Computación y Sistemas Vol. 4 No. 3 pp. 213 - 229
© 2001, CIC - IPN. *ISSN 1405-5546* Impreso en México

An Information Power Grid Resource Management Tool

Luis Miguel Campos[1], Fabricio Silva[1], Isaac Scherson[1] and Mary Eshaghian[2]
[1] Dept. of Information and Computer Science
University of California-Irvine
Irvine, CA 92697, USA.
e-mail: {lcampos, fsilva, isaac}@ics.uci.edu
[2] Departament of Computer Engineering
Rochester Institute of Technology
Rochester, NY 14623-5603
e-mail: mmeeec@grace.rit.edu

Article received on February 15, 2000; accepted on August 23, 2000

Abstract

Heterogeneous Computing (HC) is defined as a special form of parallel and distributed computing. Computations are carried out using a single autonomous computer operating in both SIMD and MIMD modes, or using a number of connected autonomous computers (a.k.a. Cluster Computing). Information Power Grid (IPG) a is form of HC in which high performance computers located at geographically distributed sites are connected via a high-speed interconnection network. It is desirable to make IPG accessible to general users as easily and seamlessly as electricity from the electric power grid. The users should be able to submit jobs at any site, the IPG should be able to handle the computations using the resources available, and return the results to the user. One of the many challenges in making IPG work is the issue of Resource Management. We present a Resource Management Tool for IPG called IPG-SaLaD (Static mapper, and Load balancer, and Dynamic scheduler). This tool uses the Cluster-M mapping paradigm for initial allocation of tasks of a given job onto resources. Later as new jobs come in or as the status of the system changes, the tool uses the Rate of Change algorithm for load balancing, and the Concurrent Gang Scheduling for scheduling. Currently a window-NT based implementation of the IPG-SaLaD using PVM is being constructed.

Keywords: Information Power Grid, Heterogeneous Computing, Resource Management, Parallel Job Scheduling, Static Mapping and allocation, Dynamic Load Balancing

*This research was supported in part by NASA under grant number NAG5-2561, and by the Irvine Research Unit in Advanced Computing

1 Introduction and Background

Since the early 90's, significant amount of attention has been given to the type of computing in which multiple computers are used concurrently in solving single problems. These computers may be of different type and brand with different operating systems and capabilities. Furthermore, they may be geographically distributed. A number of different terms have been used for introducing this concept such as Heterogeneous Computing, Cluster Computing, Meta Computing, Wide-area Computing, and recently Information Power Grid (IPG) computing. In an IPG environment a national computing infrastructure allows users to access the information resources of the nation in much the same way as one accesses electrical power today.

IPG is a topic that has been of interest to NASA and several other organizations such as DOD, DOE and NSF. To this date, a number of sites have taken part in IPG, such as Caltech/JPL, ISI/USC, SDSC, and NCSA. IPG offers many benefits such as allowing collaborative research among geographically dispersed teams in a virtual environment, enabling solutions of problems that could not be done otherwise, and cost and time savings through optimal use of scarce computing resources. The design issues in developing the information technology that enables IPG, can be classified into four groups: Application, User environment, Execution environment, and System integration.

Among the four groups named above, design of suitable execution environments for IPG stands perhaps as the most challenging one. IPG requires execution environments that are portable and scalable. This includes design of efficient resource management techniques, data storage and migration techniques. Tools need to be developed to enable the use of the execution environment such as automated tools for porting legacy code, collaborative problem solving environments, formal, portable programming paradigms, languages and tools that express parallelism and support synthesis and reuse. These execution environments should support applications for the future such as application software that uses 1000 or 10000 processors. Furthermore, the execution environment, user environment and applications all need to be integrated.

The focus of our work is in the area of execution environment design. More specifically we will concentrate on an IPG tool for resource management. The proposed tool called SaLaD (Static allocator, and Load balancer, and Dynamic scheduler), uses a three step mechanism. During the first step, using the Cluster-M static allocator, it will take a program-task (or so called a job) that is entered by the user (or by a compiler) and will break it into subtasks and will map/allocate the subtasks onto available processors and or computers so that overall execution time is minimized. As new jobs arrive and/or finish execution, it maybe required to redistributed individual subtasks using the Rate of Change load balancer. This step guarantees that no processing element ever goes idle. Finally the Concurrent Gang dynamic scheduler is used to schedule the execution sequence of multiple independent jobs residing on the processors. The optimal use of resources, and resource allocation based on the workload contents and site specific capabilities is consistent with the IPG objectives specified by NASA. In this paper, we will present our preliminary results towards the development of this integrated resource management tool. Currently a simple version which is NT-window-based and uses PVM is operational.

The rest of the paper is organized as follows. In the remaining part of this section we briefly present a brief background on the three areas studied in this paper, namely, static allocation, load balancing and dynamic scheduling. In the next section, we present preliminary results on Cluster-M (static allocation), Rate of Change (load balancing), and Concurrent Gang (dynamic scheduling). In Section 3, we will present the operation of the three integrated components of SaLaD, and in section 4, we will describe our current windows-NT implementation of SaLaD using PVM. In Section 5, we present our concluding remarks.

1.1 Static Allocation of subtasks in a single job

The mapping problem, in its general form, has been known to be NP-complete and has been studied intensively for homogeneous parallel computers during the past two decades [Ber87, Bok81, CE95, Efe82, ERL90, ES94, LA87, LRG+90, PSD+92, YG94]. In mapping, an application task and a computing system are usually modeled in terms of a task flow graph and a system graph. The problem, then, is how to map efficiently the task flow graph to the system graph. A task flow graph is a directed acyclic graph (DAG) that consists of a set of vertices and a set of directed edges. A vertex denotes a task module decomposed from the given task. Each vertex is associated with a weight that denotes the computation amount within the corresponding task module. A directed edge joining two task modules denotes that data communication and dependency exist between the two task modules. The weight of an edge represents the amount of data communication. While a task flow graph is usually directed, the system graph is usually an undirected graph. A set of vertices in a system graph denote processors and a set of undirected edges indicate physical communication links for processor pairs. The weight of a vertex (edge) represents the speed (bandwidth) of the corresponding processor (communication link). We define a graph as nonuniform if and only if the weights of all vertices or the weights of all edges are not the same; otherwise it is uniform.

Mapping can be static or dynamic. In static mapping, the assignments of the nodes of the task graphs onto the system graphs are determined prior to the execution and are not changed until the end of the execution. Static mapping can be classified in two general ways. The first classification is based on the topology of task and/or system graphs [CE95]. Based on this, the mappings can be classified into four groups: (1) mapping specialized tasks onto specialized systems, (2) mapping specialized tasks onto arbitrary systems, (3) mapping arbitrary tasks onto specialized systems and (4) mapping arbitrary tasks onto arbitrary systems. The second classification can be based on the uniformity of the weights of the nodes and the edges of the task and/or the system graphs. Based on this, the mappings can be categorized into the following four groups: (1) mapping uniform tasks onto uniform systems [CE95, Bok81, LA87, BS87, ERS90], (2) mapping uniform tasks onto nonuniform systems, (3) mapping nonuniform tasks onto uniform systems [WG90, MG89, Sar89, ERL90, YG94], and (4) mapping nonuniform tasks onto nonuniform systems

[ST85, Lo88]. In IPG, the task and system graphs can be nonuniform. Therefore, the mapping problem in HC can be viewed as mapping of an arbitrary nonuniform task graph onto an arbitrary nonuniform system graph.

In this paper, we first concentrate on static mapping of arbitrary nonuniform task graphs onto arbitrary nonuniform system graphs. The existing mapping techniques in this group include El-Rewini and Lewis' mapping heuristic algorithm [ERL90] for directed task graphs and Lo's max flow min cut mapping heuristic [Lo88] for undirected task graphs[1] The time complexity of these two heuristics are $O(M^2N^3)$ and $O(M^4N\log M)$, respectively, where M is the number of task modules and N is the number of processors. Another algorithm in this category is called Cluster-M which can map arbitrary, nonuniform, architecturally independent, directed task graphs onto arbitrary, nonuniform, undirected, task-independent system graphs in $O(M^2)$ time, where $N \leq M$. Cluster-M will be explained in detail in 3.1

1.2 Load Balancing by Redistribution of Subtasks

Dynamic Load Balancing (DLB) is an important system function aimed at distributing workload among available processors to improve throughput and/or execution times of parallel computer programs either uniform or non-uniform (jobs whose workload varies at run-time in unpredictable ways). Non-uniform computation and communication requirements may bog down a parallel computer if no efficient load distribution is effected.

Load balancing strategies fall broadly into either one of two classifications, namely static or dynamic. A multicomputer system with static load balancing distributes tasks across the processing elements (PEs) before execution using a priori known task information and the load distribution remains unchanged at run time. A multicomputer system with DLB uses no a priori task information, and must satisfy changing requirements by making task distribution decisions during run-time. DLB can be further classified as centralized or distributed. In a centralized strategy, load balancing decisions are made by one PE only, which is responsible for maintaining global load information. In distributed strategies, decisions are made locally, load information is distributed among all PEs and they all share the responsibility of achieving global load balance. The other factor often used to classify load balancing strategies, implicit or explicit, is strictly speaking a system issue rather than a load balancing one. Implicit load balancing refers to load balancing performed automatically by the system, whereas explicit means it is up to the user do decide which tasks should be migrated and when.

Many Dynamic Load Balancing strategies have been proposed in the literature. Three of the best known are: the gradient model [LK91], the sender initiated diffusion [WLR93], and the central job dispatcher [LHWS96]. The gradient model employs a gradient map of the proximities of under-loaded processors in the system to guide the migration of tasks between overloaded and under-loaded processors. The sender initiated diffusion is a highly distributed local approach which makes use of near-neighbor load information to apportion surplus load from heavily loaded processors to under-loaded neighbors in the system. Global balancing is achieved as tasks from heavily loaded neighborhoods diffuse into lightly loaded areas in the system. In the central job dispatcher strategy, one of the network processors acts as a central load balancing master. The dispatcher maintains a table containing the number of waiting tasks in each processor. Whenever a task arrives at or departs from a processor, the processor notifies the central dispatcher of its new load state. When a state change message is received or a task transfer decision is made, the central dispatcher updates the table accordingly. The network bases load balancing on this table and notifies the most heavily loaded processor to transfer tasks to a requesting processor. The network also notifies the requesting processor of the decision.

1.3 Dynamic Scheduling of Multiple Jobs or Tasks

Parallel job scheduling is an important problem whose solution may lead to better utilization of modern parallel computers. It is defined as: "Given the aggregate of all tasks of multiple jobs in a parallel system, find a spatial and temporal allocation to execute all tasks efficiently". Each job in a parallel machine is composed by one or more tasks. For the purposes of scheduling, we view a computer as a queuing system. An arriving job may wait for some time, receive the required service, and depart [FR98]. The time associated with the waiting and service phases is a function of the scheduling algorithm and the workload. Some scheduling algorithms may require that a job wait in a queue until all of its required resources become available (as in variable partition-

[1]Mapping of directed task graphs (if there is precedence relation among the task nodes) is called task scheduling [ERL90]. If the task graphs to be mapped are undirected, then it is called task allocation [ERL90].

ing), while in others, like time slicing, the arriving job receives service immediately through a processor sharing discipline.

Gang scheduling has been widely used as a practical solution to the dynamic parallel job scheduling problem. Parallel tasks of a job are scheduled for simultaneous execution on a partition of a parallel computer. Gang Scheduling has many advantages, such as responsiveness, efficient sharing of resources and ease of programming. However, there are two major problems associated with gang scheduling: scalability and the decision of what to do when a task blocks.

2 Preliminaries

The IPG tool which is presented in this paper, called SaLaD consists of three main components. In the next section we will describe how they work together in an integrated fashion. In this section, we present preliminary background on each of three components namely the Static Allocator, Load Balancer, and the Dynamic Scheduler. The static allocation technique used is based on the Cluster-M technique, Load Balancing is achieved using the Rate of Change algorithm, and Dynamic Scheduling is performed using a generalization of gang-scheduling dubbed Concurrent Gang.

2.1 Cluster-M Static Allocation

Cluster-M is a programming tool that facilitates the static allocation and mapping of portable parallel programs [CE95]. Cluster-M has three main components: the specification module, the representation module and the mapping module. In the specification module, machine-independent algorithms are specified and coded using the Program Composition Notation (PCN [FT93]) programming language [ES94]. Cluster-M specifications are represented in the form of a multi-layer clustered task graph called Spec graph. Each clustering layer in the Spec graph represents a set of concurrent computations, called Spec clusters. A Spec graph can also be obtained by applying one of the appropriate Cluster-M clustering algorithms to any given task graph. A Cluster-M representation represents a multi-layer partitioning of a system graph called a Rep graph. Given a system graph, a Rep graph can be generated using one of the Cluster-M clustering algorithms. At every partitioning layer of the Rep graph, there are a number of clusters called Rep clusters. Each Rep cluster represents a set of processors with a certain degree of connectivity. The clustering is done only once for a given task (system) graph independent of any system (task) graphs. It is a machine-independent (application-independent) clustering, therefore it is not necessary to be repeated for different mappings. For this reason, the time complexities of the clustering algorithms are not included in the time complexity of the Cluster-M mapping algorithm. In the mapping module, a given Spec graph is mapped onto a given Rep graph. In an earlier publications [CE95, Esh96] two Cluster-M clustering algorithms and a mapping algorithm were presented for both uniform an non-uniform graphs.

Below, we present a set of experimental results. The following criteria are used for comparing the performance of our algorithm with other leading techniques: (1) the total time for executing the mapping algorithm, T_c; (2) the total execution time of the generated mappings, T_m; (3) the number of processors used, N_m; and (4) the total time executing both clustering (task and system) and mapping algorithms, T_{cm}. From (2) and (3), we can obtain the speedup $S_m = \frac{T_s}{T_m}$ and efficiency $\eta = \frac{S_m}{N_m}$, where T_s is the sequential execution time of the task.

In Table 1, comparison results are shown for mapping nonuniform random task graphs ranging from 100 to 1000 nodes onto the random system graph of size 100, where the speed of the processors and communication channels is ranged between 1-5 only. In other words even though there are 100 processors but they are very slow, and therefore the speed up is expected to be low. What is really important here is that both MH and Cluster-M lead to the same type of results, but Cluster-M is very much faster than MH both asymptotically and experimentally. The running time of MH grows significantly as the size of the task graph grows. Whereas the running time of Cluster-M remains relatively stable. In most cases, Cluster-M obtains a better speedup than MH. But in all cases Cluster-M has a significantly lower time complexity.

The experimental results shown in this section were obtained by running a set of simulations on a SUN UltraSPARC I workstation, and all running times times (T_c, T_{cm}) are measured in milliseconds The non-uniform task graphs are randomly generated.

2.2 Rate of Change Load Balancing

Previously proposed load balancing strategies cast the problem as one of equalizing the absolute number of load units among all PEs. Rate of Change load balancing algorithm [CS99, Cam99] constitutes a departure from this classical approach. We define load balancing as the activity of migrating load units from one PE to another so that all PEs have at least

Size of Random Task Graph	T_s	Cluster-M $[O(M^2)]$ w/clustering $[O(M^2) + O(N^3 \log N)]$				MH $[O(M^2 N^3)]$		
		T_m	S_m	T_{cm}	T_c	T_m	S_m	T_c
100	286	88.80	3.22	831.4	4.5	95.80	2.99	6064.9
200	630	133.20	4.73	1056.6	4.5	231.82	2.72	24380.1
300	855	345.55	2.47	1440.6	4.7	240.25	3.56	55305.5
400	1162	478.40	2.43	1915.9	4.5	496.30	2.34	98888.8
500	1514	550.80	2.75	2547.9	4.5	458.07	3.31	153099.0
600	1793	358.20	5.01	3436.1	4.9	599.07	2.99	222381.1
700	2075	690.85	3.00	4334.0	4.7	685.57	3.03	299372.2
800	2376	474.00	5.01	5621.3	4.8	958.87	2.48	391142.3
900	2653	1113.80	2.38	6656.3	4.7	1120.17	2.37	495050.7
1000	2966	850.15	3.49	8074.3	4.7	1087.08	2.73	613118.2

Table 1: Comparison of Cluster-M and MH on a random system with 100 nodes.

one load unit at all times. The rationale is that if any given PE is busy executing tasks then the load differential with respect other PEs is irrelevant since no performance gain can be obtained by transferring load. Hence the decision to initiate load transfers should not depend on a PE's absolute number of load units, but on how the load changes in time.

Our novel approach uses the Rate of Change (RoC) of the load on each PE to trigger any load balancing activity. It can be described as a dynamic, distributed, on demand, preemptive and implicit load balancing strategy. Dynamic, because it does not assume any prior task (unit of load in this study) information and must satisfy changing requirements by making task distribution decisions at run-time. Distributed, because all load balancing decisions are made locally and asynchronously by each processor. On demand, because only PEs that "need" tasks are allowed to initiate any migration activity. Preemptive, because running tasks may be suspended, moved to another PE and restarted. Implicit, because all load balancing activity is done by the system, without user assistance.

Furthermore, our proposed RoC-Load Balancing strategy achieves the goal of minimizing processor idling times without incurring into unacceptably high load balancing overhead. It does so by striking a balance between the cost of evaluating load information which now is a local activity to each PE, and the cost of transferring tasks across the system.

Experimental results have been obtained and are briefly described in here. For a complete description of the experiments, algorithm and supporting data structures please refer to [CS99, Cam99]. The first experiment (table 2) simulates a stable situation where an initial set of tasks (representing several competing applications) was distributed uniformly among all 16 processors. In the second scenario (table 3), the same initial set of tasks was divided among only half of the PEs with the remaining being idle. In both experiments, new applications were not allowed to be submitted to the system. All new tasks were generated internally due to the non-uniform nature of the applications being simulated. For the third experiment (table 4), the possibility of arbitrary arrival times for new jobs was allowed, and the tasks that compose an application were distributed randomly among all PEs.

The performance metric used in the experiments is the normalized performance (NP) as used in [WLR93, LHWS96].

Normalized performance (NP) determines the effectiveness of the load balancing strategy (such that $NP \to 0$ if the strategy is ineffective and $NP \to 1$ if the strategy is effective). This is a comprehensive metric; it accounts for the initial level of load imbalance as well as the load balancing overheads. NP is formally defined as:

$$NP = \frac{(T_{noLB} - T_{bal})}{(T_{noLB} - T_{opt})}$$

where T_{noLB} is the time to complete the work on a multicomputer network without load balancing, T_{opt} is the time to complete the work on one processor divided by the number of processors in the network and T_{bal} is the time to complete the work on a multicomputer network with load balancing.

2.3 Concurrent Gang Dynamic Scheduling

Gang scheduling has been widely used as a solution to the dynamic parallel job scheduling problem. Gang service is a paradigm where all tasks of a job in the service stage are grouped into a gang and concurrently scheduled in distinct processors. Reasons

	Mesh	Hypercube	Fully Connected	Network of Workstations
Gradient Model	0.66	0.69	0.73	0.52
Sender Initiated Diffusion	0.7	0.71	0.74	0.59
Central Job Dispatcher	0.59	0.62	0.63	0.37
Rate of Change Model·	0.79	0.79	0.87	0.61

Table 2: Stable situation.

	Mesh	Hypercube	Fully Connected	Network of Workstations
Gradient Model	0.72	0.74	0.76	0.54
Sender Initiated Diffusion	0.65	0.65	0.68	0.52
Central Job Dispatcher	0.65	0.66	0.65	0.45
Rate of Change Model	0.74	0.75	0.81	0.60

Table 3: Unstable situation

to consider gang service are responsiveness [FA97], efficient sharing of resources[Jet97] and ease of programming. In gang service the tasks of a job are supplied with an environment that is very similar to a dedicated machine [Jet97]. It is useful to any model of computation and any programming style. The use of time slicing allows performance to degrade gradually as load increases. Applications with fine-grain interactions benefit of large performance improvements over uncoordinated scheduling[FR92]. One main problem related with gang scheduling is the necessity of multi-context switch across the nodes of the processor, which causes difficulty in scaling[ea98]. In our work, we use a class of scheduling policies, dubbed concurrent gang, that is a generalization of gang-scheduling and allows for the flexible simultaneous scheduling of multiple parallel jobs in a scalable manner.

Concurrent Gang is an strategy that increases utilization and throughput in parallel machines when compared with other implementations of gang service, for the same resource sharing strategy, as simulation studies indicate[SS99b, SS99a]. The utilization in Concurrent Gang is improved because, in the event of an idle slot or a blocked thread, Concurrent Gang always tries to schedule other tasks that are either local tasks or tasks that do not require, at that moment, coordinated scheduling with other tasks of the same job. This is the case, for instance, of I/O intensive tasks and Computation intensive tasks. Improved processor utilization in turn leads to better throughput.

Preliminary results have been already obtained and published. In Tables 5, 6, 7 we present a summary of those results. For a complete description of the experiments please refer to [SS99b].

3 IPG-SaLaD Components

IPG-SaLaD is composed of three inter-related components: Static Allocator, Load Balancer and Dynamic Scheduler. Their interaction can be thought of as a three stage process with a feedback loop as shown in figure 1.

During the first stage, arriving jobs one at a time, are passed to the static allocator which is responsible for performing a suboptimal matching of the individual tasks that compose a job to the resources available. The task assignment from the first stage is fed to the load balancer whose aim is to minimize idle time among the processing elements that constitute the system. Note, from the details presented in the last section, that the Rate of Change load balancing is only triggered when individual PEs predict that their future load will lead them to an idle state. As new jobs come in and are fed through the static allocator and possibly altered by the load balancer the dynamic scheduler is invoked to handle the scheduling of the multiple jobs present in the system. This may require redistribution of tasks. The load balancer also receives input from the dynamic scheduler in the form of the current task allocation for each resource. Given both the inputs from the dynamic scheduler and the static allocator, the load balancer decides based on the Rate of Change algorithm whether or not to further redistribute the load. This interaction between load balancer and dynamic scheduler continues through the execution of the entire workload.

It is important to note that at any given instance all stages can be active simultaneously which allows for a high degree of parallelism.

A detailed description of the internal functioning of each individual component follows.

	Mesh	Hypercube	Fully Connected	Network of Workstations
Gradient Model	0.7	0.77	0.81	0.54
Sender Initiated Diffusion	0.73	0.76	0.81	0.58
Central Job Dispatcher	0.52	0.53	0.53	0.42
Rate of Change Model	0.82	0.82	0.88	0.62

Table 4: Arbitrary arrival times

Simulation time	Gang		Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)	Jobs Completed	Total Idle Time (%)
5000	2	72	5	43
10000	6	68	10	26
20000	14	67	19	14
30000	23	66	34	10
40000	28	66	45	7

Table 5: Experimental results - I/O intensive workload

3.1 SaLaD Static Allocation

There are a number of reasons and benefits in clustering task and system graphs in the Cluster-M fashion. Basically Cluster-M clustering causes both task and system graphs to be partitioned so that the complexity of the mapping problem is simplified and efficient mapping results can be obtained. In clustering an undirected graph, completely connected nodes are grouped together forming a set of clusters [CE95, ES94, Esh96]. Clusters are then grouped together again if they are completely connected. This is continued until no more clustering is possible (see figure 2). When an undirected graph is a task graph, then doing this clustering essentially identifies and groups communication-intensive sets of task nodes into a number of clusters called Spec clusters (see figure 3). Similarly for a system graph, doing the clustering identifies well-connected sets of processors into a number of clusters called Rep clusters. In the mapping process, each of the communication intensive sets of task nodes (Spec clusters) is to be mapped onto a communication-efficient subsystem (Rep cluster) of suitable size (see figure 4). Clustering directed graphs (i.e., directed task graphs) produces two types of graph partitioning: horizontal and vertical. Horizontal partitioning is obtained because, as part of clustering, we divide a directed graph into a layered graph such that each layer consists of a number of computation nodes that can be executed in parallel and a number of communication edges incoming to these nodes. The layers are to be executed one at a time. Therefore, the mapping is done one layer at a time. This significantly reduces the complexity of the mapping problem since the entire task graph need not to be matched against the entire system graph.

Vertical graph partitioning is obtained because as part of the clustering the nodes from consecutive layers are merged or embedded. All the nodes in a layer are merged to form a cluster if they have a common parent node in the layer above or a common child node in the layer below. Doing this traces the flow of data. This information will be used later as part of the mapping so that the tasks are placed onto the processors in a way that total communication overhead is minimized. For example, to avoid unnecessary communication overhead, the task nodes along a path may be embedded into one another so that they are assigned to the same processor.

Both horizontal and vertical graph partitioning are accomplished by performing the clustering in a bottom-up fashion. The Cluster-M mapping will then be performed in a top-down fashion by mapping the Spec clusters one layer at a time onto the Rep clusters.

3.2 SaLaD Load Balancing

The task-resource matching provided by the Static Mapper is considered only as a suggested initial assignment by the Load Balancer. This assignment is suboptimal under the assumption that the job just mapped would be the only job scheduled to run in the system. Moreover it does not consider the spawning of new tasks during the execution of the parallel job. In this more realistic dynamic environment (multiprogrammed system with unpredictable job arrivals and internally generated parallelism) it is the responsibility of the Load Balancer to transfer load (i.e. tasks) among the pool of resources (i.e. PEs) composing the system with the goal of minimizing idle time. The rational behind the load balancing algorithm, dubbed Rate of Change-Load Balancing,

Simulation time	Gang		Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)	Jobs Completed	Total Idle Time (%)
5000	5	24	8	10
10000	11	17	12	6
20000	23	12	25	4
30000	35	11	37	3
40000	44	9	48	2

Table 6: Experimental results - Computation intensive workload

Simulation time	Gang		Concurrent Gang	
Seconds	Jobs Completed	Total Idle Time (%)	Jobs Completed	Total Idle Time (%)
5000	4	46	3	10
10000	14	43	16	6
20000	24	40	31	4
30000	41	39	54	3
40000	47	39	75	2

Table 7: Experimental results - Synchronization intensive workload

is as follows: if any given PE is busy executing tasks then the load differential with respect to any other PE is irrelevant since no performance gain can be obtained by load transfer. Moreover the decision factor to initiate a load transfer request should not be a PE's absolute load but instead how much its load has changed since the previous time interval. Our goal is then to minimize processor idling time without incurring high load balancing overhead. To do so, an optimal tradeoff between the processing (cost of evaluating load information to determine task migration) and communication (cost of acquiring load information and informing other PEs of load migration decisions) overhead and the degree of knowledge used in the balancing process must be sought.

The load balancing problem can be thought of as a four phase decision process, namely:

When to initiate task migration

- Where, which PE, to send the task migration request
- How many tasks to migrate
- Which tasks to migrate

In RoC-LB these four phases occur asynchronously at each PE and are purely distributed.

3.2.1 The When Phase

At the beginning of each sample interval, each PE calculates the change in its load since the previous interval. Let us call this quantity the difference in load *(DL)*. Two key observations should be made at this point. First, each PE may calculate this quantity independently from other PEs in the system, i.e. there is no concept of global synchronous clock in the system. Second, the length of the sample interval may vary with time for a given PE, depending, for instance, on the number of load requests received or network traffic. As a consequence different PEs may use different sampling intervals at any given time. The sampling interval is therefore an adaptive parameter. The sampling interval is usually measured as a multiple of the time slice duration. The finer the sampling the faster we detect the need for load balancing, but the greater overhead we incur.

Each PE uses its own *DL* as a predictor for how many tasks will finish in subsequent intervals. Each PE assumes that *DL* will remain the same forever. Given that it calculates the number of sampling intervals it will take to reach an idle state (no tasks to process). If the number of intervals (times its duration) is less than the network delay *(ND)*, then the PE will initiate a migration request.

From the above, one can conclude that only when *DL* is negative (reduction in load) will a PE consider requesting load from other PEs. Network delay is an adaptive parameter. It is defined as the time it takes between the initiation of a load request and the reception of load. It may vary with time and each PE may use a different value depending on its own record of past load requests. Initially, before any request has been made, *ND* is set to an arbitrary (positive) value.

There are two exceptions to the general rules described above. The first exception is the situation in which a PE will initiate a request for load even though it does not predict it will reach the idle state.

```
Clustering Nonuniform Undirected Graphs (CNUG) Algorithm
For all nodes $p_i$ do
begin Make a cluster for $p_i$ at clustering layer 1
      Set the parameters of the cluster to be $(1, B_i, 0, 0)$
end
Set cluster layer $L$ to be 1
While there is at least one edge linking two clusters of layer $L$ do
begin Sort all edges linking any two clusters in descending order
      While sorted edge list is not empty, do
      begin Take the first edge $(c_i, c_j)$ from sorted edge list
            Delete the edge from the list
            Merge $c_i$ and $c_j$ into cluster $c'$ at layer $(L+1)$
            Calculate the parameters of $c'$
            Delete clusters $c_i$ and $c_j$ from current layer $L$
            For each edge $(c_x, c_y)$ in sorted edge list
            begin
                  If ($c_x$ is a sub-cluster of $c'$) and
                  ($c_y$ is not a sub-cluster of any cluster) and
                  ($c_y$ is connected to all other sub-clusters of $c'$) then
                  begin Merge $c_y$ into $c'$
                        Recalculate the parameters of $c'$
                        Delete $(c_x, c_y)$ from edge list
                  end
                  Else if $c_x$ and $c_y$ are sub-clusters of two different clusters at layer $(L+1)$, then
                  begin Add the weight of $(c_x, c_y)$ to the edge between the two super-clusters
                        Delete $(c_x, c_y)$ from edge list
                  end
            end
      end
      Increment clustering layer $L$ by 1
end
```

Figure 2: Clustering Nonuniform Undirected Graphs (CNUG) algorithm.

In order to understand why it chooses to do so, let us introduce the three thresholds used by the algorithm. High threshold *(HT)* and Low threshold *(LT)* are used to determine the load status of the processor. If a PE's load is greater or equal to HT it is considered a Source PE. If on the other hand it is less or equal to LT it is considered a Sink PE. If its load lies between these two thresholds then the PE is in a neutral state. If however a PE's load falls below a Critical threshold *(CT)* the PE immediately initiates a request for load regardless of the predicted future load based upon the current DL value. We decide to request load even though we do not expect to reach the idle state since even a small change in load at this level will result in immediate task starvation by the PE. The only exception to this rule is that if a PE has already a request pending in the network it will not issue another until either load is received from other PE(s) or the request comes back as unfulfilled. This last observation applies in every case even when a *DL's* value would deem necessary to issue another request. The other exception to the general rule is the situation when a PE's load level is above HT. In that case even if the value of DL predicts that the PE will reach an idle state we do not initiate a load request because at this load value the value of DL necessary to force the PE to become idle must be quite high. There is a good chance that such a value is rare and short lived in which case during the next sample interval the newly calculated DL will be such that it does not warrant an initiation of a transfer request. By delaying any action until the load value falls below HT we are immune to any spikes in load that may occur over time.

3.2.2 The Where Phase

Each PE keeps two local tables containing system load information. One contains information regarding the location of sink PEs, called Sink table the other of source PEs, called Source table. Any PE that initiates a request for load is considered to be a sink by the receiving PE(s). This is true regardless of the level of load at the time the request for load transfer was issued. Selecting the PE to which send a load request is a simple operation. The sink PE (request initiator) selects a source PE from its source table (the first entry in the table) and sends a message requesting load to it. Every time the static allocator produces a task-to-resource mapping for a

```
Clustering Nonuniform Directed Graphs (CNDG) Algorithm
Divide the directed graph into horizontal layers in top-down fashion
For each layer of the task graph do
begin
    If $L = 1$ then
      Make each node at layer 1 into a cluster and calculate its parameters
    else
    begin
        For all edges $(t_i, t_j)$ where at least either $t_i$ or $t_j$ is not embedded or merged yet, do
        begin If $t_i$ is a fork-node then
            begin Embed the child node with the largest edge weight to $t_i$
                If the child nodes of $t_i$ are not in some common cluster then
                begin Merge them with $t_i$ into a cluster
                    Calculate the parameters of the new cluster
                end
            end
            If $t_j$ is a join-node then
            begin Embed the parent node with the largest edge weight to $t_i$
                If the parent nodes of $t_j$ are not in some common cluster then
                begin Merge them with $t_j$ into a cluster
                    Calculate the parameters of the new cluster
                end
            end
        end
    end
end
```

Figure 3: Clustering Nonuniform Directed Graphs (CNDG) algorithm

given job, the entries in the source/sink tables are updated with this new information. The selection process described above is used by all PEs alike, whether they are the request initiator or the recipient of a load transfer message who might need to select a source PE to whom forward the message to.

3.2.3 The How Many Phase

At the beginning of each sample interval each PE calculates its *DL*. Using *DL* it computes how many tasks it would have, assuming *DL* would stay constant, after a length of time equal to *ND*. Let us call this quantity Predicted Load (*PL*). If *PL* is greater or equal to zero then the PE does not expect to become idle within the next *ND* period and therefore does not initiate a request for load. If on the other hand *PL* is lesser than zero the PE requests load, according to the mechanism described in 3.2.2, in the amount of **abs**(*PL*). On the receiver side, a PE will only transfer tasks if its load is above the *HT* level, in which case it transfers tasks above this value up to the requested transfer amount.

3.2.4 The Which Phase

Finally, to answer the question of which tasks to migrate, we have decided to migrate older tasks because these tasks have a higher probability of living long enough to amortize their migration cost [HBD90] Age in this case refers to the CPU time a task has used thus far and not how long ago the task was created measured in wall time clock.

Figure 5 shows the internal data structures involved during the load transfer update process.

3.3 SaLaD Dynamic Scheduling

In this section we describe the Concurrent Gang algorithm. We first introduce the concepts of cycle, slice, period and slot, which are fundamental to understand the internal workings of our algorithm. Then we describe the task classification that is made by the algorithm; we shall see that this task classification is used by Concurrent Gang to decide locally which task to schedule if the current task blocks. In 3.3.3 the algorithm itself is detailed, with the description of the components of a Concurrent Gang Scheduler.

3.3.1 Time Utilization

In parallel job scheduling, as the number of processors is larger than one, the time utilization as well as the spatial utilization can be better visualized with the help of a bi-dimensional diagram dubbed *trace diagram*. One dimension represents processors while the other dimension represents time. Through the trace diagram it is possible to visualize the time utilization of the set of processors given a scheduling

```
Mapping Algorithm
For each layer of Spec graph (starting from maximal layer number) do
begin
    Sort all Spec clusters at current layer in descending order of $\sigma S_i^u$, $\delta S_i^u$, $\Pi S_i^u$, and $\pi S_i^u$.
    Sort all Rep clusters at current layer in descending order of $\sigma R_j^v$ $\delta R_j^v$, $\Pi R_j^v$, and $\pi R_j^v$.
    Calculate $f_{(u,v)}$, if $f_{(u,v)} > 1$, let $f_{(u,v)} = 1$.
    Calculate the required size of the Rep cluster matching $S_i^u$ to be $f_{(u,v)} \times \sigma S_i^u$
    For each Spec cluster at current layer sorted list, do
    begin If the cluster has only one sub-cluster, then
        Go to a lower layer where there are multiple or no sub-clusters
        If at least a Rep cluster of required size is found, then
        begin Select the Rep cluster of required size with minimum estimated execution time
            Match the Spec cluster to the Rep cluster
            Delete the Spec and Rep cluster from Spec and Rep list
        end
    end
    For each unmatched Spec cluster, do
    begin If the size of the first Rep cluster > the required size, then
        begin Split the Rep cluster into two parts with one part of the required size
            Match the Spec cluster to this part
            Insert the other part to proper position of the sorted Rep cluster list
        end
        Else begin
            Merge Rep clusters with largest size until the sum of sizes $\geq$ the required size
            If =, then
            Match the Spec cluster to the merged Rep cluster
            Else
            begin Split the merged Rep cluster into two parts with one of required size
                match the Spec cluster to this part
                Insert the other part to the sorted Rep list
            end
        end
    end
    For each matching pair of Spec cluster and Rep cluster, do
    begin If the Rep cluster contains only one processor, then
        Map all the modules in the Spec cluster to the processor
        Else if Inequality is satisfied, then
            begin Select the sub-cluster of the Spec cluster with the largest size
                Embed the nodes of other sub-clusters to the connected nodes of the selected sub-cluster
                to the connected nodes of the selected sub-cluster
                embed these sub-clusters onto the selected one
                Calculate the parameters for the new cluster
                Insert it into the sorted Spec cluster list
            end
            Else
            begin Delete the Spec cluster from Spec cluster list
                Delete the Rep cluster from Rep cluster list
                Go to the sub-clusters of the Spec and Rep cluster (thus they are pushed to current layer)
                Call the same mapping algorithm for these clusters
            end
    end
end
```

Figure 4: Mapping algorithm.

algorithm. One such diagram is illustrated in figure 6. A *Workload change* occurs at the arrival of a new job, the completion of an existing one, or through the variation of the number of eligible tasks of a job to be scheduled. The time between workload changes is defined as a *cycle*. Between workload changes, we may define a period that depends on the workload and the spatial allocation. The period is the minimum interval of time where all jobs are scheduled at least once. A cycle/period is composed of *slices*; a slice corresponds to a time slice in a partition that includes all processors of the machine. Observe that the duration of the slice for Concurrent Gang is defined by the period of the global clock. A *slot* is the processors' view of a slice. A Slice is composed of N slots, for a machine with N processors. If a processor has no assigned task during its slot in a slice, then we have an idle slot. The number of idle slots in a period divided by the total number of slots in that period defines the *Idling Ratio*. Note that workload changes are detected between periods. If, for instance, a job arrives in the middle of a period, corresponding action of allocating the job is only taken by the end of the period.

3.3.2 Task Classification

In Concurrent Gang, each PE classifies each one of its allocated tasks into classes. Examples of such classes are: I/O intensive, Synchronization intensive, and computation intensive. Each one of these classes is similar to a fuzzy set [Zad65]. A fuzzy set associated with a class A is characterized by a membership function $f_A(x)$ with associates each task T to a real number in the interval [0,1], with the value of $f_A(T)$ representing the "grade of membership" of T in A. Thus, the nearer the value of $F_A(T)$ to unity, the higher the grade of membership of T in A. For instance, consider the class of I/O intensive tasks,

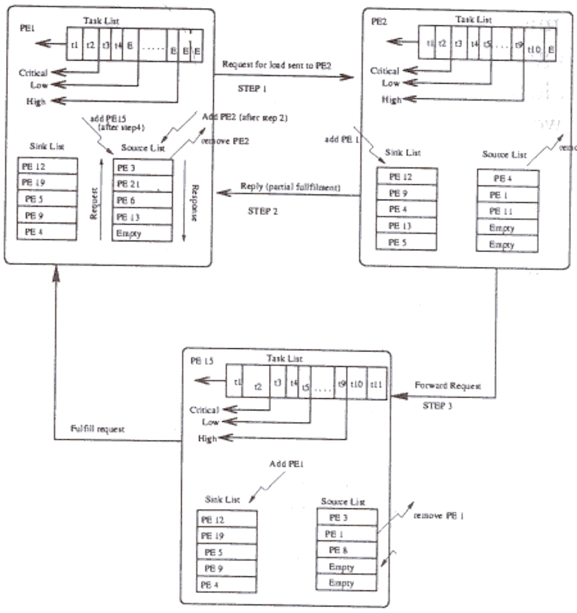

Figure 5: Load Transfer Update Process

with its respective characteristic function $f_{IO}(T)$. A value of $f_{IO}(T) = 1$ indicates that the task T only have I/O statements, while a value of $f_{IO}(x) = 0$ indicates that the task T have executed no I/O statement at all.

In principle, four major classes are possible: I/O intensive, Computing intensive, Communications (point to point) intensive and synchronization intensive. We will see in the next subsection that only a subset of them are used in Concurrent Gang.

3.3.3 Description of Concurrent Gang

A Concurrent Gang scheduler is composed by a set of local schedulers, one for each PE, and a mechanism

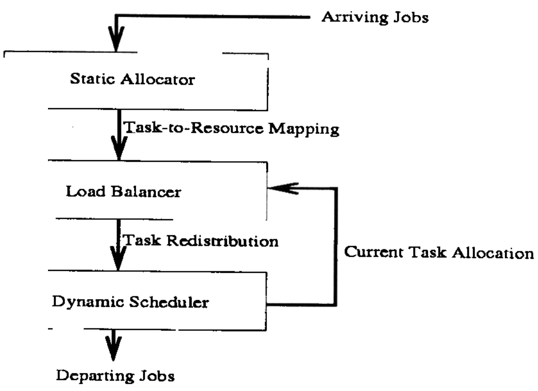

Figure Interaction between the different SaLaD components

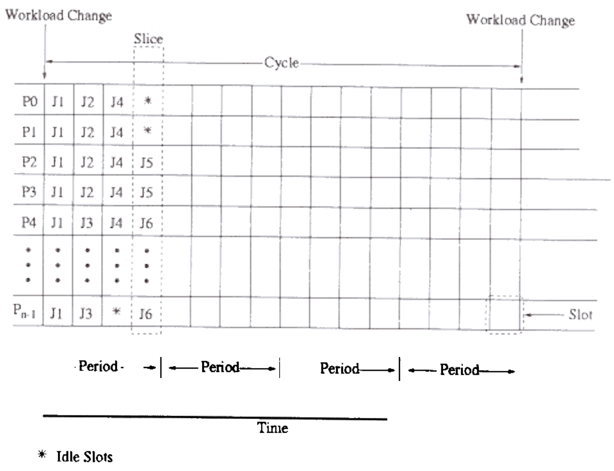

Figure 6: Definition of slice, slot, period and cycl

for the coordination of the global context switch in the machine, which can be either a central controller or a global synchronizer.

A local scheduler in Concurrent Gang is composed of two main parts: the Gang scheduler and the local task scheduler. The Gang Scheduler defines the next task to be scheduled by the arrival of the global context switch signal coming from a synchronizer or a central controller. The local task scheduler is responsible for scheduling sequential tasks and parallel tasks that do not need global coordination, as described in the next paragraph, and it is similar to a UNIX scheduler. The Gang Scheduler has precedence over the local task scheduler.

We may consider two types of parallel tasks in a concurrent gang scheduler: Those that should be scheduled as a gang with other tasks in other processors and those that gang scheduling is not mandatory. Examples of the first class are tasks that compose a job with fine grain synchronization interactions [FR92] and communication intensive jobs[ea98]. Second class task examples are local tasks or tasks that compose an I/O bound parallel job, for instance. On other side a traditional UNIX scheduler does a good job in scheduling I/O bound tasks since it gives high priority to I/O blocked tasks when the data became available from disk. As those tasks typically run for a small amount of time and then blocks again, giving them high priority means running the task that will take the least amount of time before blocking, which is coherent to the theory of uniprocessors scheduling where the best scheduling strategy possible under total completion time is Shortest Job First [MPT94]. In the local task scheduler of Concurrent Gang, such high priority is preserved. Another example of jobs where gang scheduling is not mandatory are embarrassingly parallel jobs. As the number of iterations among tasks belonging to this class of jobs are small, the basic requirement for scheduling a embarrassingly parallel job is to give those jobs the larger fraction of CPU time possible, even in an uncoordinated manner.

Differentiation among tasks that should be gang scheduled and those that a more flexible scheduler is better is made using the grade of membership information computed by the local scheduler (as explained in the last subsection) about each task associated with the respective processor. The grade of membership of the task previously scheduled is then computed at the next machine-wide context switch, and is that information that is used to decide if gang scheduling is mandatory or not for a specific task.

The local task scheduler defines a priority for each task allocated to the corresponding PE. The priority of each task is defined based on the grade of membership of a task to each one of the major classes described in the previous subsection. Formally, the priority of a task T in a PE is defined as:

$$Pr(T) = max(\alpha \times \lambda_{IO}, \lambda_{COMP}) \qquad (1)$$

Where $\lambda_{IO}, \lambda_{COMP}$ are the grade for membership of task T to the classes I/O intensive and Computation intensive. The objective of the parameter α is to give higher priority to I/O bound jobs ($\alpha > 1$). The choices made in equation 1 intend to give high priority to I/O intensive jobs and computation intensive job, since such jobs can benefit the most from uncoordinated scheduling. The multiplication factor α for the class I/O intensive gives higher priority to I/O bound tasks over computation intensive tasks, since those jobs have a higher probably to block when scheduled than computing bound tasks. By other side, synchronization intensive and communication intensive jobs have low priority since they require coordinated scheduling to achieve efficient execution and machine utilization[FR92, ea98]. A synchronization intensive or communication intensive phase will reflect negatively over the grade of membership of the class computation intensive, reducing the possibility of a task be scheduled by the local task scheduler. Among a set of tasks of the same priority, the local task scheduler uses a round robin strategy. The local task scheduler also defines a minimum priority β. If no parallel task has priority larger than β, the local task scheduler considers that all tasks are either communication or synchronization intensive, thus requiring coordinated scheduling.

In practice the operation of the Concurrent Gang scheduler at each processor will proceed as follows: The reception of the global clock signal will generate an interruption that will make each processing element schedule tasks as defined in the trace diagram. If a task blocks, control will be passed to another task as a function of the priority assigned to each one of the tasks until the arrival of the next clock signal. The task chosen is the one with higher priority.

In the event of a job arrival, a job termination or a job changing its number of eligible tasks the front end Concurrent Gang Scheduler will :

```
  Update Eligible task list
2 Allocate Tasks of First Job in General Queue
3 While not end of Job Queue
            Allocate all tasks of remaining parallel jobs
            using a defined spatial sharing strategy
4 Run
```

Between Workload Changes
- If a task blocks or in the case of an idle slot, the local task scheduler is activated, and it will decide to schedule a new task based on:

- Availability of the task (task ready
- Priority of the task defined by the local task scheduler.

All processors change context at same time due to a global clock signal coming from a central synchronizer. The local queue positions represents slots in the trace diagram. The local queue length is the same for all processors and is equal to the number of slices in a period of the schedule. It is worth noting that in the case of a workload change, only the PEs concerned by the modification in the trace diagram are notified.

It is clear that once the first job, if any, in the general queue is allocated, the remaining available resources can be allocated to other eligible tasks by using a predefined partitioning strategy.

In the case of creation of a new task by a parallel task, or parallel task completion, it is up to the local scheduler to inform the front end of the workload change. The front end will then take the appropriate actions depending on the pre-defined space sharing strategy.

4 Current Implementation of SaLaD

Software solutions for IPG are still in their infancy. As examples of some experimental meta-computing systems we have the NOW project [2] and Legion[3], which enable users to use resources across multiple machines. A number of attempts are being made to address the problem of allowing programmers to write portable code that will run across networks of computers. One such tool is HENCE[4] (Heterogeneous Network Computing Environment), an X-window based software environment designed to assist in developing parallel programs that run on a network of computers. HENCE allows a user to lay out tasks in a graphical format and execute them. HENCE uses a static table of costs that describes the cost of executing various program modules on various machines. Selection is then done using this table so that the overall cost is minimized.

In this section, a Windows NT based implementation of the SaLaD tool for distributed execution of arbitrary heterogeneous tasks onto arbitrary suite of heterogeneous systems using PVM is presented. The main components of this tool are the Cluster-M Clustering and Mapping Module, the Rate of Change Module, the Concurrent Gang Scheduling Module,the Graphical User Interface Module and the PVM based Distribution Module. The user inputs the task and system graphs using the tool's Graphical User Interface module, where the Nodes of a task graph represent an arbitrary executable program written in C, C++ or Fortran, and the Nodes of the System Graph represent any system with a PVM implementation. Once the mapping is done, the Distribution Module using PVM dispatches the tasks on the underlying heterogeneous systems and displays the results graphically. Next, the Rate of Change Load Balancer and Concurrent Gang Scheduler will redistribute the jobs and the tasks as necessary.

The current implementation of SaLAD requires the application programs to be described by graphs. SaLaD graphs are variants of directed acyclic graphs, or DAGS. There are two kinds of graphs namely Task and System graphs. Nodes of the task graph represent executables and the arcs represent data dependencies between the executables. Similarly Nodes of the system graph represent computer systems and the arcs represent the communication bandwidth available between them. SaLaD uses the Cluster-M algorithms for clustering and mapping of the task graphs and the system graphs in an efficient way, based on the data dependency between the tasks and the communication capacity available between the systems. Once the tasks have been mapped onto the systems, individual tasks are distributed using PVM. As the load of the system changes and/or as new jobs come in the Rate of Change load balancer and Concurrent Gang scheduler are invoked and they will redistribute the tasks using PVM. SaLaD is composed of integrated graphical tools for creating, executing, and analyzing parallel programs. During execution, SaLaD displays an event-ordered animation of application execution. Through SaLaD, a user can easily schedule and execute applications written in any language over an existing collection of workstations or supercomputers. SaLaD depends on PVM and hence the task graph modules can reside on any Computer system for which a PVM implementation is available.

The SaLaD tool runs on the Windows NT, Windows 95 environments. PVM implementations for Win32 must be installed before using SaLaD. The task graph the user draws is a directed graph in

[2]Network of Workstations (UC Berkeley Project)
[3]Visit http://www.cs.virginia.edu/ legion
[4]Heterogeneous Network Computing Environment (X-Windows based network programming tool)

which nodes represent executables written in C or Fortran. Arcs represent execution ordering constraints. A node may execute when all nodes that precede it in the graph have executed. The restriction placed on the task modules is that they should be using the PVM library to send back results to the SaLaD tool, if it has to be passed to other dependent modules. If the task modules have dependencies in the order of execution but do not have to pass data amongst themselves, then no change needs to be done to the executables. The above restriction is placed due to the fact those unlike tools where users write their programs graphically, SaLaD is more of a Cluster-M scheduling tool for executable modules. Unlike Task modules there is no restriction on the systems. Any system for which a PVM implementation is available can be part of the system graph.

When the SaLaD executable is run the opening screen is divided into four windows as follows:
1.Information Window - Shows the names of the currently loaded task and system graphs.
2.Drawing Window - Used to draw task and system graphs.
3.Map Window - Shows the result of the Cluster-M mapping.
4.Run Window - Show the results of executing the tasks (graph) on the systems (graph) using PVM.

The steps a user must perform to create and run a parallel program under SaLaD are as follows: 1. Draw a task and system graph in the Drawing Window that shows the desired parallel structure. 2. Once the Task and system graphs have been drawn, use the "Map" menu command to Map the task graph on the system graph. Once the mapping is done, the Rep and Spec Clusters and the Mapping results are displayed in the mapping information window. 3. After the tasks have been mapped, they can be executed on the respective systems by using the "Run" Menu command. The Run Results Window displays the status of the execution as it progresses.

5 Conclusion and Future Work

The main goal of our studies has been to concentrate on the development of tools that can be used in the execution environment of IPGs. Mainly, our focus is to complete building the SaLaD tool that is intended to automate the execution of a program inputed by a user of IPG, so that it is distributed among available resources minimizing total execution time. SaLad can be viewed as having three components, Cluster-M Static-allocator the Rate-of-Change Load-balancer, and Concurrent-Gang Dynamic-scheduler. In section 3, we presented how these three components operate in an integrated fashion, and section 4 we presented a simple implementation of SaLaD using PVM which distributes the tasks to the locations determined by the mapper.

We propose to extend the windows-NT based implementation of SaLaD using PVM with a portable programming interface. Currently a user has to specify a set of threads which can be C or C++ code, and their interdependency. Then SaLaD takes this information and produces a mapping using Cluster-M mapping routines, and then the load balancer and dynamic schedulers are triggered as needed. We propose a portable programming interface such that the user just enters the program and then the program is decomposed into a task graph showing the dependencies and extracted parallelisms, and then that information gets supplied to the Cluster-M mapper. One way of pursuing this would be to use a portable parallel language from beginning such as PCN. Our initial investigation shows that clustering constructs can be implemented in PCN so that the process of clustering the graphs is automated. Another approach would be to use a Functional and/or object oriented programming language such as SISAL or JAVA.

References

[Ber87] F. Berman. Experience with an automatic solution to the mapping problem. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 307–334. MIT Press, Cambridge, MA, 1987.

[Bok81] S. H. Bokhari. On the mapping problem. *IEEE Trans. on Computers*, c-30(3):207–214, March 1981.

[BS87] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, pages 439–458, April 1987.

[Cam99] L. M. Campos. *Resource Management Techniques for Parallel and Distributed Systems.* PhD thesis, University of California at Irvine, 1999.

[CE95] S. Chen and M. Eshaghian. A fast recursive mapping algorithm. *Concurrency. Practice and Experience*, 7(5):391–409 August 1995.

[CS99] L. M. Campos and I. D. Scherson. Rate of change load balancing on distributed and parallel systems. In *Proceedings of the Second Merged IPPS/SPDP*, pages 701–707, April 1999.

P. G. Solbalvarro et al. Dynamic Coscheduling on Workstation Clusters. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459:231–256, 1998.

[Efe82] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *IEEE Computer*, 15(6):50–56, 1982.

[ERL90] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, pages 138–153, September 1990.

[ERS90] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. *Journal of Parallel and Distributed Computing*, pages 33–44, October 1990.

M. Eshaghian and M. Shaaban. Cluster-M parallel programming paradigm. *International Journal of High Speed Computing*, 6(2):287–309, June 1994.

[Esh96] M. M. Eshaghian. *Heterogeneous Computing*. Artech House, 1996.

[FA97] D. Feitelson and M. A.Jette. Improved Utilization and Responsiveness with Gang Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1291:238–261, 1997.

[FR92] D. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[FR98] D. Feitelson and L. Rudolph. Metrics and Bechmarking for Parallel Job Scheduling. *Job Scheduling Strategies for Parallel Processing*, LNCS 1459:1–24, 1998.

I. Foster and S. Tuecke. Parallel programming with PCN. Technical report, Argonne National Laboratory, University of Chicago, January 1993.

[HBD90] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1990.

[Jet97] M. A. Jette. Performance Characteristics of Gang Scheduling In Multiprogrammed Environments. In *Proceedings of SC'97*, 1997.

[LA87] S. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Trans. on Computers*, 36:433–442, April 1987.

[LHWS96] P.K.K. Loh, W. J. Hsu, C. Wentong, and N. Sriskanthan. How network topology affects dynamic load balancing. *IEEE Parallel and Distributed Technology*, 4(3):25–35, 1996.

[LK91] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Software Eng.*, 13(1):32–38, 1991.

[Lo88] V. M. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. on Computers*, C-37(11):1384–1397, November 1988.

[LRG$^+$90] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. A. Mohamed, and J. A. Telle. Oregami: Software tools for mapping parallel computations to parallel architectures. In *Proc. International Conference on Parallel Processing*, 1990.

[MG89] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Communications of ACM*, 32(9):1073–1078, September 1989.

[MPT94] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130(1):17–47, 1994.

[PSD$^+$92] R. Ponnusamy, J. Saltz, R. Das, C. Koelbel, and A. Choudhary. A runtime data mapping scheme for irregular problems. In *Proc. Scalable High Performance Computing Conference*, pages 216–219, May 1992.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. The MIT Press, Cambridge, MA, 1989.

[SS99a] F.A.B. Silva and I.D. Scherson. Improving Throughput and Utilization in Parallel Machines Through Concurrent Gang. In *Proceedings of the IEEE/ACM International Parallel and Distributed Processing Symposium 2000*, 1999.

[SS99b] F.A.B. Silva and I.D. Scherson. Towards Flexibility and Scalability in Parallel Job Scheduling. In *Proceedings of the 1999 IASTED Conference on Parallel and Distributed Computing Systems*, 1999.

[ST85] C. Shen and W. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minmax criterion. *IEEE Trans. on Computers*, c-34(3):197–203, March 1985.

[WG90] M. Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):101–119, 1990.

[WLR93] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distributed. Systems*, 4(9):979–993, 1993.

[YG94] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):951–967, September 1994.

[Zad65] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965.

*Luis Miguel Campos received his Ph.D.. degree in Computer Science from the University of California Irvine in 1999, where he his currently a faculty member. Dr. Campos' interests are in the area of resource management in distributed and parallel systems and mobile agents. He his the co-author of the computer language Tea and the web application server I*Tea. He his a former Fulbright scholar.*

Fabricio Silva *received his Ph.D. degree in Computer Science from Universite Pierre et Marie Curie in 2000. He is currently a postdoctoral researcher at the University of California at Irvine. Dr. Silva is a member of IEEE and ACM and has authored numerous scientific articles.*

Isaac D. Scherson *is currently a Professor in the Deptartments of Information and Computer Science and Electrical and Computer Engineering at the University of California, Irvine. He received BSEE and MSEE degrees from the National University of Mexico (UNAM) and a Ph.D. in Applied Mathematics (Computer Science) from the Weizmann Institute of Science, Rehovot, Israel. He held faculty positions in the Dept. of Electrical and Computer Engineering of the University of California at Santa Barbara (1983-1987), and in the Dept. of Electrical Engineering at Princeton University (1987-1991). Dr. Scherson has been a member of the Technical Program Committee for several professional conferences and served as Workshops Chair for Frontiers '92. He is the editor of the book that resulted from the Frontier's 92 workshop and a co-editor of an IEEE Tutorial on Interconnection Networks. He was the Guest Editor of the Special Issue of The Visual Computer on Foundations of Ray Tracing (June 1990.). Dr. Scherson is a member of the IEEE Computer Society and of the ACM. Since July, 1992, he has contributed to the IEEE as member of the IEEE Computer Society Technical Committee on Computer Architecture. His research interests include concurrent computing systems (parallel and distributed), scalable server architectures, scheduling and load balancing, interconnection networks, performance evaluation, and algorithms and their complexity.*

Mary M. Eshaghian *received her Ph.D. degree in Computer Engineering from the University of Southern California in 1988. She is currently Professor of Computer Engineering at the Rochester Institute of Technology. Dr. Eshaghian has chaired and served on many program committees of international conferences and workshops in the area of parallel processing. She has edited several special volumes and has authored numerous scientific articles.*