A SEMI-DISTRIBUTED LOAD BALANCING MODEL FOR PARALLEL REAL-TIME SYSTEMS

Kayhan Erciyeş, Öznur Özkasap, Nilgün Aktaş
Ege University Computer Eng. Dept., 35100 Bornova, İzmir, Turkey
e-mail: erciyes@baum01.ege.edu.tr, ozkasap@baum01.ege.edu.tr, aktas@baum01.ege.edu.tr

Keywords: parallel processing, load balancing, real-time system, deterministic scheduling.

Edited by:
Received: **Revised:** **Accepted:**

We propose static and dynamic load balancing policies for parallel real-time systems. A parallel real-time system in this context is considered as a computational environment consisting of a number of processors where stringent timing requirements of processes should be met. This would encompass massively parallel systems at one end of the spectrum and a group of computers connected by a local network at the other end. The static and dynamic load balancing policies developed are suitable for both types of systems with parameters such as communication costs to be tuned for each environment. For massively parallel processing systems, we introduce the concept of a domain which is a pool of processors and is governed locally for various services such as dynamic load balancing. The dynamic load balancing is implemented by central load balancers per domain which make use of the group communication facility for distributed communication with the other load balancers. This semi-distributed approach eliminates the need for maintaining a central node or replicated data by providing local data and control confined to that domain. The distributed data and control transfer is performed among the servers of the domains. The static scheduler however works off line for tasks with known characteristics such as execution time, communication constraints and deadlines prior to their execution which would be the usual case for hard real-time tasks.

1 Introduction

Recent developments in hardware technologies have made it possible to build systems consisting of clusters of processors usually referred to as Massively Parallel Processing (MPP) systems. MPP systems are increasingly finding many applications in hard real-time systems such as particle physics. A heterogeneous parallel real-time system is envisioned as a MPP system and host computers connected by a real-time network as shown in Fig. 1.

Our study focuses on load balancing in the MPP system of such an environment. We

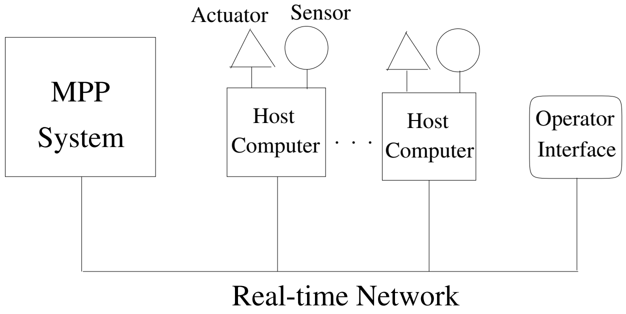

Figure 1: A Parallel Real-Time System

propose a deterministic scheduler, a dynamic load balancing mechanism and operating system modules to support dynamic load balancing. The central theme is to consider the MPP system as a collection of domains of processors which are managed centrally for various services in a domain and as distributed for interdomain services.

The system comprises minimum functionality required from a distributed memory parallel system to achieve coarse to medium grain parallelism for single or multiple applications. The deterministic scheduler accepts task graphs and the deadlines of tasks in the case of real-time tasks, as its input. It first puts the tasks with heavy communication into sets to be allocated to domains and then calculates various heuristic values for each task in the set and assigns these tasks to the processors in the domain according to these heuristic values. Different than the previous work in this area, namely, static scheduling of real-time tasks (Liu & Layland 1973), (Stankovic & Ramamritham 1988), we have considered communication costs among tasks and defined a heuristic which is a function of communication costs and is a component in the total heuristic value for a task.

The dynamic load balancing mechanism uses a semi-distributed approach. The periodically invoked central load balancer in a domain of processors tries to establish balance among them by first balancing the load within individual domains, then among different domains of the system in the second step. The operating system supports the dynamic load balancing mechanism by providing necessary group communication primitives for multicast communication.

2 Static Scheduling

Task Scheduling is one of the most challenging problems in parallel and distributed computing. Informally, the scheduling problem arises because the concurrent parts of a parallel program must be arranged in time and space so that the overall execution time of the parallel program is minimized. It is known to be NP-complete in its general form as well as several restricted cases. In an attempt to solve the problem in the general case, a number of heuristics have been introduced. The effectiveness of these heuristics depends on a number of factors such as grain size, interconnection topology, communication bandwidth and program structure (El-Rewini 1989).

The maximization of the speedup of a parallel program on a target parallel computer requires the allocation of the tasks among the processors in such a way that the total computational load is distributed as evenly as possible. To minimize the amount of processor idle time, the time required to perform necessary interprocess communication is minimized (Sadayappan & Ercal 1987).

Efe (Efe 1982) developed a heuristic allocation algorithm to balance processor load and to minimize communication cost. His algorithm consists of two phases. First, tasks are clustered with each other to optimize the communication cost and each cluster of tasks is assigned to a processor. Then, tasks are shifted from overloaded to underloaded processors in order to meet load-balance constraints. The algorithm is repeated until a satisfactory degree of load-balancing is achieved whereas we first group closely related tasks for domains and then allocate them individually to the processors of domains.

2.1 Problem Statement

This section describes the components of the static scheduling model we have developed. In general, there are four components in any scheduling system:

1. the target machine
2. the parallel tasks
3. the generated schedule
4. the performance criterion

In our model, the parallel application that contains real-time tasks is characterized by

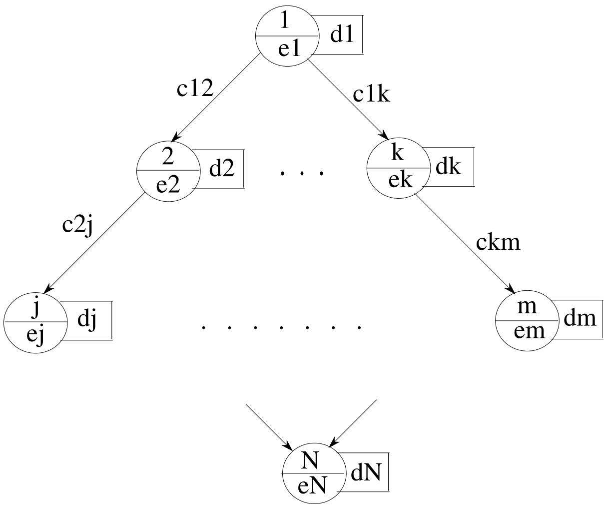

Figure 2: A General Task Graph.

an acyclic directed graph G(V,E) as shown in Fig. 2. Vertex weights $V = \{t_i : i = 1, 2, ..., N\}$ represent computational load, or worst-case execution time of the tasks, and edge weights $E = \{c_{ij}$: for $\forall t_i, t_j$ where $p_{ij} = 1\}$ represent interprocess communication costs. Precedence relation between tasks is defined as follows :

$p_{ij} = 1$ if there exists a precedence relation $>$ between t_i, t_j task pairs
$p_{ij} = 0$ otherwise

Since, real-time systems are static and it is assumed that all task characteristics are known a priori (Stankovic & Ramamritham 1988), $\forall t_i$ of the parallel application has known timing characteristics such as execution time (e_i for $\forall t_i$) and deadline (d_i for $\forall t_i$)

The topologies of the target Parallel Processing Architecture (PPA) are in the form as shown in Fig. 3. $P = \{P_i : i = 1, 2, ..., M\}$ is a set of homogenous processors with local memory, which communicate via message passing paradigm or channel structures. A delay matrix, D is introduced to represent physical overheads among processors in the parallel processing system.

$$D = \{D_{ij} : \text{number of hops between processors } P_i \text{ and } P_j \text{ for } \forall P_i, P_j \text{ where } i \neq j\}$$

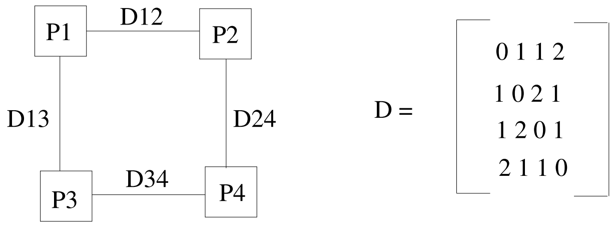

Figure 3: A Target PPA (Parallel Processing Architecture) and its delay matrix.

The intraprocessor overhead D_{ii} for $\forall i$ is assumed to be zero. Also, speeds of processors in the system are assumed as equal, that is, they are homogeneous processors.

We considered that N > M where N is the number of tasks in task graph and M is the number of processors in the system.

Definition: A *domain* is a group of fixed-size processors, which includes closely related, therefore heavily communicating tasks of an application. Processors forming a domain are selected as physically closed ones in the parallel processing architecture.

Definition: If tasks t_i and t_j are mapped to processors P_i and P_j respectively, and $p_{ij} = 1$ (that is, there exists a precedence relation between t_i and t_j), then the *cost function* is defined as $F(C) = min(\sum c_{ij} D_{ij}$ for $\forall t_i, t_j)$

2.2 Scheduling Model

The deterministic scheduler of the load balancing model proposed works off-line for scheduling tasks of a parallel application. Inputs of the scheduler are a task graph that shows precedence constraints, communication costs among tasks, execution times and deadlines of tasks, and interprocessor communication delays for the PPA. The off-line scheduler works in two phases:

Domain Allocation : Tasks with heavy communication are put into groups to be allocated to domains.

Task-to-Processor Mapping : Heuristic values for each task in a domain are calculated and the tasks are assigned to the individual processors in their domain.

Domain Allocation Algorithm: The Domain Allocation (DA) Algorithm that is shown in Fig. 4 considers timing constraints of tasks such as execution time, precedence constraints among tasks and interprocess communication. The procedure is to group tasks into domains of the parallel processing system. The objectives considered during this procedure are putting tasks with heavy communication into the same domain in order to minimize communication costs and balancing them by trying to ensure that total execution time of tasks allocated to each domain are approximately the same. The result obtained is an allocation scheme that permits parallel execution of tasks in the target PPA.

Task-to-Processor Mapping Algorithm The Task-to-Processor Mapping (TPM) Algorithm is shown in Fig. 6. Processor allocation for tasks is performed by calculating heuristic values for each task as described in section 2.4 and assigning these tasks to the individual processors in their domain according to these heuristic values.

Once the first step of the scheduling model, namely, domain allocation is performed, task-to-processor mapping should be completed for each domain of the parallel processing system. This second phase can be performed in parallel for each domain as shown in Fig. 5. This makes the scheduling process faster especially for MPP systems where there are hundreds of processors, and many domains.

2.3 Task Graph Generation

A task graph generator, which generates schedulable task graphs with tasks having timing and precedence constraints is developed. This strategy allows us to evaluate performance of domain allocation and task-to-processor mapping algorithms using various heuristic functions on different parallel applications characterized by generated task graphs.

Inputs: *Task Graph (TG) with N tasks;*
number of domains (DN);
. $TE = \sum_{i=1}^{N} e_i$;
. $AE = TE/DN$;
. *Sort c_{ij} in descending order for $\forall\ t_i, t_j$*
where $p_{ij} = 1$;
. *Get first element c_{ij} of the sorted list;*
. *Set current domain (CD) to first domain;*
. *Put t_i,t_j into CD;*
.*DO UNTIL $\forall\ t_i$ allocated to a domain*
{ *if ($\sum e$ tasks in CD < AE)*
Find $\max c_{ij}$ for t_i and t_j
that are not allocated to a domain;
Put t_j into CD
Delete_comm_cost();
else
*if(CD < DN) CD++; /*Set current domain*
to next domain/*
else set CD to domain with $min(\sum e)$;
if there exists c_{ij} where
t_i and t_j are not allocated to a domain
Put t_i, t_j into CD;
Delete_comm_cost();
else if ($\sum e$ of tasks in domain DN==0)
DN–;
AE = TE / DN;
Get next element c_{ij};
if (t_i or t_j is not allocated to a domain)
CD = Domain of task t that is allocated;
Put task into CD;
Delete_comm_cost();
}

Figure 4: Domain Allocation Algorithm

SEQ
Allocate Domains
PAR i=1 FOR DN
Do Task-to-Processor Mapping for domain i

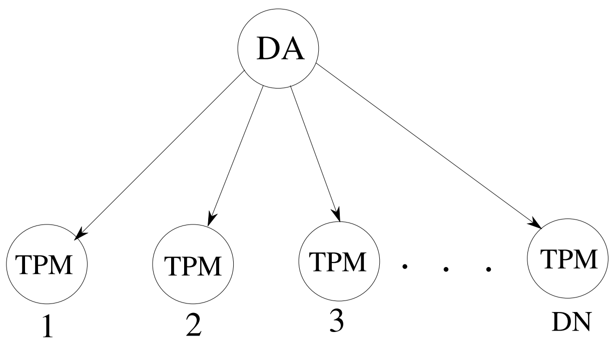

Figure 5: Task to Processor Mapping.

Inputs: *Task Graph (TG) with N' tasks allocated for the domain;*
Delay Matrix D;

. n_i = number of immediate predecessors of t_i for $i = 1, ..., N'$
. Task_type=READY for $\forall t_i$ where $n_i = 0$
. Insert READY tasks into Sched_list according to heuristic values at time =0
.WHILE (Sched_list is not empty)
{Get a task t_i from the Sched_list;
Switch(Event_type)
case READY :
map t_i to an idle processor P_j;
Event_type = FINISHED;
Time = Finish_time of t_i on P_j
Insert_event into Sched_list;
case FINISHED :
For each immediate successor t_k of t_i
$n_k = n_k$ - 1;
if (n_k == 0) Event_type = READY
Time=Finish_time of t_i on P_j;
Insert_event into Sched_list;
}

Figure 6: Task to Processor Mapping Algorithm.

Task Graph Generation Algorithm: The approach uses the strategy of randomly assignment of tasks to the degrees or positions of the graph. Then predecessor and successor task(s) of each task are determined. Finally, timing constraints of each task and communication costs between related tasks are assigned as shown in Fig. 7.

2.4 Scheduling heuristics

During task-to-processor mapping, we have used the list scheduling method. List Scheduling is a class of scheduling heuristics in which tasks are assigned priorities and placed in a list ordered in decreasing priority. Whenever tasks contend for processors, the selection of tasks to be immediately processed is done on the basis of priority with the higher priority tasks being assigned to processors first. The heuristic functions that determine the priorities of processes can be explained as follows:
EDF (Earliest Deadline First): Priority is

- *Randomly assign N tasks to d degrees in task graph*
- *Determine predecessor(s) and successor(s) of each task under the constraints such as number of tasks in a degree and number of degrees in graph*
- *Handle timing constraints (e_i for $\forall t_i$) (d_i for $\forall t_i$)*
- *$E = \{c_{ij}$: for $\forall t_i, t_j$ where p_{ij}=1$\}$ of each task generated*

Figure 7: Task Graph Generation Algorithm

given to the real-time tasks with the earliest-deadline.
MLF (Minimum Laxity First): Priority is given to the real-time tasks with minimum laxity where task laxity = task deadline - task execution time.
H3:min(EDF*W1+MLF*W2): The first two heuristic functions are combined by using weight values. We have developed a new heuristic function by using simulation results for EDF and MLF heuristic functions.

3 Dynamic Load Balancing

The random arrival of processes in a parallel processing system can cause some processors to be heavily loaded while other processors are idle or lightly loaded. Dynamic load balancing improves the performance by transferring tasks from heavily loaded processors, where service is poor, to lightly processors where the task can take advantage of computing capacity that would otherwise go unused.

Most of the methods used for dynamic load balancing are either fully distributed or centralized methods. Neither fully distributed nor centralized load balancing policies are known to yield good performance for MPP systems. Fully distributed algorithms use a small amount of information about the state of the system. Small systems can yield good performance with limited information, but this may not be true for large systems. Despite the fact that fully distributed algorithms incur less overhead due to message exchange, this overhead linearly increases with the system size. Centralized algorithms do have the potential of yielding optimal performance, but require accumulation of global information which can become a formidable task. The storage requirement for maintaining the state information also becomes prohibitively high with a centralized model of the large system. For a large system consisting of a hundred or thousands nodes, the central scheduler will become a bottleneck and lower the throughput.

Besides, centralized models are highly vulnerable to failures. The failure of any software or hardware component of the central scheduler can stop the operation of the whole system.

In our study, a semi-distributed dynamic load balancing model is developed for a distributed memory computer system. In this case, the processors of an MPP system are divided into domains of fewer processors which are managed centrally for various services and distributed for others. Domains are allocated dynamically during run time in some researches (Kremien et. al. 1993), whereas our system is divided into domains in a static manner before the system starts. A group management module designed for the underlying distributed operating system provides the necessary group communication in multicast mode for the manager processes. A system process in each domain called Central Load Balancer (CLB) first tries to balance the load within its domain. If this is not possible, it communicates with other CLBs to find a destination node for the candidate process for migration as depicted in Fig. 8.

This model proposes a two level load balancing strategy. At the first level, load balancing is carried out within individual domains where the central node of each domain acts as a centralized controller for its own domain. At the second level, the load is balanced among different domains of the system, thus providing a distributed environment among domains. The design of such a strategy involves designing an algorithm for performing optimal task scheduling and load balancing within a domain as well as among domains and developing efficient means for collecting state information at interdomain and intradomain levels.

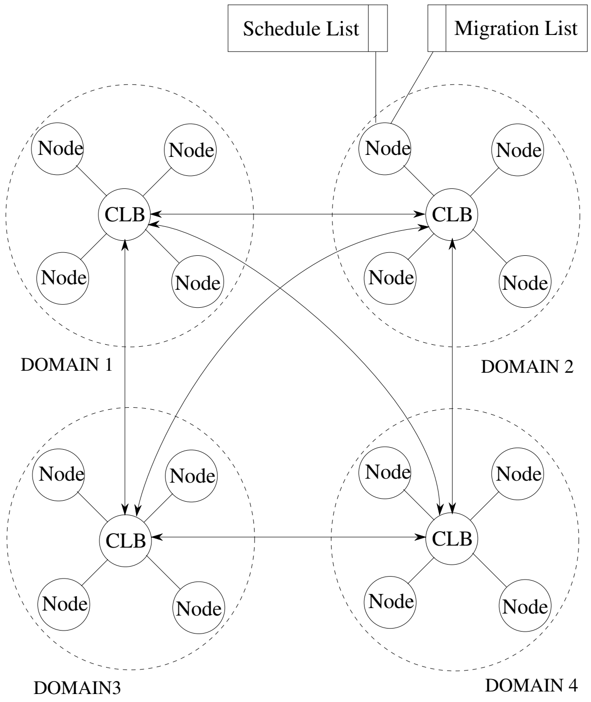

Figure 8: Semi-distributed System Model

Central load balancers are responsible for dynamically assigning processes to individual nodes of the domain, transferring the load to other domains if required, and maintaining the load status of the domain and nodes. As a result of load balancing and sharing, a process can be completed earlier due to the utilization of otherwise idle or lightly loaded processors.

3.1 Real-time Approach

In a conventional multitasking operating system, processes are interleaved with higher importance (or priority) processes receiving preference. Little or no account is taken of deadlines. This is clearly inadequate for real-time systems. These systems require scheduling policies that reflect the timeliness constraints of real-time processes .

A realistic hard real time system must guarantee both periodic and non-periodic hard real-time processes on the same processor, utilize spare time by non-critical processes, initialize static allocation of periodic processes and migrate aperiodic processes for response to changing environment conditions or local overload.

Schedulers produce a schedule for a given set of processes. If a process set can be scheduled to meet given pre-conditions, the process set is termed feasible. A typical pre-condition for hard real-time periodic processes is that they should always meet their deadlines. An optimal scheduler is able to produce a feasible schedule for all feasible process sets conforming to a given precondition. For a particular process set, an optimal schedule is the best possible schedule according to some predefined criteria. Typically a scheduler is optimal, if it can schedule all process sets.

The system model that has been used allows both periodic and aperiodic processes. Precedence constraints among processes are enforced by using the process's start times and deadlines and no process resource requirements are considered. Context switch have zero cost and multi-node, multi-domain systems with dynamic process allocation is allowed.

3.1.1 Deadline Characteristics

Periodic processes are characterized by their period and their required execution time per period. For each periodic process, its period must be at least equal to its deadline. That is, one invocation of a process must be completed before successive invocations. This is termed as the runnability constraint as shown below:

$$computation\_time \leq deadline \leq period$$

The activation of an aperiodic process is essentially, a random event and is usually triggered by an action external to the system. Aperiodic processes also have timing constraints associated with them; i.e. having started execution, they must complete within a predefined time period. It is not guaranteed that aperiodic processes will certainly meet their deadlines. If it is not possible to schedule an aperiodic process on any processor before its deadline, this process is said to be unschedulable. Aperiodic processes can be invoked at any time.

It has been showed that the algorithms that are optimal for single processor systems are not optimal for increased numbers of processors. In a multiprocessor or distributed system, processes that are considered likely to miss their deadlines have to be migrated to other processors. But it has also been showed that it is better to statically allocate periodic processes rather than let them migrate and, as a consequence, potentially downgrade the system's performance (Audsley & Burns 1990).

3.1.2 Scheduling Policy

Each process is characterized by (A,S,C,D) known at the time of process arrival, where A is the process's arrival time, S is the earliest possible time at which its execution may begin (start time), C is the maximum computation time and D is the deadline by which it must complete its execution. An aperiodic process is described by (A,S,C,D). For each periodic process, an (A,C,P) describes its arrival time, computation time, and period.

The algorithm schedules sets of processes ordered by increasing deadlines. Given such a set, the algorithm selects the first process and schedules it as near to its start time as possible (i.e. at the earliest available time after its start time). The process is scheduled by simply accumulating all unused processor time past the process's start time until sufficient computation time is found. If the resulting schedule permits this process to complete before its deadline then the process is schedulable, else it is unschedulable.

The scheduling information used by this algorithm is recorded in a list. Each element of the list represents a time slot already assigned to a process, and has four fields: starting time, ending time, a pointer to the next list element, a pointer to the previous list element. Given this list, the process schedulability is analyzed by searching the list for available time intervals between two elements. This search starts at an element compatible with the process start time and ends at a time point compatible with the process deadline or when the accumulated length of available time is equal to the process computation time. The process is schedulable if sufficient computation time is found before its deadline during this search, else the algorithm reports the process as unschedulable.

In our system, aperiodic processes do not have hard deadlines, hence they can be migrated to other processors when they can not be scheduled on present processor. On the other hand, periodic processes have hard deadlines and they can not be migrated to other processors. They are statically allocated when the system starts by the deterministic scheduler as explained in Section 2.

In order to schedule an aperiodic real-time process with a soft deadline dynamically, a modified form of Bryant and Finkel's algorithm is employed.

3.1.3 Bryant and Finkel's Algorithm

Bryant and Finkel's algorithm (Bryant & Finkel 1981) is a dynamic and physically distributed algorithm. In our system, Bryant and Finkel's algorithm is used in a semi-distributed fashion, considering the deadlines of the aperiodic processes. To make a decision, processors cooperate by sending negotiation messages. The decisions are sub optimal and heuristic approach is used to find solution.

A newly arriving aperiodic process can be called as schedulable only if its scheduling does not danger previously scheduled processes. First, the new aperiodic process is placed in order into the list, which holds all previously scheduled processes on this processor. Then processes in the list are rescheduled, using the algorithm explained above. If any of the previously scheduled processes is unschedulable now, then newly arriving aperiodic process is determined as unschedulable on this processor. Otherwise it is schedulable. When a process is determined as unschedulable at that node, a timer starts to work. When the timer reaches the value that is equal to deadline minus the execution time of that process, then this process is said to be unschedulable elsewhere.

By using semi-distributed approach, unschedulable newly coming aperiodic processes are tried to be scheduled at intradomain level. Each node in a domain sends its schedule list and unschedulable aperiodic processes list to the CLB periodically. CLB collects this information, then tries to find appropriate empty time slots on different nodes of its domain for unscheduled aperiodic processes. If it can find such a node, then this node is determined as destination node, and process migration takes place. Otherwise the process is said unschedulable within this domain. If such a condition occurs the strategy works in the interdomain level in the following way:

1. The CLB of domain A (CLB_A), sends a query to one of its nearest neighbors CLB of domain B (CLB_B), to form a temporary pair, which enables a controlled, stable environment suitable for process migration. The query has two purposes:
   a. it informs the CLB_B that CLB_A wishes to form a pair
   b. it contains a list of processes and time constraints for each processes.

2. CLB_B after receiving the query can perform one of three options:
   - rejecting CLB_A 's query; this implies that CLB_A must send a query to another neighbor domain
   - form a pair with CLB_A; this implies that CLB_A as well as CLB_B reject all incoming queries until the pair is broken.
   - postpone CLB_A when CLB_B is in a migrating state, that is, sending processes this implies that CLB_A must wait until CLB_B forms a pair with it, or rejects it- CLB_A cannot query anyone else.

3. After establishing a pair, CLB_A sends unschedulable aperiodic real-time processes list to the CLB_B. Then CLB_B broadcasts this information to all of its nodes. Nodes try to schedule these processes on their own schedule table. If this is pos sible, scheduled processor id and its response time are returned to the CLB_B. As the last step, CLB_B compares the response times of the same processes on different nodes and selects the node giving the minimum response time as the destination node.

4. If no processes can be executed on CLB_B, then CLB_B informs CLB_A of this fact and the pair is broken. Otherwise the processes are migrated. This process is repeated for all remaining unscheduled aperiodic real-time processes until no process is left.

4 Operating System Support

In order that the central load balancers can communicate efficiently, the operating system should provide some form of multicast communication. The group management and naming modules described below were added on top of the existing facilities of the NX/2 kernel of the Intel iPSC/2 hypercube simulator.

4.1 Group Management

Processes that are functionally related to finish an overall task are included in a group. These processes communicate frequently in multicast mode where one process which is the member of a group sends a message to all other members of the group as one-to-many communication (Cheriton & Mann 1988).

In our system, groups could be distributed over the processor domains. Each processor domain has a group server that initiates all local group communication primitives, like make_group, kill_group, join_group, leave_group, etc. Each group server is responsible for only the local members of any group. All local members that are located in a domain send their requests to their own group server of the domain. Only group server could be in contact with other members of the group via other group servers if needed.

When a process in a domain wants to create a group, it sends that requests to the group server. Group server creates a group control block and sends that request to the group servers on the other domains. However joining to a group or leaving a group are done locally. No interdomain communication is needed for these frequent services. When a process from a group wants to send a group message to all other group members over the system, it first sends that request to the local group server which then sends the message to all other members in that domain and pass the message to all other group servers. These group servers on the other domains do the same thing simultaneously. So group communication is handled in parallel by the group servers distributed over the domains of the MPP system.

4.2 Naming

A naming facility in a distributed system, in general should provide a mapping from system names to addresses where the object is residing, and a route to specify how to get there (Goscinski 1991). The naming is implemented by the name servers in each domain which hold a subset of the global naming space for the objects in that domain (Cheriton & Mann 1988). The name servers form a process group and communicate as described above. The name server in a domain contains the address of the objects in its domain and can receive a request from a local processor or another name server for an address of an object. If the request is local, a check is made to find if the object is located at an address in that domain. If this is not the case, a broadcast message is sent to all name servers to receive the address. In this case, only the name server which has the address of the object will respond.

5 Implementation And Results

The proposed load balancing techniques were implemented in an Intel iPSC/2 hypercube simulator running on OSF/1 MACH Unix environment with the following results obtained.

5.1 Static Scheduling Results

The performance of TPM (Task-to-Processor Mapping) and DA (Domain Allocation) algorithms were evaluated by using sample task graphs generated randomly by TGG (Task Graph Generation) algorithm, generally in the form of supervisor-worker model. Results were obtained for different number and topologies of processors, several heuristic functions for real-time tasks, and different number of domains.

Parallel system topologies are selected as 4-processor mesh topology, and 8 and 16-processor hypercube topology. Processor domain numbers are selected as follows:

- DN=1 and 2 for 4-processor case
- DN=1, 2 and 4 for 8-processor case
- DN=1,2,4 and 8 for 16-processor case

DN=1 means there is only one domain in the system. In this case, domain allocation phase is not used.

For different domain numbers, we have obtained time measures for task-to-processor mapping and percentage of real-time tasks whose deadlines are met. The following are some of the results we have obtained for different heuristic functions:

EDF (Earliest-Deadline-First) Heuristic: Fig. 9 shows time spent during static mapping of tasks to processors for DN=1, DN=2, DN=4 and DN=8 where M=16 with hypercube topology. N varies between 24 and 48 interrelated tasks.

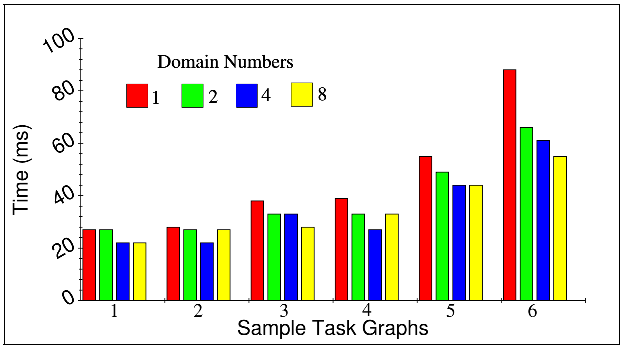

Figure 9: Time Spent for Static Mapping in EDF

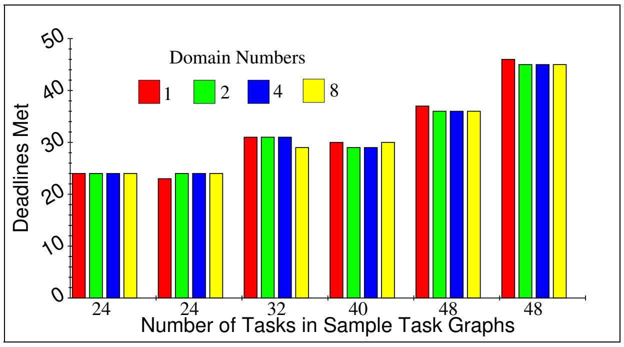

Figure 10: Deadlines met in EDF

Fig. 10 shows the number of real-time tasks whose deadlines are met at the end of our static scheduling scheme versus the number of real-time tasks in sample task graphs for the parallel system with M=16 processors and hypercube topology.

MLF (Minimum-Laxity-First) Heuristic: Fig. 11 shows time spent during static mapping of tasks to processors with varying domain size where M=16 with hypercube topology. N varies between 24 and 48 interrelated tasks.

Fig. 12 shows the number of real-time tasks whose deadlines are met at the end of our static scheduling scheme versus the number of real-time tasks in sample task graphs for the parallel system with M=16 processors and hypercube topology.

$H3 = (EDF * W1 + MLF * W2) Heuristic$: Fig. 13 shows time spent during static mapping of tasks to processors for DN=1, DN=2, DN=4 and DN=8 where M=16 with hypercube topology. N varies between 24 and 48

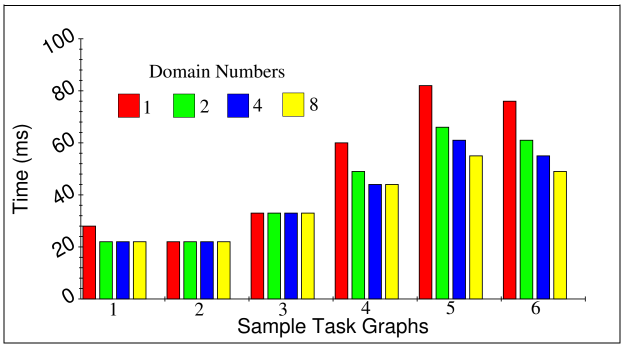

Figure 11: Time Spent for Static Mapping in MLF

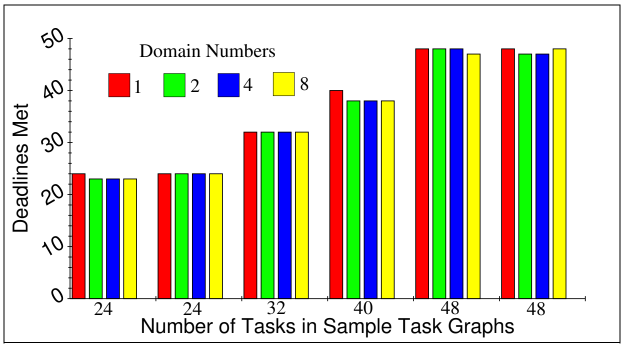

Figure 12: Deadlines met in MLF

interrelated tasks.

Fig. 14 shows the number of real-time tasks whose deadlines are met at the end of the static scheduling scheme versus the number of real-time tasks in sample task graphs for the parallel system with M=16 processors and hypercube topology.

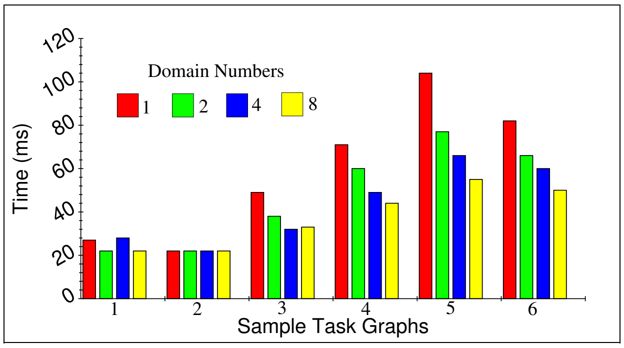

Figure 13: Time Spent for Static Mapping using H3

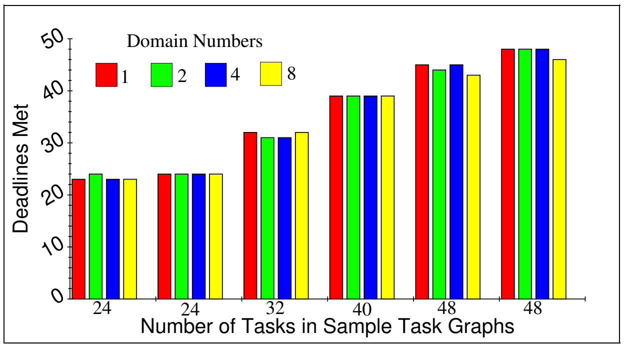

Figure 14: Deadlines met in H3

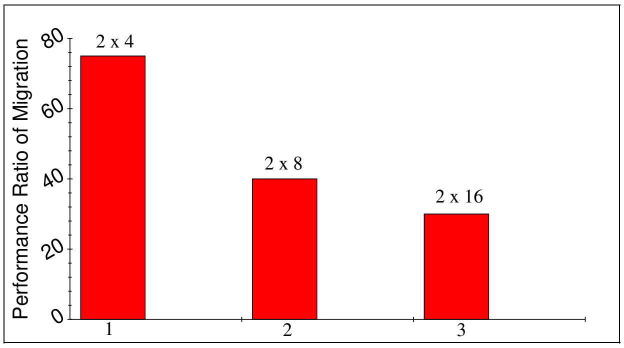

Figure 15: Test of Real-time Approach When Domain Number is Constant and Processor Numbers in Each Domain are Variable.

5.2 Dynamic Load Balancing Results

The dynamic load balancing mechanism is implemented for soft real-time processes described in Section 3 by adding group management and naming modules to the existing NX/2 kernel of the hypercube simulator. The semi-distributed central dynamic load balancers are system processes which communicate with other load balancers to perform load transfer. Process migration facility is simulated. Real-time approach is implemented and system's performance is observed for different number of domains and processor numbers. In figures 15-17, the percentages of aperiodic processes meeting their deadlines that have been in migration list are shown.

Fig. 15 shows the performance of the system when domain number is 2 but number of processors in a domain is varying as 4, 8 and 16. When the domain number is constant but

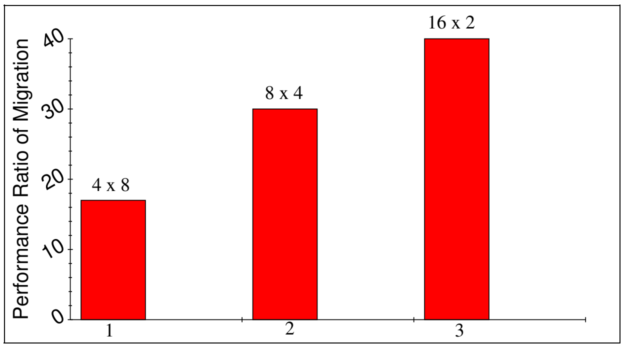

Figure 16: Test of Real-time Approach When Domain Number is Variable But Processor Numbers in Each Domain are Constant

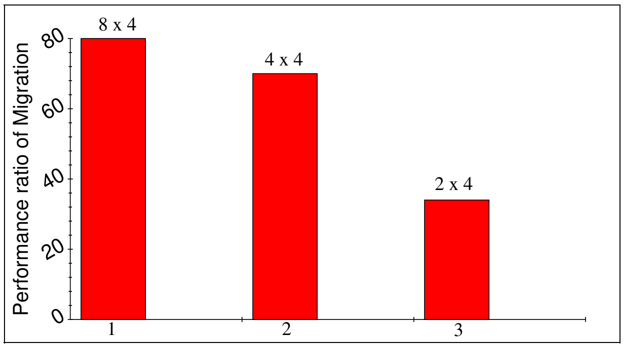

Figure 17: Test of Real-time Approach When Domain Number is Variable But Processor Numbers in Each Domain are Constant

the processor number increases, system goes to a centralized manner and performance of the system decreases.

Fig. 16 shows the system test results when domain number is varying but the total processor number in the system is constant. As the number of domains increases, the system closes to a distributed condition and the system performance increases hence the percentage of the number of aperiodic processes meeting their deadlines rise.

Performance of the system, when the domain number is varying but the processor number in each domain is constant, is shown in Fig. 17. As the domain number in the system increases, the number of aperiodic processes meeting their deadlines increases.

Schedulability ratios for 50 processes on varying number of domains are represented in Fig. 18. Each domain has 4 processors in this

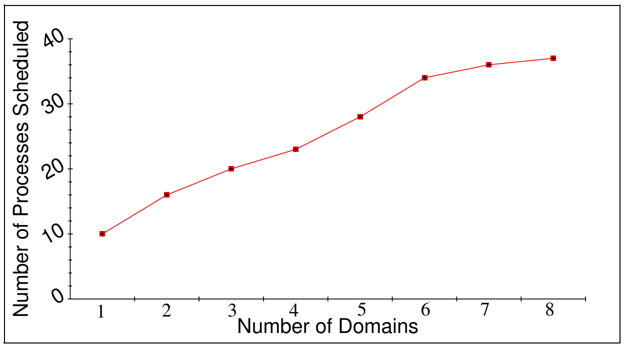

Figure 18: Number of Processes Scheduled Using Real-time Approach For Varying Number of Domains

case. It can be deduced from the figure that as the number of domains in the system increases, schedulability chance of the processes also increases.

6 Conclusions

We have proposed a framework for load balancing and for running applications in a parallel processing system. The main components of the system are the deterministic scheduler, the dynamic load balancing mechanism and operating system support modules for dynamic load balancing. The off-line scheduler is used to allocate periodic, hard real-time processes and the dynamic load balancers transfer aperiodic soft real-time processes from heavily loaded to lightly loaded nodes. The processors of the parallel real-time system are divided into domains which are a collection of processors. Both static and dynamic schedulers try to balance load at intradomain and interdomain levels, but in different directions. The static scheduler starts from global system information and ends in allocation of tasks to processors of individual domains whereas the dynamic load balancers first try to even the load within the domain and if this is not possible, communicate with load balancers of other domains to perform transfers at interdomain level.

The static scheduling model is tested and evaluated for approximately 1500 precedence related tasks. The results can be summarized as follows: It is observed that, when the system is partitioned into domains, time spent during mapping decreases as the number of domains in the system increases. This is caused by the fact that, domain allocation phase minimizes the search space of Task-to-Processor Mapping phase during finding optimal processor for a task. This result is especially important for MPP systems. The percentage of deadlines met for heuristic functions are given in Table 1.

Partitioning the system into processor domains is an advantageous approach according to the simulation results we have obtained. When the system is partitioned into domains, time spent for static scheduling decreases. Since the performance of the model we have proposed is approximately the same for domain partitioning approach and the approach without domains, the domain partitioning scheme that has less time complexity will be preferred. Also, the concept of domain will provide the base for centralized use of resources at intradomain level and distributed usage of resources at interdomain level.

For the dynamic load balancing case, the starting point of our research was the fact that neither fully distributed nor centralized load balancing policies yield good performance for MPP systems. We therefore designed a semi-distributed model which makes use of both approaches. The iPSC/2 hypercube simulator unfortunately created problems after 32 processors, so the maximum domain number we could test is 16 each with 2 processors. It is observed that when the system is tested for varying number of domains and processors, the percentage of the scheduled aperiodic real-time processes rise sharply as the number of domains increase which is in accordance with what we expected.

This semi-distributed management of resources by confining local information to domains and acquiring this information if needed by use of the underlying kernel services can be extended to other higher system functions such as file servers, etc. which can be built on this structure using the same principle. For example, a process in a domain, which wants to open a file, would send a message to the file server of that domain. If the file is not found locally, the other file servers can be informed to find the file and transfer data. We think this approach eliminates the need for central or the replicated data, which would lead to bottleneck in the first case and overhead for consistency in the latter.

Table 1: Percentage of Real-Time Tasks whose deadlines are met.

	M=4	M=8	M=16
EDF	0.96	0.97	0.94
MLF	0.96	0.98	0.98
H3	0.97	0.98	0.96

The results obtained for both static and dynamic load balancing cases are decisive in accepting that partitioning a parallel real-time system into domains and managing them accordingly yields better performance than no-domain case. Secondly, this idea can further be used for other resource management purposes. At a more abstract level, each domain can be considered a node of a distributed real-time system connected by a real-time network. Most of the methodology developed is valid for these loosely coupled systems with the modification of the communication costs accordingly. A future investigation area would be on how to configure the size of the domains to suit an application.

References

[1] Audsley N. & Burns A. (1990) Real-time System Scheduling. Technical Report YCS134. Department of Computer Science, University of York, UK.

[2] Bryant R.M. & Finkel R.A. (1981) A Stable Distributed Scheduling Algorithm.

Proceedings of the 2nd International Conference on Distributed Computing Systems.

[3] Cheriton, D.R. & Mann T.P. (1988) Decentralizing a Global Naming Facility for Improved Performance and Fault Tolerance. *ACM Transactions on Computer Systems*, 7(2), p. 147-183.

[4] Efe K. (1982) Heuristic Models of Task Assignment Scheduling in Distributed Systems. *IEEE Computer*, June 82, p. 50-56.

[5] El-Rewini H. (1989) Task Partitioning and Scheduling on Arbitrary Parallel Processing Systems. PhD. Thesis, Oregon State University.

[6] Goscinski, A. (1991) Distributed Operating Systems: The Logical Design. Addison-Wesley, ISBN 0-201-41704-9, p. 300-344.

[7] Kremien O., Kramer J. & Magee J. (1993) Scalable, Adaptive Load Sharing for Distributed Systems. *IEEE Parallel & Distributed Technology*, vol.1, no.3.

[8] Liu J. & Layland J. (1973), Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment. *Journal of the ACM*, 20(1), p.40-61.

[9] Sadayappan P. & Ercal F. (1987) Nearest Neighbor Mapping of Finite Element Graphs onto Processor Meshes. *IEEE Transactions on Computers*, vol.c-36, no.12, p. 1408-1424.

[10] Stankovic J.A. & Ramamritham K. (1988) The Spring Kernel: A New Paradigm for Real-Time Operating Systems. COINS Technical Report 88-97, University of Massachusetts, Amherst.