RESOURCE-AWARE LOAD BALANCING SYSTEM WITH ARTIFICIAL NEURAL NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALİ YILDIZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2006

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Dr. Cevat Şener
Supervisor

Examining Committee Members:

Prof. Dr. Payidar Genç	(METU, CENG)	_____
Dr. Cevat Şener	(METU, CENG)	_____
Prof. Dr. Müslim Bozyiğit	(METU, CENG)	_____
Dr. Atilla Özgit	(METU, CENG)	_____
Dr. Erdal Oktay	(EDA Eng. Design Ltd. Co.)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: ALİ YILDIZ

Signature :

ABSTRACT

RESOURCE-AWARE LOAD BALANCING SYSTEM WITH ARTIFICIAL NEURAL NETWORKS

Yıldız, Ali

MS, Department of Computer Engineering

Supervisor: Dr. Cevat Şener

August 2006, 94 pages

As the distributed systems becomes popular, efficient load balancing systems taking better decisions must be designed. The most important reasons that necessitate load balancing in a distributed system are the heterogeneous hosts having different computing powers, external loads and the tasks running on different hosts but communicating with each other. In this thesis, a load balancing approach, called RALBANN, developed using graph partitioning and artificial neural networks (ANNs) is described. The aim of RALBANN is to integrate the successful load balancing decisions of graph partitioning algorithms with the efficient decision making mechanism of ANNs. The results showed that using ANNs to make efficient load balancing can be very beneficial. If trained enough, ANNs may load the balance as good as graph partitioning algorithms more efficiently.

Keywords: load balancing, distributed systems, artificial neural networks, P-GRADE, graph partitioning

ÖZ

YAPAY ZEKA AĞLARI KULLANILARAK GELİŞTİRİLMİŞ KAYNAK HABERDAR YÜK DENGELEME SİSTEMİ

Yıldız, Ali

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Dr. Cevat Şener

Ağustos 2006, 94 sayfa

Dağıtımlı sistemler populerleştikçe, daha iyi kararlar verebilen verimli yük dengeleme sistemleri geliştirilmelidir. Dağıtımlı bir sistemde yük dengelemeyi zorunlu kılan en önemli nedenler farklı çalışma hızlarına ve dış yüklere sahip heterojen yapıdaki bilgisayarlar ve farklı bilgisayarlarda koşan fakat birbirleriyle haberleşen görevlerdir. Bu tezde, çizge bölümleme and yapay nöral ağlar kullanılarak geliştirilmiş, RALBANN adı verilen bir yük dengeleme yaklaşımı ve yazılımım anlatılacaktır. RALBANN'ın amacı çizge bölümlemeli yük dengeleme algoritmalarının başarılı sonuçlarını, yapay nöral ağların verimli karar alma mekanizmasıyla birleştirmektir. Sonuçlar, yük dengelemesi yapmak için yapay nöral ağları kullanmanın çok yararlı olabileceğini gösterdi. Yeterince eğitildikleri zaman, yapay nöral ağlar, daha verimli bir şekilde çizge bölümlemeli yük dengeleme algoritmaları kadar iyi yük dengeleyebilirler.

Anahtar Kelimeler: yük dengeleme, dağıtık sistemler, yapay sinir ağları, P-GRADE, çizge bölümleme

To My Parents, My Sister and My Fiancée

ACKNOWLEDGMENTS

I would like to thank to my advisor Dr. Cevat Şener for his friendly advices, support and guidance. It would not be possible to complete this study without his motivation and encouragement.

I would like to thank to Prof. Dr. Erol Sahin for his supports about the concepts related with artificial neural networks.

Also, I would like to thank to my family and fiancée for their limitless patient and love which motivated me to complete this study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Scope of the Thesis

During the past several decades, the widespread availability of internet and computer intranets has motivated the migration of several applications, which require high computing power, memory or network communication, from stand alone and centralized environments (PC, main frames) into distributed and parallel environments. The reason of this migration is the problems raised in centralized systems. For example, a centralized system which has to provide web services to millions of users is very likely to crash due to point of failure. In computational mechanics applications like particle simulations and transient dynamics calculations, the computation can not be run on a single processor since most of the CPU would be wasted to switch between several subtasks. In distributed systems, a single computational task is divided into several subtasks and distributed across several computers so that the power (CPU, memory, network, disk) of these computers is accumulated. One of the most important features of distributed systems is the availability to increase their overall computational power by adding new nodes. This requirement is raised because of the need to address increasingly growing computation workloads. The possibility of increasing size of the distributed system eventually results in heterogeneous environments as the newly-added nodes often have better computational power. Several efforts to develop technologies to combine the power of Internet-connected systems are being spent. Grid technologies like Globus [1] and MPICH-G2 [2] are example of such environments. They have enabled computation on heterogeneous and widely-distributed systems on the Internet or Intranet.

However, it is clear that the idea of distributing tasks over several hosts has made the problem of efficient task-resource assignment more difficult. The problem of efficiency emerges from the fact that it is likely to find hosts idle, while there are jobs queuing for execution in other hosts. Therefore it would be better to move from heavily loaded hosts to idle hosts. This and other efficiency problems can be solved via load balancing.

Load balancing can be described as dividing the amount of work that a computer must do between two or more processors or computers so that the average execution time of the application is minimized. Therefore, the main objective of load balancing can be described to decrease average execution time of distributed software. There are two situations: the tasks may communicate with each other or each task execute independently without any communication. If the tasks don't communicate with each other, the objective must be to minimize the execution time of the independent tasks. Otherwise, the objective must be to minimize the average completion times of independent tasks by reorganizing them on hosts so that the average load times of all hosts are nearly the same. The reorganization of the tasks must be done to satisfy the following main objectives [3-5]:

1. All hosts must be kept busy as much as possible.
2. The amount of inter-host communication must be minimized.

The aim of first goal is make every host have the same amount of workload. This is done basically by moving tasks from heavily loaded hosts to idle or less loaded hosts. The aim of the second goal is to decrease the usage of network resource so that the tasks do not spend their time waiting for messages from the other tasks running on different hosts. The solution to this problem is to collect highly communicating tasks on a single host as much as possible.

A lot of load balancing algorithms have been devised to fulfill the main objectives of load balancing. What they do is to find the best matching between the tasks and the resources that fulfills the 2 requirements. However, examining and finding the best matching of T tasks to P processors is an NP-complete problem [6]. Therefore, all algorithms try to find an optimal or near-optimal solution in an efficient way. Gener-

ally, load balancing algorithms is comprised of three phases: information collection, decision making and data migration (Figure 1). During the information collection phase, load balancer gathers the information of workload distribution and the state of computing environment and detects whether there is a load imbalance. The decision making phase focuses on calculating an optimal data distribution, while the data migration phase transfers the tasks from overloaded processors to under loaded ones. The load balancing algorithms is widely discussed in section 2.1.

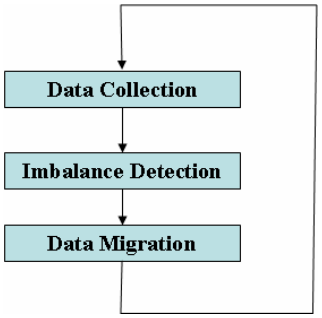


Figure 1 - Load Balancing Phases

Classification of Load Balancing Algorithms:

Load balancing algorithms can be classified according to the following criterias [7]:

- Whether balancing is done at run-time, dynamically;
- Load balancing and task migration is done at single host;
- Whether tasks that have already begun can not be rescheduled.

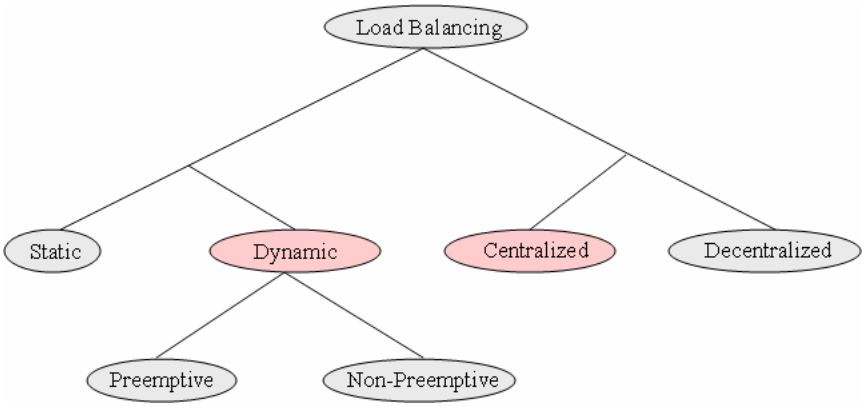


Figure 2 - Classification load balancing algorithms

In centralized algorithms, the load balancing and process migration decisions at only one host. On the contrary in decentralized algorithms, load balancing is done at all hosts. Centralized are better at balancing interdependent tasks since they can see the system globally; but decentralized algorithms are better for scheduling independent tasks.

In static algorithms, load balancing decision is determined statically for each task, without considering the run-time loading conditions. They work better when tasks have predictable resources and hosts have predictable workloads since they need no information collection at run time. However in dynamic algorithms, the host to distribute the task is chosen at run time according to the workloads of the hosts. These are more suitable when little is known about either the tasks being scheduled or the loading conditions are changeable during task execution. Dynamic algorithms [8] may be further classified as either preemptive or non-preemptive; the former permit migration of tasks that have already begun execution, but the latter do not.

Dynamic load balancing algorithms can be mainly divided into geometric and topological algorithms [9]. In geometric load balancing algorithms, the tasks are distributed to hosts according to the geometric locations. Normally, these kinds of algorithms are mostly used for problems in which the interactions are geometric, like particle simulations. RCB (Recursive Coordinate Bisection) [3,10] and space filling

curves [11] are the most commonly used geometric partitioning algorithms. On the contrary topological algorithms work using the interactions between the tasks instead of geometric positions. Topological algorithms can be used for nearly all problems since connectivity of the tasks is implicit and exploited by the algorithms. Local methods (diffusion [12], demand driven, dimension exchange), RSB (Recursive Spectral Bisection) and graph partitioning algorithms are the most commonly used topology based load balancing algorithms. Researches show that topological algorithms are much more superior to geometric algorithms in terms of quality. Especially, graph partitioning algorithms create the best load balancing results. However, they suffer from large memory usage and processor usage. Geometric and local methods run the fastest with less memory usage; however their quality is not as good as graph partitioners' quality. In Table 1, the comparison of load balancing algorithms is given [3,4].

Table 1 - Comparison of Dynamic Load Balancing Algorithms

Method	Quality	Speed	Memory
Master/Slave	***	Not Scalable	***
Geometric Methods			
RCB/URB	*	***	***
RIB	*	**	***
Octree/SFC	*	***	**
Local Methods			
Diffusion	**	***	***
Demand Driven	**	***	***
Dimension Exchange	**	***	***
Graph Partitioners			
RSB	***	*	*
Multilevel	***	**	*

The aim of this thesis is to describe a resource aware, machine learning and dynamic load balancing model to address the problems of graph partitioning based load balancing algorithms. This model is called RALBANN (**R**esource **A**ware **L**oad **B**alancing with **A**rtificial **N**eural **N**etworks). The main goal of RALBANN is the learning of load balancing based graph partitioning using artificial neural networks, to bring together the successful partitioning results of graph partitioning algorithms with the efficiency and learning capability of artificial neural networks in a resource aware scheme. This is accomplished by mapping the task partitions to the computing resources according to the capabilities of the computing resources producing a resource aware load balancing scheme. Feed-forward artificial neural networks are used as the machine learning tool.

Graph partitioners are used to teach RALBANN's learning module how to balance the network. The reason for choosing graph partitioners is that they produce the most successful load balancing results among all other load balancers (Table 1). However, their disadvantage is that they are very inefficient in terms of memory usage and speed. Therefore, RALBANN aims at balancing the load as successful as graph partitioners with better memory and processor usage using artificial neural networks. RALBANN basically tries to (1) divide the task interaction graph (TIG) of the distributed application into different task groups and (2) find the best matching between the task groups and the hosts considering their computational power and external loads. As the supervisor for artificial neural networks, a component which partitions the graphs using multilevel graph partitioning algorithms is implemented.

Traditional graph partitioning based load balancing algorithms have several drawbacks. Our model tries to find solutions to these drawbacks. These algorithms try to divide the tasks into groups of different size. Each group consists of only tasks that communicate with each other heavily. However the increase in the execution of distributed applications depends on not only the distribution of the tasks over the hosts composing the distributed environment but also the computational power and external load of the hosts. Normally, graph partitioning algorithms don't take care of these issues and are not aware of the computational power and external load of the hosts.

To further increase the efficiency of distributed applications, these external constraints must be taken care, too.

Another problem with graph partitioning algorithms is their inefficiency in terms of both memory usage and speed. As described in section 2.1, graph partitioning algorithms used for load balancing are generally inefficient since graph algorithms take too much memory and computational time. In fact, graph partitioning is an NP complete problem. To find the best solution, the algorithms must scan all possible task-to-host assignments and select the one which reduces the completion time of the distributed application at most. Since the complexity of such an operation is very large, most graph partitioning algorithms try to find an optimal solution, which is not guaranteed to be the best.

The most important contribution of this thesis is the automated learning and execution of load balancing. Most load balancing algorithms require some parameters to be tuned manually by a human designer before they are executed as explained in Mehra's work [13,14]. Therefore these algorithms work well, only when these parameters are adjusted accordingly. When the configuration of the system changes or when new applications are run on the system, the parameters must be adjusted again. On the contrary, RALBANN has capabilities to learn and adapt the distributed system from scratch without any priori information about the application and the system.

Our thesis proposes an integrated approach that addresses the problems of current problems of graph partitioning based load balancing. Our load balancing module contains 2 artificial neural networks which learn (1) to partition TIG graph and (2) decide the assignment of the tasks over the hosts by considering their heterogeneous computational power and external loads in a resource-aware scheme. The learning process can be done automatically during execution. Therefore the load balancing module can adapt itself when there are changes in the system. Using neural networks, the inefficiency of graph partitioning algorithms is replaced with efficient decision making mechanism of artificial neural networks. Once trained enough, it runs more efficient than graph partitioning based load balancing algorithms in terms of memory and CPU usage.

The load balancing module implemented has been fully integrated and test with P-GRADE (**P**arallel **G**rid Run-Time and **A**pplication **D**evelopment **E**nvironment) parallel application development environment. Using the monitoring tool of P-GRADE, our module collects the necessary information and takes the necessary load balancing decisions.

To summarize, the main contributions of this thesis are the following:

- Learning of graph partitioning and load balancing using artificial neural networks,
- Resource aware load balancing with artificial neural networks,
- Learning of computing resources and distributed application with different artificial neural networks: RALBANN can learn the distributed system (properties of the computational resources) separately from the application running on the system. For a specific distributed network, the properties of computational resources are learnt once and saved to a file. Later, it can be used for each application running on the same network. This makes learning time faster and efficient.

1.2. Outline of the Thesis

The thesis is organized as follows. The next chapter contains background information about artificial neural networks and load balancing. Further, several researches and projects about load balancing are described briefly. Chapter 3 describes our proposed model. The chapter starts by first explaining the models used in RALBANN, namely abstract distributed system model and graph model. In this chapter, the architecture and operation of RALBANN is described in detail, too. The chapter concludes by describing the steps to integrate RALBANN with other graph partitioner algorithms and other distributed systems. Chapter 4 lists the experimental tests of the model. In Chapter 5, main contributions of our research and some ideas for future work are listed. In Appendix A, developer's manual for RALBANN takes part. This chapter contains class diagram of RALBANN with the important classes and structures are described.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this thesis, improving the efficiency and effectiveness of load balancing systems used in distributed systems are concerned. Before the relevant research with similar objectives is presented, the basics of graph partitioning algorithms and artificial neural networks are summarized.

2.1. Load Balancing Algorithms

Due to its critical role in high-performance distributed computing, the load balancing issue has been studied extensively in recent years and a number of load balancing algorithms have been devised for parallel computing. The load balancing algorithms can be classified in 5 categories according to the methods and data structures they used [3,4,15]:

- Master/Slave
- Optimization
    - Simulated Annealing
- Local Methods
    - Diffusion
    - Demand driven
    - Dimensional Exchange
- Graph Partitioning
    - Geometric Methods
    - Structural Methods
    - Refinement Algorithms

- Multilevel Techniques
- Parallel Techniques

- Machine Learning
  - Unsupervised
  - Supervised

2.1.1. Master/Slave

The master/slave load balancing approach is the simplest load balancing algorithm. One of the hosts is called *the master* which is responsible for maintaining the tasks and delivering them to the slaves. The slave hosts finish their tasks and request new task from the master host.

The master/slave approach has very attractive features that make it well suited for many problems. It is quite easy to implement. However, master/slave model is only suitable for problems in which tasks can be performed independently and asynchronously by a single computer. Furthermore, it must be possible to send a task with small messages so that the message communication between the slaves and the master doesn't cause a bottleneck.

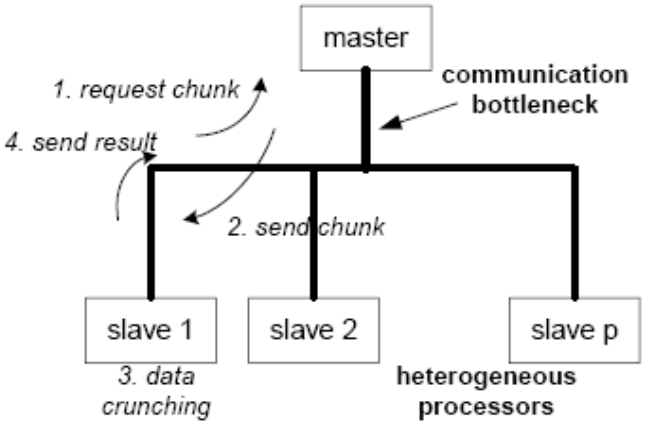


Figure 3 - Master-Slave

2.1.2. Local Methods

Most of load balancing algorithms use global data to determine the idle hosts and the processes to be migrated. That is, all hosts and processes in the distributed systems are examined globally to take the necessary steps to decrease average completion time.

On the contrary, local load balancing methods work in smaller neighborhoods [3,16]. The aim is to spread the workload of highly loaded processors to other processors in the defined local neighborhood. The neighborhoods are chosen to overlap each other so that over several iterations the workload is spread over all processors. Unlike global methods, each iteration of local methods is usually quite fast and inexpensive. Because all of the information and communication is performed within small sets of processors, the methods scale well with the number of processors. Local methods are incremental and can be parallelized easily. One iteration can reduce a single heavily loaded processor's workload significantly. Since the total time computation time is determined by the time required by the most heavily loaded processor, a small number of iterations may be sufficient to reduce imbalance.

The most important advantage of local methods is that it is quite incremental, fast and efficient since they work on small neighborhoods. In a few iterations, they can reduce the workload of an overloaded processor. Since the total computation time is determined by the most heavily loaded processor, a small number of iterations are enough to reduce the computation time to an acceptable level. These kinds of algorithms are very easy to distribute since they work on individual processors and don't need global data.

However when global balance instead of local balance is needed, a lot of iterations may be needed to converge to a good balance state. Therefore high quality load balancing is quite difficult with local methods. Local methods are generally very efficient and fast in doing small and local changes in load balance.

In distributed application that works asynchronously, the local methods may be more efficient, since global methods are necessarily synchronous. When a processor has

finished its task, it can initiate load balancing to request more jobs. While other neighborhoods continue their tasks, single neighborhoods may do load balancing. But the disadvantage here is that asynchronous models are difficult to implement.

Local methods have two simple steps which highly affect their performance. In the first step, the method calculates how much work must be moved from the processor to other processors to balance the load. In the second step, the method selects the tasks and processors to be moved. The tasks to be migrated are chosen according to their communication connections with other tasks. This guarantees the minimization of the communication costs of the tasks.

Some example algorithms in this category are diffusion, demand driven and dimensional exchange.

2.1.2. Graph Partitioning (Mesh Partitioning)

The most important part of load balancing algorithms includes *graph partitioning* algorithms. This kind of algorithms models the problem domain as a graph consisting of weighted vertices and edges [3,4]. Using this model, load balancing problem is transformed to a problem with the goal to reduce the imbalance among the nodes and minimize the communication between the nodes.

In its most general form, the graph partitioning tries to find how best to divide a graph's vertices into a specified number of subsets such that:

1. The number of vertices per subset is equal and
2. The number of edges straddling the subsets is minimized.

A graph is defined in terms of a set of vertices V, and a set of edges E. Edges connect vertices from V pair-wise and are undirected. Self-loops are not permitted. A p-way partition of a graph is a mapping P:V->[1..p] of its vertices into p subsets $S_1,S_2,,,S_p$, where $\cup_i S_i=V$ and $Si \cap Sj= \varnothing$ wherever $i \neq j$. Every partition generates a set of cut edges, E_c, defined as the subset of E whose endpoints lie in distinct partitions $E_c=\{(v_i,v_j) | (v_i,v_j) \in E, P(v_i) \neq P(v_j)\}$. The weight of each subset, $|S_i|$, is defined to be the number of vertices mapped to that subset by P.

Given a graph as input, the graph partitioning problem seeks to find a p-way partition in which each subset contains roughly the same number of vertices ($|S_i| \leq \lceil |V|/p \rceil$) and the number of cut edges, $|E_c|$, is minimized. For input graphs that represent workload as described in the introduction, each partitioned subset represents data and computation that should be assigned to a single processor. The cut edges represent the inter-host communication required by the distribution. Thus, the graph partitioning problem attempts to find a distribution that balances the computation done by each processor while minimizing the total inter-host communication.

Traditional graph partitioners share the drawback that stems from the priori knowledge of the workload and system state, and cannot adapt to system changes. On the other hands, they mainly focus on tightly-coupled systems consisting of homogenous processors thus can not handle the heterogeneity in distributed systems. Furthermore, these algorithms only consider the imbalance of sub domains and the edge-cut communication, while neglect the data movement cost that can be a crucial performance bottleneck due to the high communication latency in distributed systems.

Examples of graph partitioning based load balancing algorithms are:

- **Recursive Bisection:** Recursive bisection algorithm [17] is divide-and-conqueror approach that recursively divides the graph into 2 different parts. Although this algorithm is NP-complete, it is still the most used algorithm because of its simplicity compared to other p-way partitioning algorithms.

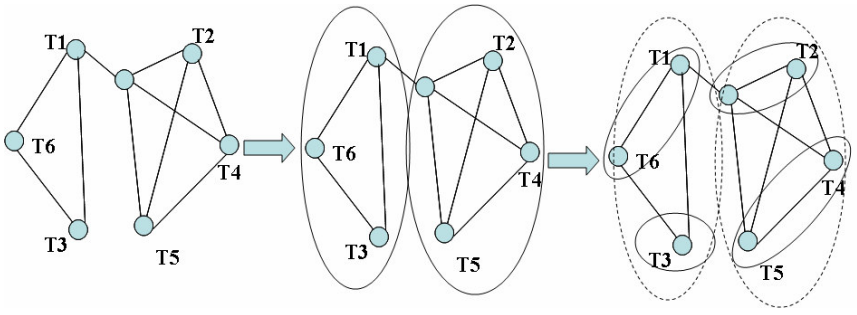


Figure 4 - Recursive Bisection Algorithm

Several variations of the basic graph partitioning problem exist. The weighted graph partitioning problem allows weights to be associated with the vertices and edges of G. In this problem, a good partition is one in which the total vertex weight of each subset is roughly equal, and the total weight of the cut edges is minimized. In the context of workload graphs, node weights can be used to encode differing computation expenses across the graph (e.g., boundary locations vs. internal locations). Similarly, edge weights can signify the volume of communication required between two nodes.

The δ-partitioning problem is one in which some imbalance in the subset weights is tolerated in hopes of significantly reducing the number of cut edges. In this problem, a tolerance δ is supplied as input where $1/p <= \delta <= 1$. Legal partitions are those that yield subsets with weight $|S_i| <= \lceil \delta |V| \rceil$. Note that the standard graph partitioning problem is simply a δ-partitioning problem in which $\delta = 1/p$. Another variation that allows for unbalanced subsets is the skewed partitioning problem. In this version, user-supplied weights are associated with each subset to specify a desired imbalance in the partition. Algorithms for this problem compute partitions whose subsets are weighted proportionally to those specified by the user.

In the context of workload distribution, these generalizations allow the characteristics of a parallel machine to have some bearing on the partitioning. For example, if a machine's inter-host communication overhead is sufficiently high, it might be worthwhile to tolerate some degree of load imbalance in order to drastically reduce the communication volume. This tradeoff can be described using δ-partitioning. As another example, if a parallel computation is to be performed on a heterogeneous processor set, the relative performance of the processors can be described by specifying appropriate target weights in the skewed partitioning problem.

- **Geometric Methods:** These methods assume that the vertices have associated geometric coordinates [3,4]. Many mechanical computations like physical simulations have an underlying geometry. The object interacts if they are near each

other. This property enables the geometric methods to divide the work to processors according to their coordinates. The main idea is to divide problem space in the several partitions recursively (starting from 2) according to the coordinates. Examples of such algorithms are Recursive Coordinate Bisection (RCB), Recursive Inertial Bisection (RIB), Recursive Orthogonal Bisection (ROB) and Circle Bisection. These kinds of algorithms have two disadvantages: the first is that they work only on graphs that have geometric coordinates associated with them. The second is that they never care the connectivity of the graph and require a lot of computation.

- **Structural Methods:** These kinds of algorithms resemble geometric methods. However they don't lack the disadvantages of the geometric methods. They define the distance between two vertices as the length of their shortest connecting path, rather than their distance in Euclidean space. The algorithm finds two vertices of near-maximal distance from one another and then performs a breadth-first search from one of the vertices, until it has reached half of the vertices in the graph. These vertices are placed in the first subset, leaving the remainder in the second. The algorithm is then applied recursively to each of the subgraphs. Examples are recursive level-structure bisection, graph walking algorithms and recursive spectral bisection (RSB).
- **Refinement Algorithms:** This class of algorithms tries to improve an initial (possibly random) partition of the graph by trading vertices from one subset to the other with the goal of reducing the number of cut edges. Kernighan-Lin (KL) [18] algorithm is based on this notion. This algorithm will be further described in section 3.2.3.
- **Multilevel Techniques:** The most famous and successful graph partitioning algorithms are multilevel algorithms [17,19-21]. These techniques are analogous to multigrid methods for solving numerical problems. Both approaches construct a hierarchy of approximations to the original problem so that a coarse solution can quickly be generated. This solution is then progressively refined at the more detailed levels of the hierarchy until a solution for the original problem is reached. In the same way graph partitioning algorithms construct a smaller approximation to the original graph in a few steps. The smallest graph is the partitioned very ef-

ficiently. In the last step the partitioned graph is propagated back to the original graph through intermediate graph. The disadvantage of these algorithms is that they are expensive in both computation and memory. However they produce the highest quality partitions among all other partitioner algorithms. This algorithm will be described in detail in section 3.2.3.

- **Parallel Techniques:** The algorithms in this class are generally the parallel forms described previously [3,4]. However they are not widely accepted since the parallelized versions don't provide an acceptable increase and are not worth implementing. However they are useful anyway since there are some compelling reasons that motivate parallel solutions to the graph. One of these is that although the parallel computers have enough memory for handling large computations, the graph of these computations may not fit in the memory. Another situation in which parallel algorithms would do better is that runtime changes in the computation's workload result in the need for dynamic load balancing. Given the choice, it would be preferable to compute a repartitioning in-place rather than to ship the entire graph to a single processor and generate a new partition from scratch. Thus, parallel solutions must be considered.

2.1.3. Optimization

In these algorithms, load balancing problem is modeled as a graph coloring problem [22]. The distributed problem is modeled as an undirected graph as in graph partitioning based load balancing algorithms. Load balancing problem is described as coloring the vertices of this graph with P (processors) different colors to minimize a cost function which is related to the time taken to execute the program for a given coloring. The choice for the cost function here is critical. An example cost function is the following: $H = H_{wl} + \eta H_{comm}$ where H_{wl} is minimized when each processor has equal workload, H_{comm} is minimal when communication is minimized, and η is a parameter expressing the balance between the two values.

To apply the optimization algorithms to load balancing, an analogy between load balancing and physical systems. The tasks to be distributed can be thought of as par-

ticles moving around in the discrete space formed by the processors. This physical system is controlled by the Hamiltonian (energy function) given as:

$$H = \frac{P^2}{N^2} \sum_q N_q^2 + \mu \left(\frac{P}{N}\right)^{\frac{d-1}{d}} \sum_{e \leftrightarrow f} 1 - \delta_{p(e),p(f)}$$

The first term in this equation ensures equal work per node and is a short-range repulsive force trying to push particles away if they land in the same node. The second term is a long-range attractive force which links ``particles'' (data points) which communicate with each other. This force tries to pull particles together (into the same node) with strength proportional to the information needed to be communicated between them. In general, this communication force depends on the architecture of the interconnections of the parallel machine.

When this equation is used as the cost function in graph coloring analogy, the load balancing problem becomes finding the equilibrium state of a system of particles with a ``conflict'' between short-range repulsive (hardcore) and long-range attractive forces.

Simulated annealing is an optimization method which simulates the slow cooling of a system. In simulated annealing, there is a cost function H, which associates a cost to any state of the system and a temperature T. The algorithm works by iteratively proposing changes ΔH in H. The change is accepted or rejected according to the following rules, if the cost function decreases $\Delta H<0$, the change is accepted unconditionally; otherwise it is accepted but only with a probability. A crucial requirement for the proposed changes is reachability that there be a sufficient variety of possible changes that one can always find a sequence of changes so that any system state may be reached from any other.

If the temperature is chosen close to zero, the changes are accepted only if H decreases. This is called hill-climbing. The system reaches a state in which none of the proposed changes can decrease the cost function, but this is generally a local optimum.

On the other hand, if the temperature is very large, all changes are accepted, and in this case, all states of the system are visited regardless of cost function because of the reachability property of the set of changes.

Simulated annealing starts by proposing a lot of changes at a high temperature to explore state space and gradually decreasing the temperature to zero while hopefully settling on the global optimum.

2.1.4. Machine Learning

The goal of machine learning algorithms is to discover some underlying structure of a set of data. In machine learning load balancing algorithms, load balancing problem is taken up as a learning task.

- **Reinforcement Learning**: Reinforcement learning [23-25] is a machine learning method which is based on mapping states to actions. The learner lives in a dynamic environment in which it can receive rewards to its action. The learner is given a goal and not told which actions to take in a state but instead must discover which actions yield the most reward by trying them. It learns how to achieve the goal by trial and error interactions with its environment. By choosing the action which takes the highest reward in a state, the learner reaches the goal. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it already knows in order to obtain reward, but it also has to explore in order to make better action selections in the future.

  The usage of reinforcement learning in load balancing [26-28] is rather restricted due to the complexity and huge number of states in load balancing problems. Anyway, there are some algorithms which can learn simple load balancing algorithms like master-slave load balancing algorithm.

- **Self-Organizing Maps (Kohonen Networks):** Kohonen network [29] is a type of artificial neural network model whose learning is unsupervised, unlike the

previous self-learning algorithms. No priori knowledge about the input data or output is needed by the network. That is, neither external rewards signaling the success of the output nor input/output pairs are feed with the input data. The network architecture consists of a set of neurons, usually arranged as a two-dimensional map. All the map neurons are connected to a set of input neurons. Any activity pattern on the input neurons gives rise to excitation of some local group of map nodes. After learning, the spatial positions of the excited groups specify a mapping of the input onto the map. The result is a system that maps similar inputs close to each other in the resulting map.

There have been some attempts in applying the idea of Kohonen networks to load balancing [30-32]. In these algorithms, generally the input is a vector representing the task interaction graph and depending on the algorithm the output signals some kind of information to balance the load in the system.

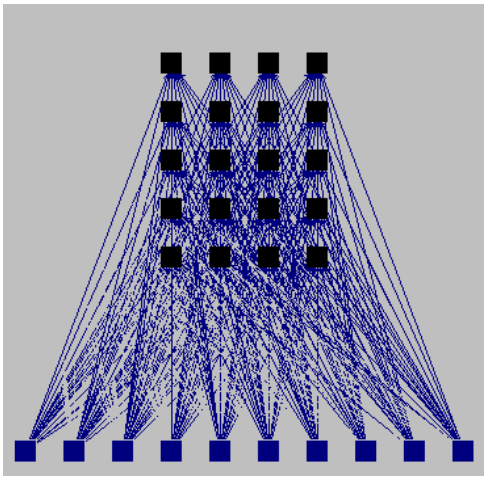


Figure 5 - Kohonen Network

2.2. Artificial Neural Networks

Artificial Neural Networks [33-35] can be described as computational model, inspired by biological findings relating to the behavior of the brain as a network of

units called neurons. Neurons are the most basic elements of the human brain which provides humans with abilities to remember, think, and apply previous experiences to every action. Basically, a biological neuron receives inputs from other neurons, combines them in some way, performs a generally nonlinear operation on the result, and then outputs the final result. The power of the human mind comes from the large numbers of these basic components and connections between them.

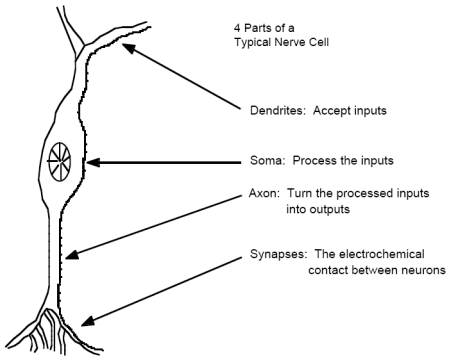


Figure 6 - A simple brain neuron

Like brain, artificial neural networks consist of interconnected processing units called neurons that send signals to one another depending on their incoming signals. The three basic components of the (artificial) neuron are:

1. The synapses or connecting links that provide weights, w_j , to the input values, x_j for j = 1, ...m;
2. An adder that sums the weighted input values to compute the input to the activation function $v = w_0 + {}^{m}\sum_{j=1} w_j x_j$, where w_0 is called the bias (not to be confused with statistical bias in prediction or estimation) is a numerical value associated with the neuron. It is convenient to think of the bias as the weight for an input x0 whose value is always equal to one, so that $v = {}^{m}\sum_{j=0} w_j x_j$;
3. An activation function g (also called a squashing function) that maps v to g(v) the output value of the neuron.

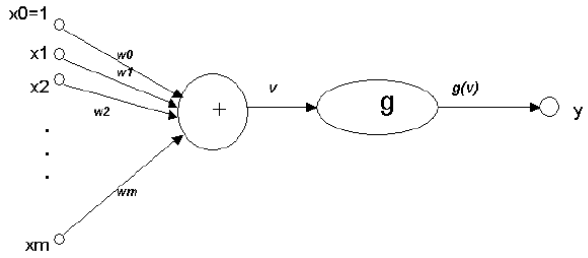
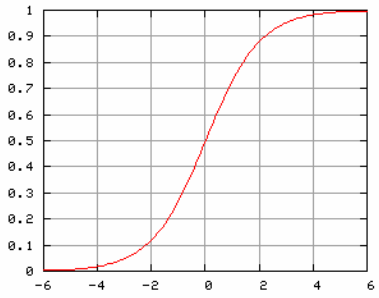


Figure 7 - An artificial neuron

Input vector is applied to the neuron via input connections. The connections have weights which changes while the neural network learns. Weights can either excite or inhibit the transmission of the input value. Mathematically, input values are multiplied by the value of that particular weight. At the neuron node, all weighted-inputs are summed. This summed value is then passed to an activation function. Some examples of activation functions are: step function, ramp function and sigmoid function:



$$g(v) = \left(\frac{e^v}{1+e^v}\right)$$

Figure 8 - Sigmoid function

The sum of the weighted inputs represents the horizontal axis in Figure 8. The curve represents the output of the function for each value on the horizontal axis. The func-

tion output is then sent to each node that is connected to an output weight of the firing neuron.

ANN architectures are used in process modeling at most. The main advantage is that understanding of the process is not necessary. Given the inputs and outputs, neural network can be constructed and trained to accurately imitate the process.

Feed Forward Networks: While there are numerous different artificial neural network architectures that have been devised to perform a range of tasks including pattern recognition, data mining, classification, and process modeling, the most successful architecture is the multilayer feed forward networks, which has also been used in this thesis. These networks consist of layers, which in turn consist of several neurons. All nodes in a given layer are connected to all nodes in a subsequent layer. The network requires at least two layers, an input layer and an output layer. In addition, the network can include any number of hidden layers with any number of hidden nodes in each layer (not necessarily the same in each hidden layer, in fact, typically not). An example of a typical feed-forward NN is as following:

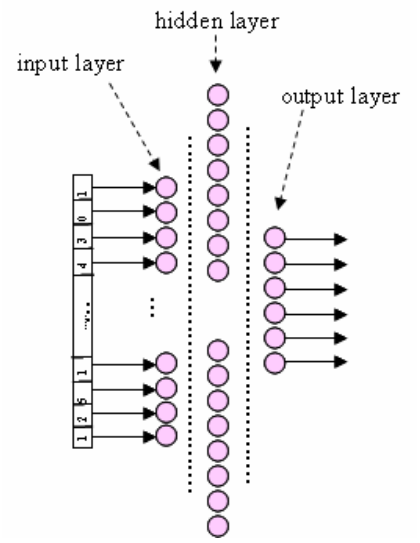


Figure 9 - Multilayer feed forward network

In this architecture, the input is given to the input network. Using algorithms described above, the output values are calculated at each neuron and they are propagated through the network until the output layer is reached. The output vector represents the predicted output. This predicted output may contain error and a tuning may be required. Tuning means adjusting the weights of the neurons so that the predicted output is closer to actual output. The tuning process is called *training*. There are a variety of learning techniques; the most popular is back-propagation. The output values are compared with the correct answer to compute the value of some predefined error-function. By various techniques the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles the network will usually converge to some state where the error of the calculations is small.

Recurrent Networks: In the contrary to feed forward networks in which all data flows contain no cycles, recurrent networks are models with bi-directional data flow. For instance, a hidden unit can be connected with itself over a weighted connection, connect hidden units to input units, or even connect all units with each other.

Each time a pattern is presented, the unit computes its activation just as in a feed forward network. However its net input now contains a term which reflects the state of the network (the hidden unit activation) before the pattern was seen. When subsequent patterns are presented, the hidden and output units' states will be a function of everything the network has seen so far. The network behavior is based on its history, and pattern presentation must be arranged as it happens in time.

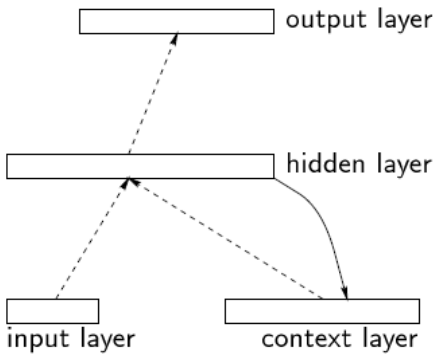


Figure 10 - Simple recurrent network

Simple recurrent network: A simple recurrent network (SRN) is a variation on the multi-layer perceptron. A three-layer network is used, with the addition of a set of "context units" in the input layer. There are connections from the middle (hidden) layer to these context units fixed with a weight of one. At each time step, the input is propagated in a standard feed-forward fashion, and then a learning rule (usually back-propagation) is applied. The fixed back connections result in the context units always maintaining a copy of the previous values of the hidden units (since they propagate over the connections before the learning rule is applied). Thus the network can maintain a sort of state, allowing it to perform such tasks as sequence-prediction that is beyond the power of a standard multi-layer perceptron.

In a fully recurrent network, every neuron receives inputs from every other neuron in the network. These networks are not arranged in layers. Usually only a subset of the neurons receive external inputs in addition to the inputs from all the other neurons, and another disjunct subset of neurons report their output externally as well as sending it to all the neurons. These distinctive inputs and outputs perform the function of the input and output layers of a feed-forward or simple recurrent network, and also join all the other neurons in the recurrent processing.

Hopfield network: The Hopfield network consists of a set of N interconnected neurons which update their activation values asynchronously and independently of other neurons. All neurons are both input and output neurons. The activation values are

binary. This network guarantees that its dynamics will converge. If the connections are trained using Hebbian learning then the Hopfield network can perform robust content-addressable memory, robust to connection alteration.

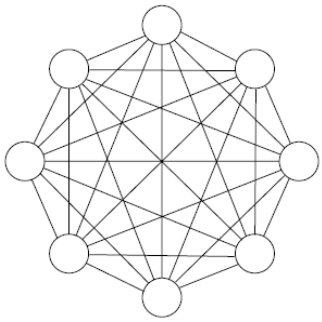


Figure 11 - Hopfield Network

Stochastic neural networks: Stochastic neural networks are built by introducing random variations into the network, either by giving the network's neurons stochastic transfer functions, or by giving them stochastic weights. This makes them useful tools for optimization problems, since the random fluctuations help it escape from local minimums. For example, Boltzmann machine can be thought of as a noisy Hopfield network.

Modular neural networks: Modular neural networks are networks that contain smaller networks which cooperate or compete with other to solve the problem. For example, a committee of machines (CoM) is a collection of different neural networks that together "vote" on a given example. This generally gives a much better result compared to other neural network models. In fact in many cases, starting with the same architecture and training but different initial random weights gives vastly different networks. A CoM tends to stabilize the result.

Associative Neural Network (ASNN) is an extension of the committee of machines that goes beyond a simple/weighted average of different models. ASNN represents a combination of an ensemble of feed-forward neural networks and the k-nearest

neighbor technique (kNN). It uses the correlation between ensemble responses as a measure of distance amid the analyzed cases for the kNN. This corrects the bias of the neural network ensemble. An associative neural network has a memory that can coincide with the training set. If new data becomes available, the network instantly improves its predictive ability and provides data approximation (self-learn the data) without a need to retrain the ensemble. Another important feature of ASNN is the possibility to interpret neural network results by analysis of correlations between data cases in the space of models.

2.3. Related Works

Learning of load balancing is a hard task which has been investigated by several researchers. The difficulty is raised from the complexity of the problem. The presence of extra workload and hardware heterogeneity on the hosts puts extra complexity to this problem. When hardware heterogeneity is concerned, one of the questions that comes to mind is what sizes should task partitions be such that they match the capabilities of the hosts or collection of hosts on which they are to be mapped. Another question is how these partitions mapped to hosts so that the average execution time of the whole application is minimized. Before the answers to these and related questions are given, recent efforts that address load balancing and machine learning are discussed.

Mehra and Wah [13,14,36,37] created a tool, SMALL, that uses artificial neural networks as the learning tool for load balancing. According to their work, load balancing problem have two components: load indices, which indicate each host's load; and decision policies, which determine both the conditions under which tasks are migrated and the destinations of incoming tasks. Load balancing systems use workload indices to dynamically schedule tasks. Load indices are the values that show how much a host is loaded. Load indices generally combine information from the key resources of contention: CPU, disk, network, and memory. Alternative destinations for each incoming task by the expected speed-ups are ranked according to their load indices. Normally, load values are computed using a manually-specified formula as functions of current and recent utilization levels of various resources. However, Me-

hra and Wah propose ranking alternative destinations for an incoming task by their respective completion times. However, completion times can only be measured for completed tasks, whereas decisions need to be made before tasks start. Therefore, completion times need to be predicted using only the information available before a task begins execution. Without knowing the resource requirements of tasks, absolute task-completion times can not be predicted. Therefore they propose to predict the relative completion times of the tasks for different hosts. It is sufficient that every host predicts the completion time of an incoming task relative to its completion time on the chosen idle file server, given only the loading conditions at task-arrival time. Predicted relative completion times can be used as load indices. Comparator neural networks, one per host, is used to learn to predict load indices (the relative speedup of an incoming job) using only the resource utilization patterns of the hosts observed prior to the job's arrival.

Another part of the work, Mehra and Wah discusses how decision policies can be self-learnt. In Figure 12, there is an example load balancing decision policy. The sender-side rules are evaluated at s, the host of arrival of a task. Reference can be either 0 or MinLoad; the other parameters - δ, θ_1, θ_2 - take non-negative real values. A remote destination, r, is picked randomly from Destinations, a set of hosts whose load indices fall within a small neighborhood of Reference. If Destinations is the empty set, or if the last sender-side rule fails, then the task is executed locally at s; otherwise, site r is requested to receive the task. Upon receiving that request, site r applies its receiver-side rule. If the receiver-side rule succeeds, the task is migrated; otherwise, the task is executed locally at s.

SENDER-SIDE RULES(s)
Destinations = {site: Load(site)–Reference(s) < $\delta(s)$}
Destination = Random(Possible_destinations)
IF Load(s) – Reference(s) > $\theta_1(s)$ THEN Send

RECEIVER-SIDE RULES(r)
IF Load(r) < $\theta_2(r)$ THEN Receive

Figure 12 - Load balancing strategy

The parameters that reside in this policy must be adjusted correctly to take correct load balancing decisions.

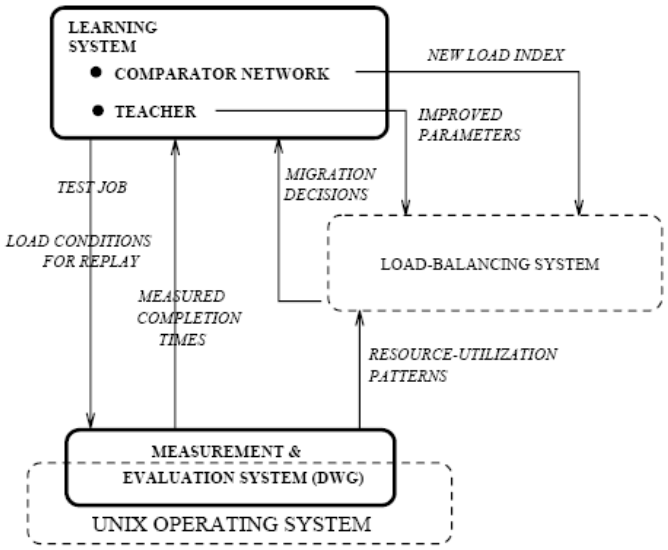


Figure 13 - Architecture of SMALL

There are 3 key components in the architecture of SMALL: a workload generator (DWG), a comparator network that trains load-index function, and a teacher that tunes the parameters of given load-balancing policies. DWG is responsible for:

- precise measurement of resource-utilization information
- precise generation of recorded loads
- measurement of job-completion time

After the raw measurements supplied by DWG have been preprocessed using filtering and extrapolation, they are used by the local load-index function for computing a load index. The given load-balancing policies use the load indices, along with other policy parameters, in order to determine the most appropriate destination for each incoming job.

Both the load-index function and the parameters of load-balancing policies can be modified based on the completion-time measurements provided by DWG. Such modifications are carried out by the learning system, which has two components: one to learn a new load-index function for each site, and another to tune the parameters of a set of site-specific policies.

The load-balancing system implements the policy shown in Figure 12, and converts the primitive measurements provided by DWG's measurement facilities into the more meaningful load indices. This system includes support for communicating the load indices among the sites, as well as for computing abstract performance metrics such as MinLoad, which denotes the minimum predicted load index.

Although Mahra's work contain some innovative ideas in strategy learning and comparator neural networks areas, the quality of its load balancing decisions can not compete with the quality of complex load balancing algorithms. The reason is that SMALL works distributed in small neighborhoods and therefore doesn't collect global data and determine the global balance.

Heiss and Dormanns [30] described how to use Kohonen networks to create a self learning load balancing system. The objective of a Kohonen network is to map input vectors (patterns) of arbitrary dimension N onto a discrete map with 1 or 2 dimensions. Patterns close to one another in the input space should be close to one another in the map. A Kohonen network is composed of a grid of output units and N input units. The input pattern is fed to each output unit. The input lines to each output unit are weighted. These weights are initialized to small random numbers. With a given input, each neuron computes the weighted sum:

$$\sum_{i=1}^{m} w_{ip} x_i$$

where w_{ip} is the weight of input I for neuron p and x_i is the i^{th} input value.

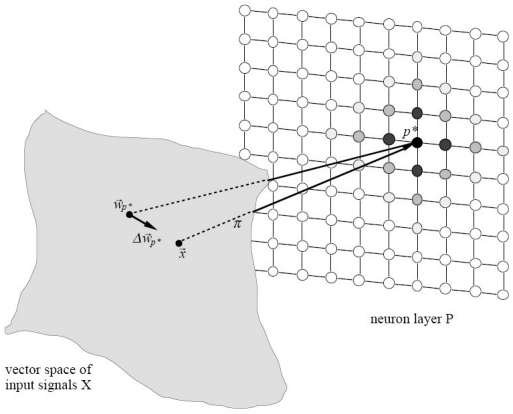


Figure 14 - Kohonen Network

The learning process is as roughly as follows:

initialise the weights for each output unit
loop until weight changes are negligible
for each input pattern
present the input pattern
find the winning output unit
find all units in the neighbourhood of the winner
update the weight vectors for all those units
reduce the size of neighbourboods if required

The winning output unit is simply the unit with the weight vector that has the smallest Euclidean distance to the input pattern. The neighborhood of a unit is defined as all units within some distance of that unit on the map (not in weight space). In the demonstration below all the neighborhoods are square. If the size of the neighborhood is 1, then all units no more than 1 either horizontally or vertically from any unit fall within its neighborhood. The weights of every unit in the neighborhood of the winning unit (including the winning unit itself) are updated using

$$\vec{w}_i = \vec{w}_i + \alpha(\vec{x}_i - \vec{w}_i)$$

This will move each unit in the neighborhood closer to the input pattern. As time progresses the learning rate and the neighborhood size are reduced. If the parameters are well chosen the final network should capture the natural clusters in the input data.

Heiss and Dormanns propose to replace the input space of Kohonen networks with task interaction graph (TIG) and the output map with processor connection graph (PCG). To map the tasks with hosts, first tasks are transformed to feature vectors which contain all topological features of the tasks. In the second step, regular Kohonen process is applied to map the vectors (input) to a host (processor) in neuron map.

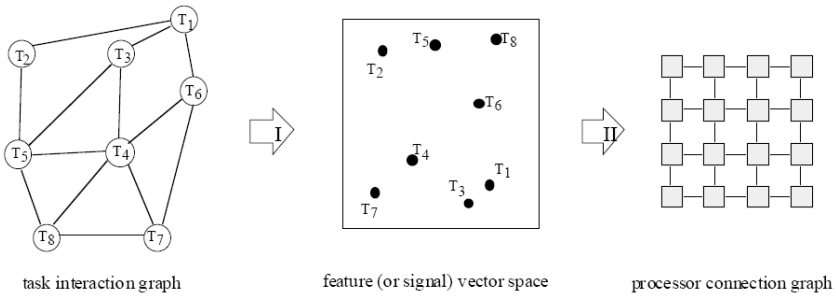


Figure 15 – Load Balancing using Kohonen Networks

The aim of Heiss and Dormanns' work is to find mapping between the tasks and processors which minimizes the communication overhead between tasks and load imbalance. The main disadvantage of their work is that it doesn't take into account the resource capabilities which can cause serious imbalances. The reason is that the tasks are mapped to processors randomly because of the random initial weights of the output neurons. To solve this problem, they implemented an external load balancing routine which is activated once per a determined number of steps after a globally ordered map is handled. This load balancing routine mainly changes the receptive fields of PCG nodes according to their load where changing the magnitude of a receptive field corresponds to transferring the loads between these receptive fields thus making tasks assigned to another processor(s).

In another work, Atun and Gürsoy [31,32] use Kohonen networks in another way to create a self learning system. They use Kohonen networks like Heiss and Dormanns do but in reverse order. They divide the input space into p regions each of which represents a processor (host) as shown in Figure 16. Each task to be assigned to a host is represented by a neuron in the output map. The weight vectors of these neurons represent the positions of the processors on the input space. A task i, is mapped to a processor Pij if the weight of the task i, Wi , is in the region of Pij. Figure 16 shows an input space with 4 processors where tasks are distributed over the S. The algorithm distributes the tasks to processors randomly at the beginning. Then weights of the tasks evolve to cover the range of input values (processor space) according to regular Kohonen learning algorithm.

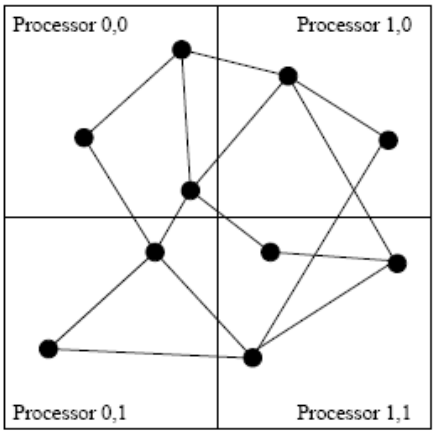


Figure 16 - Input and Output Spaces

Their algorithm chooses the tasks that are in the same neighborhood (that is that communicate with each other) so that they are processed on the processor minimizing the communication overhead. To balance the load of the tasks for each processor equally, least loaded processors must be selected for task assignment. By this way, the tasks will be shifted towards to least loaded processor, thus minimizing the load imbalance.

Atun and Gürsoy's work seems to be better than Heiss and Dormanns's work. Because, they begin by choosing the host rather than the task. Therefore if they enrich their algorithm with a powerful host selection mechanism, it will correctly move tasks from or to the selected host. However their algorithm has a disadvantage. They don't take into account the resource usage ratios of the tasks. That is, they don't take into account whether the tasks needs more processor power than memory. This can cause load imbalances in which the tasks that needs high processor power are assigned to hosts with slower CPU.

In another work produced by Abd and Bendary [38], they used winner-Take-All (WTA) neural network model for implementing the selection and location policies of a typical dynamic load balancing algorithm. All delays due to any usage of the communications network resource are taken into account. The WTA neural network employed has architecture as depicted in Figure 17. Each neuron represents a task of the waiting processes queue or the active processes queue. The winner neuron will denote the task with the largest difference between the estimated execution time and the inter-host communication cost. The reasoning behind this is the heuristic that the task with the highest execution time is more likely to be worth the transfer overhead, and that the task with minimum inter-host communication requirements would yield a least cost transfer.

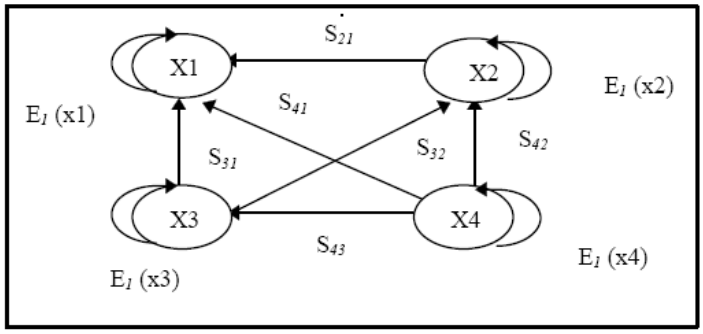


Figure 17 - Selection Policy

The WTA neural network used for implementing the location policy has architecture as shown in Figure 18. Each neuron represents a host whose active queue is not at maximum capacity. This ensures that a transferred task will receive immediate service at the receiver host. The winner neuron identifies the candidate host to which the combined cost of execution and communication is minimum. It should be noted that the winner neuron may be the one representing the local host, in such case the job will be executed locally in spite of the fact that there are other lightly loaded nodes in terms of the queue length alone. The worst case communication cost between tasks is incorporated so that a selected task will either be transferred to a host along the worst communication link or will be placed on a host to which less communication cost exists.

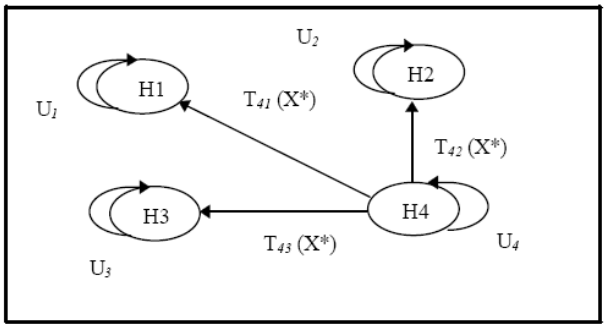


Figure 18 - Location policy neural network at host H4 and task X* is candidate for transfer

CHAPTER 3

RESOURCE AWARE LOAD BALANCING USING ANNS

In this chapter, the details of our proposed model to address the problems of load balancing algorithms are described. RALBANN brings together the successful partitioning results of graph partitioning algorithms with the efficiency and learning capability of artificial neural networks in a resource aware scheme. RALBANN maps the task partitions produced with the capabilities of the computing resources producing a resource aware load balancing scheme. RALBANN has the following features:

- RALBANN learns load balancing using artificial neural networks. In the runtime environment, RALBANN can monitor the system, train itself and do better load balancing. Automated learning capability makes RALBANN to adapt any changes in the distributed system.
- RALBANN models and encapsulates the distributed system as an undirected graph. Namely, it turns the information coming from the monitoring tool of the distributed into a generic graph. RALBANN can easily be integrated to any parallel application development environment. The only thing needed is a driver that turns monitoring information of the specific environment into a specific XML, which will then be turned to our generic graph model. In this study the P-GRADE environment together with its monitoring module is selected for testing RALBANN.
- During training, a multilevel partitioner module supervises the neural networks by providing the necessary input/output pairs for training. The integration of multilevel partitioner and the rest of RALBANN are accomplished us-

ing a generic interface. Therefore other partitioners can replace multilevel partitioner easily.

- It monitors the distributed environment using the monitoring tool and adapts to the system changes very quickly. Once trained and provided the necessary system information, RALBANN can balance the system load faster than any graph partitioning based load balancer and better than other load balancing algorithms.
- It is fully integrated into P-GRADE parallel application development environment replacing its default load balancer. The only thing needed to integrate to any version of P-GRADE is the execution of a batch file which replaces current load balancer of P-GRADE.

In the following sections, the architecture of RALBANN will be described in detail. Then P-GRADE will be introduced. The details of integrating RALBANN with other graph partitioners and distributed systems are described in the following sections.

3.1. Distributed System Models Used in the Thesis

The abstract distributed system model that is used in this thesis is illustrated in Figure 19.

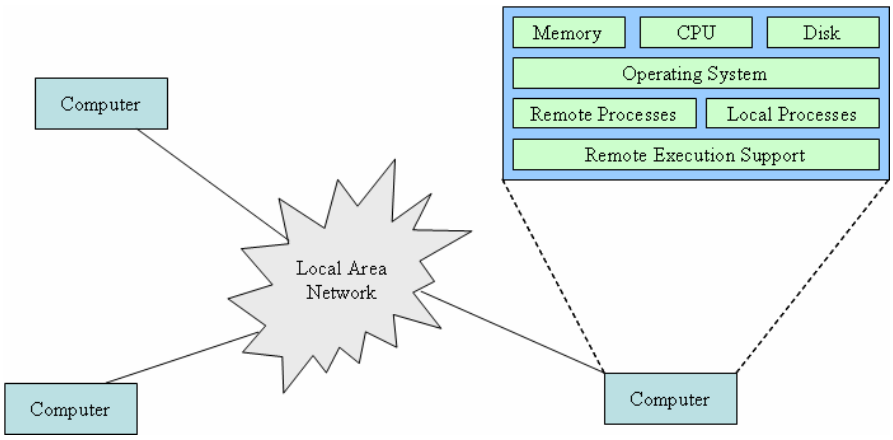


Figure 19 - Abstract Distributed Model

All of the hosts in our model have its own CPU and memory; and all hosts communicate with each other. They have the same architecture. However they have different configuration. That is, each has different memory capacity and CPU power. Although some resources, such as network and secondary storage, can be shared transparently; others, such as processing power and virtual memory, can be accessed by local tasks only. Therefore these resources are shared using a remote execution support, such as PVM. Since a host can have its own local tasks beyond the running computational task, the local resources of this host can be in use partially or completely. This demand of use on local resources is called *external load*. The absolute and relative use of resources at each host can be very dynamic. The dynamic nature of external load causes frequent: certain resources local to a host may be overloaded even similar resources at a remote host are underutilized or idle. With increases in the speeds of individual processors, and with growth in the scale of typical systems, there are increases in both the magnitude and the frequency of load imbalances. This is one of the main reasons why dynamic load balancing is more popular.

Our model assumes that tasks are dependent, which means each task may communicate with any of the other tasks. Further tasks may originate at any host of the distributed system. Further, the existence of an external load is assumed, which varies outside the control of the load balancing module. No prior knowledge of the behavior of the tasks to be balanced and external load imbalances is assumed. Further, the lack of knowledge about task length, along with the aforementioned difficulty of long-term prediction, necessitates preemptive strategies, which can ‘‘undo’’ the effects of poor initial placements. Therefore, the focus of this thesis is on dynamic, centralized, preemptive load-balancing strategies (Figure 2).

Another model that is used in this thesis for distributed systems is the graph model [15]. In this model, distributed system is modeled using graphs. Modeling distributed systems using graphs lets several problems of distributed systems to be reduced to useful graph algorithms. Once the problem is solved on the graph, the solution can be applied the distributed system directly.

In this model, each node of the graph represents the tasks and the edges represent the data communication between the tasks. The weights of nodes represent the CPU load of the tasks and the weights of the edges represent the data communicated between the tasks. This graph is called Task Interaction Graph (TIG).

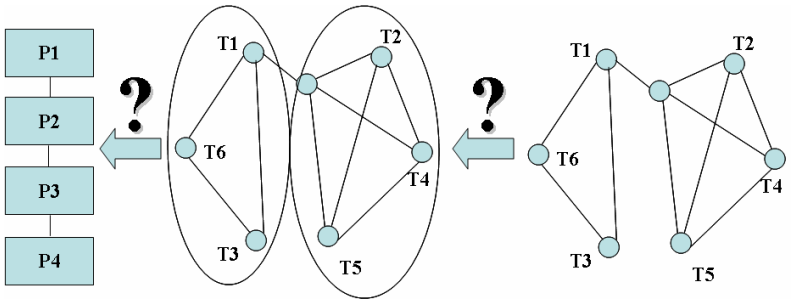


Figure 20 - TIG, graph partitioning and load balancing

Based on the described model, the basic load balancing problem can be described as the assignment of a group of tasks to a suitable host so that each host has nearly equal load. The tasks that highly communicate with each other must be grouped together so that the communication between the groups is minimized. Describing in another way; the problem of grouping the nodes together can be reduced to graph partitioning problem, which produces partitions with minimum edge cut.

3.2. Operation and Architecture of RALBANN

RALBANN creates the graph model of the distributed application and the underlying network using the information retrieved from the monitoring tool. The monitoring tool is periodically polled to collect data about the distributed system and the application that is running on this system. However, in order this data to be used in RALBANN, a conversion from the environment specific trace file format to RALBANN specific XML file format must be accomplished. This conversion is done by a C++ driver class that has a well-defined interface. The usage of such a driver concept makes RALBANN easy to integrate with another distributed system easier. The only

thing to do is to code the driver class of distributed environment and put compile RALBANN with the driver codes.

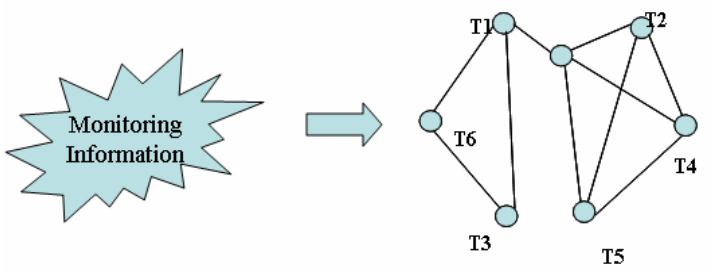


Figure 21 - RALBANN converts the monitoring information to graph model

The graph created contains information about the tasks and their communication. Each node of the graph represents a task that is assigned to a host. Each edge represents the data dependence between two vertices. The weight of the node represents the CPU load of the task and the weight of the edge represents communication load of the two tasks. This graph is called Task Interaction Graph (TIG). Actually RALBANN contains a modified version of TIG. It is modified in a way to store information about the hosts, too (Figure 22). The information stored for hosts is external load, total load and CPU power of the hosts. This information is used to map the partitions to the hosts considering which resources are more suitable for which partitions. For example, the partitions which need considerable CPU power must be matched with hosts which have high external load or low CPU load. Therefore one of the aims of RALBANN is to distribute partitions to hosts in a resource aware scheme preventing bottlenecks and overloads.

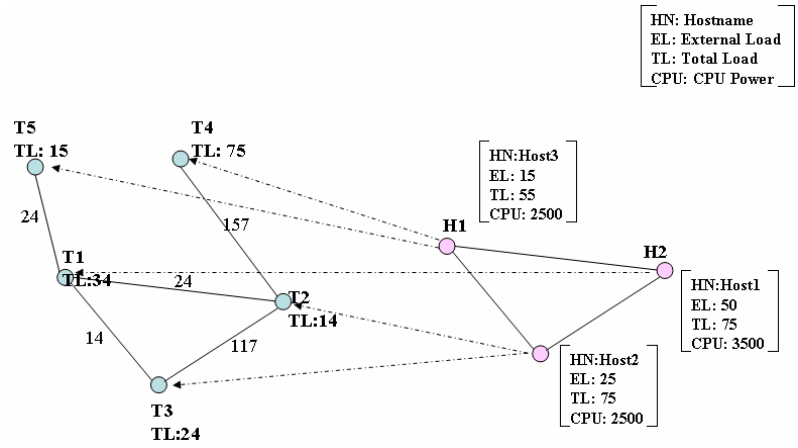


Figure 22 - Modified TIG that is used in RALBANN

After task interaction graph is created, it is partitioned by a graph partitioning module so that intercommunication between the partitions is minimized and the total computational load of the tasks in each partition is nearly equal. This module uses multilevel graph partitioning algorithms because of its efficiency and quality among other graph partitioning algorithms.

After TIG is partitioned, the TIG graph and partitioning information is encoded into a vector. The vector is optimized and is used to train an artificial neural network whose task is to partition the graph. The output of this ANN (the partitions) is given to another artificial neural network whose job is to learn how to map the partitions to the hosts. The partitions are mapped to the hosts by taking the computational load of the partitions, host external load, host CPU power and memory size into consideration.

As described above, the load balancing task is divided into 2 steps in RALBANN. This division makes the training of ANNs easier by shortening the training time by reducing the complexity of the inputs. Another benefit is that the weights of the host ANN are stored and loaded independently from the weights of the task partitioner ANN, making training more efficient. The reason of this efficiency is as following: RALBANN can be trained to learn different applications on a specific distributed system. Further RALBANN can save the weights and some parameters of the parti-

tioner and mapper ANNs into text files. These text files can later be loaded if RALBANN detects that the application is learnt before. When RALBANN is run, it automatically loads (if it exists) the weight file for the running distributed application. Especially, the weight file of mapper ANN is important. The reason is that only mapper needs to concern about the properties of the hosts. The existence of this weight file means that RALBANN has been trained with this distributed system before and has knowledge about how a given partition is mapped to one of the hosts. When a totally new application is run on this distributed system, the mapper ANN may not be trained again, if it was trained enough before. In this case, it is enough to train only the partitioner ANN which must know about the application. This partial loading mechanism provides a very efficient training time for RALBANN.

Figure 23 shows the architecture of RALBANN. It is mainly comprised of the following components:

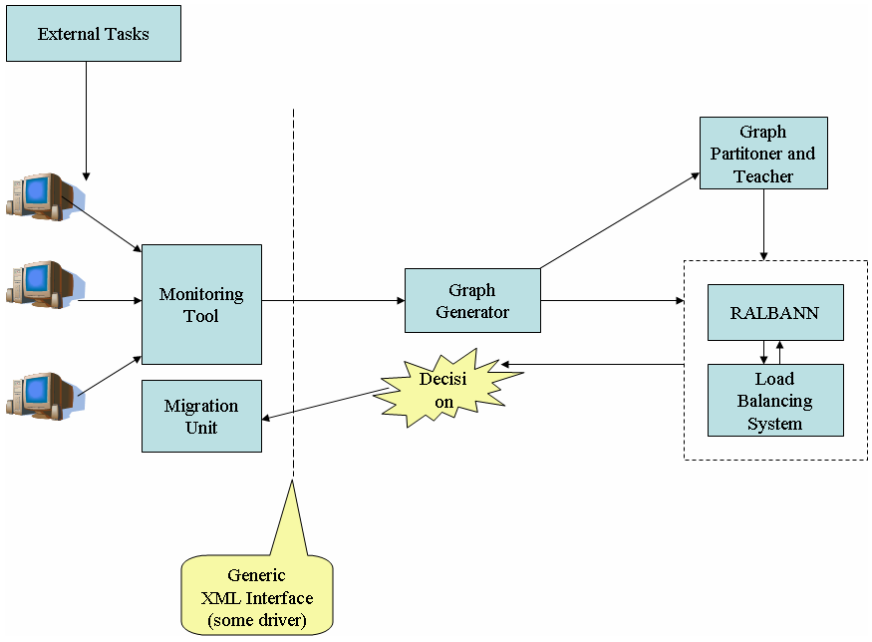


Figure 23 - Architecture of RALBANN

RALBANN (main module): This module is the main module containing the load balancing decision making and learning mechanism of RALBANN. The decision making process is accomplished using 2 artificial neural networks. These artificial neural networks are responsible for learning the 2 different stages of the load balancing:

1. Partitioning TIG graph to minimize the communication load of the tasks (the partitioner ANN)
2. Mapping the partitions to the hosts to minimize average execution time of the application (the mapper ANN)

The ANNs learn these 2 steps using the input data provided by the Teacher module. The graphs, which are generated by Graph Generator module, are encoded to 2 vectors by the Teacher. The first ANN, called the partitioner ANN, is trained to learn how to partition the task interaction graph and therefore it is trained with the task communication data.

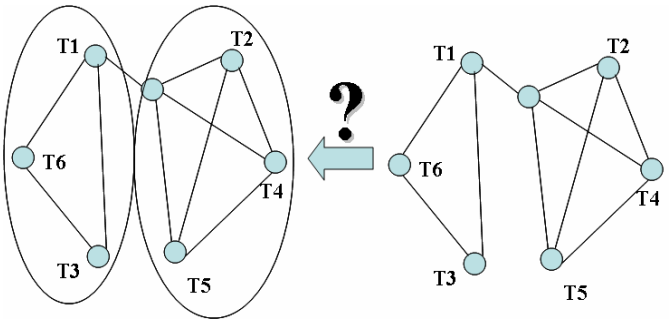


Figure 24 - The task of the partitioner ANN

The second one, called the mapper ANN, is trained to learn how to map partitions to hosts. Therefore it is trained with total processor usage of each partition and the processor power of each host. Given this data, it tries to find the best host for each partition.

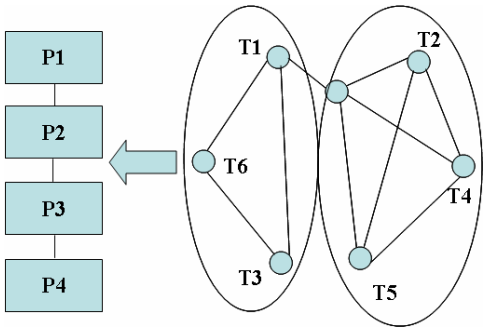


Figure 25 - The task of the mapper ANN

The aim of this separation is to make the job of ANNs easier by simplifying the input vectors. Another benefit of this separation is that the mapper ANN doesn't need to be trained again and again for different distributed applications that are run on the same distributed network. As its main responsibility is to learn the network (hardware properties of hosts) specific data, once it is trained enough, its weights are saved to a file and loaded again at the initialization phase of the RALBANN. For different distributed applications, it is enough to train only the partitioner ANN. For different distribute application environments, both the mapper and partitioner graph must be trained.

Graph Generator: This module is responsible for generating graph model of the distributed network. The graph is a kind of task interaction graph (TIG) and created from a well-defined XML file which is in turn generated by the XML trace file conversion driver.

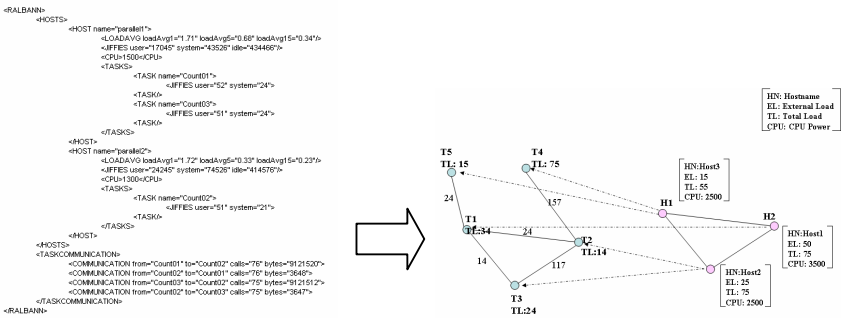


Figure 26 - Conversion XML to TIG

The graph generated contains information about:

- Message communication patterns of tasks
- Processor usage of tasks
- Processor power of hosts

Generic XML Interface: This interface is used to convert the format of monitoring information coming from the monitoring tool of the distributed environment to a well-defined XML format. The reason for this conversion is to make graph generator module generic as much as possible. This provides RALBANN to work with other distributed computing environment with minimal modification. Actually, the only thing needed to adapt RALBANN to other environments is to code the necessary class that converts the specific monitoring information to well-defined XML interface.

Graph Partitioner (Teacher): This module is responsible for providing the necessary input to train the artificial neural networks. Its main task is to partition the TIG graph and produce the 2 input vectors per TIG graph which is then used to train ANNs. This module performs its work in 2 steps. In the first step, it partitions the TIG graph into different partitions using graph partitioning algorithms. Then it converts the partitioning information into a vector encoding the partitioning related information, which is used to train the partitioner ANN in RALBANN main module. In the second step, it maps the partitions produced to the hosts in a resource aware

scheme. To train the mapper ANN, it creates the second vector encoding the CPU and memory loads of the partitions and CPU power and memory size hosts.

RALBANN's operation can be broken into phases of information collection, graph generation, neural mapping, learning and decision making.

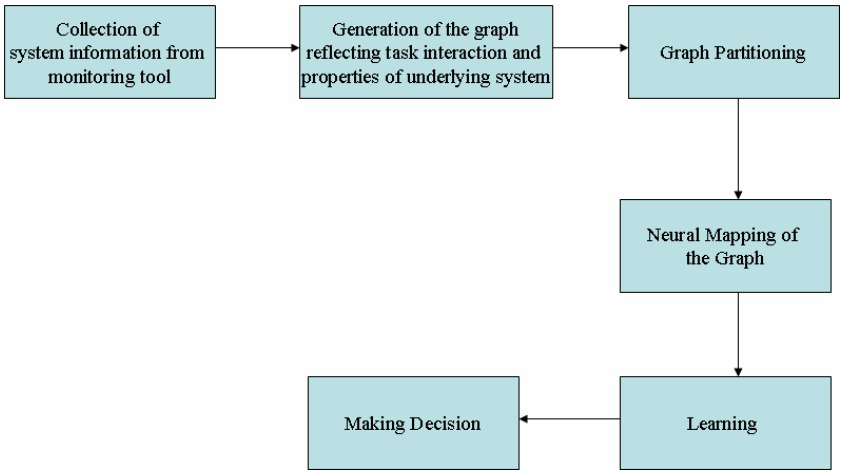


Figure 27 - Operation of RALBANN

3.2.1. Information Collection

RALBANN collects the following monitoring information from the distributed environment:

- communication pattern of the tasks
- computational loads of the tasks (processor loads)
- computational power of the hosts (processor power)
- external loads of hosts

The tools that are used collect load balancing information form the environment is called monitoring tool. RALBANN must connect such a tool to collect the necessary information. To make the usage of RALBANN with all distributed environments easier, the interface of the graph generator is fixed to a well-defined XML format.

The data sent by monitoring tools must be converted this XML format. Our purpose for using XML is to make it easier to integrate with distributed platforms other than P-GRADE. The XML generated must conform to the following DTD:

```
<?xml version="1.0"?>
<!DOCTYPE note [

<!ELEMENT RALBANN (HOSTS,TASKCOMMUNICATION)>
<!ELEMENT HOSTS (HOST*)>
<!ELEMENT TASKCOMMUNICATION (COMMUNICATION*)>
<!ELEMENT COMMUNICATION EMPTY>
<!ELEMENT HOST (LOADAVG,JIFFIES,CPU,TASKS)>
<!ELEMENT LOADAVG EMPTY>
<!ELEMENT JIFFIES EMPTY>
<!ELEMENT CPU (#PCDATA)>
<!ELEMENT TASKS (TASK*)>
<!ELEMENT TASK (TASK_JIFFIES)>
<!ELEMENT TASK_JIFFIES EMPTY>

<!ATTLIST COMMUNICATION from CDATA #REQUIRED
                        to CDATA #REQUIRED
                        calls CDATA #REQUIRED
                        bytes CDATA #REQUIRED>
<!ATTLIST HOST name CDATA #REQUIRED>
<!ATTLIST LOADAVG loadAvg1 CDATA #REQUIRED
                  loadAvg5 CDATA #REQUIRED
                  loadAvg15 CDATA #REQUIRED>
<!ATTLIST JIFFIES user CDATA #REQUIRED
                  system CDATA #REQUIRED
                  idle CDATA #REQUIRED>
<!ATTLIST TASK name CDATA #REQUIRED>
<!ATTLIST TASK_JIFFIES user CDATA #REQUIRED
                       system CDATA #REQUIRED>
]>
```

Figure 28 - RALBANN XML DTD

Once collected, this information is used to generate the graph model which will then be used for both training and decision making.

```
<RALBANN>
        <HOSTS>
                <HOST name="parallel1">
                        <LOADAVG loadAvg1="1.71" loadAvg5="0.68" loadAvg15="0.34"/>
                        <JIFFIES user="17045" system="43526" idle="434466"/>
                        <CPU>1500</CPU>
                        <TASKS>
                                <TASK name="Count01">
                                        <TASK_JIFFIES user="52" system="24"/>
                                <TASK/>
                                <TASK name="Count03">
                                        <TASK_JIFFIES user="51" system="24"/>
                                <TASK/>
                        </TASKS>
                </HOST>
                <HOST name="parallel2">
                        <LOADAVG loadAvg1="1.72" loadAvg5="0.33" loadAvg15="0.23"/>
                        <JIFFIES user="24245" system="74526" idle="414576"/>
                        <CPU>1300</CPU>
                        <TASKS>
                                <TASK name="Count02">
                                        <TASK_JIFFIES user="51" system="21"/>
                                <TASK/>
                        </TASKS>
                </HOST>
        </HOSTS>
        <TASKCOMMUNICATION>
                <COMMUNICATION from="Count01" to="Count02" calls="76" bytes="9121520"/>
                <COMMUNICATION from="Count02" to="Count01" calls="76" bytes="3648"/>
                <COMMUNICATION from="Count03" to="Count02" calls="75" bytes="9121512"/>
                <COMMUNICATION from="Count02" to="Count03" calls="75" bytes="3647"/>
        </TASKCOMMUNICATION>
</RALBANN>
```

Figure 29 - Example XML File

3.2.2. Graph Generation

The XML file created in *Information Collection* phase is converted to TIG graph in this phase. Four different types of data about the distributed application and the execution environment are monitored:

- For each host k, the total computational load of tasks and external load of the hosts can be calculated.
- For each task i, its CPU usage ($c_p(i)$): This type gives information about the load of each task.
- For each pair of processes (i; j), the number and total size (in bytes) of messages sent from i to j

In the created graph, each node represents a task that is assigned to a host. Each edge represents the data dependence between two vertices. The weight of the node represents the CPU load of the task and the weight of the edge represents communication load of the two tasks. RALBANN extends this graph to store information about the

hosts, too. The information stored for hosts is external load, total load and CPU power of the hosts.

The graph is represented with an instance LBGraph class. LBGraph is the class that provides the necessary functionalities to manipulate and create TIG graph. It contains an array of hosts, an array of processes and two dimensional array of messages.

Each node of the LBGraph is an instance of LBGraphProcess class. Each instance contains the CPU and memory load of the process and points to an array which contains the messages sent to/received from the other tasks. The so-called neighbor tasks can be derived from this structure. The messages are stored as instances of LBGraphMessage class. This class contains the message count and message size in bytes. The message array provides the communication information patterns of the tasks.

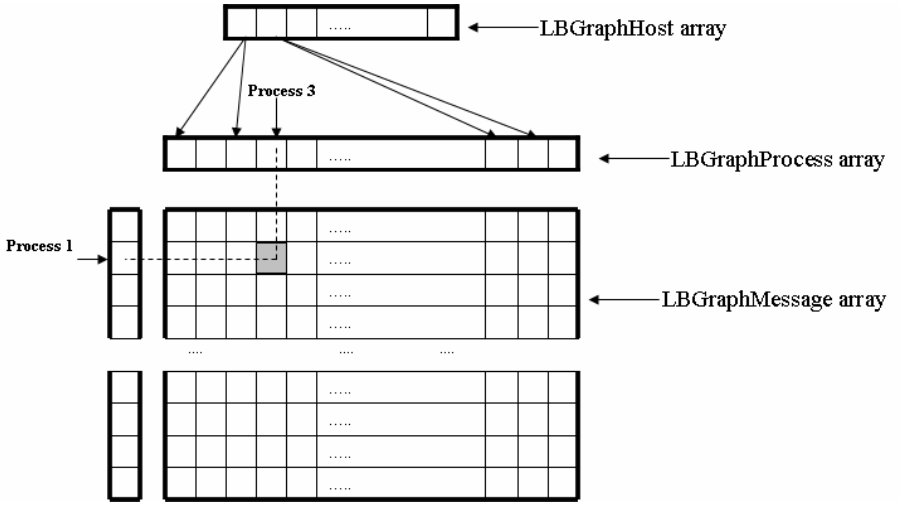


Figure 30 - LBGraph structure

3.2.3. Graph Partitioner (Teacher)

In this phase, the graph created in "Information Collection" phase is partitioned. RALBANN uses graph partitioning algorithms to train the ANNs how to do load

balancing. The reason for using graph partitioning algorithms is that they produce the highest quality partitioning results over all other load balancing algorithms. However they have one big disadvantage: inefficient usage of memory and processor. In this thesis, multilevel graph partitioning algorithm is chosen because of its better partitioning quality and efficiency.

Multilevel partitioning algorithms are analogous to multigrid methods [9,39-41]. These methods construct a coarsened version of the original problem. The solution is generated on the coarsened problem. Then this solution is refined until the original problem is found. In graph partitioning context, this can be explained as creating a simplified (lesser edges and vertices) graph, partition it and then propagate the partitioned graph to the original graph by refining at each level. This technique is efficient because partitioning smaller graph is easy and produces good results if the refinement is done with a good algorithm. Multilevel partitioning algorithm has 3 phases [17,19-21]:

1. Coarsening Phase: Given the input graph G, construct a series of increasingly smaller graphs G_1, G_2 ,..., G_m each of which retains some sense of G's global structure.
2. Partitioning Phase: Partition the coarsest graph, G_m, using a standard algorithm.
3. Uncoarsening Phase: Propagate the solution for G_m up to the finer graphs, refining it at each level.

Coarsening Phase:

Coarsening phase is comprised of several phases. At each phase, RALBANN coarsens the graph to smaller graphs by combining some vertices into single vertices called multinodes. The new graph G_{i+1} is constructed from G_i by combining a set of vertices V_i^v to create vertex v in G_{i+1}. The set of vertices are selected according to *matching* concept. A matching of a graph is defined as the set of edges in which no two edges are incident on the same vertex. That is, a matching contains edge pairs none of which are incident. The next level coarser graph G_{i+1} is constructed from G_i by combining the edges in the matching of G_i. The unmatched vertices are simply

copied over to G_{i+1}. Since the goal of collapsing vertices using matchings is to decrease the size of the graph G_i, the matching should contain a large number of edges. For this reason, maximal matchings are used to obtain the successively coarse graphs. A matching is maximal if any edge in the graph that is not in the matching has at least one of its endpoints matched.

Here the important thing which determines the vertices in the partitions that will be created at the end is the how the matchings are determined at each level. In RALBANN, 2 important conditions are applied while constructing the matchings. The aim of both conditions is to create task partitions having approximately equal computational load and containing highly communicating the tasks:

1. The vertex with the smallest load is selected as the initial vertex for each pair of edges in the matching. The aim is to combine the vertices with small loads with other vertices to increase the possibility of constructing more balanced partitions in the partitioning phase.
2. The total edge-weight of the coarser graph is reduced by the weight of the matching. Hence, by selecting a maximal matching whose edges have a large weight, the edge-weight of the coarser graph can be decreased by a greater amount. Since the coarser graph has smaller edge-weight, it also has a smaller edge-cut. Therefore the matching vertex with the highest communication size (edge weight) is selected. The aim of this selection is to bring together the vertices which highly communicate with each other by minimizing the edge-cut.

In figure, T1 and T6 are combined to form new vertex T1:T6. The CPU loads and memory loads of T1 and T6 are summed up in T1:T6. Then

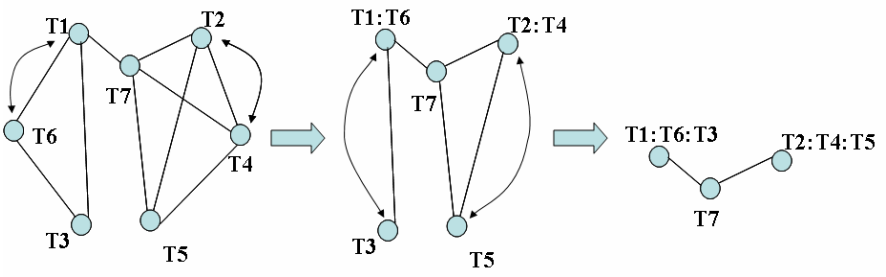


Figure 31 - Coarsening Phase

Partitioning Phase:

Coarsening phase continues until no more matches are possible or the weights of all remaining edges are under the average edge-weight size. The average edge-size is a quantity which is calculated as the arithmetic mean of the sizes of all messages. RALBANN continues coarsening the graph only if there exist edges whose weights are above the average message size. This makes it possible to discard the messages (edges) with low weights and helps to detect highly communicating partitions.

Figure 32 - Partitioning Phase

Uncoarsening Phase:

During the uncoarsening phase, RALBANN projects the partition P_m of the coarser graph G_m back to the original graph, by going through the graphs G_{m-1}, G_{m-2},…, G_1. Since each vertex of G_{i+1} contains a distinct subset of vertices of G_i, obtaining P_i from P_{i+1} is done by simply assigning the set of vertices V_i^v collapsed to $v \in G_{i+1}$ to the partition $P_{i+1}[v]$.

Even though P_{i+1} is a local minimum partition of G_{i+1}, the projected partition P_i may not be at a local minimum with respect to G_i. Since G_i is finer, it has more degrees of freedom that can be used to improve P_i, and decrease the edge-cut. Hence, it may still be possible to improve the projected partition of G_{i-1} by local refinement heuristics. For this reason, after projecting a partition, a partition refinement algorithm is used. The basic purpose of a partition refinement algorithm is to select two subsets of vertices, one from each part such that when swapped the resulting partition has a smaller edge-cut.

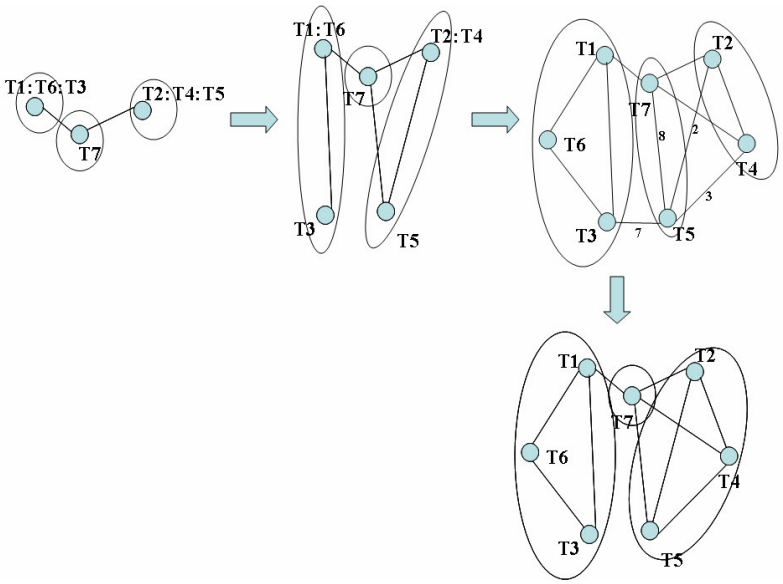


Figure 33 - Refinement Phase

To refine the partitions, *Kernighan-Lin* Heuristic [18] is used in RALBANN. The Kernighan-Lin (KL) algorithm is based on the gain concept, which is used to show the benefit of moving a vertex from one subset to the other. In KL, a vertex's gain is simply the total edge weight connecting it to the other subset minus that which connects it to its own, which is the reduction on the edge-cut:

$$g_v = \sum_{(v,u)\in E \wedge P[v]\neq P[u]} w(v,u) - \sum_{(v,u)\in E \wedge P[v]=P[u]} w(v,u)$$

Figure 34 - Gain formula in KL algorithm

KL algorithm repeatedly selects a vertex and calculates its gain. If gain is positive, then by moving v to the other partition the edge-cut decreases by gain; whereas if gain is negative, the edge-cut increases by the same amount.

For each vertex v for graph G_i, the neighborhood of the v is defined as N(v), the union of partitions that the vertices adjacent to v belongs to: $N(v) = U_{u\in adj(v)}P_i[u]$, where P_i is the partitioning vector of G_i. During refinement, v can move to any of the partitions in N(v). For each vertex v, the gain of moving v to one of its neighbor partitions is computed. In particular, for every $b\in N(v)$, $ED[v]_b$ is computed as the sum of the weights of the edges (v, u) such that $P_i[u]=b$. $ED[v]_b$ is called *external degree* of vertex v for partition b. Then ID[v] is computed as the sum of the weights of the edges (v, u) such that $P_i[u]=P_i[v]$. ID[v] is called *internal degree* of vertex v. The gain of moving vertex v to partition b is defined as $ED[v]_b$ - ID[v]. Negative gain means, if this vertex is moved to the partition b, then edge-cut will be increased and since the aim is to decrease it, the vertex is not moved. If the gain is positive, the vertex can be moved provided that it doesn't create an unbalanced partition.

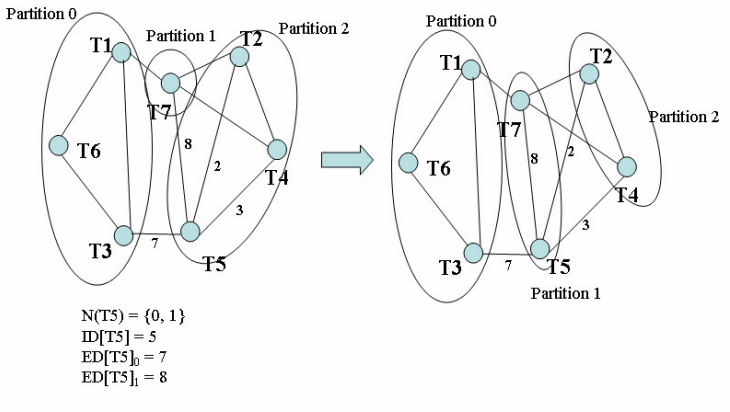


Figure 35 - Refinement of Task 2 using KL. T2 is moved from one partition to another neighbor partition (this movement provides better edge cut)

Therefore before moving the vertex, the following conditions must be checked:

1. $W_i[b] + w(v) <= W_{max}$
2. $W_i[a] - w(v) >= W_{min}$

where:

- Wi[b] is the total weight of the vertices in partition b for graph G_i containing k partitions,
- w(v) is the weight of vertex v,
- $W_{max} = A*|W_{all}|/k$ for A>=1.0,
- $W_{min} = B*|W_{all}|/k$ for B<=0.9,
- W_{all} is the total weight of all vertices.

The first condition ensures that movement of vertex does not make the weight of the target partition higher than W_{max}. The second condition ensures the load of partitions is not too small. By adjusting the value of A and B, the imbalance of partitions can be varied. For RALBANN, the experiments show that B=0.2 and A = 3 produce good load balancing results.

After the partitions are created according to the rules above, the partitions are mapped to hosts according to their CPU power and external load. This mapping al-

lows RALBANN to place the partitions to hosts without causing any potential overloading or bottleneck problems:

- By taking external load into account, RALBANN can learn to make the total load (external load + load due to distributed application) of every hosts equal.
- By taking CPU power and memory size of the hosts in to account, RALBANN prevents any bottlenecks due to CPU overloading and insufficient memory.

The mapping is done according to the *load parameter* values of all hosts and partitions. The load parameter is a parameter that shows how good a host can deal with an upcoming partition of tasks. This parameter is calculated using the following formula:

$$L_i = H_i^{cpu}/(C*H_i^{ext} + \sum_{v \sim P} W(v))$$

where H_i^{cpu} is the CPU power of host i, H_i^{ext} is the external load of host i and W(v) is the cpu load of task which is represented by vertex v belonging to partition P in the graph model. C is the adjustment parameter which can be used to increase or decrease the effect of external load of the host. After the load parameter is calculated for all hosts for partition P, the host with the highest parameter value is chosen for partitions P. For the second partition, the load parameter of the host with which the previous partition is mapped, is calculated again. This continues until all partitions are mapped one host.

3.2.4. Artificial Neural Networks and Neural Mapping

After partitions are produced and mapped to hosts by the graph partitioning (teacher) module, the next step is the training of ANNs to learn the task of load partitioning task. To accomplish this, the graph model and task partitions are converted to vectors as inputs to artificial neural networks in such way that one can decipher a solution for balancing the loads from outputs of the neurons. Neural networks are generally used as a model of the computation for solving a variety of problems in fields like computer vision, pattern recognition. In RALBANN, another field namely load balancing

and graph partitioning is aimed as a computational model for ANNs. Therefore special attention is paid on artificial neural networks and encoding the graph partitioning and load balancing task into vectors, which are then used to train ANNs.

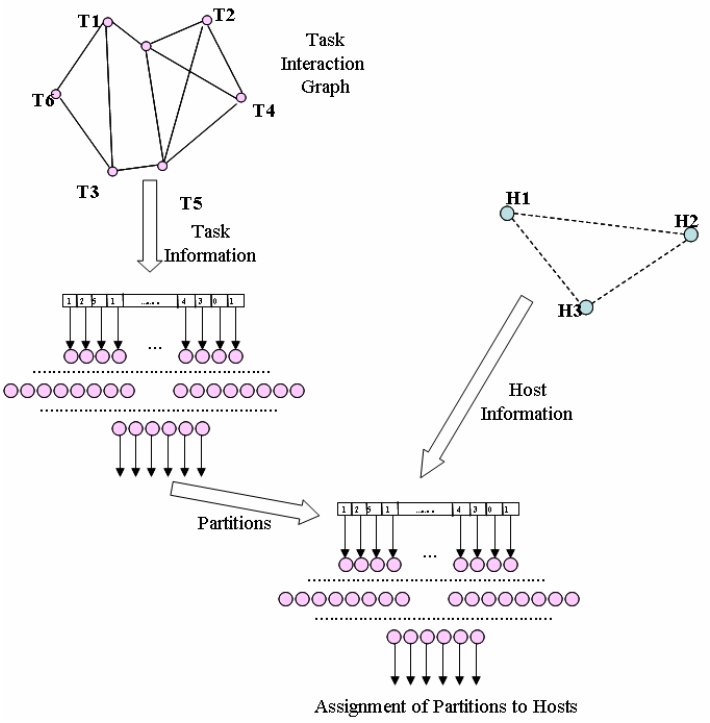


Figure 36 - Neural Mapping

The partitioner ANN is given only the task communication and CPU load information. Therefore the encoded vector contains (CPU load) + (message sizes with other tasks) information for each task in its input part. In the output part, the partitioning data which was produced by graph partitioning module is encoded. There exists one integer for each task, which represents the partition number to which the task is assigned. Using this training data, the partitioner ANN is trained to learn how to partition the vertices of the graph whose topological structure is encoded in the input part. Since in RALBANN the graph is partitioned by graph partitioner module using multilevel algorithms, (if trained enough) the partitioner module imitates multilevel partitioning algorithm in the way implemented.

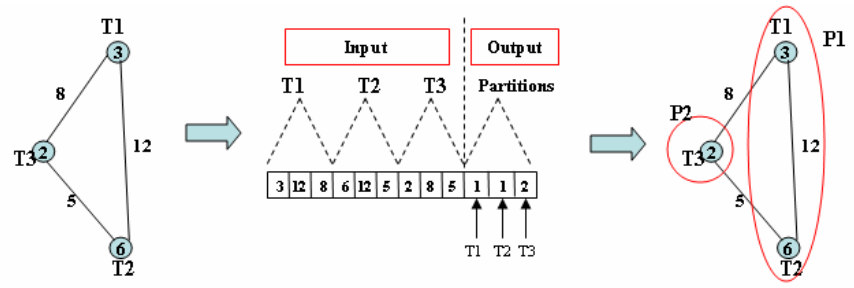


Figure 37 - Input vector for partitioner ANN

The mapper ANN, on the other hand, is given as input the computational load and power of the hosts and computational loads of the partitions produced. The computational loads of the partitions are the summation of the computational loads of the individual tasks that belongs to the partition.

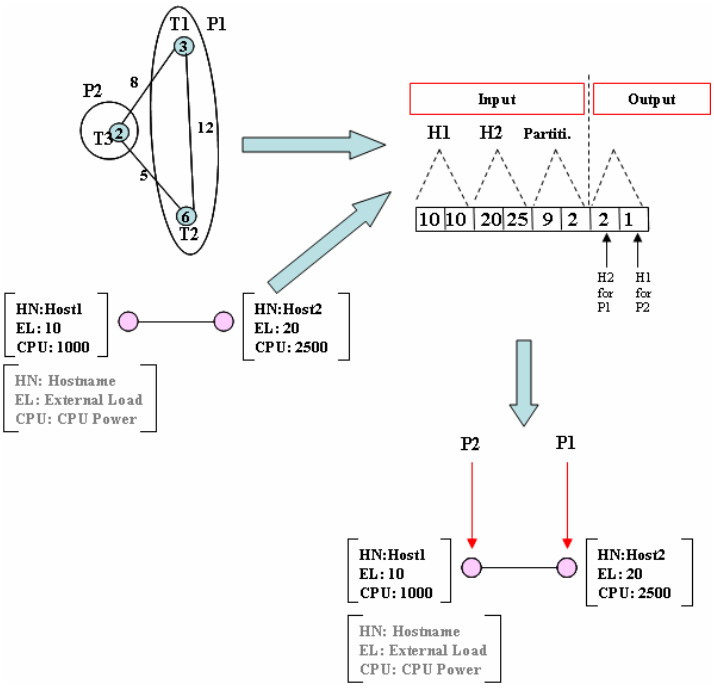


Figure 38 - Input vector for mapper ANN

RALBANN can work both in offline and online mode. In online mode, the data coming from distributed environment is directly converted to graph model, partitioned, converted to vectors and send to artificial neural networks for training. On the other hand, in offline mode RALBANN can train itself by creating random graphs using random graph generator module. This module is responsible for generating random graphs for a given number of hosts and number of tasks. An example usage of RALBANN with offline learning is "ralbann -offline 2 5". With this call, RALBANN creates random graphs containing 2 hosts and 5 tasks with random communication and computation loads, convert them to vectors and train neural networks. Later, the trained networks can be used to balance a network with 2 hosts and 5 tasks.

After the number of vectors created reaches a predefined number, they are optimized so that the neural networks converge and learn better and the learning phase begins which continues until the finish criteria is accomplished. The finish criteria can be one of the followings:

- RALBANN predicts the partitions of at least, for example, 80% of the graphs given.
- The network error of the neural network is less than some predefined error constant.

In RALBANN, multilayered feed forward artificial network architecture is used. There are a number of reasons for using feed forward networks to implement learning of load balancing task. The most important is the following: When the output of each unit of a feed-forward network is given by the sigmoidal function of its net input, then a network having the feed forward architecture can approximate any function with arbitrary accuracy, provided that the network has sufficiently many units in the hidden layer. In RALBANN, the number of hidden layers is 2, since it provided more accurate results.

A second important reason for using feed-forward neural network is the availability and success of supervised learning algorithms; given the actual and desired outputs

for a feed-forward neural network, these methods can determine the appropriate modifications for the weights of that network and make them converge to the desired results. The most popular learning algorithm which is also used in RALBANN is the back-propagation algorithm.

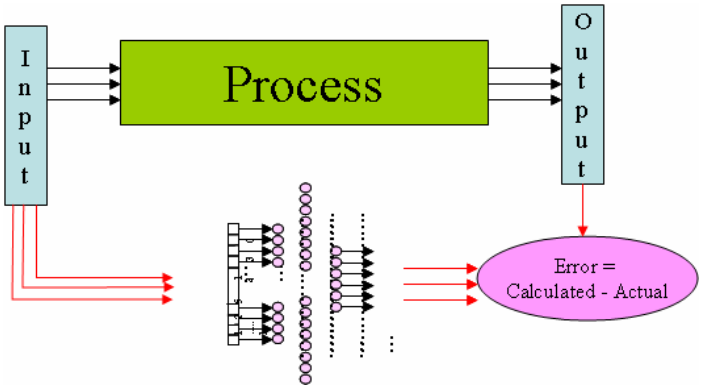


Figure 39 - Training of ANN

Back-propagation algorithm consists of 2 phases. In the first phase, using the input, the erroneous output is the calculated. In the second phase, the weights are adjusted using the error between predicted outputs and actual outputs. The purpose is to tune the network to more accurately predict the input in the future. Similar to the way for progressing forward through the neural network layer by layer to calculate the predicted output, the weights are adjusted by running backwards through the network layer by layer. To do this, first the error term for each neuron is calculated as: $\delta_k = o_k(1 - o_k)(y_k - o_k)$ (o_k is the calculated output and y_k is the actual output) for output neurons and $\delta_j = o_j(1 - o_j)\Sigma_k w_{jk}\delta_k$ for hidden neurons. The new value of each weight w_{jk} of the connection from neuron j to neuron k is given by: $w^{new}_{jk} = w^{old}j_k + \eta o_j \delta_k$.

η is an important tuning parameter that is chosen by trial and error by repeated runs on the training data. Typical values for η are in the range 0.1 to 0.9. Low values give slow but steady learning; high values give erratic learning and may lead to an unstable network. For RALBANN, this value is set to 0.3.

Typically, the training data has to be scanned several times before the error is reasonably small. A single scan of all training data is called an epoch. The aim is to find the weights that minimize the error function of the output network which is calculated as:

$$\sum(o_k - y_k)^2 \text{ for each output neuron } k$$

However, it is not always possible to find the correct the optimum weights to minimize error. The algorithm can be caught at a local minima when the input contains very complicated data. Another weakness of back-propagation is the overfitting problem, causing the error rate to be small for only training data but very big for other data. In this case, the network can not do generalization. This is caused when the network is overtrained.

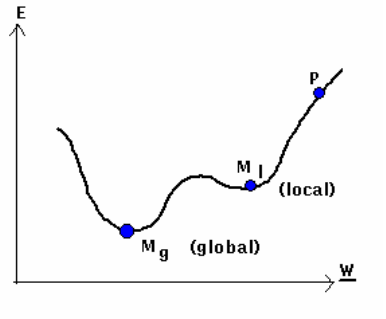


Figure 40 - Local minima

To prevent these situations, the following methods are applied in RALBANN:

- The order of training data is randomized before each epoch.
- The tuning parameter η is decreased as the number of epochs increases and the network error doesn't decrease. This is useful because it avoids overfitting that is more likely to occur at later epochs than earlier ones.

- A momentum term is used in the weight adjustment formula. This term causes the algorithm to jump over small minimas and find the correct global minima. This resembles to the affect of speed over the physical objects.

3.3. Integration with P-GRADE

3.3.1. P-GRADE

P-GRADE (Parallel Grid Run-Time and Application Development Environment) is a parallel application development environment for Grid, clusters and supercomputers that provides both run-time and application development environment [42,43]. It provides several facilities for the user to develop parallel applications quickly using the visualized components of P-GRADE using application development environment. The run-time system, on the other hand, provides the facilities necessary to run the developed application on Grid and dedicated resources, like load balancer and monitoring tool.

Application development environment is comprised of several graphical user interfaces:

- GRED: a graphical user interface environment to construct parallel applications and workflows
- a mapping tool to assign tasks of a parallel program to processors of a cluster or to assign components of a workflow to sites of a Grid
- DIWIDE: a distributed debugger GUI to animate, watch and control the execution of parallel programs on a desktop or cluster
- PROVE: a visualization tool to visualize program execution
- GRP2C: a compiler to generate C code with PVM or MPI library calls

In the run time system, P-GRADE uses Globus, Condor-G and MPICH-G2 as grid-aware middleware to execute applications. P-GRADE uses GRAPNEL language to create and manage workflows in Grid environment. GRAPNEL allows a programmer to set a workflow of objects/library calls. Special tool GRED is used to generate MPI code from a GRAPNEL program. P-GRADE workflow is used to compose jobs consisting of MPI (MPI-CH2), Condor, and executables. The workflow produces events

that can be visualized to show workflow history. The run time system provides the following facilities:

- a mapping facility to map processes or workflow components to resources
- DIWIDE: a distributed debugger to debug parallel and distributed applications
- GRM: a distributed monitoring infrastructure to collect run-time trace information on application execution both in desktop and dedicated cluster environments (In non-dedicated cluster or Grid environments a real Grid monitor support is needed. Such a Grid monitor is Mercury.)
- a checkpoint system to checkpoint PVM programs developed by P-GRADE
- a migration unit to automatically migrate processes of P-GRADE programs inside a cluster or among sites of a Grid
- a load-balancer unit to provide well balanced execution of the parallel program among the nodes of a cluster
- a workflow engine that controls the workflow execution and takes care of the necessary file transfers.

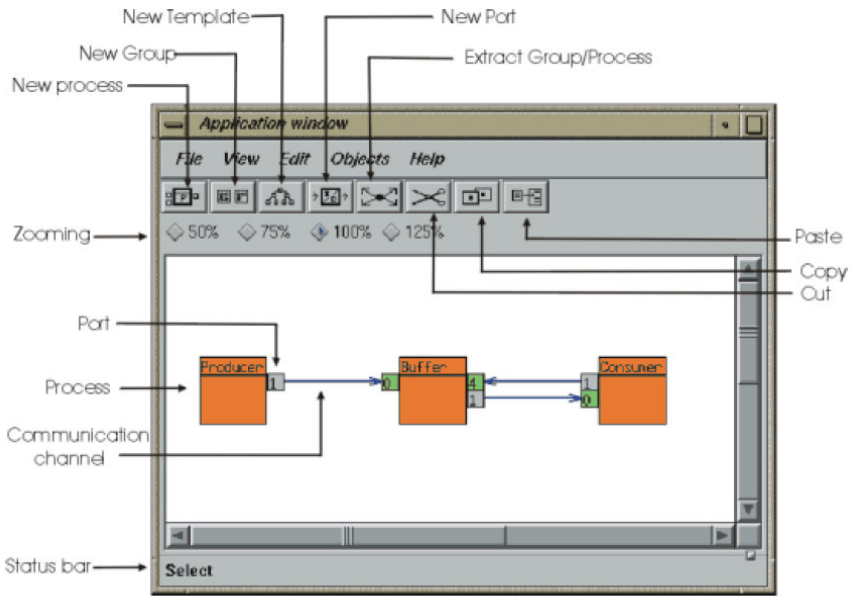


Figure 41 - P-GRADE Application window

The separation of P-GRADE Application Development Environment (GUI) and Run-Time system (RS) lets the user to separately install these components in different computers. Installation of RS on different computers gives rise to creation of different clusters on different computers. A cluster can be dedicated or non-dedicated. Dedicated cluster means that the whole cluster or several nodes of the cluster are dedicated to the execution of the parallel program. Non-dedicated cluster means that several users' application jobs can simultaneously run on the cluster and typically a local job manager (such as Condor, SGE, PBS, etc.) takes care of queuing and launching user jobs (Figure 42). The desktop on which the GUI is installed is used to edit, compile, submit and visualize the application. The actual execution is performed on the clusters. The user can debug and monitor the program in the cluster environment. The P-GRADE load-balancer can be used. The P-GRADE checkpoint and migration unit can be used to migrate processes of PVM programs (developed by P-GRADE) among the nodes of the cluster according to the decisions coming from the load balancer. The parallel program monitoring is done by GRM and execution visualization is provided by PROVE. Even the migration and load-balancing activities can be visualized using PROVE.

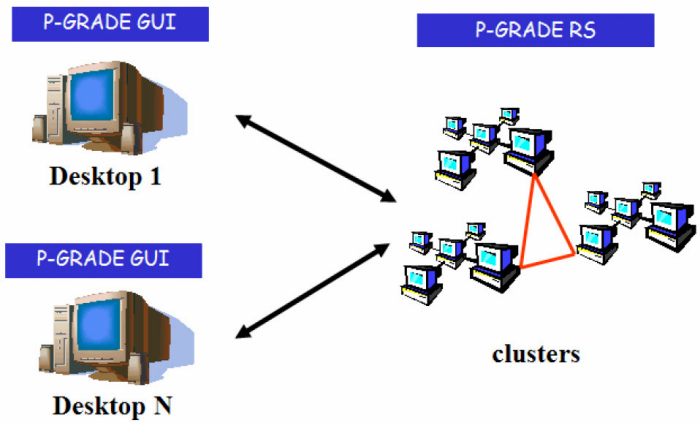


Figure 42 - General P-GRADE usage

P-GRADE includes a centralized load balancer unit. The load balancer continuously collects information about the system using the monitoring tool of P-GRADE called GRM. Based on the coming information, if LB concludes that there is a problem with the current mapping, it reorganizes the process-host mapping using the migration unit. This load balancer is centralized which means it runs on the desktop host and it uses Simulated Annealing algorithm.

3.3.2. Integration of RALBANN with P-GRADE

RALBANN is designed to run with any distributed environments since it is designed as a separate application with generic interfaces. Specifically, RALBANN is integrated and tested on P-GRADE environment. It is invoked by P-GRADE and collects the system information from GRM module. After finding an optimal mapping of processes and hosts, the new mapping is applied to the distributed system using the migration tool. The integration of RALBANN to any P-GRADE version is done straightforwardly by executing the following batch file provided in the context of our application for the sake of full integration:

```
#!/bin/sh
if [ -n "$1" ] && [ -n "$2" ];
then
sed 's/GRADE_LB_EXE="$SCRIPTDIR\/$SYSARCH\/lb $GRM_HOST $GRM_PORT $CHKPT_LB_PERIOD
$OBJDIR\/$APPLNAME.map"/GRADE_LB_EXE="'$1' $GRM_HOST $GRM_PORT $CHKPT_LB_PERIOD
$OBJDIR\/$APPLNAME.map $APPLNAME -learning"/' $2/bin/start-appl > /tmp/start-appl && mv -f /tmp/start-
appl $2/bin/start-appl
else
echo "Usage: patch.sh <LB_EXECUTABLE> <PGRADE_HOME>"
echo "CAUTION: Escape / characters in <LB_EXECUTABLE>"
fi
```

3.4. Usage with other Distributed Environments

In this section, the steps necessary to use RALBANN with other distributed environments other than P-GRADE are described. RALBANN is designed to run as a separate application. When it is run, it can periodically connect to the monitoring tool and collect information about the system. It is invoked in online mode using the following command:

```
ralbann HOSTNAME PORT LB_PERIOD APPLNAME -learning{nonlearning}
```

HOSTNAME and PORT: Hostname and port on which the monitoring tool running

LB_PERIOD: The period in seconds that RALBANN collects data and invokes its balancing procedure.

APPLNAME: Name of the application currently running in the distributed environment.

To integrate RALBANN with other distributed system, the only thing needed is a class that connects to remote monitoring tool, collects the monitoring information (trace data) and converts it to RALBANN's XML trace format. This class must inherit *DataMonitoring* abstract class and implement the following virtual functions:

- *DataMonitoring(char *hostname, int port)*: This is the constructor of the *DataMonitoring* interface. *hostname* is the hostname of the monitoring tool and *port* is the port number on which the monitoring tool is running.
- *virtual void connect():* This function must abstract the connection protocol of the monitoring tool that listens on the remote server. The socket number of the connection socket is reserved in a variable *m_sock*.
- *virtual char *readBlockData(int byteCount):* This function must abstract to read *byteCount* number of characters from the monitoring tool. It must implement the necessary protocol to read the data. If no more data is available it must return NULL.
- *void retrieveTraceData():* This function is already implemented and is used to read data from monitoring tool continuously.
- *virtual convertTraceDataToRALBANNXMLFormat(string filename):* This function is the most important function to be implemented. It converts the information collected from monitoring tool, which has an environment specific format to well-defined RALBANN XML format.
- *void migrate(string hostname, int pid):* This function abstracts the migration process. It contacts the migration tool of the distributed environment to migrate a process to another host.

3.5. Usage with other Graph Partitioners

RALBANN can run with any graph partitioning algorithm provided that the graph partitioner module inherits *IGraphPartitioner* interface and implements its only function:

- *int *getPartitions(LBGraph *graph):* This function gets a pointer to an LBGraph object and returns an array containing the partition id of each task in the graph.

CHAPTER 4

EXPERIMENTAL STUDY

To evaluate the load balancing results of RALBANN, a test platform using random graph generator module is prepared. This random graph generator generates random graphs with various task and host configurations as if they are generated from real distributed systems. The graphs are converted to input vectors, and these vectors are used to train the artificial neural networks.

The benefit of using ANNs reaches to the peak when the network is trained enough, since in this case, RALBANN can provide the most accurate results with a very efficient CPU and memory usage. In online training mode, RALBANN can both adapt the system and do load balancing. Therefore the using RALBANN in this mode may create some overhead. In a case where RALBANN's overhead exceeds its load balancing advantages, RALBANN can be trained in offline mode using random graph generator. In offline mode, prior to real time execution, RALBANN is trained with several random graphs which resemble the real application's graph structure. In this case, RALBANN's adaptation is much faster.

The potential benefit of resource-aware load balancing is at maximum if the system is heterogeneous. If the execution environment is homogeneous, very little can be gained by accounting for heterogeneity. In such a situation, the overhead may even slow down the computation slightly. Therefore, the most appropriate systems for RALBANN are heterogeneous systems.

4.1. Experiment 1: Communication Pattern

The aim of this set of experiments is to test how successfully RALBANN can learn to distribute the tasks to partitions according to their communication pattern. In the graphs that are produced randomly for this experiment, the tasks are running on 3 hosts and a variable number of tasks. All hosts have the same configurational properties.

The following table shows the results of this experiment. The values show how successfully RALBANN can distribute the tasks to partitions compared with multilevel partitioning algorithm. In all experiments, RALBANN is trained until it can successfully partition 90% of all graphs given.

Table 2 - Success rates of the Partitioner ANN using several tasks on 3 hosts compared with multi-level load balancer

Number of Graphs Trained	5 Tasks	10 Tasks	15 Tasks	20 Tasks	25 Tasks	30 Tasks
200	70 %	54 %	45 %	38 %	34 %	29 %
400	83 %	75 %	66 %	44 %	42 %	31 %
600	97 %	89 %	72 %	59 %	45 %	38 %
800	99 %	95 %	81 %	68 %	52 %	46 %
1000	100 %	99 %	85 %	81 %	69 %	63 %

As seen in the table, RALBANN show a superior success in partitioning the TIG graphs. When trained enough, it can predict the partitions as good as multilevel partitioning algorithms in a more efficient way using artificial neural networks. As the number of random graphs used for training increases, the success rate increases naturally since RALBANN is more experienced with various different graphs. However, as the number of tasks increases, RALBANN must be trained with more graphs to increase the success rate to achieve better results.

Increase in the number of hosts doesn't affect the results of this experiment because the ANN (partitioner ANN) is fed with only communication pattern of the tasks which has nothing to do with hosts.

These experiments are done with 2 hidden layers having 20 neurons. Since 2 hidden layers were sufficient for successful results, experiments are not repeated more hidden layers.

4.2. Experiment 2: Heterogeneous Hosts

The aim of this experiment is to evaluate the success of RALBANN in mapping the task partitions to the hosts which have configurational heterogeneity. The experiment is done with a variable number of tasks over 3 hosts, which have processors with 512 MHz, 1024 MHz and 2048 MHz clock speed respectively.

The following table shows the results of this experiment. The values show how successfully RALBANN can distribute the partitions to hosts according the computation loads of the partitions and hosts. In all experiments, RALBANN is trained until it can successfully predict 70% of the mappings given. The results are compared with the mapper of RALBANN.

Table 3 - Success rates of the Mapper ANN using several tasks on 3 hosts compared with multi-level load balancer

Number of Graphs Trained	5 Tasks	10 Tasks	15 Tasks	20 Tasks
200	30 %	25 %	23 %	21 %
400	33 %	27 %	26 %	21 %
600	33 %	28 %	26 %	22 %
800	37 %	30 %	27 %	24 %
1000	40 %	33 %	29 %	25 %

As seen in the table, RALBANN can not show the same perfect performance for the learning of mapping problem. The complexity of the mapping algorithm used in RALBANN prevents neural networks to do a generalization although it can work very well for the training inputs. However it is clear that a simpler algorithm would cause much better results. As the number of hosts increases, the training rates drop slightly since the input vectors become more complex. The following table shows the same experiment with 6 hosts:

Table 4 - Success rates of the Mapper ANN using several tasks on 6 hosts compared with multi-level load balancer

Number of Graphs Trained	5 Tasks	10 Tasks	15 Tasks	20 Tasks
200	25 %	21 %	21 %	20 %
400	26 %	24 %	21 %	19 %
600	28 %	24 %	23 %	22 %
800	32 %	27 %	26 %	23 %
1000	37 %	30 %	26 %	25 %

The experiments are repeated with both 1 hidden layer having 20 neurons and 2 hidden layers having 20 neurons each. However the results with 2 hidden layers were slightly better than the results of the results with 1 hidden layer.

The importance of this experiment is that RALBANN was able to detect non-dedicated hosts. That is, RALBANN distributed the task partitions to the hosts according to hosts' external loads. This proved resource aware load balancing skills of RALBANN.

4.3. Experiment 3: Comparison with Graph Partitioning based Load Balancer

This set of experiments is performed to compare the performance of RALBANN with graph partitioner in terms of CPU usage while RALBANN is running in non-learning mode. The comparisons are done using the graph partitioner of RALBANN which is implemented using multilevel graph partitioning algorithms. In multi-level algorithms [3,4], in the first phase (coarsening phase), heavy edge matching is used. In heavy edge matching, the vertices are visited in random order. A vertex u is matched with one of its adjacent unmatched vertex v such that the weight of the edge (u, v) is maximum over all valid incident edges (heavier edge). The complexity of computing a maximal matching is O(v*e), where v is the number of vertices and e is the number of edges. Since this phase runs until the number of multi-nodes is sufficiently small, the stop criteria of coarsening phase is the equality of the number of multi-nodes to the number of hosts. Therefore, this phase runs v times at the worst case, making the overall complexity of this phase $O(v^2*e)$. The second phase is the partitioning phase and the coarsened graph is partitioned into k partitions. Since the graph is coarsened until it has only k multi-node vertices, the partitioning phase has the complexity O(1). In third phase (uncoarsening phase), the coarsened graph is propagated back to the initial graph with the partitioning information. In this phase, each vertex is visited and moved to another partition if some conditions are valid. Therefore the complexity of this phase is O(v). Considering all 3 phases, the overall complexity of the algorithm is $O(v^2*e)$.

For a feed-forward ANN, the complexity for producing an output directly depends on the number of neurons and the size of the input vector. The number of hidden and output neurons is generally constant and determined for the quality and accuracy of the output. On the other hand, the number of input neurons and therefore the number of calculations between the weights of the hidden neurons in the first layer and the output of input neurons depends on the size of input vector. Therefore the complexity of generating an output from an input vector of size n is O(n). In RALBANN the input vectors are created from TIG graphs. For mapper ANN, the size of input vector is determined by the number of hosts and therefore the complexity is O(h*k), where

h is the number of hosts and k is the number of partitions, which can be the number of tasks (v) at the worst case. For partitioner, the input vector is constructed from the tasks and task communications and therefore the complexity is $O(v^2)$ where v is the number of tasks (vertices). The overall complexity is $O(v^2)$ since $O(v^2)$ overrides O(h*k).

From the complexity calculations, it is evident that RALBANN run faster and uses less memory than the multilevel graph partitioner in theory.

The following table shows the results of CPU usage for RALBANN and multilevel graph partitioner for a network with 3 hosts and a variable number of tasks which is specified in left-most column.

Table 5 - Comparison of the execution times of multi-level load balancer and RALBANN

Number of Tasks	CPU Usage in ms (Multi Level)	CPU Usage in ms (ANN)
10	6	4
20	32	14
100	290	78
200	710	172
500	1901	381

These results show that RALBANN's performance is superior to the performance of graph partitioner. Provided that RALBANN is trained well enough, its load balancing decisions will be as successful as the decisions of graph partitioner.

4.4. Comparison with P-GRADE Load Balancer

P-GRADE has its own centralized load balancer. This load balancer is based on simulated annealing algorithm. This load balancer uses the following cost function [44] which replaces the Hamiltonian function in simulated annealing method [22]:

$$IB(i) = |L(i) - L_{avg}| + w_{ul} \max(0, n_{CPU} - L(i))$$
$$F_0(M) = \sum_{1 \le i \le m} IB(i) + w_{comm} \sum_{1 \le i < j \le n, M(i) \neq M(j)} c(i,j)$$

Figure 43 - P-GRADE Load Function

where,

- $L(i) = \sum_{M(j)=i} l(j)$ and $L_{avg} = 1/m \sum_i L(i)$ and l(i) is the computation demand of process i on its host.
- $l_{CPU}(i)$ is CPU usage of task i.
- n_{CPU} denotes the number of CPUs on the hosts.
- Processes are numbered 1...n and hosts are numbered 1…m.
- M is a random mapping of the tasks to hosts.
- c(i,j) is the communication load of process i and j.
- w_{ul} and w_{comm} are the constants that are chosen between the interval [0,1].
- And lastly F(M) is the cost function and IB(i) is the weighted imbalance of process i.

Although simulated annealing is a good method which can find the best load balancing decisions, it has the following disadvantages over RALBANN; To ensure a balanced state [45], special care must be taken to decide the correct set of changes and choose the correct sequences of the temperature values. Otherwise, the balancer will not be able to find the correct optima. Another problem of simulated annealing is that it may not find the optimal solution at the early steps of the algorithm since the cost function calculates the cost for a random mapping at each iteration.

On the other hand, RALBANN uses graph partitioning methods which takes care of the global system data and therefore can find the best or a nearly best solution only in

one step. Combining the success of graph partitioners with artificial neural networks, RALBANN is a successful and efficient load balancer.

The following table shows the comparison of the execution times of P-GRADE load balancer and RALBANN in non-learning mode. RALBANN runs much faster than simulated annealing based load balancer.

Table 6 - Comparison of the execution times of P-GRADE LB and RALBANN

Number of Tasks	CPU Usage in ms (P-GRADE)	CPU Usage in ms (ANN)
10	110	4
20	156	14
100	397	78
200	603	172
500	1211	381

4.5. Training Time

The most important disadvantage of RALBANN is its training time. The time to train RALBANN for getting load balancing decisions with sufficient quality is much longer than the time to get the decision itself. However, once trained, RALBANN can very efficiently make load balancing decisions since its uses artificial neural networks.

In RALBANN, there are 2 neural networks, namely: partitioner and mapper ANNs, each having 40 hidden neurons separated into 2 hidden layers. For partitioner ANN, the size of the input vector is in the order of $O(n^2)$, where n is the number of tasks. On the other hand, for mapper ANN, the size of the input vector is in the order of O(n*m), where n is the number of tasks and m is the number of hosts. Since n^2 suppresses n*m, the overall complexity of RALBANN for taking a load balancing decision is $O(n^2)$. This assumption is true only if n>m.

However, calculating the complexity of training algorithm for an ANN is not straightforward. Because training time depends on the size of the input vector, the number of input vectors and mostly the complexity of the process that must be learnt. The training algorithm terminates when the network error of the ANN is sufficiently small for all input vectors. Therefore, if the process to be learnt is complex, the network error decreases slowly and training takes more time. Figure 44 shows the training time of partitioner ANN:

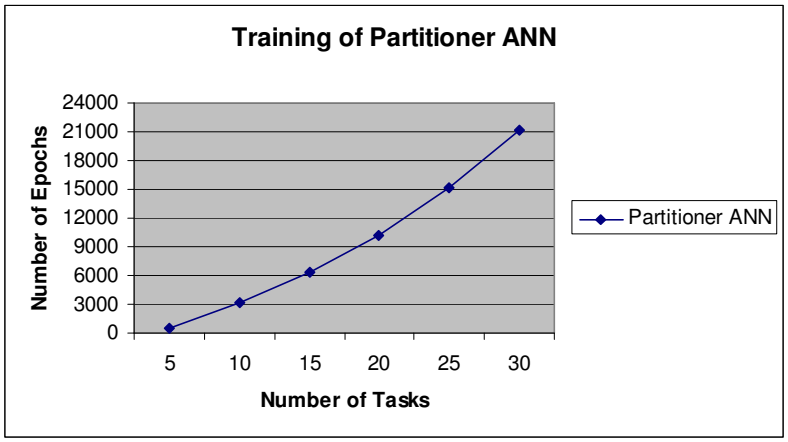


Figure 44 - Training time of Partitioner ANN with n*40 input graphs, where n is the number of tasks

Figure 45 shows the training time of mapper ANN:

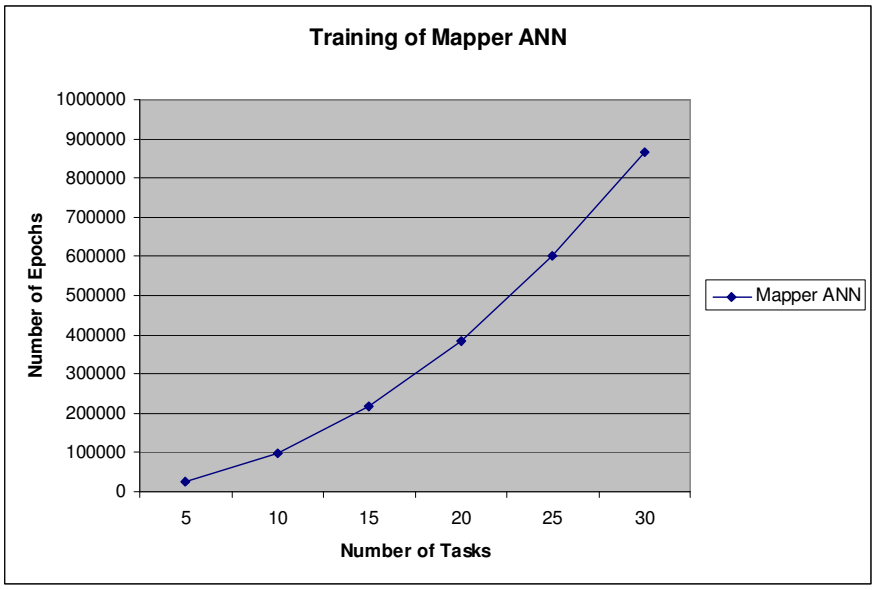


Figure 45 - Training time of Mapper ANN with n*40 input graphs, where n is the number of tasks

As seen in the figures, the training time of mapper ANN consumes much more time than the partitioner ANN. The reason for this is the complex mapping algorithm. If a simpler algorithm were to be used, it would have a better training time.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, a resource aware load balancing model entitled as RALBANN is described. The main goal of RALBANN is the learning of graph partitioning based load balancing using feed-forward neural networks. Graph partitioner based load balancers models the distributed system using graphs and thus load balancing problem is reduced to graph partitioning problem. Therefore, load balancing problem can be described as creating task partitions which has minimum edge cut which means minimizing communication load between the partitions. Although graph partitioners are counted among the most successful load balancers, they have a major disadvantage; they are inefficient in terms of CPU usage and memory. RALBANN is designed as a solution to the drawbacks of the graph partitioners by bringing together the successful results of graph partitioners with efficient decision making mechanisms of feed-forward neural networks. In other words, RALBANN learns load balancing using feed-forwards neural networks. There are 2 neural networks which are charged to learn the 2 steps of load balancing: partitioner ANN whose job is to learn how to partition the graph and mapper ANN whose job is to learn how to map the partitions to hosts. Resource awareness is accomplished by mapping the task partitions to the computing resources according to the capabilities of the computing resources producing a resource aware load balancing scheme.

RALBANN can train itself offline using its random graph generator module, which produces random TIG graphs for a specific configuration. After trained enough, RALBANN saves the weights of the neural networks into files which can then be loaded and used again. This self-training process speeds up training process very

much. One of the good aspects is that RALBANN differentiates between the partitioning and mapping tasks, which are the basic tasks of graph partitioning based load balancing algorithms, by assigning each phase to a different artificial neural network. The weights of the partitioner and mapper ANNs are saved to different files. This separation speeds up the training process when the same host configuration is used repeatedly for different distributed applications. The reason for this speed up is that the task of partitioner ANN is only to partition the tasks into groups according tasks' computational loads. Therefore it has nothing to do with the configuration of the hosts. On the other hand, the task of mapper ANN is to map the partitions to hosts according to computational load of the partitions, computational power and external load of the hosts.

One of the good results of RALBANN software which has been produced as a result of the research within the scope of this thesis is that it is a self-sufficient tool that can be used with any existing load balancing algorithms and any distributed environments with only small changes. RALBANN has simple interfaces that must be implemented to adapt to any distributed systems. In this context, it is very clear that our model has advantages when compared to similar same-goal efforts that generally propose tools that are tightly-coupled to the specific distributed environments. In this thesis, RALBANN is fully integrated with P-GRADE. RALBANN can periodically poll the monitoring tool of P-GRADE to upload the necessary data. Then it is used for both training and producing load balancing decisions. The decisions are then sent to migration unit of the specific distributed environment.

Since ANNs are trained to learn the partitioning and mapping problem as a function of several parameters (that is, not only specific cases), RALBANN can react any changes in the atrributes of the hosts or tasks , such as CPU load and CPU power and produce the right load balancing decisions. However, RALBANN has limitations for the dynamic changes in the size of hosts and tasks. One of the main limitations of RALBANN is that dynamic computational models are not supported. Therefore, for the applications in which the number of tasks change dynamically during run-time, RALBANN doesn't provide any mechanisms to adapt the changes. If the number of hosts changes, RALBANN has no mechanisms to adapt the changes, either. How-

ever, since the host data, which is learnt by mapper ANN, is separately learnt and saved, RALBANN can be re-run to load the new weight file for the new resource situation. This shows the benefit of separating and saving the weights of mapper and separator ANNs in different files and for different configurations.

The results obtained from RALBANN are very promising. The experiments show that RALBANN can perform as successful as multilevel graph partitioners which are among the most successful load balancers. RALBANN's major advantage over graph partitioners is that it runs much faster than graph partitioning methods because of the efficient run time of neural networks. Training time is the major disadvantage of RALBANN. However, this can be compensated by training RALBANN for the most commonly used distributed applications and saving the weight values for later usage.

Future Work

In terms of the tasks partitions produced, RALBANN shows very good performance, while it can not produce the same successful results for the mapping task. A certain improvement could be achieved if a simpler mapping algorithm were used. One of the necessary future works is to improve the performance of mapper ANN by doing the necessary modifications to the learning process and learning algorithm.

It is necessary to include the memory factor, too. The load balancing results are effected very much by the memory usage of the tasks and memory size of the hosts. Currently RALBANN doesn't care this factor.

Another improvement is to change *Kernighan-Lin* algorithm to consider the computational power and external loads of the hosts while creating the partitions. In this case, the conditions which are checked before the movement of vertices must be changed to include also the relevant properties of the hosts. Instead of creating equally balanced partitions in terms of CPU usage, the algorithm will create imbalanced partitions in terms of CPU usage but balanced when the conditions of the hosts are considered.

Considering the dynamic computational models is another planned future work for RALBANN. In this case, RALBANN will consider and provide solutions for the dynamic changes in the number of tasks.

REFERENCES

[1] The Globus Toolkit, http://www.globus.org, last accessed on 05.08.2006.

[2] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the Message Passing Interface. J. Parallel Distrib. Comput., http://www3.niu.edu/mpi, May 2003.

[3] Bruce Hendrickson and Karen Devine: Dynamic Load Balancing in Computational Mechanics, Computer Methods in Applied Mechanics and Engineering, 184:485-500, 2000.

[4] Bradford L. Chamberlain: Graph Partitioning Algorithms for Distributing Workloads of Parallel Computations, 1998.

[5] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, Luis G. Gervasio: New Challenges in Dynamic Load Balancing, Technical Report Technical Report CS-04-02, Williams College Department of Computer Science, 2004.

[6] Eric D. Vaughan, Ezequiel Di Paolo, Inman R. Harvey: The tango of a load balancing biped, Proceedings of the Seventh International Conference on Climbing and Walking Robots, CLAWAR Madrid, September 22th-24th, 2004.

[7] Samphel Norden, Distributed Computing : An Overview, http://www.geocities.com/SiliconValley/Vista/4015/pdcindex.html, last accessed on 13.05.2006.

[8] M. Kara, "Using dynamic load balancing in distributed information systems" Research Report Series, School of Computer Studies, The University of Leeds, Report 94.18, May 1994.

[9] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editors, CRPC Parallel Computing Handbook. Morgan Kaufmann, 2000 (in press), 2000.

[10] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic octree load balancing using space-filling curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[11] Levi Valgaerts: Dynamic Load Balancing Using Space-Filling Curves, 2005.

[12] Kirk Schloegel, George Karypis, Vipin Kumar, Rupak Biswas, and Leonid Oliker: A Performance Study of Diffusive vs. Remapped Load-Balancing Schemes, 11th Intl. Conference on Parallel and Distributed Computing Systems, 1998.

[13] Pankaj Mehra: AUTOMATED LEARNING OF LOAD-BALANCING STRATEGIES FOR A DISTRIBUTED COMPUTER SYSTEM, Ph. D. Thesis, Dept. of Computer Science, University of Illinois, 1993.

[14] PANKAJ MEHRA: LOAD BALANCING: AN AUTOMATED LEARNING APPROACH, World Scientific Publishers, 1995

[15] James D. Teresco, Karen D. Devine, and Joseph E. Flaherty: Partitioning and Dynamic Load Balancing for the Numerical Solution of Partial Differential Equations, Chapter in Numerical Solution of Partial Differential Equations on Parallel Computers, Are Magnus Bruaset, Petter Bjørstad, Aslak Tveito, editors. © Springer-Verlag, 2005. Also available as Williams College Department of Computer Science Technical Report CS-04-11.

[16] N.G. Shivaratri, P. Krueger, M. Singhal, Load distributing for locally distributed systems, IEEE Comput. 25 (1/2) (1992) 33-44.

[17] S.T. Barnard, PMRSB: Parallel multilevel recursive spectral bisection, Cray Res. Inc., 1996.

[18] B Kernighan, S Lin, "An efficient heuristic procedure for partitioning graphs", The Bell System Technical Journal, V-29, 1970

[19] Kirk Schloegel, George Karypis, Vipin Kumar: Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes, Journal of Parallel and Distributed Computing, 1997.

[20] B. Hendrickson, R. Leland, A multilevel algorithm for partitioning graphs, in: S. Karin (Ed.), Proceedings of Supercomputing'95, San Diego, CA, ACM Press, New York, 1995.

[21] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Par. Dist. Comput. 48 (1) (1998) 96-129.

[22] Geoffrey C. Fox, Roy D. Williams, Paul C. Messina: Parallel Computing Works from http://www.netlib.org/utk/lsi/pcwLSI/text, last accessed on 31.08.2006.

[23] Leslie Pack Kaelbling, Michael L. Littman, Andrew W. Moore: Reinforcement Learning: A Survey, Journal of Artificial Intelligence Research, 1996.

[24] M. E. Harmon and S. S. Harmon, "Reinforcement learning: A Tutorial", http://citeseer.ist.psu.edu/harmon96reinforcement.html, last accessed on 20.08.2006.

[25] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, Massachusetts: MIT Press, 1998.

[26] A. Galstyan, K. Czajkowski, and K. Lerman. Resource allocation in the grid using reinforcement learning. In Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS-04), 2004.

[27] C. Claus and C. Boutilier, "The dynamics of reinforcement learning in cooperative multiagent systems", in Proceedings of the Fifteenth National Conference on Artificial Intelligence, Madison, WI, 1998, pp. 746-752.

[28] Johan PARENT, Katja VERBEECK, Jan LEMEIRE: Adaptive Load Balancing of Parallel Applications with Reinforcement Learning on Heterogeneous Systems, 2002.

[29] Xiaojin Zhu: Semi-Supervised Learning Literature Survey, Computer Sciences TR 1530, University of Wisconsin - Madison, 2005.

[30] Hans-Ulrich Heiss, Marcus Dormanns: Task Assignment by Self-Organizing Maps, Internal Report No. 17/93, Dep. of Computer Science, University of Karlsruhe (May 1993).

[31] Murat Atun and Attila Gürsoy: A New Load-Balancing Algorithm Using Self-Organizing Maps, Proc. of the 14th Intl. Symp. on Computer and Information Sciences, ISCISXIV, Ed. M Turksever et al. October 18-20, 1999, Kusadasi - Turkey, pp. 787-794.

[32] Murat Atun and Attila Gürsoy: Neighborhood Preserving Load Balancing: A Self Organizing Approach, Vol 1900 pp. 234-241 (Europar 2000).

[33] Dave Anderson and George McNeill: ARTIFICIAL NEURAL NETWORKS TECHNOLOGY, State-of-the-Art Report, August 20 1992.

[34] Bishop, Christopher: Neural Networks for Pattern Recognition, Clarendon Press, Oxford, 1995.

[35] Shashi Shekhar and Minesh B. Amin: Generalization by Neural Networks, Knowledge and Data Engineering, 1992.

[36] Pankaj Mehra, Benjamin W. Wah: POPULATION-BASED LEARNING OF LOAD BALANCING POLICIES FOR A DISTRIBUTED COMPUTER SYSTEM, Proc. 9th Computing in Aerospace Conf., pp. 1120-1130, American Institute of Aeronautics Conf., pp. 1120-1130, American Intitute of Aeronautics and Astronautics, San Siego, CA, 1993.

[37] Pankaj Mehra and Benjamin W. Wah: AUTOMATED LEARNING OF WORKLOAD MEASURES FOR LOAD BALANCING ON A DISTRIBUTED SYSTEM, Proc. of the 1993 International Conference on Paralled Processing, pages 263-270, section III.

[38] Aly E. El-Abd, Mohamed I. El-Bendary: A Neural Network Approach for Dynamic Load Balancing In Homogeneous Distributed Systems, IEEE Computer Society Washington, DC, USA, 1997

[39] C. Walshaw, M. Cross, Mesh partitioning: a multilevel balancing and refinement algorithm, SIAM J.Sci. Comput. 22 (1) (2000) 63-80.

[40] George Karypis and Vipin Kumar: Multilevel k-way Partitioning Scheme for Irregular Graphs, Technical Report: 95-064, 1998.

[41] Bruce Hendrickson and Karen Devine: A Multilevel Algorithm for Partitioning Graphs, In Proc. Supercomputing '95, (1995).

[42] P-GRADE, Parallel Application Developemnt Environment, http://www.lpds.sztaki.hu, last accessed on 01.09.2006.

[43] Pter Kacsuk, Gbor Dzsa, Jzsef Kovcs, Rbert Lovas, Norbert Podhorszki, Zoltn Balaton, and Gabor Gombs: P-GRADE: A Grid Programing Environment, Journal of Grid Computing (JGC), 1(2):171--197, 2003.

[44] M. L. Tóth, N. Podhorszki, P. Kacsuk: Load Balancing for P-GRADE Parallel Applications, Proceedings of the 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS 2002), Linz, Austria, pp. 12-20, 2002.

[45] Roy D. Williams, Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations, 1990.

APPENDIX A

DEVELOPER'S MANUAL

In this appendix, the important data structures implemented in the RALBANN software is described. RALBANN is an application which is composed of 5 important components as depicted in the following diagram:

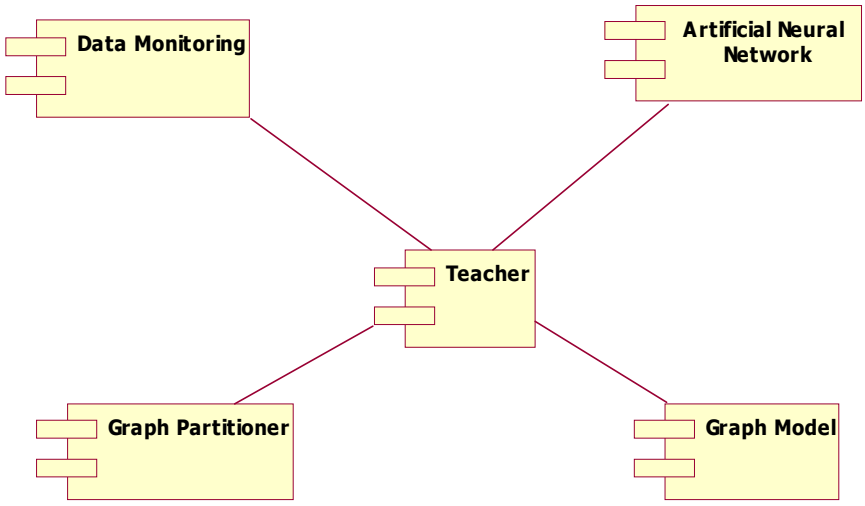


Figure 46 - Component Diagram of RALBANN

In the following sections, the classes and functions used in each component are described in short detail.

Graph Model Component:

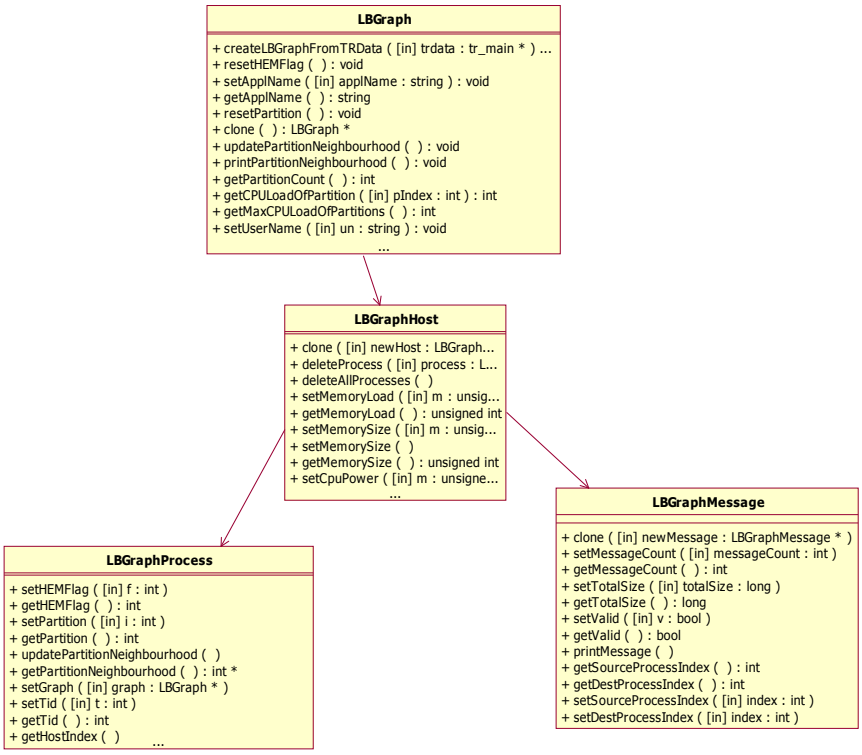


Figure 47 - Class Diagram of Graph Model

This component is responsible for representing the TIG graphs. The graph is created from an XML file containing the host and task information necessary to do load balancing. The most important class is LBGraph class. This class provides all functions needed to create a TIG Graph. LBGraphHost represents a host and abtracts the host information. LBGraph Process abtracts the processes (tasks) running on the hosts and lastly LBGraphMessage class represents the messages sent from one process to another.

1. LBGraph Class:

– **createLBGraphFromTRData:** This static function is used to create LBGraph object from P-GRADE trace file.

- **createLBGraphFromXMLFile:** When this static function is called it creates and returns an LBGraph object from an XML file which contains load balancing data.
- **setApplName:** Sets the name of the distributed application. This name is used for searching, loading and saving the weights of the neurons.
- **resetPartition:** Resets the partitioning data which is created after running graph partitioner.
- **clone:** Creates a clone of the graph.
- **getPartitionCount**: Returns the partition count created after running graph partitioner.
- **addHost**: Add a new host to the graph and returns its index in the host array.
- **getHostCount**: Returns the host count added to the graph.
- **getHosts**: Returns a LBGraphHost array containing ter host data.
- **getHost**: Searchs and returns the host at the specified index or name given.
- **addProcessToHost**: Adds a new task (process) to a host. Processes are hold in an array.
- **getProcessCount**: Returns task (process) count.
- **deleteProcess**: Deletes a previously added task (process) from the graph.
- **addMessageToProcess**: Add a new communication message to a process. Messages are hold in a 2 dimensional array. Each of the elements of the array is a LBGraphMessage object.

ANN Component:

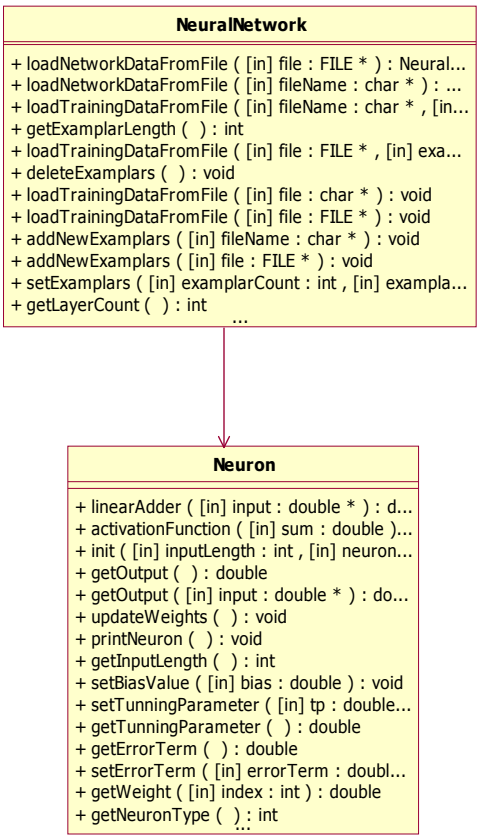


Figure 48 - Class Diagram of Artificial Neural Network

This component is used to construct a feed forward artificial neural network. NeuralNetwork class provides methods for constructing ANN's with several hidden layers using Neuron objects. The parameters susch as tunning parameter, adjustment parameter, bias values can be configured from configuration file. The configuration file can contain the following values:

InputLength: Input length of the ANN.

HiddenLayers: Number of hidden layers and number of neurons in each hidden layer is given with this attribute

OutputType: This can be FINDBIGGEST or EXACTMATCH. In EXACTMATCH, ANN tries to learn the output exactly. But for FINDBIGGEST, ANN learns to classify the input vectors and highlights one of the classes for each input data.

OutputLength: Length of output vector and number of output neurons.

TunningParameter: The initial value for tunning parameter. This parameter is weaved continosly so that ANN doesn't drop into a local minima/maxima.

BiasValue: The bias value for the neurons.

NumberOfEpochs: The number of epochs that ANN will train itself.

NumberOfScans: The number of scans that ANN will go in each epoch.

When trainNetwork() function is called, ANN begins to train itself until success condition is reached, which can be network error or succes rate for the examplars. The weights and the parameters are saved to a file periodically so that it can be loaded again later and continue training.

1. Neuron Class:

- **init:** This method reconstructs the data structures used inside the Neuron class.
- **getOutput**: Given an input vector, this method returns the output of neuron with the current weights.
- **updateWeights**: This method updates the weights of the neuron according to the tunning parameter, error term for the given input and output.
- **linearAdder**: This function calculates the sum of multiplication of each input and weight.
- **activationFunction**: This method abstracts the activation function which is equal to sigmoid function in RALBANN.
- **printNeuron**: Prints the neuron data in a human readable form.
- **getInputLength**: Returns the input length.
- **setBiasValue**: This function sets the bias value for this neuron.

- **setTunningParameter**: This function sets the tunning parameter.
- **setErrorTerm/getErrorTerm**: This function sets/gets the error term. Error term is equal to (desired_output-calculated_output).
- **getNeuronType**: Returns neuron type. It may be OUTPUT or HIDDEN neuron.
- **copyWeights**: Gets a copy of the weights. The copies are used during back-propogation algorithm.
- **restoreWeights**: The weights previously copied are restored again.
- **getBiasValue**: Returns the bias value.
- **setWeight/getWeight**: Sets/Gets the weights of the neuron.

2. NeuralNetwork Class:

- **setStopCondititon**: This sets the stop condition. It can be NETWORK_ERROR or SUCCESS_RATE. If it is set to NETWORK_ERROR, the exemplars are trained until the network error is smaller then the error value which is set using setNetworkErrorLimit() function. Otherwise (SUCCESS_RATE) the examplars are trained until ANN guesses a high percent (for example 90%) of the examplars. The success rate is set using setSuccessRateLimit() function.
- **setNetworkErrorLimit**: Sets the error limit that ANN must give at most for the registered examplars.
- **setSuccessRateLimit**: Sets the success rate limit. This must be set if stop condition is SUCCESS_RATE.
- **getOutputForSingleCaseWithNormalization**:
- **setSaveFileName**: Sets the file name in which the weights and configuration information will be saved.
- **setTunningParameter**: Sets the tunning parameter.
- **trainNetwork**: Start training the given examplars.
- **loadNetworkDataFromFile**: ANN parameters and weights are loaded from file.
- **setParameters**: Tunning parameter, tunning adjustment value, bias value are set using this function.
- **saveToFile**: The parameters and weights are saved to file.
- **loadTrainingDataFromFile**: Training examplars are loaded from file.

- **addNewExamplar**: Adds a new training examplar.
- **normalizeExamplars**: The input vectors (examplars) are normalized to values between 0.0 and 0.9 for faster training.
- **copyWeights**: The weights are copied to internal structures. This is needed for back- propagation learning algorithm.
- **restoreWeights**: The weights copied previously are restored again. This is needed for back- propagation learning algorithm.

Graph Partitioner:

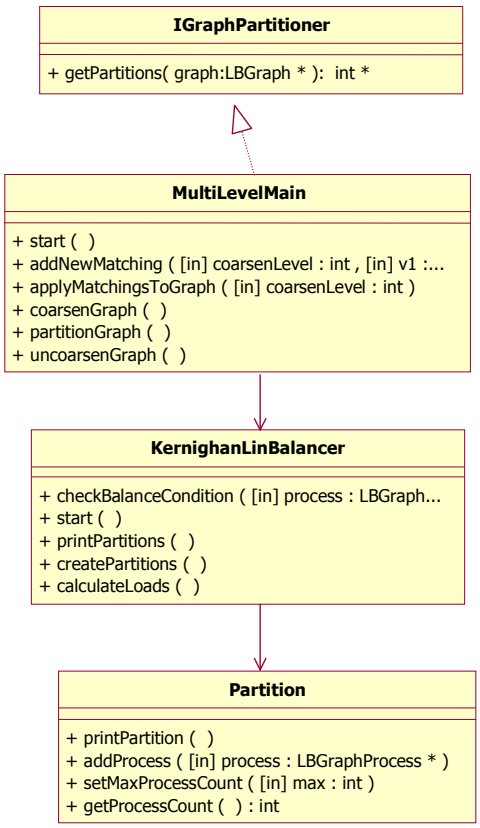


Figure 49 - Class Diagram of Graph Partitioner

This component is used by Teacher module for partitioning the TIG graphs to train artificial neural networks. RALBAAN uses IGraphPartitioner interface which provides the necessary functions for graph partitioning. Therefore any graph partitioning algorithm implementing this interface can be used with RALBANN. In RALBANN, multi level graph partitioner is implemented specifically.

1. IGraphPartitioner Interface:

– **getPartitions:** This function gets a LBGraph object and return an array which contains the partition ids for each task (process).

Data Monitoring:

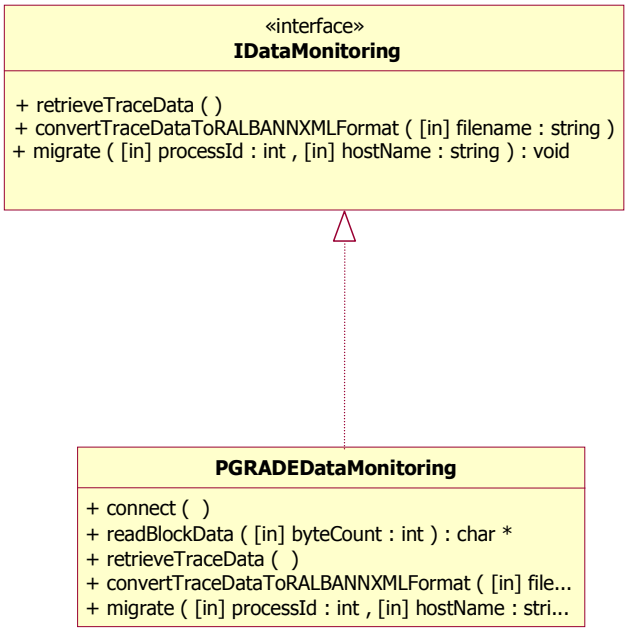


Figure 50 - Class Diagram of Data Monitoring

The data monitoring component is the interface of RALBANN to the distributed environment. RALBANN can be integrated with any environment provided that the necessary class implementing IDataMonitoring interface is coded. RALBANN uses this interface to comminucate with the distributed environment. This interface provides the following methods:

- **retrieveTraceData**: This function connects to the monitoring tool of the distributed environment and collects the load balancing data. The data is saved in a file which is then given **convertTraceDataToRALBANNXMLFormat()** function so that it is converted to special XML format.
- **convertTraceDataToRALBANNXMLFormat**: The trace file containing load balancing information is converted to a XML file.
- **migrate**: This function contacts the migration tool of the distributed environment to migrate a process to a host.

Teacher:

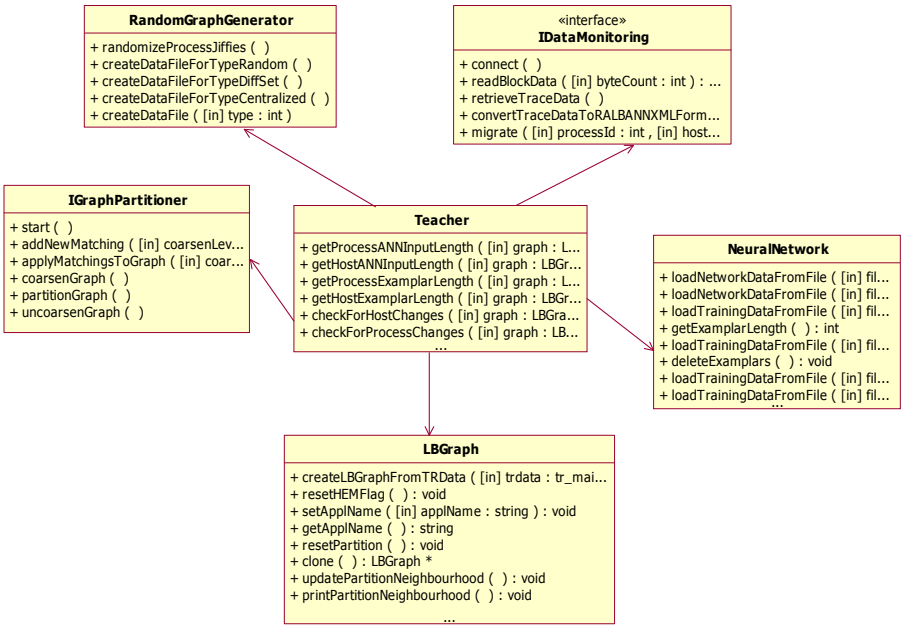


Figure 51 - Class Diagram of Teacher

The Teacher uses all other modules to train the artificial neural networks. Teacher can used the load balancing information collected using *Data Monitoring* component, or it can generate random TIG graphs using RandomGraphGenerator class. The random graphs are then converted to input vectors which are used to train the ANNs. Using random graph makes training process faster. The weights of trained ANNs can be saved and later loaded again to be used with real load balancing data.