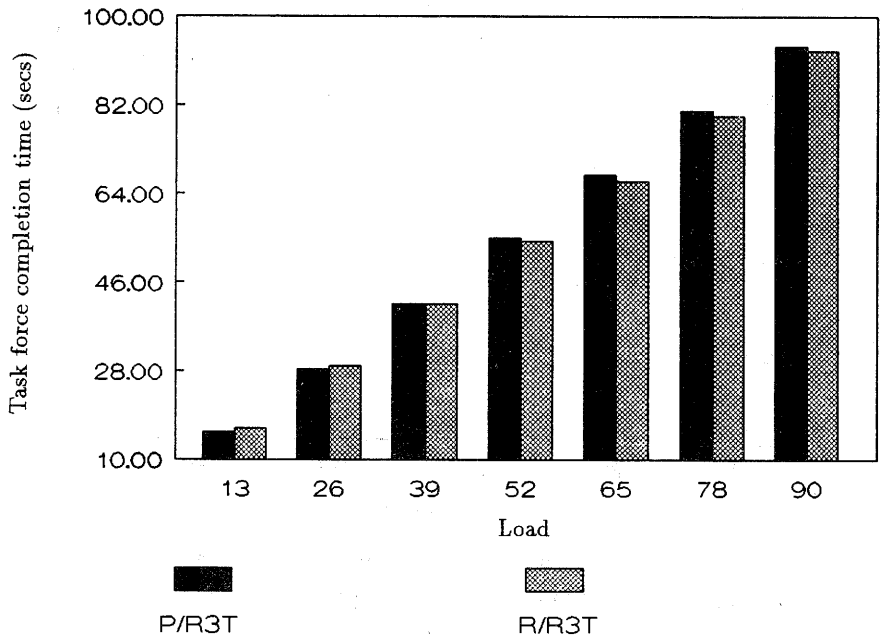


Fig. 8. Comparison of task force completion times.

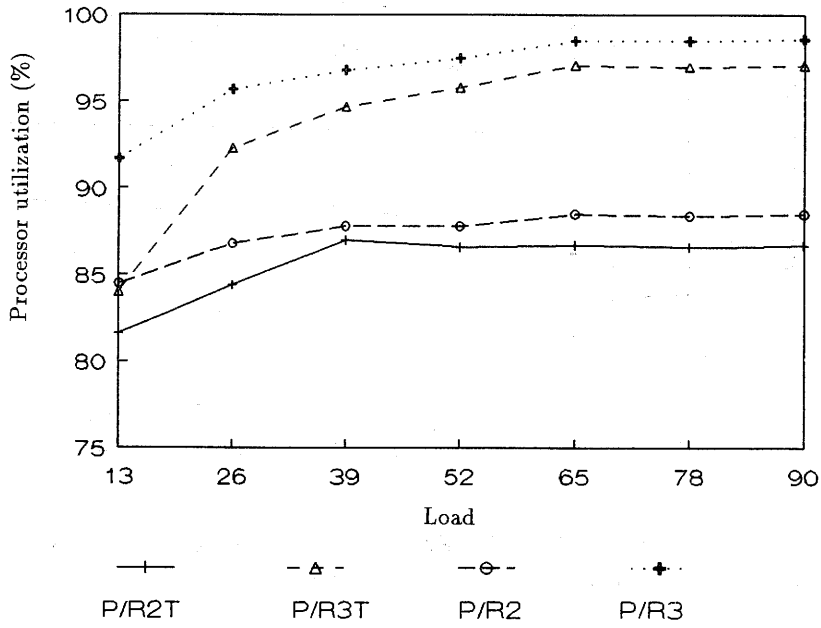


Fig. 9. Processor utilization comparison.

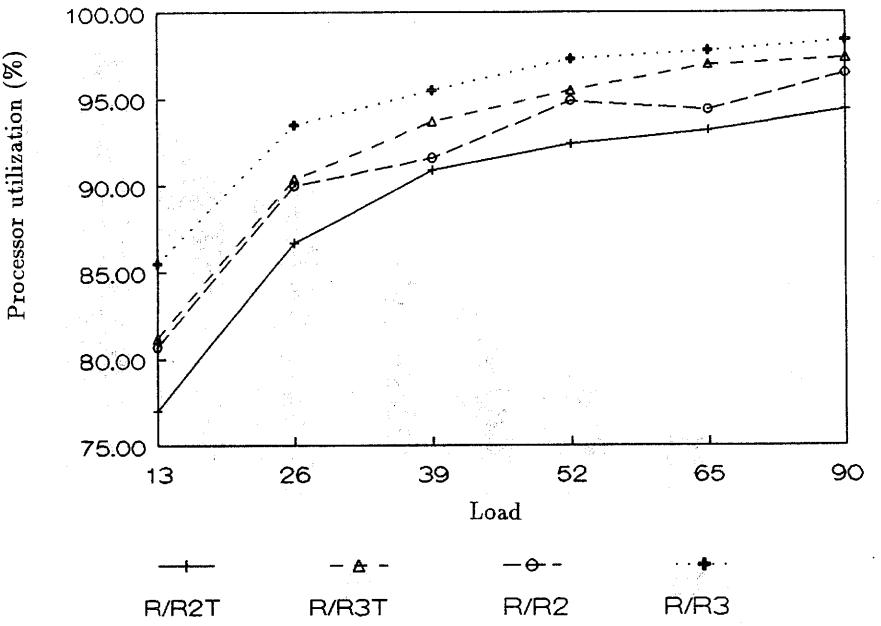


Fig. 10. Processor utilization comparison.

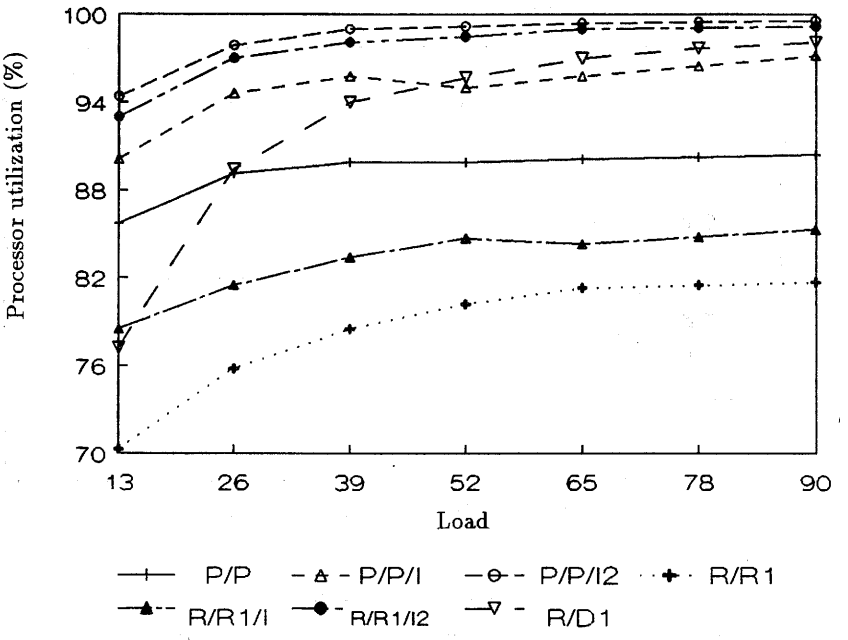


Fig. 11. Processor utilization comparison.

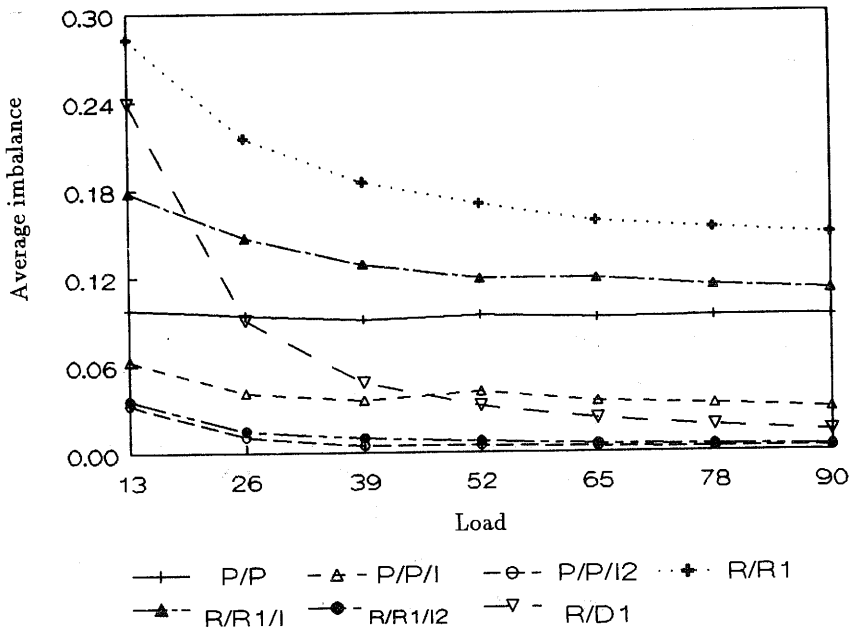


Fig. 12. Load imbalance comparison.

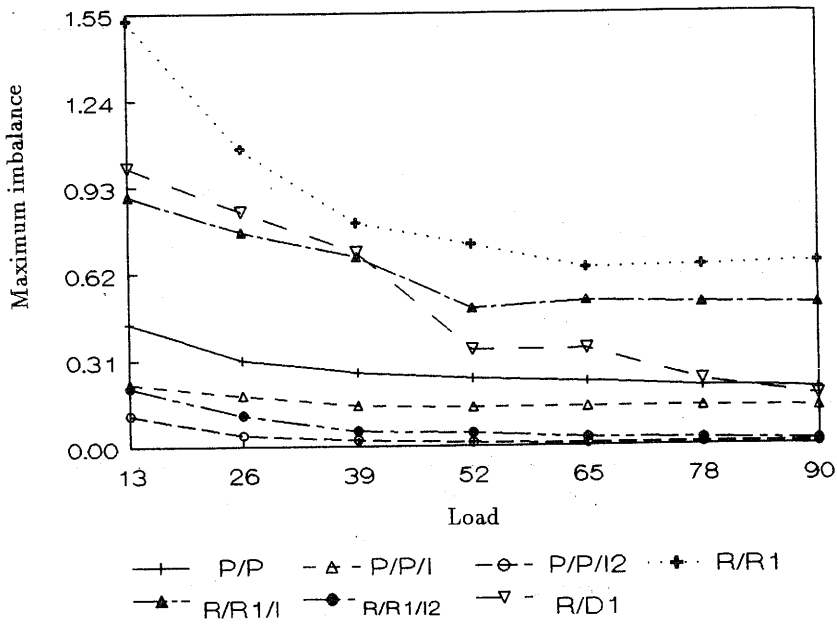


Fig. 13. Load imbalance comparison.

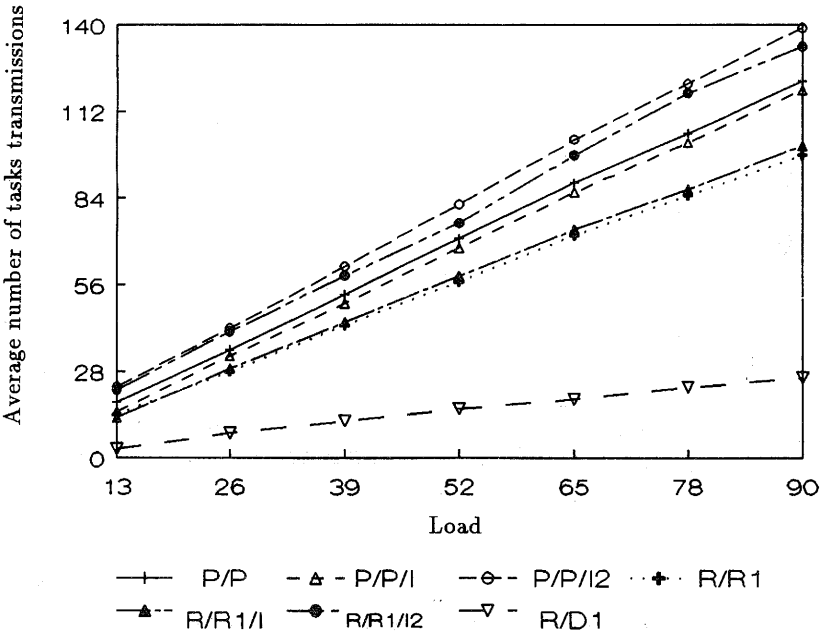


Fig. 14. Comparison of task transmissions.

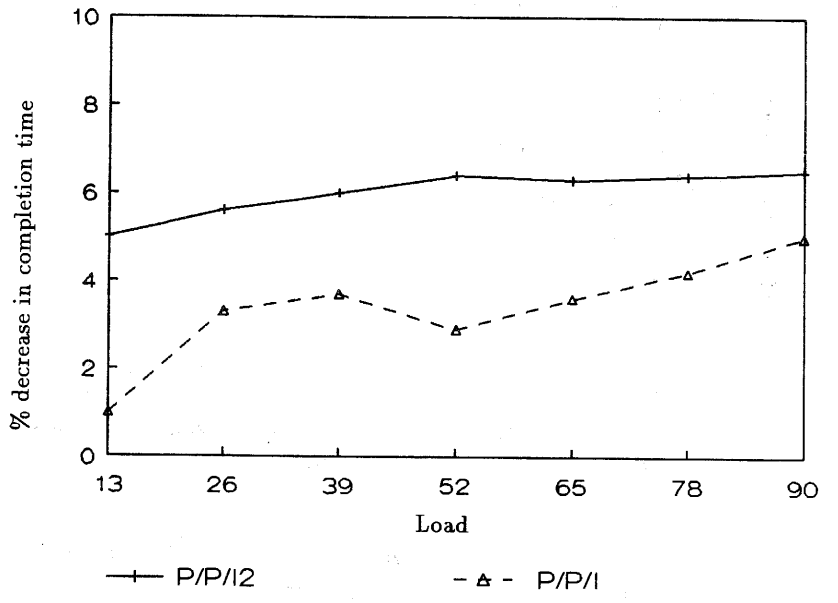


Fig. 15. Decrease in the task force completion time as against P/P.

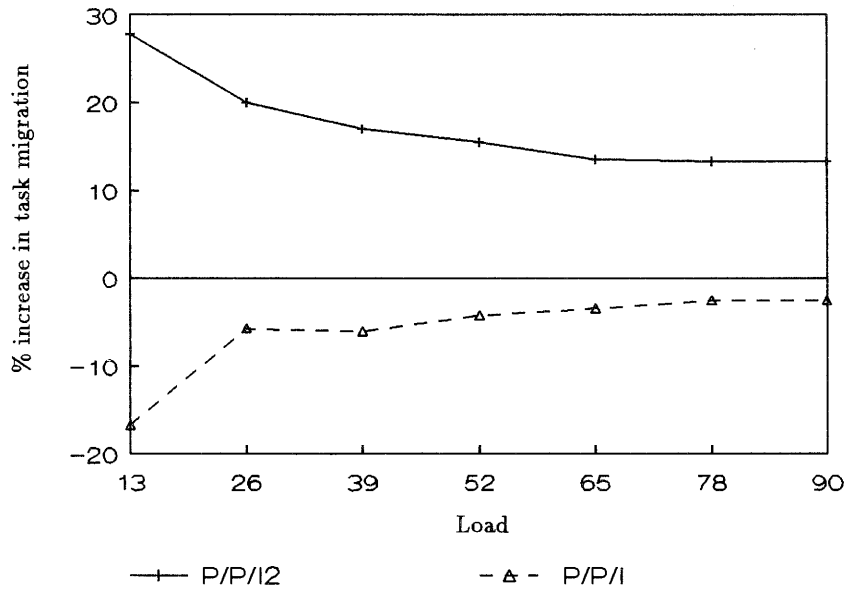


Fig. 16. Increase in the tasks transmissions as against P/P.

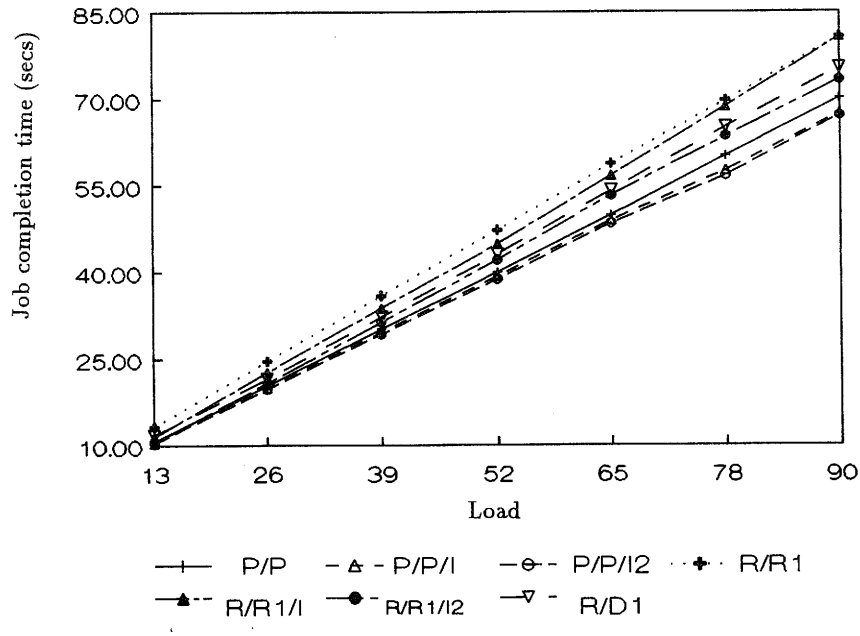


Fig. 17. Comparison of job completion times.