BALANCING COMPUTATION LOAD AND COMMUNICATION OVERHEAD WITH MULTILEVEL SELF ORGANIZING MAPS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Erdoğan BIKMAZ
July, 2001

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

Asst. Prof. Attila Gürsoy (Supervisor)

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

Assoc. Prof. Özgür Ulusoy

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

Asst. Prof. Uğur Güdükbay

Approved for the Institute of Engineering and Science:

Prof. Mehmet Baray
Director of Institute of Engineering and Science

ABSTRACT

BALANCING COMPUTATION LOAD AND COMMUNICATION OVERHEAD WITH MULTILEVEL SELF ORGANIZING MAPS

Erdoğan BIKMAZ
M.S. in Computer Engineering
Supervisor: Asst. Prof. Attila Gürsoy
July, 2001

Today, execution time of big programs such as the programs for data-analysis tasks, scientific computations, and engineering problems remains as a big bottleneck. To reduce the execution time, a common approach is to run such applications on parallel machines. A major task in the parallelization of these applications or computations is to distribute the computational load to processors in a balanced way. We argue that distributing only the computational load equally is not enough for load balancing because communication cost, which is inevitable in parallel computations, brings some extra overhead. We used Kohonen Self-Organizing Maps (SOM) that preserves the neighborhood relationship of tasks to minimize and balance the communication overhead. We balance not only computation load but also communication overhead by balancing the number of messages. The performance experiments show that our algorithm outperforms the other static task mapping algorithms on the view of load balancing. One general drawback of Self-Organizing approaches is the high running time. We decreased the execution time of SOM algorithm with multilevel approach.

Key words: Neural networks, Kohonen Self-Organizing Maps, task mapping, load balancing, communication overhead.

ÖZET

HESAP VE HABERLEŞME YÜKÜNÜ ÇOK KATMANLI KENDİNDEN DÜZENLENEN HARİTALARLA DENGELEME

Erdoğan BIKMAZ
Bilgisayar Mühendisliği Bölümü, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Attila Gürsoy
Temmuz, 2001

Bugün, veri analizi, bilimsel hesaplamalar, ve mühendislik hesaplamaları gibi uygulamalarda kullanılan bazı büyük programların bilgisayarlarda yürütme zamanı büyük bir problem olarak karşımıza çıkmaktadır. Yürütme zamanını azaltmak için genel yaklaşım, bu tür uygulamaları paralel makinalarda yürütmektir. Bu tür uygulamaları ve hesaplamaları paralelleştirmekte önemli olan hesaplama yüklerini işlemcilere dengeli dağıtmaktır. Sadece hesaplama yüklerini eşit olarak dağıtmanın yük dengesi açısından yeterli olmadığını iddia ediyoruz, çünkü paralel makinalarda kaçınılmaz olan haberleşme fazla yük getirir. Haberleşme yükünü azaltmak ve dengelemek için komşuluk ilişkilerini topolojik olarak muhafaza eden Kohonen Kendinden Düzenlenen Harita Algoritmasını kullandık. Mesaj sayısını dengeleyerek sadece hesaplama yükünü değil, aynı zamanda haberleşme yükünü de dengeledik. Başarım deneyleri, yük dağılımı açısından bizim algoritmamızın diğer statik yük dağılım algoritmalarından daha iyi olduğunu gösterdi. Kendinden Düzenlenen Haritalar Algoritmasının genel kötü yanı uzun yürütme zamanıdır. Kendinden Düzenlenen Haritalar Algoritmasının işlem süresini çok katlı yaklaşımla azalttık.

Anahtar Kelimeler: Sinir ağları, Kohonen Kendinden Düzenlenen Haritalar Algoritması, görev atama, yük dengeleme, Haberleşme yükü.

To my dear wife; ÖDÜL

Acknowledgement

First of all, I would like to express my deepest gratitude to my advisor Asst. Prof. Attila Gürsoy for his great supervision, guidance and patience for the development of this thesis.

I would like to thank to committee members Özgür Ulusoy and Uğur Güdükbay for reading the thesis and for their constructive comments.

I would like to express my deepest gratitude to Turkish Armed Forces for their support and giving me opportunity to make this study.

I would like to express my thanks to Murat Atun, who formerly studied on Self Organizing Maps whose research shed light on my studies.

I also would like to thank to my wife Ödül for her great patience and advices, finally to Nedret and Bülent Munzur and my office mates for their moral support and encouragement.

Contents

1 Introduction **1**

2 Load Balancing with SOM **4**

2.1 Kohonen Self-Organizing Map 4

2.2 Implementation of SOM for Load Balancing 6

2.3 Motivation of Our Studies 11

3 Communication Overhead **12**

3.1 Communication Cost 13

3.2 Solution of Laplace's Equation by Jacobi Iteration 13

3.3 Experiments 16

3.3.1 Experiment 1: Effect of Communication Cost on Total Cost. 16

3.3.2 Experiment 2: Size of Load Experiment 17

3.3.3 Experiment 3: Effect of the Number of Communicating Processors 20

3.3.4 Experiment 4: Effect of Number of Sends 23

3.3.5 Experiment 5: The Message Length 25

3.4 Configuration of the Machine 27

3.5 Conclusion 28

4 Communication Optimized Mapping 29

4.1 Processor Mapping 29

4.2 Models for Representing Processors 30

4.3 Discussion of Processor Connection Models 35

4.4 Improved SOM Algorithm for Load Balancing 37

5 Comparison of Processors Mappings 38

5.1 Related Programs Used for Comparison 38

5.2 Comparison of SOM Mappings 39

5.3 Comparison of SOM Processor Mappings with pMeTis and kMeTis 40

5.4 Comparison of Algorithms 43

5.5 Comparison of Hexagonal Mapping and kMeTis with and without Communication Overhead 48

6 Multilevel Implementation of SOM Algorithm 54

6.1 Multilevel Implementation of Self-Organizing Maps 56

6.2 Performance Results 57

6.3 Conclusion 59

7 Conclusion and Future work 60

7.1 Conclusion 60

7.2 Future Work 61

REFERENCES **66**

Appendix A MeTis 66

Appendix B Data Sets Used for Experiments 68

Appendix C Chaco file Format 71

List of Figures

2.1 Processor mappings of SOM algorithm in[13] (a)Processor Mapping, (b)Processor mapping when each processor area divided to boxes. 7

2.2 Typical SOM Algorithm 10

2.3 Airfoil data is distributed to 16 processors. 10

3.1 5 pointed stencil. 14

3.2 Square grid. 15

3.3 Main loop of the iterative solver. 15

3.4 Nodes are distributed to processors, each processor has 100×100 nodes. Weight of nodes are given randomly at the beginning of the program. 17

3.5 (d) 9 processors are communicating with each other 18

3.6 Blocking communication of Laplace Equation. 19

3.7 Non-blocking communication of Laplace Equation 19

3.8 Load overhead of blocking communication 20

3.9 Load overhead of non-blocking communication 20

3.10 P_0 is communicating with 5 other processors 21

3.11 Communicating processors number overhead of blocking communication . . . 22

3.12 Communicating processors number overhead of non-blocking communication . . . 22

3.13 Communicating processors number overhead of blocking communication, while receiver processors making computation . . . 23

3.14 Communicating processors number overhead of non-blocking communication, while receiver processors making computation . . . 23

3.15 Sender processor sending 1 to 6 messages to the receiver processor. 24

3.16 Number of send overhead of blocking communication when the receiver processor is idle. . . . 25

3.17 Number of send overhead of non-blocking communication when the receiver processor is idle. . . . 25

3.18 Number of send overhead of blocking communication, while the receiver processor making computation. . . . 26

3.19 Number of send overhead of non-blocking communication, while the receiver processor making computation. . . . 26

3.20 Length of message overhead on communication cost. . . . 27

4.1 Processor Mappings. (a) Square Mesh, (b) Square Processor Mapping, (c) Staggered Processor Mapping, (d) Hexagonal Processor Mapping . . . 32

4.2 Finding processor identification number of each box in hexagonal processor mapping . . . 34

4.3 Finding processor identification number of each box in staggered processor mapping . . . 35

4.4 SOM Algorithm that balances communication overhead and computational load. . . . 37

5.1 Deterioration of load imbalance graphs under different communication cost percentage for airfoil data set and when convergence is 3. 42

5.2 Deterioration of load imbalance graphs under different communication cost percentage for biplane data set and when convergence is 3. 44

5.3 Deterioration of load imbalance graphs under different communication cost percentage for crack data set and when convergence is 3. 46

5.4 Deterioration of load imbalance graphs when communication cost included under different communication cost percentage when airfoil, crack and biplane datums are taken and when convergence is 3. 50

6.1 The various phases of the multilevel SOM algorithm. 56

C.1 Chaco Input File Format 72

List of Tables

5.1 Number of communicating processors according to the algorithms and processor shapes. 39

5.2 airfoil data is distributed to 25 processors, weights of nodes are all 1. 41

5.3 airfoil data is distributed to 25 processors, weights of nodes are given randomly between 1 and 10, weights are read from a text file. 41

5.4 airfoil data is distributed to 60 processors, weights of nodes are all 1. 43

5.5 airfoil data is distributed to 60 processors, weights of nodes are given randomly between 1 and 10, weights are read from a text file. 43

5.6 biplane data is distributed to 25 processors, weights of nodes are all 1. 45

5.7 biplane data is distributed to 25 processors, weights of nodes are given randomly between 1 and 10 and are read from a text file. . 45

5.8 biplane data is distributed to 60 processors, weights of nodes are all 1. 45

5.9 biplane data is distributed to 60 processors, weights of nodes are given randomly between 1 and 10 and are read from text file. . . 47

5.10 crack data set is distributed to 25 processors, weights of nodes are all 1. . . . 47

5.11 crack data set is distributed to 25 processors, weights of nodes are given randomly between 1 and 10 and are read from a text file. . . . 48

5.12 crack data set is distributed to 60 processors, weights of nodes are all 1. . . . 48

5.13 crack data is distributed to 60 processors, weights of nodes are given randomly between 1 and 10 and are read from a text file. . . 49

5.14 crack data set is distributed to 60 processors when communication cost included, weights of nodes are 1. . . . 49

5.15 crack data set is distributed to 25 processors when communication cost included, weights of nodes are 1. . . . 51

5.16 biplane data set is distributed to 60 processors when communication cost included, weights of nodes are 1. . . . 51

5.17 airfoil data set is distributed to 60 processors when communication cost included, weights of nodes are 1. . . . 52

6.1 Comparison of SOM and MSOM for 25 processors. . . . 57

6.2 Comparison of SOM and MSOM for airfoil data set with 3% convergence constraint for different number of processors. . . . 58

6.3 Execution time of SOM Algorithm for different data sets with the 3 percent convergence constraint for 5 PCG. . . . 58

B-1 Test Graphs 70

Chapter 1

Introduction

To solve computationally large problems we use parallel programs. The execution time of a parallel program is determined by the processor with the largest execution time. This time is dependent upon both the local calculations performed on each processor and any required inter processor communication. To be able to get good performance from multi-computer systems, computational load of each processor should be maintained in a balanced way. Balancing means that total load (computation and communication) of each processor should be assigned such that each processor spends nearly equal time on the problem.

In order to develop a parallel program for a multi-computer, first of all, the problem is decomposed into a set of interacting sequential sub-problems (or tasks) that can be executed in parallel. Then, each one of these tasks is mapped to a processor of the parallel machine, in such a way that, the total execution time is minimized. This mapping phase is called the task mapping problem, and is known to as NP-hard. However, there exist heuristic approaches to map tasks to processors that significantly minimize the execution time. According to their task assignment order (at the beginning or during the execution), mappings can be grouped in two categories: "static" and "dynamic". Static mapping deals with the assignment of tasks to processor at the beginning only once. On the other hand, dynamic mapping requires changes on mapping (which is also called task migration) during the execution, according to the states of processors. In this study, we consider static mapping with Self-Organizing

Maps (SOM). We have considered both communication load and computation load of each processor. The performance of SOM algorithm is compared with the performance of other well-known algorithms.

Self-Organizing Map algorithm is a stochastic optimization algorithm, which is first introduced by Tuevo Kohonen [28, 29]. The idea of SOM is originated from the organizational structure of human brain and the learning mechanism of biological neurons. It is based on competitive learning, where the neural units compete in order to be activated and eventually corrected. During this process, the neurons organize themselves according to given inputs and according to signals from other neurons. After training, some neurons become sensitive to particular inputs. This sensitivity forms a topological relation (ordering) between the inputs and neurons. That is after training, the neurons become organized in such a way that their ordering reflect the topological properties of inputs. So, SOM algorithm forces neurons to be topologically ordered during the execution according to given inputs.

In this work we discuss different processor connection graphs for SOM based load balancing for a class of parallel programs where the communication between tasks are localized. Many real-life parallel scientific computations such as fluid dynamics, particle simulation, finite element methods have this kind of communication pattern. Furthermore, these problems can be represented by computation graphs where the nodes of the graph represent tasks and edges represent communication or interaction among the tasks. A common way of mapping these tasks to processors is to partition the graph so that the computational load of each partition (that is, sum of node weights) is balanced while the number of edges (or some of edge weights) between partitions is minimized (minimizing edge-cut). Then, each partition is assigned to a processor. Edge cut minimization reduces total volume of communication, results in improved communication time and in general better total execution time. Until now, a lot of different algorithms designed to solve the mapping problem like genetic algorithms [11], simulated annealing [7], mean field annealing [9], greedy approaches [12], Kernighan-Lin heuristic [13] etc. Additionally, many multilevel algorithms/tools are also designed using the above algorithms/approaches including MeTiS [23], Chaco [17], Jostle [10]. Most of these approaches considered reducing total volume of communication. However, in current parallel architectures (message passing), most of the communication overhead takes place

at the preparation of sending and receiving messages at the operating system and parallel runtime system levels [24]. Therefore, in many parallel computations such as finite element method based computations, minimizing the number messages that each processor is sending becomes more important. Recently, this problem is addressed within the graph partitioning context [32, 19]. In this thesis, we address this problem and propose algorithms, which is balancing the number of messages, that balances not only computational load but also communication overhead. The proposed algorithms are based on the neighborhood preserving load balancing with Self-Organizing Maps approach developed by Gürsoy and Atun [14]. In their work, the Self-Organizing Map approach reduce the communication overhead inherently. We improve the load balancing algorithm further to minimize and balance the communication overhead by proposing new processor connection graphs.

The rest of the thesis is organized as follows: For completeness, the SOM Algorithm is explained in Chapter 2. In Chapter 3, communication overhead factors, namely, the effect of load size, the effect of communication type, the effect of the number of communicating processors, the effect of start up time, the effect of message length are discussed. In Chapter 4, the mapping problem is described and given new models for processor connection graph, and processor connection graph models are compared. In Chapter 5, processor mapping approaches are compared and presented. In Chapter 6, multilevel implementation of Self-Organizing Map (MSOM) is presented. In Chapter 7, experimental results are presented. In Appendix A, properties of MeTiS program are given. In Appendix B, list and properties of input files are given. In Appendix C, the Chaco input file format is explained.

Chapter 2

Load Balancing with SOM

2.1 Kohonen Self-Organizing Map

Self-Organizing Maps, which are introduced by T. Kohonen [29], are artificial neural networks modelling the brain's cortex. One of the best properties of these maps is neighborhood preservation from input space to output space. The Kohonen's algorithm is a well established learning rule in fields like robot control and speech recognition. A biological analogue to the SOM are the computational maps in the celebral cortex, where the spatial location of neurons correspond to a particular domain on the human body input sensors. The human brain is arguably the most fascinating structure in all the human physiology. Although the brain is vastly complex on a microscopic level, it reveals a consistently uniform structure on a macroscopic scale from one brain to other. Centers of different actions such as hearing, vision, speech and motor functions lie in specific areas of the brain and these regions have a relation between each other. Furthermore, individual areas exhibit a logical ordering of their functionalities. The SOM was inspired by the way in which various human sensory impressions are neurologically mapped into the brain such that spatial or other relations among stimuli correspond to spatial relations among the neurons. A SOM tries to find clusters such that any two nodes that are close to each other in the grid space have code book vectors that are close to each other in the input space. However, the reverse may not be true: code book vectors that are close to each other in the input space, do not necessarily correspond to clusters

that are close to each other in the grid.

SOMs are widely used in engineering and data-analysis tasks, but rarely in very large-scale problems. A lot of studies are done to decrease the computation time. A lot of methods are proposed to reduce the amount of computation, such as Fast Winner Search [26], and also hardware supported system that enables the parallel computing of Euclidean distance with Kohonen's Self-Organizing Map [2] that computes the distances of all neurons and finds rapidly the position of the winning neuron.

The SOM is a competitive learning algorithm, where the processing elements compete in order to be activated and eventually corrected. During the learning process, neurons become selectively tuned to subregions of the input space, which results in a code book (reference to each other) represented by prototypes of typical features in the data.

The Kohonen SOM (KSOM) is composed of neurons that are on one or two dimensional grid. Each neuron, which is directly connected with the input signal, has lateral connections with other neurons determining the state of activation of the whole map. After the learning process is completed, the Kohonen map has two important properties: topological ordering and density matching. The map is topologically ordered because there are two prototypes (features) that are neighbors in the grid and also neighbors in the input domain space. The map matches the density of the input distribution because regions with high probability of accuracy are mapped to proportionally larger zones. The KSOM usually has an output layer of interconnected neurons that are fully connected to the input layer, so every neuron from the output layer is connected to every neuron from the input layer. As in the domain of supervised networks the connection has certain weights. Every neuron from the output layer has consequently as many weights as the neural network inputs. furthermore the output neurons are ordered in a particular way, usually two dimensional grid, where each neuron has neighbors. The KSOM was inspired by the way in which various human sensory impression are topographically mapped into the neurons of the brain. Like any other neural network, the usage of the Kohonen map follows two steps: the learning step and the testing step. During learning phase, the input examples are sequentially used as input of the neural network, every time the weights of the connections are being changed. The input data

are repeatedly used until the neural network converges. During the testing, the weights do not change and the output of the neural network is used as the response of the neural network to the given input data.

2.2 Implementation of SOM for Load Balancing

The basic architecture of Kohonen's map is n neurons that are generally connected to a $d-$dimensional space, for example a grid, where each neuron is connected with its neighbors. The map has two layers: an input layer and an output layer that consists of neural units. Each neuron in the output layer is connected to every input unit. A weight vector w_i is associated with each neuron i. An input vector, v, which is chosen randomly, is forwarded to the neuron layer during the competitive phase of the SOM. Then, excitation center is determined (the determination of the excitation center will explained later). The weight vectors of the winner neuron and its topological neighbors are updated so as to align them towards the input vector. This step corresponds to the cooperative learning phase of the SOM.

SOM Algorithm is used for load balancing is implemented as follows [15]: in the input layer, unit square $S = [0,1] \times [0,1]$ is accepted as the input space of the Self-Organizing Map. Then, unit square S is divided into p square regions, called processor regions (Figure 2.1(a)), and if the number of tasks is larger than predetermined number of tasks, then each processor area is divided to 5×5 square boxes (Figure 2.1(b)), where $p = P_x \times P_y$ is the number of processors. Every processor P_{ij} has a region of coordinates that is a subset of S bounded by $i \times width_x, j \times width_y$ and $(i+1) \times width_x, (j+1) \times width_y$ where, $width_x = 1/p_x$ and $width_y = 1/p_y$.

Processors are connected to each other with the Processor Connection Graph (PCG), which is a commonly used model for this purpose. A PCG, is an undirected graph $G_P(V_P, E_P)$ such that

- each $v \in V_P$ represents a processor,
- each $(u, v) \in E_P$ represents the physical communication link between processors u and v.

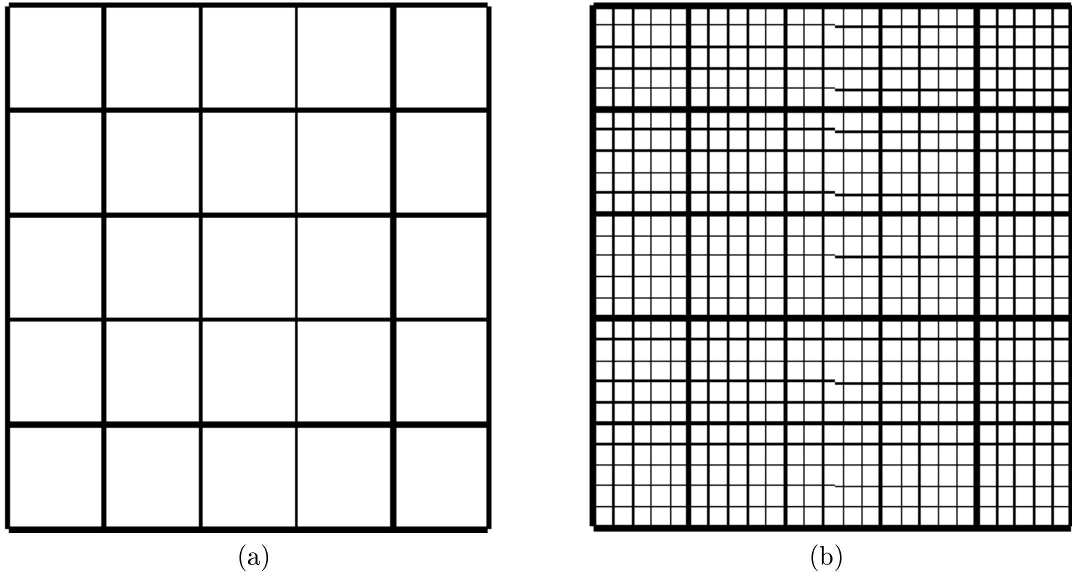


Figure 2.1: Processor mappings of SOM algorithm in[13] (a)Processor Mapping, (b)Processor mapping when each processor area divided to boxes.

In order to allow different architectures, a weight can be assigned to each vertex, indicating the speed of that processor and a weight to each edge representing the bandwidth of the communication link between processors.

In distributed memory architectures, parallel program tasks are generally thought as a limited number of command lines to be executed sequentially with interchanging sequences of computation and communication phases. This definition allows the tasks to be executed at different processors at once (no precedence constraint between tasks), exposing the advantage of distributed architectures and forms a way of assigning a load to each task. The load of a task is the average computation time between successive communication phases. With these in mind, the parallel program's tasks are represented by Task Interaction Graph (TIG), where TIG is a weighted undirected graph $G_T(V_T, E_T)$ such that

- each $v \in V_T$, represents a task,
- each $(u, v) \in E_T$ represents the data exchange between tasks u and v.

In [15], each task corresponds to a neuron in the Self-Organizing Map. The weight vectors of these neurons are selected to be positions on the unit square S. This means that, each weight vector, $W = (x, y) \in S$, is a point in S. A

task, i, is mapped to a processor P_{ij} if the weight of task i, W_i, is in the region of P_{ij}.

Similar to PCG, a weight is assigned to each vertex indicating the load of that task and a weight of each edge (u, v) representing the communication between tasks u and v.

With this representation, the mapping problem is reduced to partitioning of the vertices of a graph G_T into $|V_P|$ roughly equal parts (load balancing) such that the number of edges that are connecting vertices of G_T in different parts are to minimize communication. In other words, given a TIG $G_T(V_T, E_T)$ and PCG $G_P(V_P, E_P)$, the optimum partition can be obtained with $P : V_T \rightarrow V_P$ such that the differences between the loads of processors are minimized:

$$\begin{aligned} AvgLoad = & \quad \frac{1}{p} \times \sum_{i=1}^{|V_p|} load(i) \\ MaxLoad = & \quad max_{i \in V_p} load(i) \\ LoadImbalance = & \quad \frac{MaxLoad - AvgLoad}{AvgLoad} \times 100 \end{aligned} \tag{1}$$

At the beginning of the algorithm, weights of nodes are given randomly and these tasks (nodes) are randomly distributed to the subregions (sub squares) of the processors. Then processor loads and initial load imbalance value are calculated. Then the SOM algorithm is executed in a loop (number of predetermined times), execution is terminated when convergence load imbalance value is reached before this predetermined times. In the loop, we choose any task (node) in the area of the least loaded processor's least loaded box and find the excitation center. As we know each processor region was divided into square boxes. The excitation center is determined as follows: each input vector is compared to the weight vector of every neuron from the output layer. The neuron with the most similar weight vector is selected to be the excitation center c and is permitted to update its weights towards the values of the input vector [15]

$$||W_c^t - I^t|| = \min_{k=1}^{n} ||W_k^t - I^t|| \tag{2}$$

That is why the output layer is often called the competitive layer: each neuron fights over the input and the winner is granted an update of its weights. If a similar input is represented afterwards, this neuron will be more likely to win again because it will update its weights while the weights of the other neurons remained unchanged. The neuron that wins the competition is called "Best Matching Unit (BMU)" or "excitation center".

After the excitation center is determined, the weight vectors of some neurons are updated according to a learning function by which the neurons organize themselves. The learning function of the SOM Algorithm is generally formulated as Equation (3)

$$W_i^{t+1} = W_i^t + \epsilon^t * h(c, i, t) * ||I^t - W_i^t|| \tag{3}$$

In Equation (3),

- W_i^t and W_i^{t+1} are the weight vectors of neuron i at time t and $t + 1$ respectively,
- I^t is the input vector forwarded to output layer at time t,
- $h(c, i, t)$ is the neighborhood function and
- ϵ^t is the learning rate at time t.

Neighborhood function mainly defines the neurons that are effected and the ratio of this effect for current input at each step. The neurons that are effected at each step are specified by neighborhood diameter function θ, which is generally embedded into the definition of neighborhood function. It is mostly an exponential function and decreases with increasing time step. Generally, it defines the vicinity of an excitation center at each step. As it is seen in Equation (3), the learning rate, the neighborhood function and the difference between weight vector and input vector at time t, determine at what ratio the weight vector of a neuron should be updated.

The typical SOM is given in Figure 2.2. The steps of selecting an input, determining the excitation center and the weight vector update are repeated

```
Initialize Weight Vectors
Initialize Input Vectors
  for (i=0; i$<t_{max}$; i++) {
        Select an input
        Determine excitation center
        for (j=0; j< Number_Of_Neurons; j++)
             if (neuron j is inside the vicinity of input vector)
              Update neuron j
        }
```

Figure 2.2: Typical SOM Algorithm

until an ordered map formation is completed. The stopping criterion is generally a predetermined number of steps or the result of an error function whose value is expected to become less than a predetermined threshold value [14]. An example of Airfoil data, which has 4253 nodes, is distributed to 16 processors randomly at the beginning of the algorithm and after the execution of the algorithm, these nodes are distributed to the processors in a balanced way as shown in Figure 2.3.

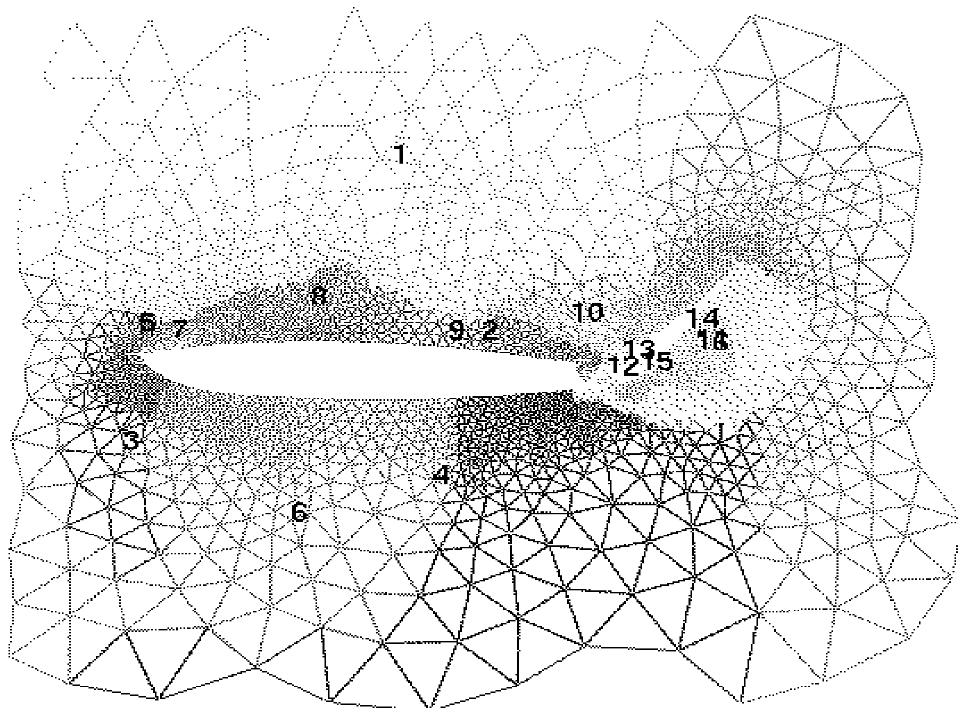


Figure 2.3: airfoil data is distributed to 16 processors.

2.3 Motivation of Our Studies

Up to know, the research on the SOM Algorithm neglected the communication overhead. We thought that, inter processor communication is inevitable for parallel computers, and this overhead will be very important during the load balancing. On the light of this hypothesis, we tried to analyze the effects of communication overhead and elements of communication cost.

Chapter 3

Communication Overhead

Recent advances in message passing implementations [31] and improved network interfaces [6, 20] have significantly reduced the software cost of message passing. However, software communication overhead still continues to be bigger problem than the hardware routing cost [24]. The interactions influencing a parallel program's overall performance can be very complex, so changing the performance of one aspect of the system may cause subtle changes to the programs behavior. For example, changing the communication overhead may change the load balance, the synchronization behavior, the contention, or other aspects of a parallel program [30].

Communication overhead is defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.

The load balancing SOM algorithm, which is discussed in previous chapter, did not include communication cost overhead or did not consider the effect of communication overhead on load imbalance. In this chapter, we studied the effect of communication cost on load balance. Our main aim is to quantify the impact of communication cost on execution time and to determine the elements of communication cost, which is very important for total cost.

3.1 Communication Cost

The time spent to communicate information from one processor to other processor is a big source of overhead when executing programs on a parallel computer. The time taken to communicate a message between two processors in the network is called *communication latency.* Communication latency is the sum of the time to prepare a message for transmission and the time taken by the message to travel the network to its destination.

$$CommunicationCost = \sum_{(i,j) \in E_T} \alpha(i,j) + l * t_w \tag{1}$$

In Equation (1),

- $\alpha(x, y)$ is the startup time needed for processors x to reach the processor y,
- l is the length of message that will be sent to the neighbor processor,
- t_w is per-word transfer time.

To see the effect of various parameters on the communication cost, some experiments were performed. These parameters are the number of communicating processors, the number of messages that are sent to the other processors, the effect of communication type, the effect of message length, the size of load and the condition of receiver processor.

3.2 Solution of Laplace's Equation by Jacobi Iteration

For our experiments, we used the solution of Laplace's equation with Jacobi iteration. In this section, we will briefly explain numerical solution of Laplace's equation. This problem is chosen since many numerical computations involve similar computational patterns.

There are a variety of different methods of simulating the effective conductivity of an inhomogeneous medium. One of them is the Laplace's equation. Laplace's equation is the second order partial differential equation that is widely used in many disciplines, because its solutions (known as harmonic functions) occur in problems of electrical, magnetic and gravitational potentials, of steady-state temperatures, and of hydrodynamics. Laplace's equation states that the sum of the second-order partial derivatives, with respect to the Cartesian coordinates, equal to zero. Laplace's equation is given as

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{2}$$

We want to solve for $u(x, y)$, an unknown function subject to conditions on its boundaries. We defined a square grid consisting of points (x,y) (Figure 3.2) and use Jacobi iteration to compute the value of $u(x, y)$ at the grid point. With Jacobi iteration, a grid value can be approximated by a centered difference using the Equation 3

We used 5-pointed stencil to represent the position of each node (Figure 3.2).

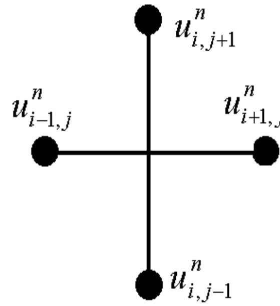


Figure 3.1: 5-pointed stencil.

$$u^{n+1}_{i,j} = \frac{u^n_{i+1,j} + u^n_{i-1,j} + u^n_{i,j+1} + u^n_{i,j-1}}{4}; i = 1, ..., m; j = 1, ..., m \tag{3}$$

where n and $n+1$ are the current and next iterations.

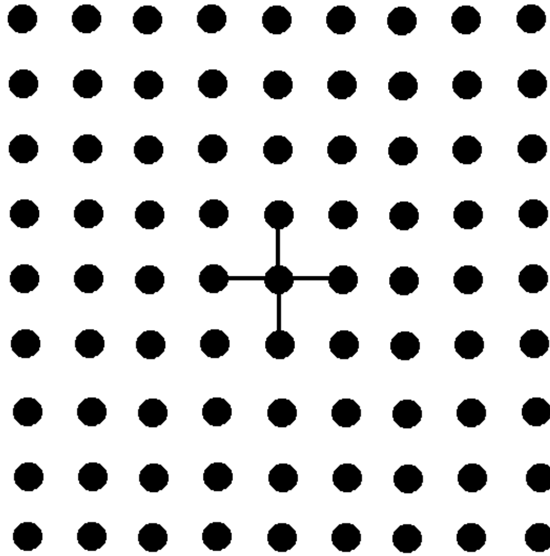


Figure 3.2: Square grid.

The code fragment given in Figure 3.3 describes the main loop of the iterative solver where, the value at a point is replaced by the average of the north, south, east and west neighbors at each iteration, (a 5-pointed stencil is used to keep example simple). Boundary values do not change. We focus on the inner loop, where most of the computation is done.

```
from left to right nodes
 from up to down nodes
  u[i][j] = 0.25*u[i+1][j]+u[i-1][j]+u[i][j+1]+n[i][j-1])
```

Figure 3.3: Main loop of the iterative solver.

We used LAM implementation of *Message-Passing Interface (MPI)* program, which provides a parallel processing environment for a network of independent computers.

3.3 Experiments

3.3.1 Experiment 1: Effect of Communication Cost on Total Cost.

Our main aim in this experiment is to quantify the overhead of communication cost. To understand the communication overhead on parallel computers, we performed some experiments. On the solution of *Laplace Equation with Jacobi Iteration* experiment, each processor has 100 nodes (Figure 3.4) and each node is initialized randomly at the beginning of the program. Each processor makes Jacobi Iteration for each of its nodes.

In this experiment, each processor has to communicate with its neighbor processors to be able to finish its own tasks, because it needs the values of neighbor processor's edge node values. After all nodes' new values are calculated, edge nodes values are sent to neighbor processors, and neighbor processors edge nodes values are received by the owner processor. For example, *P1* needs values of the first column of *P2* and values of the first row of *P3* to be able to finish its tasks (Figure 3.4). It is expected that this will cause some extra overhead.

To see the effect of communication overhead, different values for communicating processors is experimented. Each experiment is executed six times and average of the values is taken (Figure 3.5).

The same experiment is performed by using blocking communication type and non-blocking communication type and the effect of communication type on total cost is investigated.

We know that when the number of processors is one, communication cost is zero, so the total cost of this experiment with one processor gives only the computation cost of the program (Figure 3.5(a)). When the number of processor is two (Figure 3.5(b)), there is one communication on the side of each processor because each processor is sending one message and receiving one message from its neighbor processor. When the number of processors is four (Figure 3.5(c)), each processor is communicating with at most its two neighbor processors. This means that each processor is using at most two send and two receive messages. When the number of processors is nine (Figure 3.5(d)), each

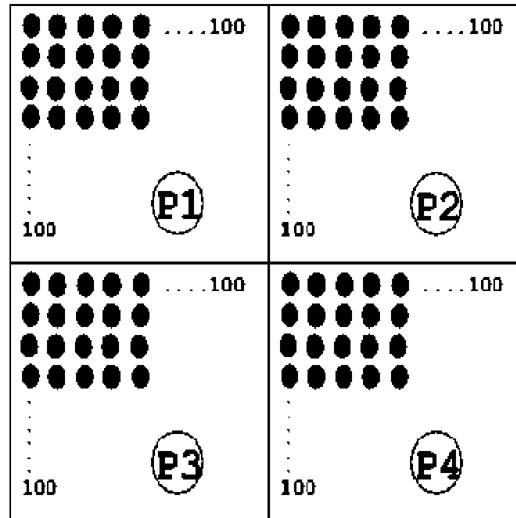


Figure 3.4: Nodes are distributed to processors, each processor has 100×100 nodes. Weight of nodes are given randomly at the beginning of the program.

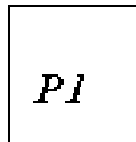
processor is communicating with at most its four neighbors. This means that, there will be at most four send and four receive messages.

When we look at the results of this experiment, we can say that every communicating processor brings some extra communication overhead to the total cost. So the number of communicating processors has a big effect on total cost. This means that communication cost can not be neglected. If we want to have better load balance values, we have to take communication cost into account. The second important result of this experiment is blocking communication's cost is lower than the non-blocking communication's cost.

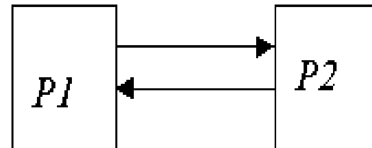
3.3.2 Experiment 2: Size of Load Experiment

Our main aim for this experiment is to see the effect of "load size" on total cost and effect of "communication type".

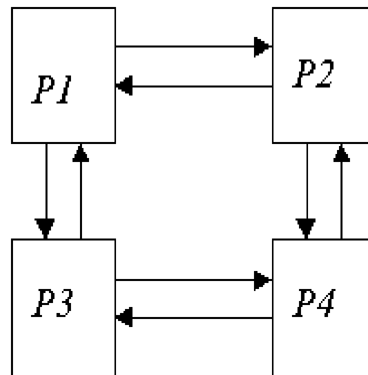
To see the effect of the load size on the communication cost, different load sizes are experimented. The number of nodes per processor increased from $10 \times 10 = 100$ to $60 \times 60 = 3600$. We performed the same experiment with non-blocking communication type and blocking communication type to be sure about the effect of communication type.



(a) 1 processor, there is no communication



(b) 2 processors communicating with each other



(c) 4 processors are communicating with each other

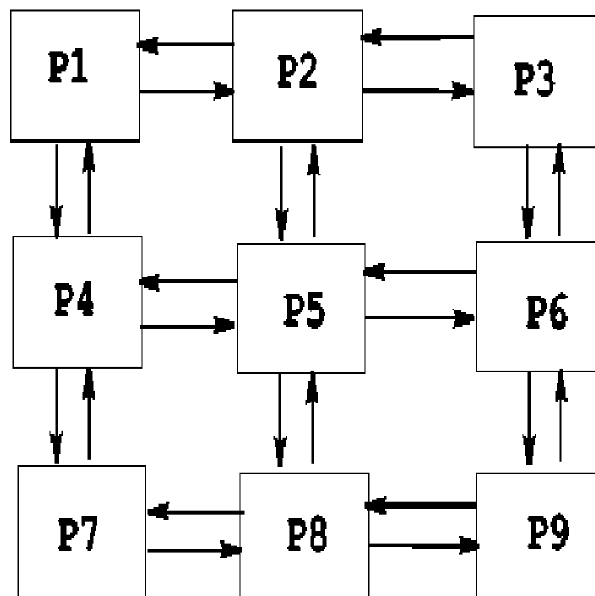


Figure 3.5: (d) 9 processors are communicating with each other

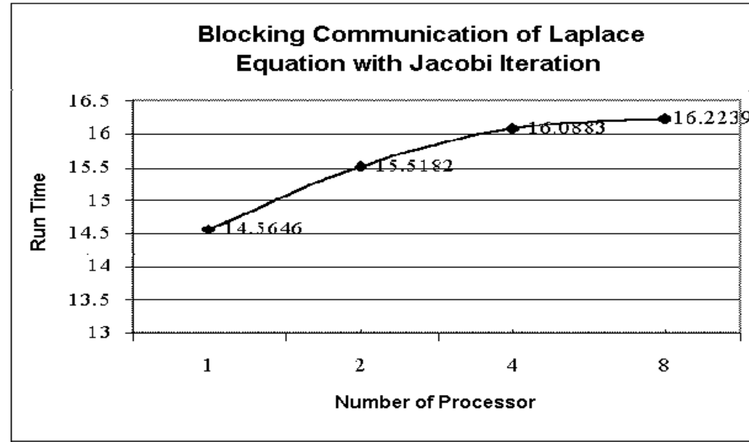


Figure 3.6: Blocking communication of Laplace Equation.

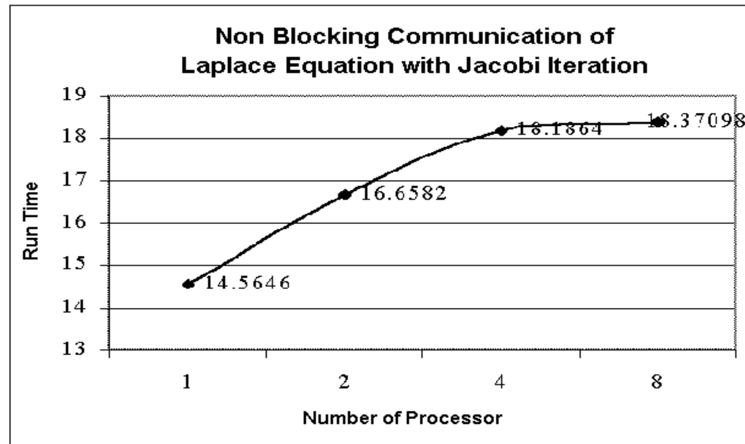


Figure 3.7: Non-blocking communication of Laplace Equation

In this experiment while the size of load increasing n^2 times, the communication length is increasing n times. For this reason, the slopes of the graphs in Figures 3.8 and 3.9 are increasing with the slope of n^2.

As a result of this experiment, we can say that there is a direct relation between the load and the computation cost; when the size of load is increased, communication cost also increases but not with the same proportion. Second result of this experiment is non-blocking communication cost is more expensive than blocking communication cost.

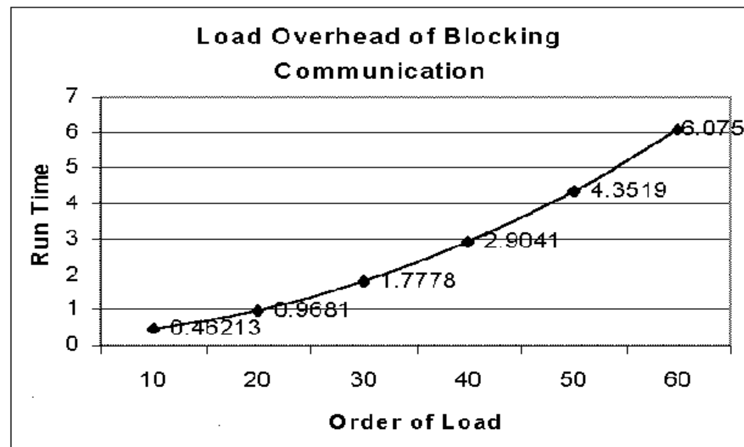


Figure 3.8: Load overhead of blocking communication

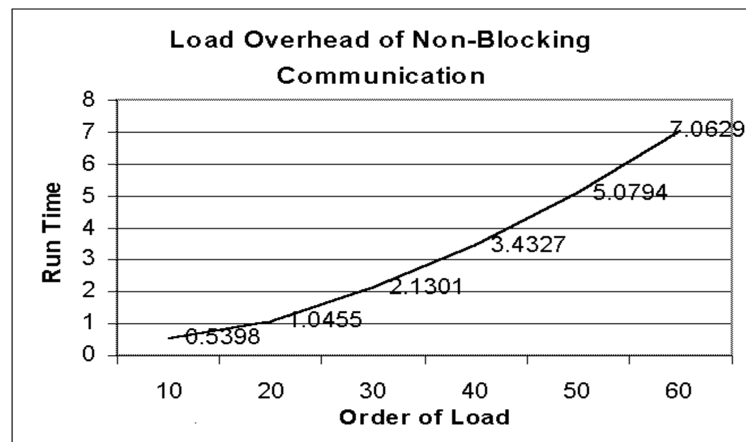


Figure 3.9: Load overhead of non-blocking communication

3.3.3 Experiment 3: Effect of the Number of Communicating Processors

In this experiment, we tried to see the effect of startup time of when each processor is busy. Startup time is the time needed to arrange data into a message and inject it into the communication network.

In this experiment, we increased the number of communicating processors from 0 to 6 and took the results (Figure 3.10).

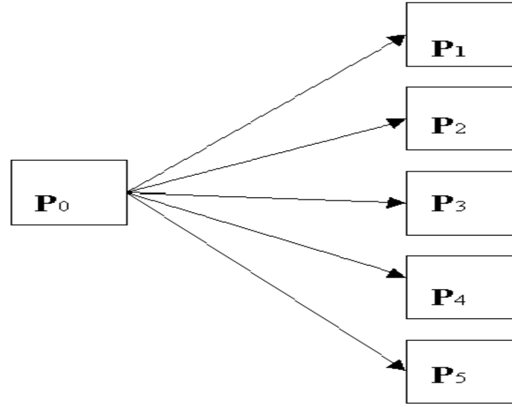


Figure 3.10: P_0 is communicating with 5 other processors

After first processor (P_0) finished its computational job, it sends the edge nodes values of the right column nodes to its neighbor processors. The number of neighbor processors is changed from 0 to 6. For example, if the number of neighbor processors is five, first processor who makes the computation sends the edge nodes values of the right column nodes to the five different processors that have neighborhood relation with the first processor. When the number of processor is zero, it means that communication is zero. Load size is not changed during this experiment. There are two versions of this experiment. In the first version (Figure 3.11), the processors who receive data (edge nodes values) are idle, they are not doing any computation. However, the other processors that receive data from the sender processor are making the same computation as the sender processor in the second version (Figure 3.12). Besides, the same experiment is done to see the effect of non-blocking communication (Figures 3.13 and 3.14).

As a result of this experiment, it can be said that, blocking communication is better than non-blocking communication. The number of communicating processors (neighbor processors) has important effect on total cost and communication cost, so it can not be neglected. There is a direct relation between "the communication cost" and "the number of communicating processors". When the number of communicating processors is increased, the communication cost is increasing. For this experiment, if the computation cost is 100 seconds, then each communicating processor brings approximately 0.32 seconds overhead. Non-blocking communication brings extra overhead at the initial step. Except

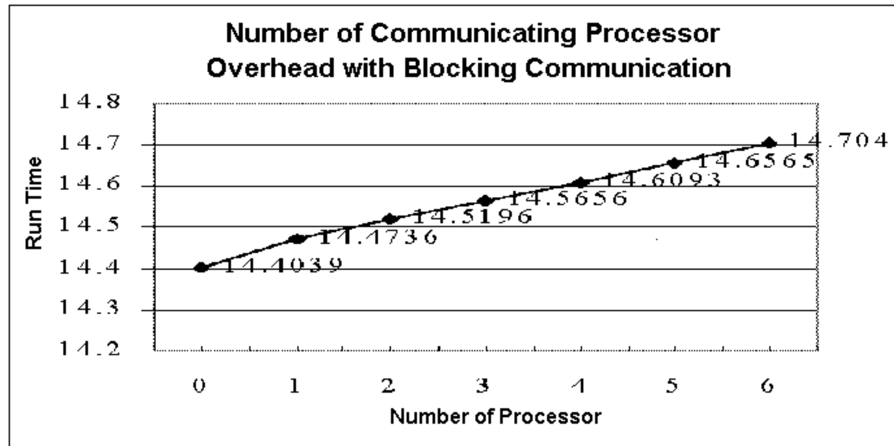


Figure 3.11: Communicating processors number overhead of blocking communication

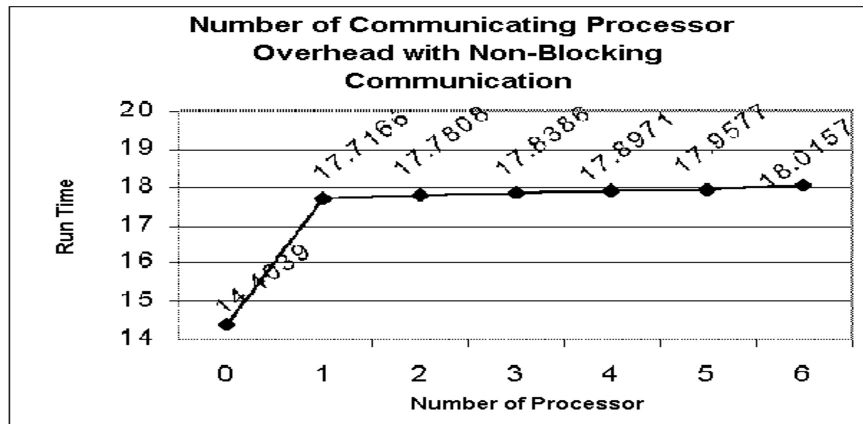


Figure 3.12: Communicating processors number overhead of non-blocking communication

the initial overhead of non-blocking communication, both communication types behaves similarly.

On the other hand, the receiver processors' condition (whether it is idle or not) have no relation with the number of communicating processors.

3.3.4 Experiment 4: Effect of Number of Sends

In this experiment, two processors are used with different number of messages. Instead of different processors, the sender processor communicates with the receiver processor. In this way, effect of communication cost is investigated,

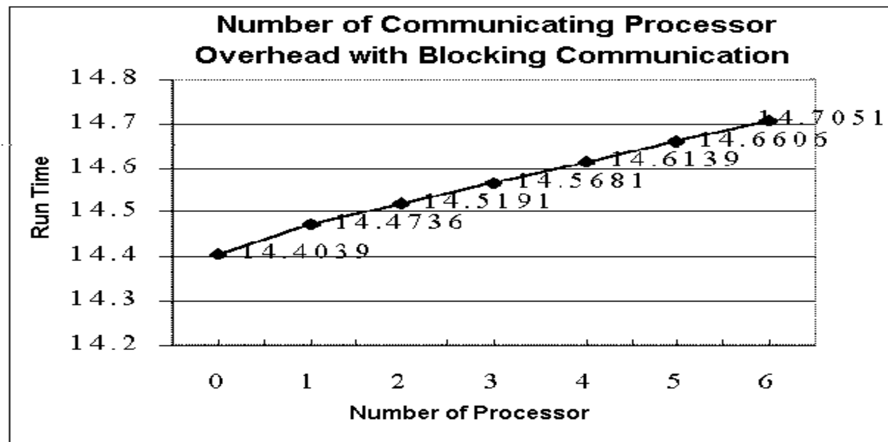


Figure 3.13: Communicating processors number overhead of blocking communication, while receiver processors making computation

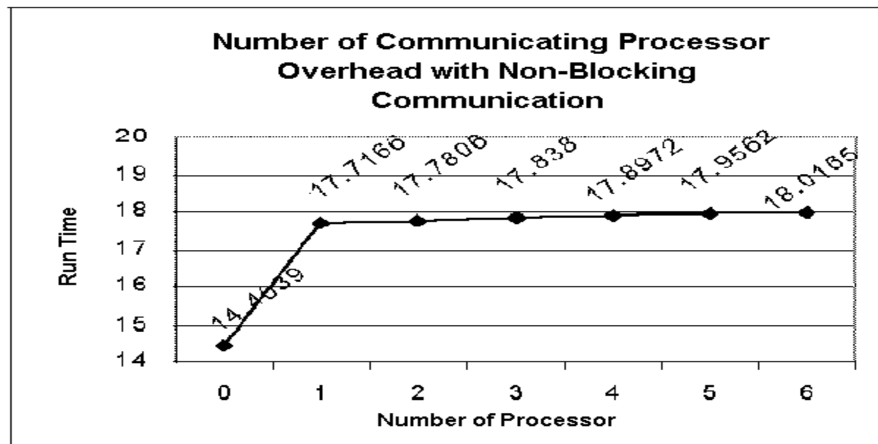


Figure 3.14: Communicating processors number overhead of non-blocking communication, while receiver processors making computation

the number of receiver processor is not changed, but the number of message is changed from one to six (Figure 3.15).

This experiment has two versions. In the first version, one processor making computation and sending edge nodes values to the other processor (Figures 3.16 and 3.17). In the first version, the receiver processor is idle. In the second version, both processors are making computation (Figures 3.18 and 3.19).

As a result of this experiment, we can say that "number of messages", which is send from one processor to the other, is very important for total cost; every send message brings some extra overhead. It has direct relation with execution

Figure 3.15: Sender processor sending 1 to 6 messages to the receiver processor.

time of a program. Non-blocking communication is more expensive than blocking communication. The receiver processor's condition is very important. If the receiver processor is not idle, the execution time of the program obviously increases, because the receiver processor is waiting to finish its current job.

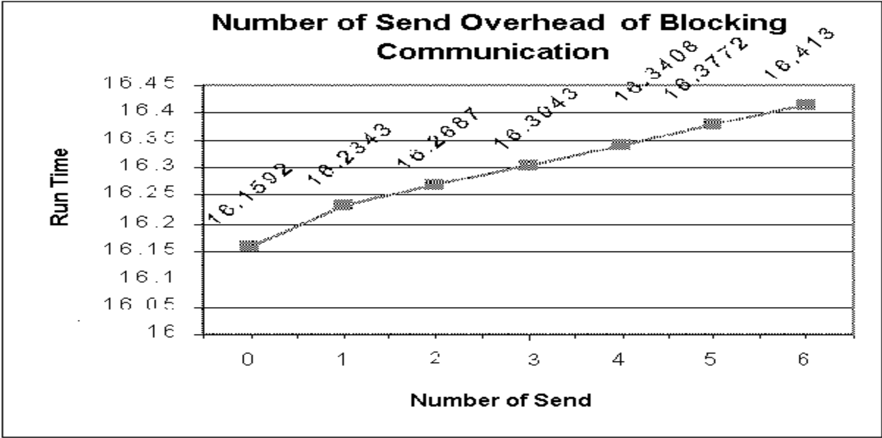


Figure 3.16: Number of send overhead of blocking communication when the receiver processor is idle.

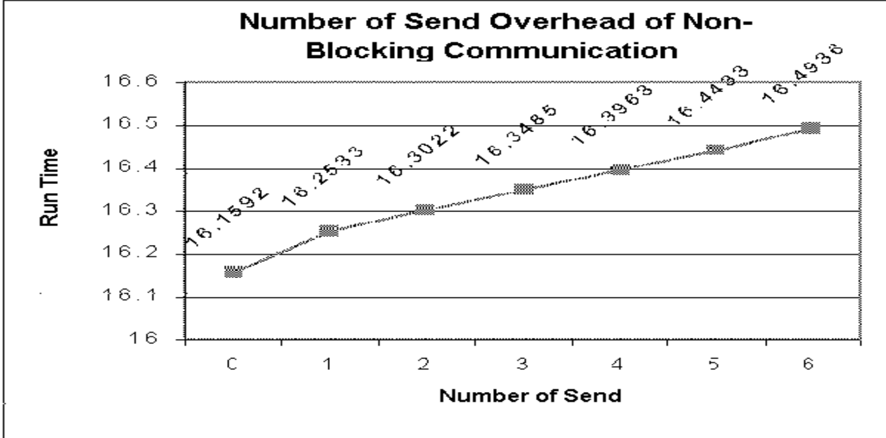


Figure 3.17: Number of send overhead of non-blocking communication when the receiver processor is idle.

3.3.5 Experiment 5: The Message Length

This experiment is performed to see the effect of the "message length" over total cost of a program. We know that one of the components of the communication cost is message length. Our main aim in this experiment is to detect whether the message length has an important effect on communication cost or not.

According to this experiment, each processor is communicating with its six neighbor processors. We changed the length of the message from 1 to 100 floating point numbers, and investigated the effect of message length over

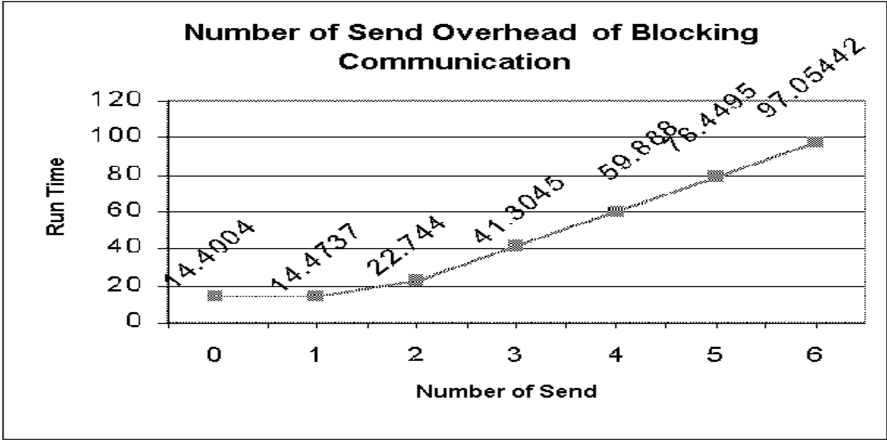


Figure 3.18: Number of send overhead of blocking communication, while the receiver processor making computation.

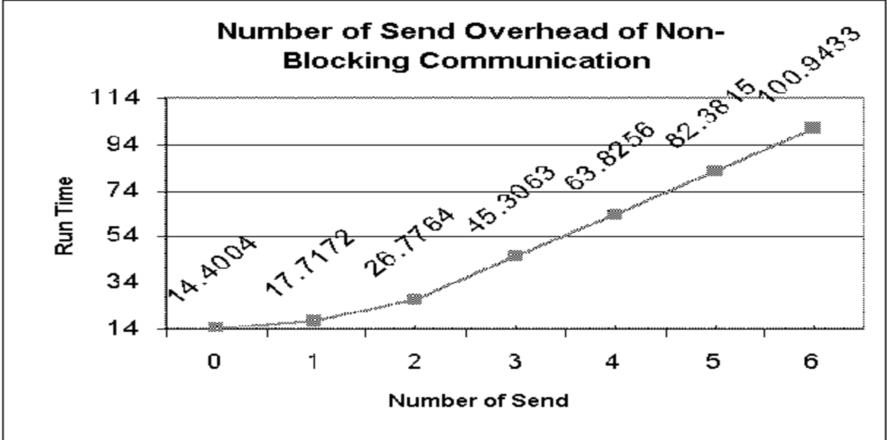


Figure 3.19: Number of send overhead of non-blocking communication, while the receiver processor making computation.

communication cost and total cost.

According to the results, we see that the message length has not a significant effect on communication cost. In Figure 3.20, it is seen that when the message length is increased, the execution time also increases but with a much less rate. As a result of this experiment, we can say that, when the computation cost is 100 seconds, each unit of message length brings approximately 0.000410059 seconds extra overhead to the total cost. This overhead is negligible when compared to the startup time of communicating processors.

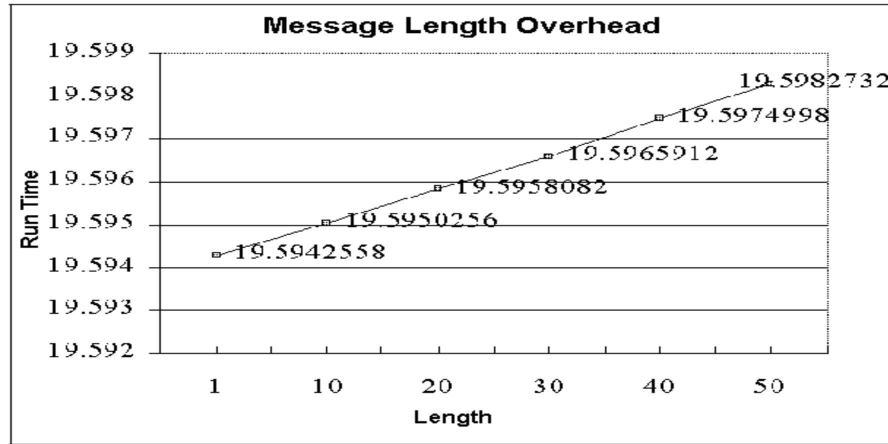


Figure 3.20: Length of message overhead on communication cost.

3.4 Configuration of the Machine

We used the Beowulf [5] cluster, called borg, at Bilkent University, Department of Computer Engineering. Beowulf systems are high performance parallel computers built with cheap commodity hardware connected with a low latency and high bandwidth interconnection network and equipped with free system software, such as GNU/Linux or FreeBSB [3, 4]. The hardware of this cluster consists of three components:

- **Nodes** There are 32 identical nodes with Intel Pentium II 400 Mhz CPU, 64 MB PC100 RAM, 6GB UDMA IDE hard drive and Intel EtherExpress Pro 10/100 MIC.,
- **Interconnection Network** The interconnection network is a 3COM SuperStack II 3900 smart switch which has 36 100Base-TX ports and a Giga bit uplink. The ports connect to nodes and uplink connect to the interface machine.,
- **Interface Computer** The interface computer is a workstation with Pentium III 500 Mhz, 512 MB Ram and 26 GB hard drive. It has a gigabit NIC which connects to the uplink of switch and a fast Ethernet to connect to the Net. The interface Computer provides communication with developers through console and network.

The Borg uses Debian GNU/Linux distribution as the operating system. This distribution has a host of development tools ready to-use, that makes it preferable.

3.5 Conclusion

Result of experiments are as follows: non-blocking communication type is more expensive than the blocking communication type. Load size is very important for total cost. Number of communicating processor is very important and has direct relation with communication cost and total cost. Receiver processors' condition is very important for the execution cost. The number of sends has a direct relation with total cost. If the receiver processor is not idle, the execution time increases, so the condition of receiver processor is very important. The message length has not a significant effect on total execution cost.

Experiment results show that in communication performance the sensitivity to software overhead is much stronger than the other factors of communication. Similar behavior is observed in [30]. The major factor that effects the communication cost is the number of communicating processors and the number of messages.

For these experiments and this machine configuration, communication cost can be formulated as in Equation 4.

$$CommunicationCost = \sum_{(i,j)\in E_T} (0.32 + l \times 0.000410059) \times Computationcost/100 \tag{4}$$

Chapter 4

Communication Optimized Mapping

On the light of the previous chapter's results we have seen that communication cost is very important, it brings some overhead and it can not be neglected. Another important result of the previous chapter is "communication cost overhead should be minimized".

In this chapter, new processor mappings are proposed to reduce the communication cost.

4.1 Processor Mapping

Processor mapping is one of the most complex problems in parallel computing. The efficiency of any parallel algorithm that assigns tasks to the processors of multi-computer system can be measured by checking its load imbalance efficiency and its total execution time. Total execution time can be defined optimal, if summation of computation and communication time is minimal. Communication time can be zero, if we assign all tasks to a single processor, which is the worst case of parallel approach, because it will give the biggest load imbalance value.

While distributing tasks to the processors, our major aim is to give equal

amount of load to each processor. We know that the load of each task is not the only factor of total processor load. Communication of processors with each other also cause some overhead. If we take the communication overhead of processors into account, we will have fair and real load imbalance. In the ideal case, the total load of a parallel program is distributed to p processors where each processor gets the $\frac{1}{p}$th part of the total load. During the distribution of loads to the processors, if we do not ignore inter processor communication overhead which is inevitable during data exchange between processors, we can decrease job completion time difference between processors.

In order to efficiently run a parallel program, tasks should be assigned to processors in such a way that the load of every processor is more or less equal, and at the same time, the amount of communication between processors should be minimum.

4.2 Models for Representing Processors

To reduce the communication cost, we should decrease inter processor communication. One approach of decreasing inter processor communication is decreasing the number of neighbor processors of each processor. If we decrease the number of neighbor processors, we will automatically decrease number of communicating processors, because neighbors of any node probably will be in the owner processor or neighbor processors of the owner processor. Based on this, two more processor mappings are proposed in addition to *Square processor mapping* (Figure 4.1(b)). These are *Hexagonal processor mapping* (Figure 4.1(d)) and *Staggered processor mapping* (Figure 4.1(c)).

In the implementation, the dimension of the mesh and the type of the processor mapping is user-specifiable, as opposed the previous approaches. For this, we initially produced the square meshes (boxes). Then, according to the position of each box and the mapping type choice, the owner processor of each box is defined. With this approach, new processor mapping types can be added to `Processor` class of our program by defining the new mapping type's method.

The main aim of usage of square boxes (mesh) is to gain time during the execution phase of the program. According to SOM algorithm, we are choosing

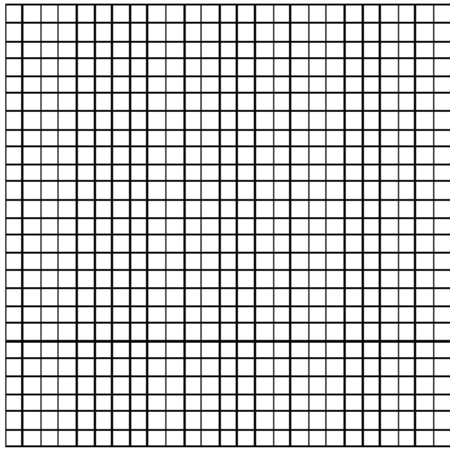
any node in the least loaded processor (which has minimum load) randomly and finding the nearest node of randomly chosen node as the excitation center of that iteration. However, the nearest node finding phase, scanning all nodes to determine whether it is the nearest node or not will be a very costly operation. To avoid from this costly operation, we used a two dimensional mesh Figure 4.1(a). With this approach, every box of the mesh is given to a processor according to the processor type. The owner processor of each box is determined by *Processor* class of our program. With this method, to find the nearest node of the randomly chosen node, which is in the least loaded processor region, we search the excitation center beginning from the subregion that owns the input vector and continue in all surrounding subregions until the excitation center is found. This method is also known as *grid method* in the literature. By this method, we find the excitation center more quickly than the *sequential* scanning method. The identification number, the position of each box (subregion) and the processor region's of two dimensional coordinates are specified in a row wise fashion.

Commonly used symbols during the calculation of the processor identification number are as follows:

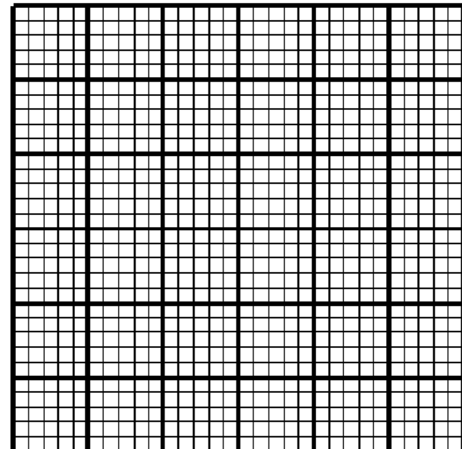
- **nBPP.i** represents the number of boxes per processor in x axis,
- **nBPP.j** represents the number of boxes per processor in y axis,
- **ppos.i** represents the processor position in x axis,
- **ppos.j** represents the processor position in y axis,
- **nPE.i** represents the number of processor in x axis,
- **nPE.j** represents the number of processor in y axis.

It should be noted that all divisions are integer division in assigning processor identification numbers.

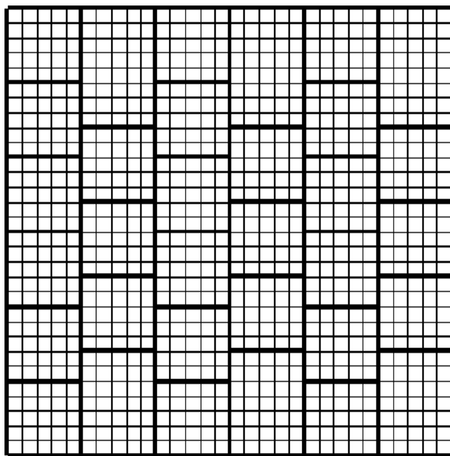
Square processor mapping: In square processor mapping, each processor has eight neighbors. So, the number of processors a processor may communicate will be at most eight (Figure 4.1(b)). Each processor has 25 square boxes, except edge and corner processors.



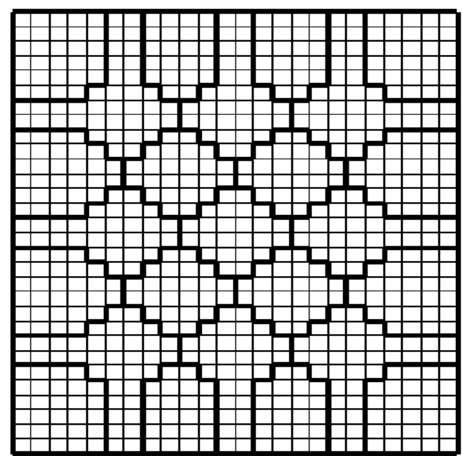
(a)



(b)



(c)



(d)

Figure 4.1: Processor Mappings. (a) Square Mesh, (b) Square Processor Mapping, (c) Staggered Processor Mapping, (d) Hexagonal Processor Mapping

In this mapping, the owner processor of each box is numbered starting from 0 in a row wise fashion. Processor identification number of each box is assigned as follows: for square processor mapping, number of Box Per Processor (nBPP) in both dimension is 5. We can find the *processor position (ppos)* of each box with Equation 1.

$$\begin{aligned} ppos.i &= (i/nBPP.i) \\ ppos.j &= (j/nBPP.j) \end{aligned} \tag{1}$$

The number of Processors (nPE.i and nPE.j) in both directions specified by the user. So, the processor identification number can be found with equation 2.

$$Processorid = ppos.i \times nPE.j + ppos.j \tag{2}$$

Hexagonal Processor Mapping: In hexagonal processor mapping, each processor has six neighbor processors, except corner and edge processors. The neighbor of nodes may be in the owner processor or in one of the six neighbor processors. So, the number of communicating processors will be at most six (Figure 4.1(d)). Each processor area includes 24 square boxes, except corner and edge processors. The hexagonal processor mapping is given in Figure 4.1(d)

In hexagonal processor mapping, the owner processor of each box is numbered in a row wise fashion. For square processor mapping, *number of Boxes Per Processor(nBPP)* in both dimension is 6.

The algorithm in Figure 4.2 used to find the processor identification number of each box in hexagonal processor mapping.

Staggered Processor Mapping: This mapping is similar to hexagonal processor mapping. Every processor region has six neighbors. However, its application is simpler than the hexagonal mapping. Each processor area has 25 boxes, except edge and corner processors (Figure 4.1(c)).

In this mapping, the owner processor of each box is numbered in a row wise fashion. The processor identification number of each box is calculated

```
for(i=0 to  number of Boxes in x direction )
  for(j=0 to  number of Boxes in y direction)
    if(i > 15) then ii=(i mod 16)
    else ii=i
        if(((ii mod 10) < 6) or (ii (mod 10) > 7)
          and(i mod 16 < 14)) then
             (yy = j(mod 6))
             (xx = ii(mod 8))
        if(((yy == 0 or yy == 5) and  (xx == 2 or xx == 3))or
               ((yy == 1 or yy == 4)
           and  (xx == 1 or xx == 2 or xx== 3 or xx ==4)) or
               (( yy == 2 or yy == 3 )
           and  (xx==0 or xx==1 or xx==2 or xx==3 or xx==4 or xx==5))
                or (j<2 ) or (j>nBox.j-3) or
                (i<2) or (i>nBox.i-3))  then
            ppos.i=(i/8)*2 and  ppos.j=j/6
         else
         ppos.i =(((i-4) /8)*2 +1) and  ppos.j =(j-3)/6

      if ((j > (nBox.j-4)) or (j < 3))
                if((i mod 8) == 6)
                    ppos.i =((i /8)*2 );
                if(j>(nBox.j-4) )
                    ppos.j = (ppos.j+1);
                else  if((i \% 8) == 7)
                    ppos.i =(((i /8)*2) +2);
                if(j>(nBox.j-4) )
                    ppos.j = (ppos.j+1);
        div =ppos.i/2;
      if(ppos.i mod 2 == 0 )
            Pnumb= ((div *nPE.j)+(div*(nPE.j - 1))+ppos.j);
        else
            Pnumb= ((div+1) *nPE.j)+(div*(nPE.j-1))+ ppos.j;
        Pmesh->grid[i][j].pe(Pnumb,ppos);
```

Figure 4.2: Finding processor identification number of each box in hexagonal processor mapping

```
for(i=0 to  number of Boxes in x direction)
  for(int j=0 to number of Boxes in y direction)
       if (i (mod 10) < 5) than
          ppos.i=(i/5) and  ppos.j=j/5
     else
          if(j > (nBox.j-5))
               ppos.j=(j-5)/5
            else
               ppos.j=(j-2)/5
            ppos.i=(i/5)
div =ppos.i/2
if(ppos.i (mod 2) is equal to 0 )
   Processor id = (div *nPE.j)+ div*(nPE.j - 1) +ppos.j
else
   Processor id = (div+1) *nPE.j + div*(nPE.j-1)+ ppos.j
```

Figure 4.3: Finding processor identification number of each box in staggered processor mapping

as follows: for staggered processor mapping number of Box Per Processors (nBPP) in both dimensions are 5.

The algorithm in Figure 4.3 is used to find the processor identification number of each boxes in staggered processor mapping.

4.3 Discussion of Processor Connection Models

Implementation of different processor mappings become easier by *Grid Method*. If we examine processor mappings, we can see that in square processor mapping each processor has 8 surrounding neighbors, where as in hexagonal and staggered processor mappings each processor has 6 surrounding neighbors. When the top left corner region of each mapping is examined, in square processor mapping it has 3 surrounding neighbors, whereas in staggered processor mapping, it has 2 surrounding neighbors and in hexagonal processor mapping it has one surrounding neighbor. Since reducing the number of neighbor processors reduces communication overhead, in hexagonal and staggered processor mappings, the number of communicating neighbors for each processor is decreased 25% as compared to square processor mapping. In terms of the total volume of communication, the processor mappings are ordered from the best to the worst

as hexagonal, staggered and square processor mappings. Hexagonal processor mapping performs better than the staggered processor mapping for the edge and corner processors.

The efficiency of the solution produced by the mapping is highly related with the representation of processors and tasks. In order to solve the mapping problem efficiently, processors and tasks must be represented by suitable models that exploit the features of the architecture of the multi-computer and properties of parallel program tasks.

By proposing the staggered processor mapping and hexagonal processor mapping, we tried to minimize the communication cost overhead of each processor. With this approach, we topologically decreased the number of neighbor processors and hence, the inter processor communication traffic.

4.4 Improved SOM Algorithm for Load Balancing

In Figure 4.4 communication overhead is included to the total cost during the execution of SOM algorithm.

Initialize square mesh S and define owner processor of each box according to user processor type choice
for all neurons i **do**
  initialize weight vectors w_i randomly.
  initialize position of each neuron w (x, y coordinates $\in S$) randomly.
end for
for all processors i **do**
  calculate load of each processor
end for
set initial and final values of diameter θ_i and θ_f
set initial and final values of learning constant ϵ_i and ϵ_f
for $t = 0$ to t_{max} **do**
  let S_p be the region of the least loaded processor p
  select a random input vector $v = (x, y) \in S_p$
  determine the excitation center c such that for all neurons n
  $\|w_c - v\| = min\|wn - v\|$
  for $d = 0$ to θ **do**
    for all neurons k with distance d from center c **do**
      update positions of $w_k \leftarrow w_k + \epsilon e^{\frac{-d}{2\theta^2}} \|w_k - v\|$
    end for
  end for
  update diameter$\theta \leftarrow \theta_i \left(\frac{\theta_f}{\theta_i}\right)^{\frac{t}{t_{max}}}$
  update learning constant $\epsilon \leftarrow \epsilon_i \left(\frac{\epsilon_f}{\epsilon_i}\right)^{\frac{t}{t_{max}}}$
  update load of each processor which includes communication overhead
  $P_k(Load) \leftarrow \sum_{i=1}^{|V_n^k|} v_i^k(load) \times \left(1 + P_k(neigh) \times \frac{t_{comm}}{t_{comp}} \times 100\right)$
end for

Figure 4.4: SOM Algorithm that balances communication overhead and computational load.

Chapter 5

Comparison of Processors Mappings

5.1 Related Programs Used for Comparison

Algorithms that find a good partitioning of highly unstructured graphs are critical for developing efficient solutions for a wide range of problems in many application on both serial and parallel computers. For example, large-scale numerical simulations on parallel computers, such as those based on finite element methods, require the distribution of the finite element mesh to the processors. This distribution must be done so that the number of elements assigned to each processor is the same and the number of adjacent elements assigned to different processor is minimized. The goal of the first condition is, to balance the computations among the processors. The goal of the second condition is to minimize the communication resulting from the placement of adjacent elements to different processors. Graph partitioning can be used successfully satisfy these conditions by first modelling the finite element mesh by a graph and then partitioning it into equal parts [23]. We used MeTiS program, which is the best of graph partitioning algorithms, for comparison. Properties of MeTiS programs can be seen from Appendix A.

5.2 Comparison of SOM Mappings

Before comparison of SOM mappings with other algorithms, we compared the SOM mappings with each other. We used airfoil, biplane and shock data. In this section, we only compared the number of communicating processors (startup times), if the number of nodes per processor is not high, each processor may not communicate with all of its neighbor processors.

Number of nodes	Data	Number of processors	Shape of processor	PMeTis	KMeTis	SOM
21,701	biplane	60	Square			3-8
21,701	biplane	60	Staggered			2-6
21,701	biplane	60	hexagonal			1-6
21,701	biplane	60		2-9		
21,701	biplane	60			2-8	
4,253	airfoil	25	Square			3-8
4,253	airfoil	25	Staggered			2-6
4,253	airfoil	25	hexagonal			2-6
4,253	airfoil	25		2-8		
4,253	airfoil	25			2-6	
36,476	shock	60	Square			3-7
36,476	shock	60	Staggered			2-6
36,476	shock	60	hexagonal			1-6
36,476	shock	60		2-13		
36,476	shock	60			2-7	
4,253	airfoil	60	Square			2-9
4,253	airfoil	60	Staggered			2-7
4,253	airfoil	60	hexagonal			2-7
4,253	airfoil	60		2-8		
4,253	airfoil	60			2-7	

Table 5.1: Number of communicating processors according to the algorithms and processor shapes.

If we look at Figure 5.1, we can see that hexagonal and staggered processor mappings perform better than the square processor mapping. Because in hexagonal and staggered processor mappings, the number of communicating processor is smaller than that of the square processor mapping. In other words, these two new mappings decrease the inter processor communication. When the number of nodes per processor is increased, effectiveness of the hexagonal processor mapping is becoming clear. Except the edge and corner processors,

the average communication traffic is decreased by 25 percent. At the edge and corner processors, communication traffic is better than the other processors.

5.3 Comparison of SOM Processor Mappings with pMeTis and kMeTis

In this section, we examine the impact of various processor mappings. By new processor mappings (hexagonal and staggered processor mappings), we tried to decrease topological neighborhood.

As it is explained in Chapter 4, the startup time has a considerable effect on total execution cost as compared to the message length. So, while we are comparing different processor mappings, we ignored the length of a message. Therefore, startup time of processors to communicate with each other is the main factor of communication cost.

To be able to get exact loads of each processor, communication cost overhead of each processor added to their loads. After calculating real loads, real load imbalance values are calculated. We tried to understand the real load imbalance values of processors with different processor mappings of SOM algorithm, PMeTis and KMeTis programs. We tried to see the behavior of load imbalance graphs of different algorithms (Figure 5.1, 5.2 and 5.3), while the amount of communication cost takes 0.1% to 3% of the computation cost.

To see the behavior of load imbalance graphs, we made the same experiments with different data sets. These data sets are airfoil, biplane and crack data sets. Properties of data sets are as follows:

- *airfoil:* airfoil data set has 4253 nodes and 12289 edges. It is in the FEM(2) graph class.
- *biplane:* biplane data set has 21701 nodes and 42038 edges. It is in the Square grid graph class.
- *crack:* crack data set has 10240 nodes and 30380 edges. It is in the FEM(2) graph class.

Communication cost (%)	KMeTis	PMeTis	Square	Staggered	Hexagonal
0.000	3.456384	1.105102	0.517282	0.517282	0.517282
0.001	3.353363	1.5482203	0.661528	0.58536	0.653423
0.002	3.25116	1.988124	0.804471	0.652865	0.788315
0.003	3.173581	2.424919	0.946127	0.719782	0.921969
0.004	3.273653	2.858605	1.086517	0.786134	1.054407
0.005	3.37294	3.289229	1.225653	0.851922	1.185643
0.01	3.857	5.397416	2.611502	1.538973	1.824364
0.02	5.459783	9.401321	5.608154	3.020489	3.020473
0.03	7.141433	13.14493	8.364479	4.388699	4.119251

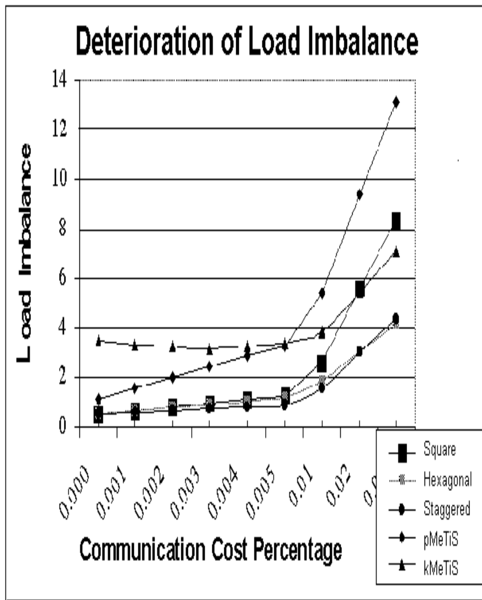
Table 5.2: airfoil data is distributed to 25 processors, weights of nodes are all 1.

Communication cost (%)	KMeTis	PMeTis	Square	Staggered	Hexagonal
0.000	3.309905	1.068845	0.694701	0.525818	0.618829
0.001	3.3367536	1.169669	0.830875	0.693983	0.752543
0.002	3.424773	1.317603	0.965796	0.860708	0.871599
0.003	3.48162	1.465406	1.09993	1.02601	0.989528
0.004	3.59455	1.630922	1.329224	1.189909	1.106346
0.005	3.750285	1.872792	1.639284	1.352422	1.222069
0.01	4.513426	3.057468	3.207681	2.144796	1.784785
0.02	5.966016	5.309942	6.144634	3.633502	2.836284
0.03	7.328012	7.418984	8.89	5.011264	3.799549

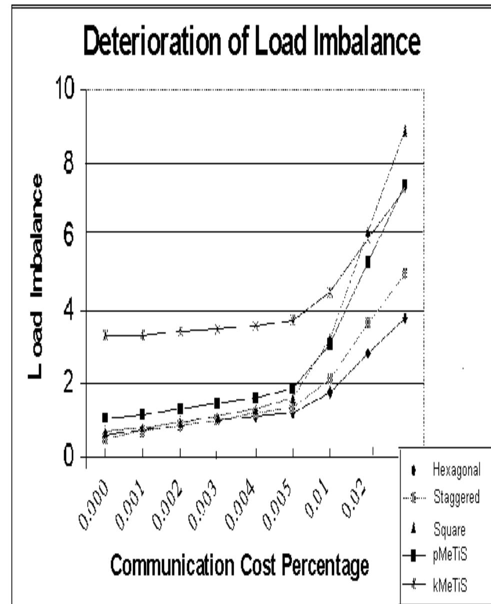
Table 5.3: airfoil data is distributed to 25 processors, weights of nodes are given randomly between 1 and 10, weights are read from a text file.

While calculating load imbalance values, we made four different experiments. At the first two parts of the experiments, we distributed airfoil data to 25 processors ((a) and (b) parts of Figures 5.1, 5.2 and 5.3). In the other two parts of the experiments, we distributed the same data set to the 60 processors ((c) and (d) parts of Figures 5.1, 5.2 and 5.3).

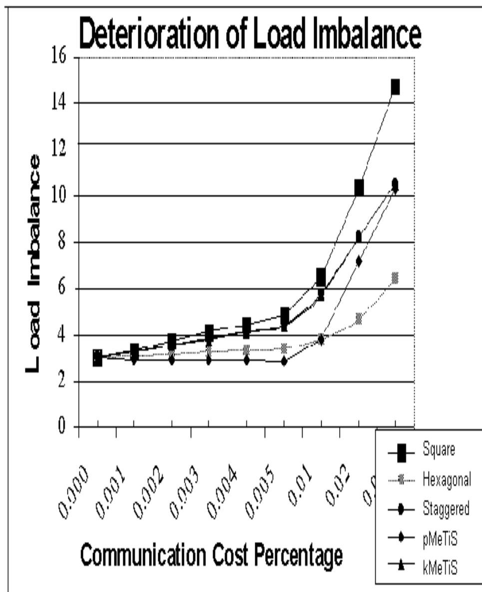
In (a) and (b) parts, all the node weights are given as 1. In the (c) and (d) parts, all the node weights are given randomly between the range of 0 and 10. For the sake of fairness, node weights are written to the text file and then they are read from the text file. With this method same node weights are used in each experiment.



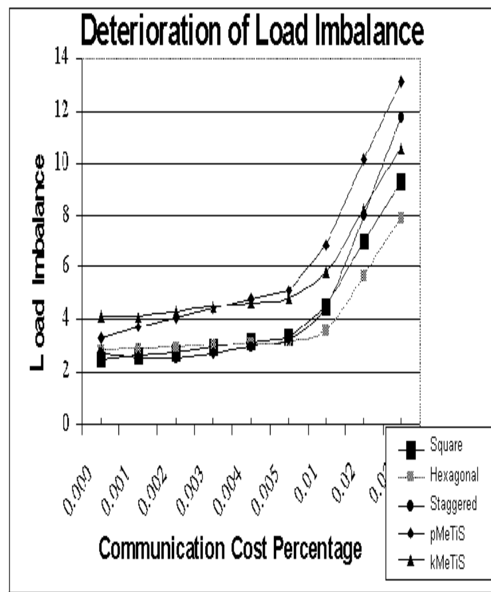
(a) airfoil data set with 25 processors, weights are all 1



(b) airfoil data set with 25 processors, weights are read from a text file.



(c) airfoil data set with 60 processors, weights are all 1



(d) airfoil data set with 60 processors, weights are read from a text file.

Figure 5.1: Deterioration of load imbalance graphs under different communication cost percentage for airfoil data set and when convergence is 3.

Communication cost (%)	KMeTis	PMeTis	Square	Staggered	Hexagonal
0.000	2.986127	2.986127	2.986127	2.986127	2.986127
0.001	3.269501	2.965679	3.357792	3.275645	3.071359
0.002	3.550493	2.945401	3.725505	3.562196	3.155719
0.003	3.829134	2.925291	4.089331	3.845828	3.239221
0.004	4.105454	2.905347	4.449331	4.126584	3.321876
0.005	4.379481	2.885568	4.805563	4.404508	3.403699
0.01	5.716200	3.818034	6.532246	5.753109	3.800751
0.02	8.233009	7.188408	10.345823	8.260603	4.948216
0.03	10.560605	10.307384	14.729931	10.543477	6.408273

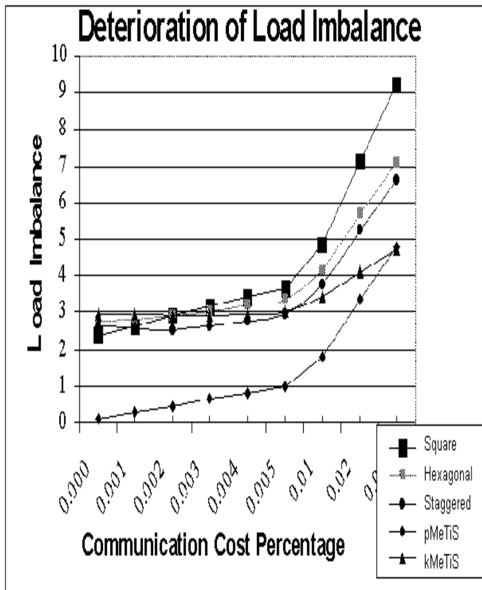
Table 5.4: airfoil data is distributed to 60 processors, weights of nodes are all 1.

Communication cost (%)	KMeTis	PMeTis	Square	Staggered	Hexagonal
0.000	4.184151	3.351072	2.468213	2.728475	2.827582
0.001	4.154592	3.718410	2.616816	2.59888	2.905802
0.002	4.328910	4.082523	2.804840	2.578009	2.983211
0.003	4.501746	4.443452	2.990962	2.695835	3.059822
0.004	4.673121	4.801238	3.175210	2.964152	3.135648
0.005	4.843053	5.155924	3.357613	3.229715	3.2107
0.01	5.800849	6.884213	4.489160	4.51766	3.574754
0.02	8.257424	10.129797	7.002007	8.001262	5.692687
0.03	10.526802	13.121556	9.291199	11.678745	7.869767

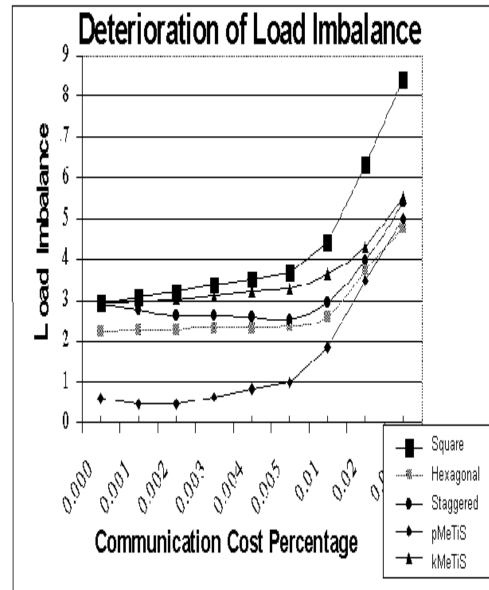
Table 5.5: airfoil data is distributed to 60 processors, weights of nodes are given randomly between 1 and 10, weights are read from a text file.

5.4 Comparison of Algorithms

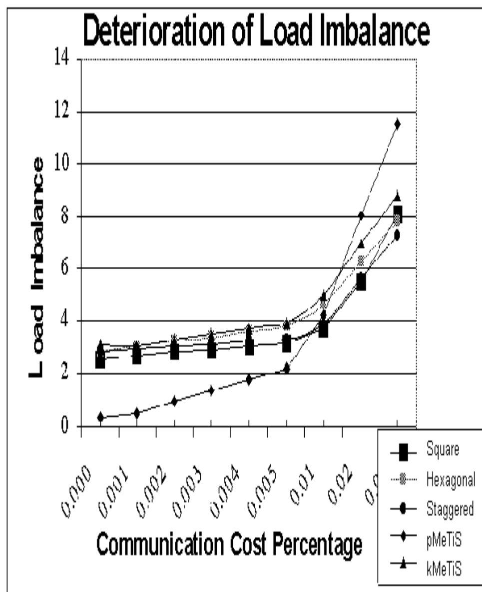
When we carefully examined the load imbalance graphs of different data sets, in distributing the load to the processors, we see that hexagonal processor mapping of SOM algorithm performs best. The reason for this behavior is its graph increases almost linearly On the other hand, PMeTis behaves worst, because its graph is increasing abruptly, especially when the communication cost is more than 0.6% of the total cost. We know that communication cost percentage of a parallel program can be up to 0.4 of the total cost. In some cases, staggered processor mapping of SOM algorithm can have the similar attitude with hexagonal mapping of SOM algorithm but we can not say that, it will show the same attitude every time.



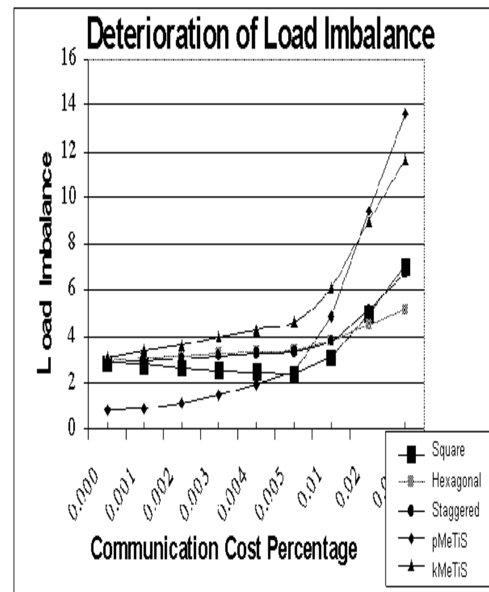
(a) biplane data set with 25 processors, weights are all 1



(b) biplane data set with 25 processors, weights are read from a text file.



(c) biplane data set with 60 processors, weights are all 1



(d) biplane data set with 60 processors, weights are read from a text file.

Figure 5.2: Deterioration of load imbalance graphs under different communication cost percentage for biplane data set and when convergence is 3.

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	0.110594	2.990646	2.389980	2.612720	2.772708
0.001	0.286037	2.966222	2.646315	2.579517	2.841921
0.002	0.460004	2.942004	2.900371	2.546598	2.91054
0.003	0.632515	2.917989	3.152179	2.618047	3.0392
0.004	0.803587	2.952627	3.401767	2.784023	3.205992
0.005	0.973238	3.027927	3.649166	2.948594	3.371373
0.01	1.800793	3.395250	4.854298	3.750982	4.177713
0.02	3.355886	4.086839	7.115751	5.259936	5.694128
0.03	4.799721	4.726420	9.1984	6.653356	7.094470

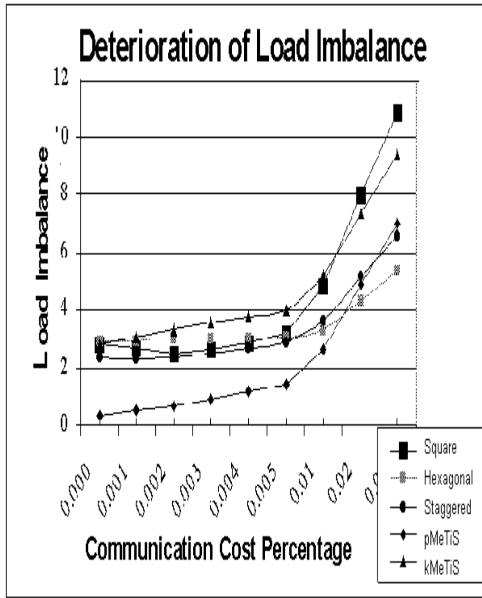
Table 5.6: biplane data is distributed to 25 processors, weights of nodes are all 1.

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	0.584783	2.45453	2.48574	2.87345	2.2264
0.001	0.434495	2.71315	3.74167	2.60343	2.54873
0.002	0.26259	3.48698	3.28369	2.34409	2.83081
0.003	0.06694	3.25433	3.81208	2.07750	2.11026
0.004	0.085853	3.01529	3.327	2.83861	2.38712
0.005	0.63154	3.76993	3.82865	2.60170	2.66143
0.01	1.29396	3.45107	4.14365	2.22934	2.91449
0.02	3.62097	4.38188	6.40969	3.60663	3.23297
0.03	4.74054	5.15903	8.11844	5.10314	4.61547

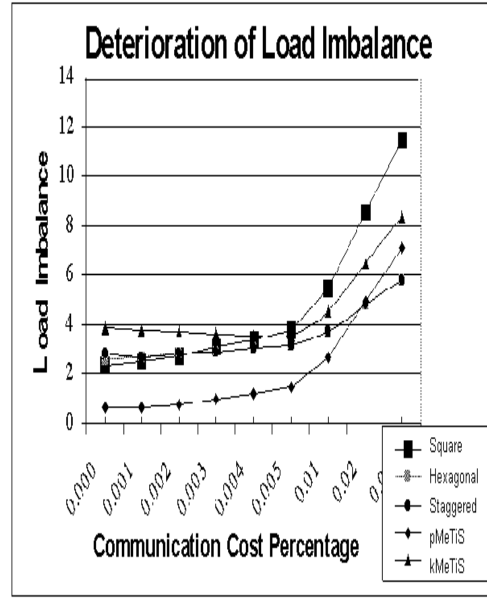
Table 5.7: biplane data is distributed to 25 processors, weights of nodes are given randomly between 1 and 10 and are read from a text file.

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	0.364039	3.128888	2.575918	2.852403	2.852403
0.001	0.519253	3.077502	2.694803	2.9551071	3.043047
0.002	0.946961	3.300460	2.812550	3.048756	3.231753
0.003	1.370732	3.521308	2.929174	3.145471	3.418549
0.004	1.790620	3.740075	3.044691	3.241231	3.603465
0.005	2.206679	3.956790	3.159118	3.336050	3.786529
0.01	4.231321	5.010586	3.715419	3.930757	4.675014
0.02	8.021353	6.979697	5.571256	5.663998	6.327819
0.03	11.501091	8.783540	8.086485	7.245584	7.833492

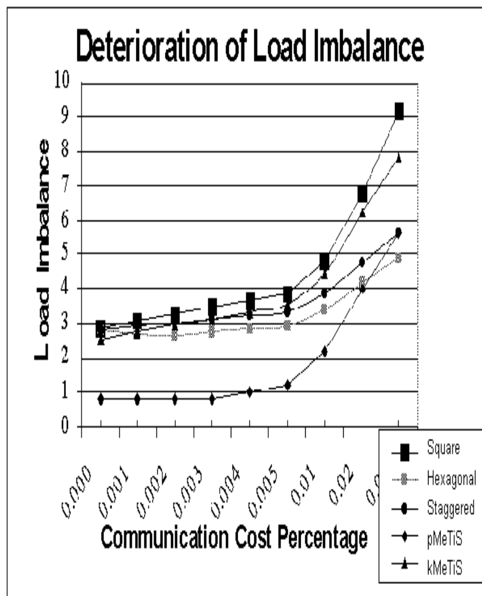
Table 5.8: biplane data is distributed to 60 processors, weights of nodes are all 1.



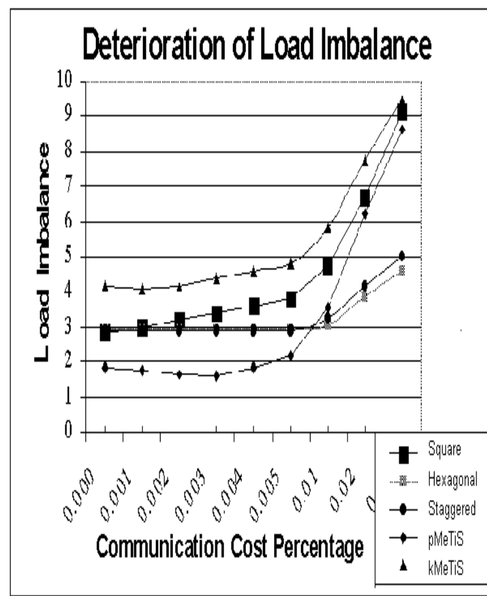
(a) crack data set with 25 processors, weights are all 1



(b) crack data set with 25 processors, weights are read from a text file.



(c) crack data set with 60 processors, weights are all 1



(d) crack data set with 60 processors, weights are read from a text file.

Figure 5.3: Deterioration of load imbalance graphs under different communication cost percentage for crack data set and when convergence is 3.

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	0.767671	3.11763	2.24710	2.832275	2.965332
0.001	0.877847	3.432841	2.735984	2.934253	3.050303
0.002	1.099512	3.659258	2.648114	3.035221	3.134405
0.003	1.422143	3.935915	2.561087	3.135194	3.217650
0.004	1.927867	4.257161	2.474891	3.234186	3.300052
0.005	2.428883	4.575407	2.389514	3.332211	3.381623
0.01	4.865584	6.123130	3.119498	3.808326	3.777448
0.02	9.420897	9.016207	4.992723	5.145886	4.513454
0.03	13.596303	11.667632	7.059425	6.747123	5.183570

Table 5.9: biplane data is distributed to 60 processors, weights of nodes are given randomly between 1 and 10 and are read from text file.

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	0.341797	2.783203	2.792780	2.326512	2.927258
0.001	0.501670	3.032509	2.641090	2.271999	2.947367
0.002	0.660148	3.279560	2.490748	2.340791	2.967285
0.003	0.868334	3.524387	2.606382	2.503001	2.987015
0.004	1.120783	3.767020	2.838245	2.669442	3.006558
0.005	1.371059	4.007487	3.181098	2.834476	3.025918
0.01	2.590798	5.178341	4.851237	3.639149	3.28013
0.02	4.8823	7.373163	7.985408	5.152536	4.343577
0.03	6.995675	9.391790	10.871917	6.550204	5.317725

Table 5.10: crack data set is distributed to 25 processors, weights of nodes are all 1.

Although kMeTis shows similar behavior with hexagonal mapping, its initial load imbalance is larger than the hexagonal mapping and when communication cost increased, load imbalance value gets larger, so it is not preferred to the hexagonal mapping of SOM algorithm. On the other hand, in some cases square processor mapping of SOM algorithm may have good behavior, but in other cases it may behave badly. Since it does not show stable behavior, it is not preferred.

To see the behaviors of graphs, we run SOM algorithm until it reached to 3 percent load imbalance value except the first experiment's (a) and (b) sections (Figure 5.1(a),(b)). We see that SOM algorithm is distributing the loads more balanced than others.

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	0.680139	3.855281	2.378378	2.816133	2.603389
0.001	0.640262	3.780296	2.532506	2.676916	2.676916
0.002	0.710215	3.706012	2.744997	2.798105	2.798105
0.003	0.965298	3.632421	3.09546	2.918148	2.918148
0.004	1.218175	3.559512	3.442833	3.037059	3.037059
0.005	1.468877	3.526564	3.787156	3.154856	3.154856
0.01	2.6907	4.508673	5.464406	3.727648	3.727648
0.02	4.986179	6.538449	8.611691	4.797958	4.797958
0.03	7.1032602	8.400212	11.51001	5.778431	5.778431

Table 5.11: crack data set is distributed to 25 processors, weights of nodes are given randomly between 1 and 10 and are read from a text file.

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	0.781250	2.539063	2.861283	2.829714	2.798826
0.001	0.788086	2.740175	3.065787	2.934941	2.686067
0.002	0.794855	2.939285	3.268265	3.039131	2.667567
0.003	0.807910	3.136423	3.468748	3.142301	2.757248
0.004	1.008159	3.331617	3.667264	3.244465	2.846031
0.005	1.06480	3.524895	3.863843	3.345638	2.933931
0.01	2.170122	4.463526	4.818645	3.837126	3.360629
0.02	3.967630	6.222122	6.780114	4.753429	4.154757
0.03	5.61069	7.807906	9.179823	5.590447	4.878602

Table 5.12: crack data set is distributed to 60 processors, weights of nodes are all 1.

As a result, we can say that SOM algorithm is better than other algorithms for balancing problem and when we used hexagonal processor mapping of SOM algorithm, we can obtain better balanced results with any communication cost percentage.

5.5 Comparison of Hexagonal Mapping and kMeTis with and without Communication Overhead

In this section, we examined the behavior of different algorithms after communication cost overhead values are added to the processor loads. In order to test the performance of different algorithms, we use the graphs listed in Table B-1

Communication cost (%)	PMeTis	KMeTis	Square	Staggered	Hexagonal
0.000	1.868896	4.188371	2.845795	2.894263	2.961032
0.001	1.770854	4.097782	2.996013	2.897620	2.954242
0.002	1.673777	4.154184	3.198148	2.900944	2.9472
0.003	1.57765	4.36944	3.39829	2.904235	2.940865
0.004	1.857415	4.582639	3.596468	2.907494	2.934277
0.005	2.149494	4.793809	3.792711	2.910722	2.927753
0.01	3.568426	5.820218	4.74587	3.265887	3.023635
0.02	6.213953	7.736293	6.705969	4.181112	3.847644
0.03	8.630748	9.489424	9.10226	5.017206	4.599175

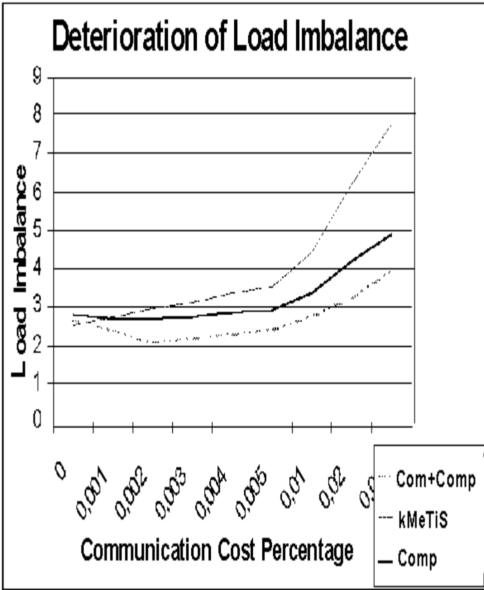
Table 5.13: crack data is distributed to 60 processors, weights of nodes are given randomly between 1 and 10 and are read from a text file.

in Appendix B.

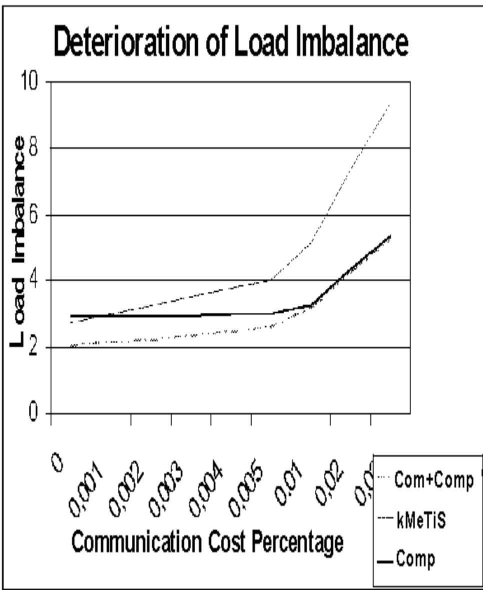
In this experiment, we compared hexagonal processor mapping with and without communication overhead. Then we compared hexagonal processor mapping with kMeTis, which is the best of the graph partitioning algorithms. In the previous experiments, we see that communication cost takes approximately 0.4% of the computation cost. We performed our experiments and we saw that SOM shows the best performance. Then, communication cost included in the total cost and compared kMeTis and SOM. When the communication cost is taken into account, edge processors will have much less communication cost since corner processors have less neighbors (Figure 5.4).

Communication cost (%)	KMeTis	Computation	Computation + Communication
0.000	2.539063	2.798826	3.710938
0.001	2.740175	2.686067	3.391547
0.002	2.939285	2.667567	3.075376
0.003	3.136423	2.757248	2.813170
0.004	3.331617	2.846031	2.902714
0.005	3.524895	2.933931	2.991368
0.01	4.463526	3.360629	3.421745
0.02	6.222122	4.154757	4.222496
0.03	7.807906	4.878602	4.953038

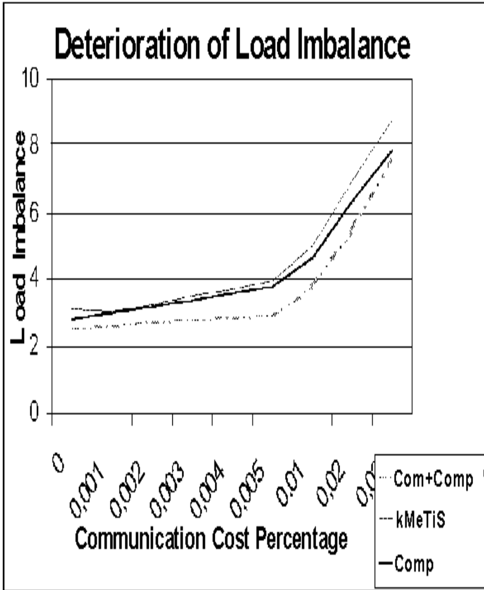
Table 5.14: crack data set is distributed to 60 processors when communication cost included, weights of nodes are 1.



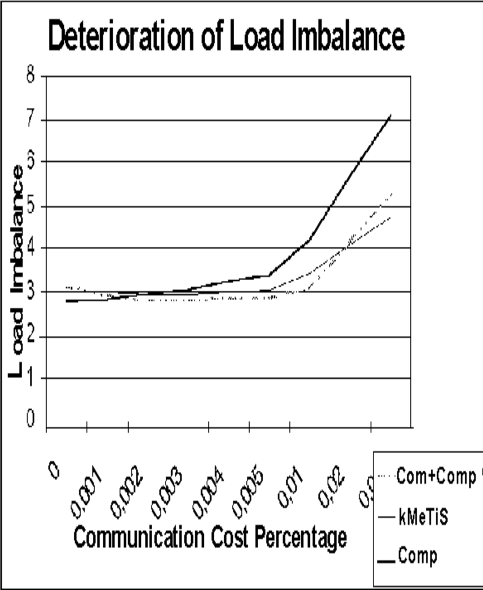
(a) crack data set for 60 processors



(b) crack data set for 25 processors.



(c) biplane data set for 60 processors



(d) airfoil data set for 60 processors.

Figure 5.4: Deterioration of load imbalance graphs when communication cost included under different communication cost percentage when airfoil, crack and biplane datums are taken and when convergence is 3.

Communication cost (%)	KMeTis	Computation	Computation + Communication
0.000	2.783203	2.927258	2.050781
0.001	3.032509	2.947367	2.172945
0.002	3.279560	2.967285	2.293948
0.003	3.524387	2.987015	2.413807
0.004	3.767020	3.006558	2.532537
0.005	4.007487	3.025918	2.650155
0.01	5.178341	3.280130	3.222091
0.02	7.373163	4.343577	4.290860
0.03	9.391790	5.317725	5.269987

Table 5.15: crack data set is distributed to 25 processors when communication cost included, weights of nodes are 1.

Communication cost (%)	KMeTis	Computation	Computation + Communication
0.000	3.128888	2.852403	2.575918
0.001	3.077502	3.043047	2.657513
0.002	3.300460	3.231753	2.738268
0.003	3.521308	3.418549	2.818196
0.004	3.740075	3.603465	2.897309
0.005	3.956790	3.786529	2.975621
0.01	5.010586	4.675014	3.768188
0.02	6.979697	6.327819	5.454193
0.03	8.783540	7.833492	7.655794

Table 5.16: biplane data set is distributed to 60 processors when communication cost included, weights of nodes are 1.

For this analysis, we performed the experiment for 25 and 60 processors. Each execution, in each test, repeated 5 times for both 25 and 60 processors and average results are taken. The (speed of) processors and their communication link bandwidths are accepted to be identical.

For this and all other analysis in the rest of this thesis, we use the formulae given in Section 5.3 for load imbalance and communication cost calculation. We calculated execution time in seconds, discarding the time spent for input file reading. For input files, we use Chaco format [17, 1] for compatibility with

Communication cost (%)	KMeTis	Computation	Computation + Communication
0.000	2.986127	2.986127	2.986127
0.001	3.269501	3.071359	2.853478
0.002	3.550493	3.155719	2.722219
0.003	3.829134	3.239221	2.592328
0.004	4.105454	3.321876	2.651669
0.005	4.379481	3.403699	2.913807
0.01	5.716200	3.800751	4.184950
0.02	8.233009	4.948216	6.544655
0.03	10.56065	6.408273	8.688727

Table 5.17: airfoil data set is distributed to 60 processors when communication cost included, weights of nodes are 1.

other tools, since it is the most common format. The Chaco file format is explained in Appendix A.

As we mentioned before, SOM algorithm satisfies the second requirement of mapping, which is neighborhood preservation. In other words, Kohonen algorithm tries to place neighboring neurons closer and this is exactly what a mapping should satisfy (placing neighboring tasks closer). However, there is the need to explicitly force the algorithm to take care of load imbalance. Uniformly distributed input selection models are generally used in SOM algorithms. This type of selection forces the SOM algorithm to distribute equal number of tasks to processors without taking into account of task loads. However, such a distribution is not a desired criteria for load balancing. So, SOM algorithm should be forced to distribute the loads of the tasks to processors in a balanced way. This can be achieved by selecting inputs from the regions closer to least loaded processor. By this way, such an input will probably force the algorithm to shift the tasks towards to least loaded processor, which means assignment of more tasks to that processor, thus minimizing the load imbalance. During input selection, least loaded processor information can be directly used and an input within the region of the least loaded processor may be chosen or such an information can be used to determine another processor. In either case, the possibility of selecting the least loaded processor should be high. When we added the communication cost overhead of each processor to the total load of each processor, corner processors will have more loads than the other processors because communication cost of the corner processors are less than the

others. When both of the loads are take into account SOM performs better performance than other well known graph partitioning algorithms. From the point of view load balance, SOM Algorithm performs better than other algorithms that we have concerned, but its execution time is much more than the some of other algorithms.

Chapter 6

Multilevel Implementation of SOM Algorithm

The SOM Algorithm is slower than some other graph partitioning algorithms. To decrease the execution time, we investigate the possibility of a multi-level implementation of the SOM Algorithm (MSOM).

As we know, graph partitioning is an important problem that has extensive application in many scientific computing, task scheduling, geographical information systems, VLSI design and operations research. The graph partitioning problem is NP-complete. However, many polynomial time algorithms have been developed to find a reasonable good partition. Generally, multilevel partitioning algorithms produce high quality partitions in a very small amount of time. Some of the best known graph partitioning algorithms are MeTiS, Jostle and Chaco. Each of them uses different methods for graph partitioning. They mainly use *recursive graph bisection method* and *spectral method*. In recursive graph bisection method: the graph is partition into two parts and each part is partitioned recursively until there are as many pieces as the number of processors of the parallel machine. On the other hand, the spectral graph methods are invariant under geometric transformations of the computational domain, as well as under renumbering of the computational graph. They also seem to generate good partitions in practice [18].

The basic idea behind the multilevel graph partitioning is very simple. The

graph G is first coarsened down to a few hundred vertices, and the algorithm is executed on these coarsened nodes. Then nodes are partitioned, and this partition is projected back towards the original graph (finer graph) by periodically refining the partition. Since the finer graph has more degrees of freedom, such refinements usually decrease the edge-cut.

During the coarsening phase, a sequence of smaller graphs $G_1 = (V_1, E_1)$, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_l| > |V_{l+1}|$. This coarsening procedure has a number of attractive properties. First, any partition of the coarse graph corresponds naturally to a partition of the fine graph. Second, since vertex weights are summed, constraints on the set sizes that depend on the number of vertices in a set are preserved in a weighted sense in the coarse graph. Thus, a good partition of the coarse graph will correspond to a good partition of the fine graph. Graph G_{l+1} is constructed from G_1 by finding a maximal matching $M_1 \subseteq E_1$ of G_1 and collapsing together the vertices that are incident on each edge of matching. Maximal matchings can be computed in different ways [8, 16, 23, 21]. The method used to compute the matching greatly affects the quality of the partitioning and the time required during uncoarsening phase. One simple scheme for computing a matching is the *random matching* (RM) scheme [8, 16]. In this scheme vertices are visited in random order and for each unmatched vertex, we randomly match it with one of its unmatched neighbors. An alternative matching scheme that we found to be quite effective is called *heavy-edge matching* (HEM) [23, 25]. The heavy-edge matching is computed using a randomized algorithm as follows. The vertices are visited in random order. However, instead of the adjacent unmatched vertices, the vertices which has the largest weight is chosen for matching. HEM scheme quickly reduces the sum of the weights of the edges in the coarsest graph. The coarsening phase ends when the coarsest graph G_m has a predetermined number of vertices.

During the uncoarsening phase, the partition of the coarser graph G_m is projected back to the original graph, by going through the graphs $G_{m-1}, G_{m-2}, ., G_1$. Since each vertex $u \in V_{l+1}$ contains a distinct subset U of vertices of V_l, the projection of the partition from G_{l+1} to G_l is constructed by simply assigning the vertices in U to the same part in G_l such that vertex u belongs in G_{l+1}. After projecting a partition, a partitioning refinement algorithm is used. The basic purpose of partitioning refinement algorithm is to select vertices such

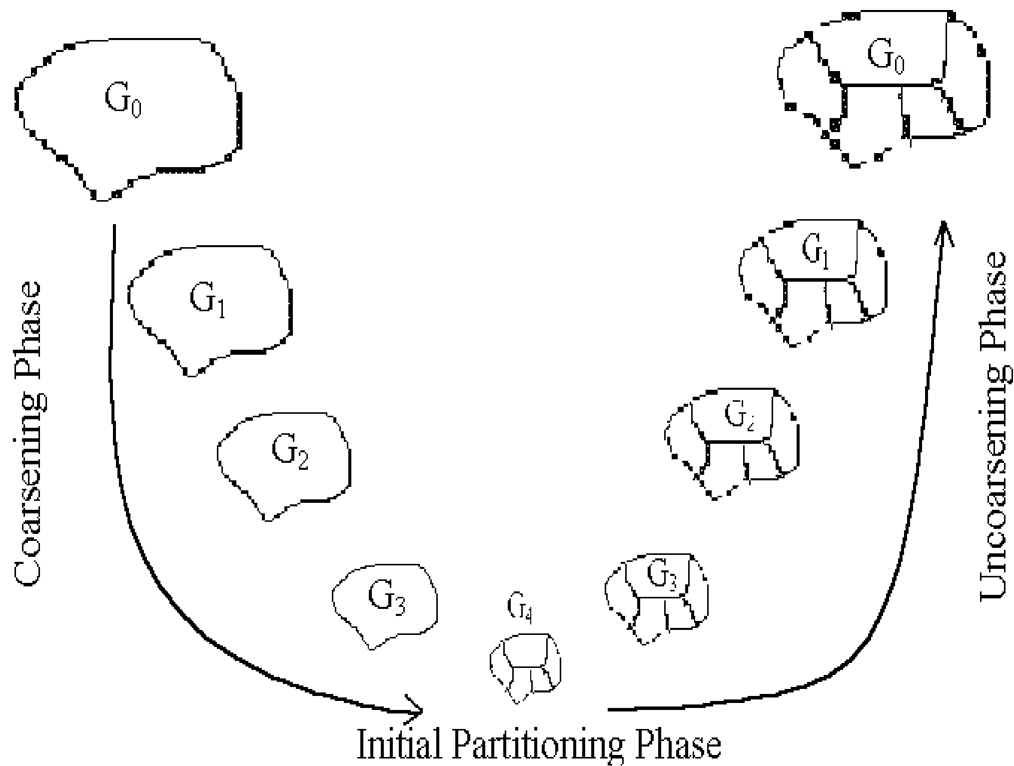


Figure 6.1: The various phases of the multilevel SOM algorithm.

that when moved from one partition to another the resulting partitioning has smaller edge-cut and remains balanced. A class of local refinement algorithms that tend to produce very good results are those that are based on Kernighan-Lin (KL) partitioning algorithm [27] and its variants.

6.1 Multilevel Implementation of Self-Organizing Maps

Multilevel implementation of Self-Organizing Maps Algorithm has three steps.

1. *Coarsening Phase:* In this phase, we used heavy edge matching(HEM) approach, which chooses the neighbor of each node that has the maximum communication load as a candidate node to combine. According to this schema, we chose any node randomly and determine the candidate node with the HEM approach. Then these two nodes are combined as a new coarsened node. Their loads are added as the new node load. The average of their x and y coordinates are taken as the new node's x and y positions. The neighbors of these two nodes are combined and taken as new node's neighbor nodes. This process was done

as many times as the number of total nodes. The nodes that are not combined with any of its neighbors are directly transferred to the next coarsening level. This coarsening step continues until the graph has less than 100 nodes.

2. *Initial Partitioning Phase:* In this phase, the coarsened nodes are distributed to the processors by the Self-Organizing Maps Algorithm. The algorithm is executed until it reaches a predefined load imbalance convergence.

3. *Uncoarsening Phase:* After initial partitioning phase, coarsest nodes are restored as many times as the coarsening phase. After each uncoarsening phase, SOM algorithm is re-executed until it reaches a predetermined load imbalance convergence. All properties (values) of the uncoarsened nodes remain the same as the initial values of each node.

6.2 Performance Results

To see the results of MSOM, we performed some experiments. In the first experiment, we compare the sequential SOM algorithm and the multilevel version. In this experiment, we used different data files and we have used hexagonal processor mapping. We chose 3 percent load imbalance convergence and we distributed the loads to 25 processors. The results are given in Table 6.1.

TIG	*PCG*	*Execution time(sec)*		*Load Imbalance(%)*		Dec.Ratio
		SOM	MSOM	SOM	MSOM	SOM/MSOM
airfoil	7*4	1068.53	258.71	2.97578	0.862605	4.13
crack	7*4	1306.72	158.62	2.97296	2.8487	8.23
Whitaker	7*4	1383.22	194.82	2.89845	2.79069	7.1
L	7*4	167.1	41.16	2.29425	2.93442	4.06
4elt	7*4	1979.02	616.55	2.41158	2.65932	3.2
FFT10	7*4	1508.17	502.95	2.56874	2.85365	2.998

Table 6.1: Comparison of SOM and MSOM for 25 processors.

At the second experiment, we distributed airfoil data to different number of processors. We executed both SOM and MSOM algorithms with the same data and tabulate the results in Table 6.2.

TIG	*PCG*	*Execution time(sec)*		*Load Imbalance(%)*		Dec.Ratio
		SOM	MSOM	SOM	MSOM	SOM/MSOM
airfoil	1*2=2	2.31	2	1.06	0.0738648	1.155
airfoil	3*2=5	191.73	5.03	2.04611	1.7982	38.12
airfoil	3*3=8	534.71	14.27	2.28073	2.93645	37.47
airfoil	3*4=11	641.96	32.97	2.80216	2.93084	19.47
airfoil	5*3=13	713.47	51.74	2.73127	2.47687	13.79
airfoil	5*4=18	832.13	87.33	2.49624	2.97152	9.53
airfoil	5*5=23	983.33	157.88	2.60422	2.6898	6.23
airfoil	7*4=25	991.43	149.59	2.27691	2.82668	9.53

Table 6.2: Comparison of SOM and MSOM for airfoil data set with 3% convergence constraint for different number of processors.

If we examine the results, we can say that MSOM is faster than SOM algorithm. However, the performance degrades when the number of processors is increased.

In the third experiment, MSOM algorithm is executed with different data sets for the same processor connection graph. Convergence constraint was 3 percent and the number of processors is 5. The results are given in Table 6.3.

TIG	Number of Nodes	PCG	Execution time (sec)	Load Imbalance (%)
jagmesh	939	3*2=5	2.69	2.73329
L	956	3*2=5	4.14	2.73008
airfoil	4253	3*2=5	2.59	2.76962
BFLY	4608	3*2=5	120.9	2.93497
CCA9	4608	3*2=5	35.81	2.59671
CCC9	4608	3*2=5	49.67	1.49698
3elt	4720	3*2=5	3.55	0.91733
FFT9	5120	3*2=5	59.46	2.71536
whitaker	9800	3*2=5	4.68	1.1647
crack	10240	3*2=5	3.62	2.60948
BFLY10	10240	3*2=5	269.18	2.35755
CCA10	10240	3*2=5	79.34	1.94254
FFT10	11264	3*2=5	97.2	1.28397
big	15606	3*2=5	7.07	2.92696
4elt	15606	3*2=5	5.42	2.38087
bplane	21701	3*2=5	6.01	2.48589
brack2	62631	3*2=5	38.07	2.85953

Table 6.3: Execution time of SOM Algorithm for different data sets with the 3 percent convergence constraint for 5 PCG.

Table 6.3 shows that the execution time of MSOM is highly dependent on the type of data. The Number of Nodes is important for execution time but not as much as the type of data. Although square data sets, like Butterfly, are taking too much time, FEM(2) type graphs gives the best performance.

6.3 Conclusion

It is clear that multilevel graph partitioning approach is able to find high quality partitions for a variety of unstructured graphs. By multilevel approach, we reduced the execution time of the SOM Algorithm.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented Kohonen's Self-Organizing Map Algorithm that includes communication cost overhead for static load balancing problem. The efficiency of a mapping can be determined by checking its load balance values. However, balancing not only means the balancing of the computation loads, but also balancing the communication load of the each processor. On the other hand, communication cost should be reduced as much as possible. To minimize the communication cost, inter processor communication of multi-computer system should be minimized.

One of the important properties of SOM Algorithm is its topology preserving property. By selecting suitable processor mapping (hexagonal) type, we balanced and minimized communication overhead caused by inter processor communication. This minimized communication overhead is included the total load of each processor. By selecting the inputs from the least loaded processor's region, we force the algorithm to distribute the loads as equally as to the processors and incorporate load balancing mechanism into the SOM Algorithm. For a given input, in order to find the nearest task quickly, we divide the processor regions into subregions and apply a range searching algorithm.

As all neural network algorithms, one of the important disadvantage of

SOM algorithm is its execution time. To alleviate this disadvantage a multilevel implementation of the SOM Algorithm is realized. In the multilevel implementation, we used recursive bisection method, which is the best of multilevel approaches. The multilevel implementation of SOM algorithm reduced the execution time by a factor of 40 times. Studies show that our algorithm outperforms the other algorithms in terms of load balance.

7.2 Future Work

Other multilevel implementation approaches can be implemented to improve execution time of the SOM algorithm. To decrease the execution time of SOM algorithm, parallel implementation of SOM algorithm can be an another brilliant idea.

Bibliography

[1] Chaco home page `http://www.cs.sandia.gov/CRF/chac.html`.

[2] Beowulf Hardware
`http://www.ti.informatik.uni-tuebingen.de/fang/`, 2000.

[3] Beowulf Howto.
`http://www.linuxdoc.org/HOWTO/Beowulf-HOWTO.html`, 2000.

[4] Beowulf Mailing List Faq.
`http://www.dnaco.net/kragen/beowulf-faq.txt`, 2000.

[5] The Beowulf Project.
`http://www.beowulf.org`, 2000.

[6] B. M. Boghosian. Cellular Automata Simulation of Two-phase Flow on the CM-2 Connection Machine Computer. Technical Report TR-19 CA88-1, Thinking Machines Corporation, 1988. (Appeared in Supercomputing 88, Vol. II:Science and Applications, J. L. Martin and S. F.Lundstrom, eds.: IEEE Computer Society Press [1989]: pp. 34-44.).

[7] S.W. Bollinger and S.F. Midkiff. Heuristic Technique for Processor and Link Assignment in Multicomputers. *IEEE TOC*, Vol. 40, No. 3, pp. 325-333, 1991.

[8] T. Bui and C. Jones. A Heuristic for Reducing Fill in Sparce Matrix Factorization. *Parallel Processing for Scientific Computing*, In the proceedings of 6th SIAM Conf.:pp. 445–452, 1993.

[9] T. Bultan and C. Aykanat. A New Mapping Heuristic Based on Mean Field Annealing. *Journal of Parallel snd Distributed computing*, Vol. 16, pp. 292-305, 1992.

[10] M. Cross C. Walshaw and M. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructered Meshes. *Journal of Parallel and distributed Computing*, Vol. 47, No. 2, pp. 102-108, 1997.

[11] T. Chockalingam and S. Arankumar. A Randomized Heuristic for the Mapping Problem: The Genetic Approach. *Parallel Computing*, Vol. 18, pp. 1157-1165, 1992.

[12] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Computation and Structure*, Vol. 28, No. 5, pp. 579-602, 1988.

[13] C.M. Fiduccia and R.M Mattheysey. A Linear Time Heuristic for Improving Network Partitions. *In the Proceedings of the 19th IEEE Design and Automation Conference*, pp. 175-181, 1982.

[14] A. Gürsoy and M. Atun. Neighbourhood Preserving Load Balancing: A Self-Organizing Approach. *Lecture Notes in Computer Science*, pp. 234-241, 1999.

[15] A. Gürsoy and M. Atun. A New Load-balancing Algorithm Using Self-Organizing Maps. *In the Proceedings of the 14th International Symposium on Computer and Information Sciences, ISCIS14, Ed. M Turksever et al*, Kuşadası, Turkey, 1999.

[16] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. Technical report, Sandia National Laboratories, SAND93-1301, 1993.

[17] B. Hendrickson and R. Leland. The Chaco User's Guide: Version 2.0. Technical report, Sandia National Laboratories, SAND94-2692, 1994.

[18] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, Vol. 16, No. 2, pp. 452-469, 1995.

[19] B. Hendricson and T. G. Kolda. Graph Partitioning Models for Parallel Computing. *Parallel Computing*, Vol. 26, pp. 1519-1534, 2000.

[20] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. *In the Proceedings of the IEEE Hot Interconnects Symposium*, pp. 245-271, 1993.

[21] G. Kapris and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Meshes. Technical Report TR 95-035, 1995.

[22] G. Kapris and V. Kumar. Multilevel Algorithms for Multi-constraint Graph Partitioning. Technical Report TR 98-019, Department of Computer Science, University of Minnesota, 1998.

[23] G. Kapris and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, Vol. 48, No. 1:pp. 96–129, 1998.

[24] V. Karamcheti and A. A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–60, San Jose, California, 1994.

[25] G. Karypis and V. Kumar. Analysis of Multilevel Graph Partitioning. Technical Report TR 95-037, 1995.

[26] S. Kaski. Fast Winner Search for Som-based Monitoring and Retrieval of High-dimensional Data. *In the Proceedings of the Ninth International Conference of Artificial Neural Networks*, Vol 2, pp. 940-945, 1999.

[27] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, Vol 49, pp. 291-297, No 2, 1970.

[28] T. Kohonen. Analysis of a Simple Self-Organizing Process. *Biological Cybernetics*, Vol. 44:pp. 135–140, 1982.

[29] T. Kohonen. Self-Organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics*, Vol. 43:pp. 59–69, 1982.

[30] D. E. Culler R. P. Martin, A. M. Vahdat and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwith in a Cluster Architecture. International Symposium on Computer Architecture, Denver, CO, 1997.

[31] S. Goldstein T. V. Eicken, D. Culler and K. Schuser. Active Messages: A Mechanism for Integrated Communication and Computation. *In the Proceedings of the International Symposium on Computer Architecture*, pp. 256-266, 1992.

[32] B. Uçar. Partitioning Sparce Rectangular Matrices for Parallel Computing AA^TX. Master's thesis, Bilkent University, Department of Computer Engineering, September 1999.

[33] AG-Monien (Parallel Computing Group) University of Paderborn. `http://www.uni-paderborn.de/fachbereich/AG/manien/`, 2000.

Appendix A

MeTis

MeTis is a software package for partitioning large irregular graphs, partitioning large meshes, and computing fill-reducing ordering of sparse matrices. The algorithms in *MeTis* are based on multilevel graph partitioning described in [22, 23]. The important properties of *MeTis* are as follows:

- **Multi-Constraint Partitioning:** MeTis includes partitioning routines that can be used to partition a graph in the presence of multiple balancing constraints. The idea is that, each vertex has a vector of weights of size m associated with it, and the objective of the partitioning algorithm is, to minimize the edge cut subject to the constraints that each one of the m weights is equally distributed among the domains.
- **Minimizing the Total Communication volume:**The objective of the traditional graph partitioning problem is, to compute a balanced *k-way* partitioning such that the number of edges that straddle different partitions is minimized. When partitioning is used to distribute a graph or a mesh among the processors of a parallel computer, the objective of minimizing the edge cut is only an approximation of the true communication cost resulting from the partitioning. Despite that, for a wide range of problems, by minimizing the edge cut, the partitioning algorithms also minimize the communication cost reasonably well.
- **Minimizing the Maximum Connectivity of the Subdomains:** The

> communication cost resulting from a *k-way* partitioning in general depends on following factors: (i) the total communication volume, (ii)the maximum amount of data that any particular processor needs to send and receive; and (iii) the number of messages a processor needs to send and receive. MeTis tries to minimize all these factors.

MeTis provides two programs $pMeTis$ and $kMeTis$ for partitioning an unstructured graph into k equal size parts. The partitioning algorithm used by $pMeTis$ is based on multilevel recursive bisection described in [21], whereas the partitioning algorithm used by $kMeTis$ is based on multilevel $k - way$ partitioning described in [23]. Both of these programs are able to produce high quality partitions. However, depending on the application, one program may be preferable than the other. In general, $kMeTis$, is preferred when it is necessary to partition graphs into more than eight partitions. For such cases, $kMeTis$ is considerably faster than $pMeTis$. On the other hand, $pMeTis$ is preferable for partitioning a graph into a small number of partitions.

Appendix B

Data Sets Used for Experiments

In order to test the performance of our algorithm and analyse the effect various methods and/or parameters we use a collection of graphs which are listed in the below table. Most of these graphs are collected from AG-Monien (Parallel Computing Group) Web pages [33]. The common properties all graphs is that they all have some spatial connections for which SOM algorithm perform well and expose the advantage of neighbourhood preservation. The table includes graphs with different interconnection properties with various degrees from different known collections like NASA and Harwell-Boeing. In the below part we give some details about each class of graphs.

- FEM(2): FEM(2) stands for 2-dimensional finite element meshes. We have six graphs in this class. All graphs in this class are not ordinary meshes since L9, 4elt and Airfoil have some holes in them.
- FEM(3): FEM(3) stands for 3-dimensional finite element mesh. The package obtained from [33] includes just two graphs and we select Brack2 since the other one named Wave has more than 156000 nodes which we thought more than enough for a non-multilevel algorithm.
- Harwell-Boeing Collection(HB): Within this class we have quite a high number of graphs with various number of nodes ranging from 500 to 30000. The package contains the series of known graphs named bcsstk. But as far as we realize most of the graphs with high number of nodes

in this package are disconnected ones which is not desired for SOM algorithm. We select three from connected ones where Jagmesh is a FEM graph with a hole in the center, Dwt2680 is a one with a rectangular shape with average degree of 8 for inner nodes, and finally Bcspwr10 is a power graph with variuos degree nodes.

- NASA: The obtained NASA graph collection contains three graphs and the selected one has the maximum number of nodes.
- Grid: Within this class we have four graphs used. The three of them are square while the other one is rectangular. All graphs in this class are ordinary grids, that is all inner nodes have degree of four. The graphs named "200_200" and "200_300" are automatically generated grids by our program with 40000 and 60000 nodes respectively.
- Butterfly, CCC & CCA: CCC stands for "Cube Connected Cycle" and CCA stands for CCC graphs without wrap around edges. These three class of graphs have various dimension, connection and degree properties.

Test Graph	$\#Vertices$	$\#Edges$	Class
Crack	10240	30380	FEM(2)
L9 (Big)	17983	35596	FEM(2)
4elt	15606	45878	FEM(2)
3elt	4720	13722	FEM(2)
Brack2	62631	366559	FEM(3)
Jagmesh	936	2664	Harwell-Boeing Collection (HB)
DWT2680	2680	11173	Harwell-Boeing Collection (HB)
Bcspwr10	5300	8271	Harwell-Boeing Collection (HB)
Nasa4704	4704	50026	NASA
Shock	36476	71290	Square Gird
Biplane	21701	42038	Square Grid
FFT9	5120	9216	Butterfly
FFT10	11264	20480	Butterfly
BFLY9	4608	9216	Butterfly
BFLY10	10240	20240	Butterfly
CCC9	4608	6912	CCC
CCC10	10240	15360	CCC
CCA9	4608	6400	CCA
CCA10	10240	14336	CCA
Airfoil	4253	12289	FEM(2)
Whitaker	9800	28989	FEM(2)
L	956	1820	Square Grid
Stufe10	24010	46614	Square Grid

Table B-1: Test Graphs

Appendix C

Chaco file Format

The Chaco file format contains the adjacency information of a graph as a text file. The lines in such a file beginning with characters # and % are comment lines. In its simplest form the file contains $n + 1$ lines. The first line contains some necessary information about the graph and file. That is the first two integers on the first line correspond to number of vertices and number of edges respectively. The remaining n lines contain neighbour lists for each vertex from 1 to n in order. Neighbour lists of vertices are a set of integers separated by single spaces where each integer identifies the id of each neighbour. Chaco file format supports graphs with weights on both edges and/or vertices. In order to include weights on edges and/or vertices, a third integer value should be specified on the first line. This number may have up to three digits. If the 1's digit is nonzero this means file also includes vertex weights. If 10's digit is nonzero this means file also includes edge weights. And finally if 100's digit is nonzero this means file includes vertex numbering information. It is important that if by setting the related digits to nonzero values, it is mentioned that vertices and/or edges have weight information than all vertices and/or edges should have weights specified. If vertex weight information is included each weight should appear before the neighbour list of each vertex. On the other hand if edge weight information is included each weight value should appear immediately after the corresponding entry in the neighbour list.

If any vertex has many neighbours it may be difficult to list all neighbours on a single line. In such a case the neighbour list of vertices can be split up

```
\% This is a sample Chaco file with all its
\% options presented
Number_Of_Vertices      Number_Of_Edges     (1)    {1}    [1]
(Vertex_Number)  {Vertex_Weight} neighbour(1) [Edge_Weight].....
......
....
..
```

Figure C.1: Chaco Input File Format

into a few lines by using vertex numbers on each line which should be the first entry on each line.

The most general form of Chaco input file is given in Figure C.1 where different types of options are indicated by different parenthesis.