Designing a scalable dynamic load -balancing algorithm for pipelined single program multiple data applications on a non-dedicated heterogeneous network of workstations

Ashraf Osman
West Virginia University

DESIGNING A SCALABLE DYNAMIC LOAD-BALANCING ALGORITHM FOR PIPELINED SINGLE PROGRAM MULTIPLE DATA APPLICATIONS ON A NON-DEDICATED HETEROGENEOUS NETWORK OF WORKSTATIONS

Ashraf Osman

Dissertation submitted to the College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Engineering

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2003

Keywords: Load-balancing algorithms, Distributed Computing, Heterogeneous Network of Workstations, Simulation and Modeling, SPMD

ABSTRACT

DESIGNING A SCALABLE DYNAMIC LOAD-BALANCING ALGORITHM FOR PIPELINED SINGLE PROGRAM MULTIPLE DATA APPLICATIONS ON A NON-DEDICATED HETEROGENEOUS NETWORK OF WORKSTATIONS

Ashraf Osman

Dynamic load balancing strategies have been shown to be the most critical part of an efficient implementation of various applications on large distributed computing systems. The need for dynamic load balancing strategies increases when the underlying hardware is a non-dedicated heterogeneous network of workstations (HNOW). This research focuses on the single program multiple data (SPMD) programming model as it has been extensively used in parallel programming for its simplicity and scalability in terms of computational power and memory size.

This dissertation formally defines and addresses the problem of designing a scalable dynamic load-balancing algorithm for pipelined SPMD applications on non-dedicated HNOW. During this process, the HNOW parameters, SPMD application characteristics, and load-balancing performance parameters are identified.

The dissertation presents a taxonomy that categorizes general load balancing algorithms and a methodology that facilitates creating new algorithms that can harness the HNOW computing power and still preserve the scalability of the SPMD application.

The dissertation devises a new algorithm, DLAH (Dynamic Load-balancing Algorithm for HNOW). DLAH is based on a modified diffusion technique, which incorporates the HNOW parameters. Analytical performance bound for the worst-case scenario of the diffusion technique has been derived.

The dissertation develops and utilizes an HNOW simulation model to conduct extensive simulations. These simulations were used to validate DLAH and compare its performance to related dynamic algorithms. The simulations results show that DLAH algorithm is scalable and performs well for both homogeneous and heterogeneous networks. Detailed sensitivity analysis was conducted to study the effects of key parameters on performance.

DEDICATION

To my wife, (Al-Hamd Lellah) I praise Allah for His blessings. He blessed me with a sincere and patient wife who was always by my side.

To our baby son, Mostafa, you are too young to read this now, but I am just reminding you that you attended my PhD defense and I hope one day that I will attend yours (In Shaa Allah).

TABLE OF CONTENTS

Abstract .... **ii**

Dedication .... **iii**

Table of Contents .... **iv**

List of figures .... **vii**

List of Tables .... **ix**

Acknowledgments .... **x**

Glossary .... **xi**

Chapter 1 .... **1**

Introduction **1**

1.1 Motivation 1
1.2 Objectives 2
1.3 Contributions 2
1.4 Thesis Organization 3

Chapter 2 .... **5**

Overview of Load-Balancing Algorithms For HNOW **5**

2.1 Overview of Heterogeneous Network of Workstations 5
2.2 Cluster Heterogeneity Parameters 7
2.3 Application Type 7
2.4 Algorithm Characteristics 8
2.5 Load-Balancing Algorithms Classification 8
2.5.1 Initiation 9
2.5.2 Load-Balancer Location 9
2.5.3 Information Exchange 10
2.5.4 Load Selection 10

Chapter 3 .... **12**

Related Work **12**

3.1 Automatic Heterogeneous Supercomputing (AHS) [DCG93] 12
3.2 Self-Adjusting Scheduling for Heterogeneous Systems (SASH) [HLA95] 13
3.3 Support for Parallel Loop Execution (SUPPLE) [OP97] 14
3.4 Data Migration Environnent (DAME) [CCN97] 15
3.5 Asymmetric Load Balancing on a Heterogeneous Cluster of PCs [B99] 16
3.6 GR Protocol [LL96] 16
3.7 Compile-time Scheduling Algorithms [CZL97] 17

Chapter 4 .... **20**

Problem definition **20**
4.1 Problem Statement 20
4.2 Problem Significance 20
4.2.1 Non-Dedicated Heterogeneous Network of Workstations 20
4.2.2 SPMD applications 21
4.2.3 Scalable Dynamic Load-Balancing Algorithm 22
4.3 Formal Problem Description 22
4.4 Problem Assumptions 23
4.5 Problem Parameters 23
4.5.1 Pipelined SPMD Application 23
4.5.2 Heterogeneous Network of Workstations Parameters 24
4.6 Problem NP-Completeness 25
4.6.1 Problem Description 25
4.6.2 Problem Statement: 26

Chapter 5 .... **29**

Pipelined Applications: Implementation and Analysis **29**
5.1 Domain Decomposition of Large Eddy Simulations of Ship Wakes' 29
5.1.1 Introduction 30
5.1.2 Methodology 32
5.1.3 Computer Resources 33
5.1.4 Domain Decomposition Strategy 34
5.1.5 Simulation Results 40
5.1.6 Conclusion 45
5.2 Distributed Algorithm for Partially Clairvoyant Dispatchers 45
5.2.1 Introduction 46
5.2.2 Problem Statement 48
5.2.3 Motivation and Related Work 50
5.2.4 Computer Resources 51
5.2.5 Experiment Design 51
5.2.6 Empirical Analysis 53
5.2.7 Conclusion 56

Chapter 6 .... **57**

HNOW Modelling **57**
6.1 HNOW Measurable Parameters 57
6.1.1 Cluster Heterogeneity Parameters 57
6.1.2 HNOW Performance Parameters 59
6.2 Theoretical Model of Parallel Computers 60
6.2.1 PRAM Model 60
6.2.2 BSP Model 62
6.2.3 LogP Model 64
6.3 HNOW Simulation Model 65
6.3.1 What is OMNeT++? 65
6.3.2 Modeling Concepts 65
6.3.3 Building and Running Simulations 66
6.3.4 Simulation Model 67

Chapter 7 .... **70**

Dynamic Load Balancing Algorithm Outline **70**
7.1 Dynamic Load Balancing Algorithm (DLAH) for HNOW Overview 70
7.2 Convergence of the Diffusive Policy 71
7.3 DLAH Algorithm 75
7.4 DLAH Analysis and Bounds 79

Chapter 8 .... **86**

Simulations **86**
8.1 Simulation Model Verification and Validation 86
8.2 DLAH Performance 87
8.2.1 Homogenous Network of Workstations 88
8.2.2 HNOW with Different Processing Power 96
8.2.3 HNOW with Different Memory Capacities 101
8.2.4 HNOW with Different Network Parameters 103
8.2.5 Complete HNOW 106
8.3 DLAH Influencing Factors 107
8.3.1 The Effect of the Workload 107
8.3.2 The Effect of the Threshold Level 108
8.3.3 Scalability of the DLAH Algorithm 109
8.4 DLAH Compared to Other Dynamic Load Balancing Algorithms 110
8.4.1 DLAH Compared to Other Diffusion Algorithms on an HNOW 110
8.4.2 DLAH Compared to Other Diffusion Algorithms on a Non-Dedicated Homogeneous Network of Workstations 112
8.5 Summary of Results 113

Chapter 9 .... **116**

Conclusions and Future Work **116**
9.1 Conclusions and Contributions 116
9.2 Future Work 118

References **120**

LIST OF FIGURES

Figure 2.1: A simple scheme of an HNOW 5
Figure 2.2: HNOW parameters 6
Figure 2.3: Dynamic load balancing taxonomy 11
Figure 4.1: Data partitioning and distribution 21
Figure 5.1: CFD lab's cluster [SS01] 33
Figure 5.2: Decomposition strategy: a,b,c – cells of the grid in different computational domains 34
Figure 5.3: Wake decomposition 35
Figure 5.4: Domain decomposition of the application 36
Figure 5.5: Load assignment to processing elements 38
Figure 5.6: Speedup and granularity 39
Figure 5.7: The schematic of the flat plate wake [SS01] 40
Figure 5.8:The stream-wise velocity contours of one-way decomposition scheme 41
Figure 5.9: Execution time versus the simulation time 42
Figure 5.10: Speedup experienced while increasing the problem size with the number of processors 43
Figure 5.11: Execution time versus number of grid nodes per workstation. 44
Figure 5.12: Single and Parallel implementation performance with respect to the number of jobs 53
Figure 5.13: Histogram of the communication time observed in our experiments 54
Figure 5.14: Undispatchable jobs for packed nodes 55
Figure 5.15: Undispatchable jobs while using one processor per node 56
Figure 6.1: PRAM model 61
Figure 6.2: BSP model 62
Figure 6.3: BSP execution 62
Figure 6.4:LogP model 64
Figure 6.5: Building and running a simulation 67
Figure 7.1: DLAH algorithm 76
Figure 7.2: Workload exchanged between two workstations with a load-balancing algorithm designed for a homogenous network 77
Figure 7.3: DLAH Analysis 80
Figure 7.4: Relation between the threshold ratio and the upper bound of the number of steps required to balance 85
Figure 8.1: Loop time vs. memory available 87
Figure 8.2:Number of steps obtained from the simulation vs. analytical performance bounds with $Steps_i < P$ 89
Figure 8.3: Number of steps obtained from the simulation vs. analytical performance bounds with $Steps_i > P$ for some cases 90
Figure 8.4: Loop time of a homogeneous network of 5 workstations 91
Figure 8.5: Smoothed loop time of a homogeneous network of 5 workstations 92
Figure 8.6: Workload assignment for a homogeneous network of 5 workstations. 93

Figure 8.7: Loop time of a homogeneous network with workstation where WS[0] has been assigned a workload 4 times its neighbors... 94
Figure 8.8: Workstation WS[0] loop time with different workloads..................................... 95
Figure 8.9 Workstation WS[0] workload distribution with time 95
Figure 8.10: Workload distribution of an HNOW of different processing power................. 97
Figure 8.11: Smoothed loop time of an HNOW of different processing power. 97
Figure 8.12: Histogram of the steps required for balance... 98
Figure 8.13: 95% confidence interval vs. number of simulations... 99
Figure 8.14:Workstation WS[0] with different processing power...................................... 100
Figure 8.15: Number of steps vs. processing power ratio 100
Figure 8.16: Loop time of an HNOW of 20 workstations of different memory capacities.. 102
Figure 8.17: Workload distribution of an HNOW with different memory capacities......... 102
Figure 8.18: Number of steps vs. memory capacity ratio 103
Figure 8.19: Loop time for an HNOW with different network parameters. 104
Figure 8.20: Loop time for workstation WS[0] with different network parameters. 105
Figure 8.21: Number of steps vs. network ratio 105
Figure 8.22: Histogram of the number of steps required to reach balance state................. 106
Figure 8.23: Average number of steps vs. number of workstations.................................. 107
Figure 8.24: The effect of the workload difference between neighbors............................. 108
Figure 8.25: The effect of the threshold level 109
Figure 8.26: Average number of steps vs. the number of workstations 110
Figure 8.27: Comparison between the average numbers of steps required for balance....... 111
Figure 8.28: Comparison between the average DP exchanged.. 111
Figure 8.29:The Algorithms reactions to a step disturbance... 113

LIST OF TABLES

Table 3.1: Summary of different load balancing algorithms for HNOW............................. 19
Table 6.1: HNOW metrics expressed in datapoints.. 68
Table 8.1 Simulation parameters.. 88
Table 8.2: Confidence interval for an HNOW of workstations of different processing power.. 98
Table 8.3: Average number of steps for an HNOW with different memory capacities. 101
Table 8.4: Average number of steps for an HNOW with different network parameters..... 104
Table 8.5: Average number of steps for a complete HNOW. 107
Table 8.6: Average number of steps.. 112
Table 8.7: Average number of datapoints exchanged 112

ACKNOWLEDGMENTS

"Glory be to You, we have no knowledge except what You have taught us.
Verily, it is You, the All-Knower, the All-Wise." 2:32

I would like to take this opportunity to express my sincere gratitude to all those who provided time, effort and support to make this work possible. Special thanks to Dr. Ammar for his guidance and patience and to Dr. Subramani for his invaluable discussions. Also, special thanks to Dr. Celik for providing an enriching experience that inspired this work.

Finally yet importantly, I would also like to express that I would always be in debt to my parents and family who have always supported me and included me in their prayers.

GLOSSARY

CFD. Computational Fluid Dynamics.

Confidence Interval. A range of values constructed around a point estimate that makes it possible to state that an interval contains the population parameter between its upper and lower confidence limits. The most frequently used confidence interval is the 95% confidence interval. This can be interpreted, as there is only a 5% chance that the sample is so extreme that the 95% confidence interval calculated will not cover the population mean.

Discrete Event Simulation. (DES) concerns the modeling of a system as it evolves over time by representing the changes as separate events.

DLAH. Dynamic Load-balancing Algorithm for HNOW.

DP. Data Points.

HNOW. Heterogeneous Network of Workstation.

NP-Complete. The complexity class of decision problems for which answers can be checked for correctness, given a certificate, by an algorithm whose run time is polynomial in the size of the input (that is, it is NP) and no other NP problem is more than a polynomial factor harder. Informally, a problem is NP-complete if answers can be verified quickly, and a quick algorithm to solve this problem can be used to solve all other NP problems quickly.

PSPACE. The set of decision problems that can be solved by a Turing machine using a polynomial amount of memory, and unlimited time.

SPMD. Single Program Multiple Data.

Chapter 1

INTRODUCTION

1.1 Motivation

Due to the recent advances in high-speed network, network based distributed computing has become a low-cost alternative to dedicated parallel supercomputer systems. These systems are becoming widely available in academic and industrial environments. To benefit from the maximum computation power of these systems, it is necessary to use all available resources, namely old machines in addition to more recent ones. Such a network is called *heterogeneous network of workstations* (HNOW). Accordingly, a dynamic load-balancing algorithm is required to harness the computing power potential of this HNOW.

Dynamic load balancing strategies have been shown to be the most critical part of an efficient implementation of various algorithms on large distributed computing systems, as load imbalance can cause poor efficiency. A load-balancing algorithm must deal with different unbalancing factors, according to the application and to the environment in which it is executed. Unbalancing factors may be static, as in the case of processor heterogeneity, or dynamic like the unknown computational cost of each task, dynamic task creation, task migration, and variation of available computational resources due to external loads.

Each application type requires a different load-balancing strategy. Actually, it is very crucial to define the application type clearly and understand the underlying hardware architecture before attempting to design a load-balancing algorithm.

In this research, we focus on the *single program multiple data* (SPMD) programming model as it has been extensively used in parallel programming, due to the ease of designing a program that consists of a single code running on different processors. Moreover, data decomposition is a natural approach for the design of parallel algorithms for many problems. In addition, the SPMD model provides attractive scalability in terms of computational power and memory size.

1.2 Objectives

The main objective of this research is to design a scalable dynamic load-balancing algorithm for pipelined SPMD applications on a non-dedicated HNOW. Load balancing for heterogeneous parallel computing systems is a relatively new topic and has been investigated less frequently. Thus, we need to identify the HNOW parameters and its measuring units, study the pipelined SPMD application characteristics, and develop a taxonomy that categorizes current load-balancing strategies and enables us to design new strategies that suit our application. Accordingly, we will come up with an optimal solution if possible.

In addition, we need to identify performance parameters that measure the quality of the load-balancing algorithms; this will allow us to compare the performance of different load-balancing strategies.

The analysis of load-balancing algorithms involves performing exhaustive tests on a controlled HNOW environment. This is not applicable in real life so we turn to modeling. Unfortunately, there is not any theoretical model that captures the complexity of an HNOW, especially, its dynamic behavior. Therefore, we need to design a general reusable simulation model.

Besides designing a new load-balancing algorithm, we will develop a general framework that facilitates the design for new load-balancing algorithms in general.

1.3 Contributions

The main contributions for this research are summarized as follows:

- Proving that load-balancing for an HNOW is an NP-Complete problem (section 4.6). This means that there does not exist any optimal algorithm that can balance the HNOW.
- A general taxonomy for categorizing load-balancing algorithms for HNOW (section 2.5). A number of classifications have been proposed, but each classification is focused on certain applications. This taxonomy is more general, it allows us to classify current load-balancing algorithms and facilitate the design of new algorithms.
- Detailed scalability analysis for pipelined SPMD applications (chapter 5). Before designing a new load-balancing algorithm, it is imperative to study the underlying hardware architecture and the application type. The analysis shows that the pipelined SPMD

applications are able to adopt the cluster computation potential and to scale up with the cluster capabilities. In addition, the pipeline paradigm permits the overlapping between communication and computation, which eliminates the need for synchronization. However, the pipelined SPMD application shows that the performance of the system is very sensitive to the slowest workstation in the pipeline and the communication time.

- General reusable simulation model of HNOW (section 6.3). In order to construct a simulation model, we need to identify the input variables that define the system, output variables that define the performance measures, and a mathematical/logical relationship between the inputs and outputs that defines the system behavior. We have defined these points and discussed the discrete event simulation environment.
- Analytical performance bounds for estimating the performance of diffusive load-balancing algorithms on a homogeneous network of workstations (chapter 7). The diffusive strategy relies on neighboring workstations that communicate with each other to eliminate any load imbalance between them. It has been proved that this strategy drives the whole system to a global balance. We derived analytical bounds for the number of steps required to reach the balanced state.
- DLAH: a scalable dynamic load-balancing algorithm for SPMD applications on non-dedicated HNOW (section 7.3). DLAH algorithm is based on the diffusive strategy while incorporating the HNOW parameters. Its performance has been studied (chapter 8) and compared to other related algorithms. DLAH shows better performance.
- A general framework for designing new load-balancing algorithms (section 9.1). During this whole process we present the necessary steps needed to design new load-balancing algorithms.

1.4 Thesis Organization

Chapter 2 provides an overview of load-balancing algorithms in general, discussing the HNOW parameters, application types, and characteristics of load-balancing algorithms. It concludes by presenting a general taxonomy used to classify any load-balancing algorithm. We use this taxonomy in chapter 3 to categorize load-balancing algorithms of related work. Then, we formally define the problem in chapter 4 and prove that it is an NP-complete problem.

In chapter 5, we implement two case studies of pipelined SPMD applications. From these applications, we analyze the scalability of the pipelined SPMD applications and measure the different cluster parameters that will be used to build our simulation model in chapter 6.

In chapter 6, we discuss the current theoretical models of parallel computers and show that they are inadequate to capture the complexity of the HNOW, particularly, its dynamic behavior. Thus, we create a simulation model for the HNOW.

In chapter 7, we propose our algorithm DLAH and derive its analytical performance bounds. We implement the DLAH algorithm on the HNOW simulation model in chapter 8 and study its performance for different HNOW settings. Finally, we conclude our research in chapter 9, summarize our contributions, and provide several pointers for further research.

Chapter 2

OVERVIEW OF LOAD-BALANCING ALGORITHMS FOR HNOW

2.1 Overview of Heterogeneous Network of Workstations

Network based distributed computing has attracted a lot of attention lately, due to the recent advances in high-speed networks. It has become a cheap alternative to dedicated parallel supercomputer systems. As these systems are widely available in academic and industrial environments, it is becoming increasingly popular to use these resources. To benefit from the maximum computation power, it is necessary to use all available resources, namely old machines in addition to more recent ones. Such a network is called heterogeneous network of workstations (HNOW). Figure 2.1 illustrates a simple HNOW example.

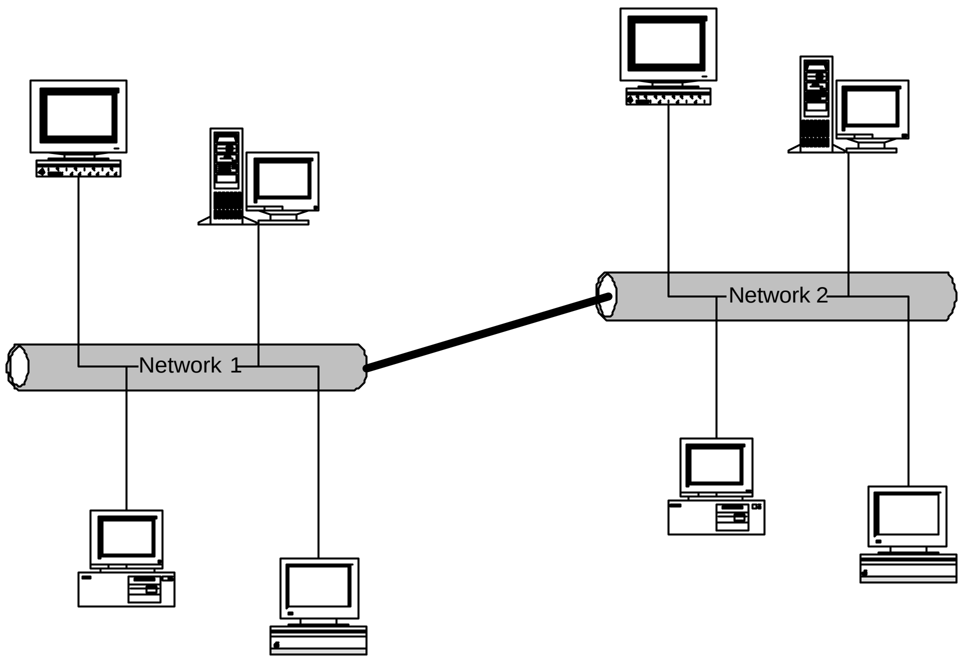


Figure 2.1: A simple scheme of an HNOW

The sources of heterogeneity in an HNOW include the processors of different speed; the memory, with different amount of available memory on different machines; the network, with varying cost of communication among pairs of processors; and the software level, with the various operating systems and environments. Accordingly, a dynamic load-balancing algorithm is required to deal with these different parameters to provide the best performance.

A dynamic load-balancing algorithm must deal with different unbalancing factors, according to the application and the environment in which it is executed. Unbalancing factors maybe static, as in the case of processor heterogeneity, or dynamic. Examples of dynamic unbalancing factors include the unknown computation task for each task, dynamic task creation and migration, and variation of available computational resources due to external loads from other applications.

For the last ten years, load-balancing problems for homogeneous parallel computing systems have been thoroughly studied. A large number of results have been gained with the help of simulations, theoretical investigations, or experimental applications {[Hil85], [PFK93], [FMD98], [DFM95], [SS94], [VS90], [WLR93], [CLZ99], [ZLP96]}.

Load balancing for heterogeneous parallel computing is a relatively new topic and has been investigated less frequently. In order to design load-balancing algorithms for HNOW, a number of parameters need to be well defined. These parameters should cover the heterogeneity of the network of workstations, the applications considered and the characteristics of the required load-balancing algorithm. Figure 2.2 summarizes the HNOW parameters that are discussed in details in the following sections.

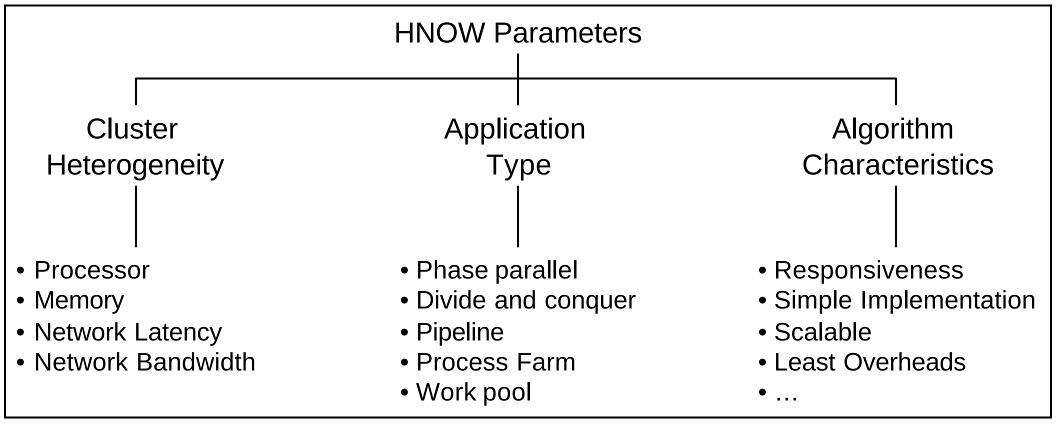


Figure 2.2: HNOW parameters

2.2 Cluster Heterogeneity Parameters

The sources of heterogeneity in a network of workstations are mainly attributed to processor speed, available memory, network latency, and network bandwidth.

- **Processor parameters:**

In a fully detailed processor model, we need to consider the speed of a processor in terms of the number of floating-point operations per second, and the number of integer operations per second. Multiple instructions and instruction pipelining would further complicate the model.

- **Memory:**

The required memory required by the application and the available memory should be considered in scheduling computations and data. Usually the total amount of memory in the cluster limits the data size considered by numerical scientific applications like weather modeling and computational dynamics. The amount of physical memory varies for different machines.

- **Network latency and bandwidth:**

This is one of the primary concerns for heterogeneous systems. Slow networks can make communication extremely expensive, and restrict the scalability of the system.

A detailed description of these parameters and how to measure them are discussed later in chapter 6.

2.3 Application Type

Parallel applications fall into a number of categories according to the parallel paradigm employed. Phase parallel, divide and conquer, pipeline, process farm, and work pool are some examples [WA99]. These programs may have regular or irregular computation and communication. Other characteristics, such as the communication to computation ratio, could dictate the decision to parallelize and the parallelization used. Accordingly, there is no load-balancing algorithm, which will have good performance for all the different application types.

2.4 Algorithm Characteristics

Before designing a load-balancing algorithm, it is necessary to define the essential features for the load-balancing algorithm besides the load balancing itself like scalability, responsiveness, least overheads, simplest implementation, etc. These characteristics are also used to measure the performance of the load-balancing algorithm accordingly. The most common parameters used to measure performance are convergence, extra load exchange, and load-balancing overheads.

- **Convergence:**

Convergence is a measure for the responsiveness, which is the number of steps required by the algorithm to reach a load balance state. The fewer steps it takes to reach the balanced state, the more responsive the algorithm is.

- **Extra load exchange:**

Extra load exchange is the total amount of extra load exchanged during execution for the load balancing. The fewer loads exchanged to reach a balanced state, the better the algorithm in estimating the load to be exchanged.

- **Load-balancing overheads:**

Load-balancing overheads are the total amount of overheads added by the load-balancing algorithm. These overheads are divided into computation overhead, and the communication overhead, which represents the extra messages produced by the load-balancing algorithm to exchange status values.

Before attempting to design a new load-balancing algorithm, we need to define a taxonomy that provides a terminology for describing different load balancing algorithms.

2.5 Load-Balancing Algorithms Classification

A number of classifications have already been proposed, but each classification was focused on certain applications. For example, [CK98] deals with scheduling of processes in distributed operating systems and with scheduling of jobs in parallel applications based on functional decomposition. [PRR03] deals with strategies for load distribution in SPMD applications. We will use our taxonomy proposed in [OA02], which, incorporates dynamic load balancing algorithms designed for homogeneous and heterogeneous systems, task migration and data parallel algorithms, central and distributed algorithms. In addition, we have shown how this taxonomy is used to design load-balancing algorithms for any kind of the application types.

In order to completely define a dynamic load-balancing algorithm, the main four sub-strategies (initiation, location, exchange, and selection) have to be well defined. A detailed discussion of these sub-strategies is presented in the following sections.

2.5.1 Initiation

The initiation strategy specifies the mechanism, which invokes the load balancing activities. This may be a *periodic* or *event-driven* initiation. Periodic initiation is a timer-based initiation in which load information is exchanged every pre-determined time interval. The event-driven is a usually a load dependent strategy based on the observation of the local load.

The load dependent strategies can be either *sender-initiated* or *receiver-initiated*. Sender-initiated means that over-loaded processors initiate activities while the receiver-initiated means that the under-loaded processors initiate the activities. Obviously, these initiation methods can be combined. Conditions like "over-loaded" and "under-loaded" are often defined by two-load thresholds L and H: if the load of a processor is less than L then it is under-loaded, if it is between L and H then it is normally loaded, and if it is larger than H, it is over-loaded.

Event-driven strategies are more responsive to load imbalances, while periodic strategies are easier to implement. However, periodic strategies may result in extra overheads when the loads are balanced.

2.5.2 Load-Balancer Location

This strategy specifies the location at which the algorithm itself is executed. The load-balancing algorithm is said to be *central* if it is executed at a single processor, determining the necessary load transfers and informing the involved processors. On the other hand, if all the processors take part in the load balancing decisions, the algorithm is classified as *distributed*.

Distributed algorithms are further classified as *synchronous* and *asynchronous*. A synchronous load-balancing algorithm must be executed simultaneously at all the participating processors. When a synchronous algorithm is invoked the processors stop processing the application and turn to load balancing. Usually this algorithm is used in applications in which application processing must stop at synchronization points for synchronism.

As for asynchronous algorithms, it can be executed at any moment in a given processor, with no dependency on what is being executed at the other processors.

Although the use of centralizing processor may lead to a bottleneck, it is important to remember that distributed strategies require load information to be propagated to all the processors, leading to higher communication costs.

2.5.3 Information Exchange

This specifies the information and load flow through the system. The information used by the dynamic load-balancing algorithm for *decision-making* can be *local* information on the processor or gathered from the surrounding neighborhood. Contrarily, it can be *global* information gathered from all the processors.

Although local information exchange strategies may yield to less communication costs, global information exchange strategies tend to give more accurate decisions.

The communication policy specifies the connection topology of the processors in the system, which determines the neighborhood of each processor. This topology does not have to represent the actual physical topology of the processors. A *uniform* topology indicates a fixed set of neighbors to communicate with, while in a *randomized* topology the processor randomly chooses another processor to exchange information with it.

In addition, the communication policy specifies the task/load exchange between different processors. In *global* strategies, task/load transfers may take place between any two processors, while *local* strategies define group of processors, and allow transfers to take place only between two processors within the same group.

2.5.4 Load Selection

The load exchange policy specifies the processors involved in the load exchange *(processor matching).* Apart from that, it specifies the appropriate load items *(load matching)* to be exchanged. Local averaging represents one of the common techniques. The overloaded processor sends load-packets to its neighbors until its own load drops to a specific threshold or the average load.

This taxonomy may be summarized as shown in Figure 2.3.

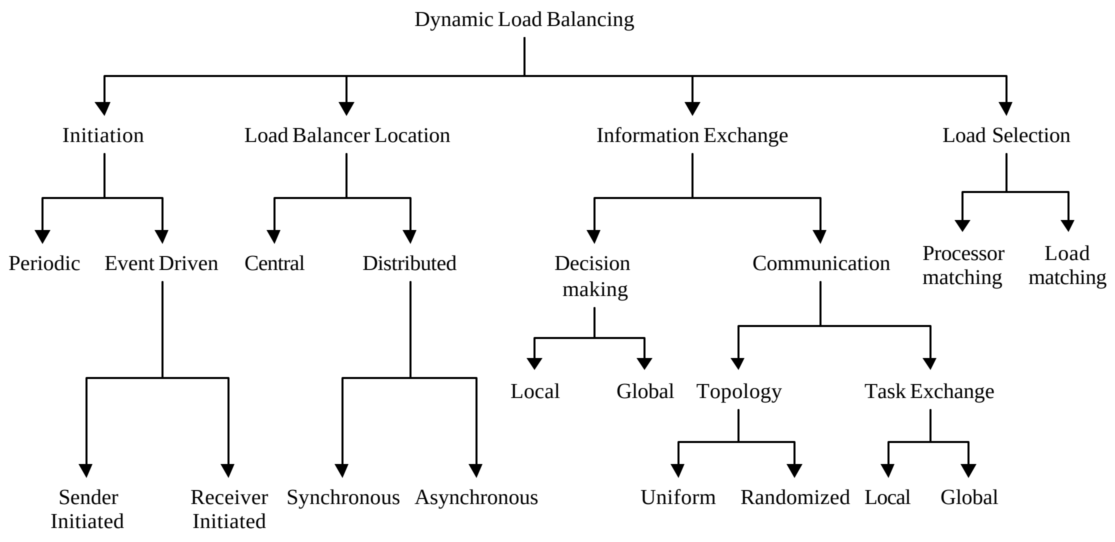


Figure 2.3: Dynamic load balancing taxonomy

Chapter 3

RELATED WORK

With the previous taxonomy, it is now possible to review load-balancing algorithms for HNOW and compare their different characteristics.

3.1 Automatic Heterogeneous Supercomputing (AHS) [DCG93]

AHS uses a quasi-dynamic scheduling strategy for minimizing the response time observed by a user when submitting an application program for execution. This system maintains an information file for each program that contains an estimate of the execution time of the program on each of the available machines. When a program is invoked by a user, AHS examines the load on each of the networked machines and executes the program on the machine that it estimates will produce the fastest turn-around time. Once the program is initiated on a specific processor, however, no further scheduling is performed.

- **Heterogeneity parameters:**

  Work load on every networked machine,

  Estimate for the execution time of each program on each of the available machines
- **Application type:**

  Whole independent programs executed on a single machine (no processor communication)
- **Algorithm strategy:**

  Initiation: Event driven (user initiated by submitting program)

  Load balancer location: Central

  Decision making/Communication: Global/ Randomized-Global

  Processor/Load matching: it estimates the fastest turn-around time obtained from the load on every workstation / the whole program is distributed.

This algorithm is limited to whole independent programs. It uses a central scheduler, which would be sufficient for a small number of jobs, but as the number of jobs increase the scheduler may become a bottleneck. Also, the algorithm requires providing an estimate time for the execution for each program on each workstation, which is not practical especially if there exists a lot of heterogeneous workstations.

3.2 Self-Adjusting Scheduling for Heterogeneous Systems (SASH) [HLA95]

It utilizes a maximally overlapped scheduling and execution paradigm to schedule a set of independent tasks onto a set of heterogeneous processors. Overlapped scheduling and execution in SASH is accomplished by dedicating a processor to execute the scheduling algorithm. SASH performs repeated scheduling phases in which it generates partial schedules. At the end of each scheduling phase, the scheduling processor places the tasks scheduled in that phase on to the working processors' local queues. The SASH algorithm is a variation of the family of branch-and-bound algorithms. It searches through a space of all possible partial and complete schedules.

The cost function used to estimate the total execution time produced by a given partial schedule consists of cost of executing a task on a processor and the additional communication delay required to transfer any data values needed by this task to the processor.

- **Heterogeneity parameters:**

  Cost function: CP(Tn,Pm) + CC(Tn,Pm) where:

  CP(Tn, Pm): Cost of executing a task Tn on a processor Pm,

  CC(Tn, Pm): Cost of additional communication delay required to transfer any data values needed by task Tn on processor Pm.

- **Application type:**

  Independent tasks.

- **Algorithm strategy:**

  Initiation: Event driven (depends on the shortest execution time achieved by the processors)

  Load balancer location: Central

Decision making/Communication: Global/Random-Global

Processor/Load matching: Using the cost function, the processor that will give the least cost will be selected.

This algorithm has an advantage of accounting for the non-uniformity in the tasks' communication and processing costs when scheduled on specific processors (although the calculation of the processor and the communication costs are not defined). Also, it overlaps scheduling with execution. On the other hand, it has a central (dedicated processor), which hinders scalability. Also, this algorithm assumes the computation cost and communication cost are constant throughout the scheduling which is valid for dedicated systems only.

3.3 Support for Parallel Loop Execution (SUPPLE) [OP97]

This strategy groups iterations into load chunks. Each processor executes its load chunks. Once a processor load decreases than a certain threshold, it asks other processors for chunk loads. A round-robin strategy is used by the underloaded processors to find an overloaded processor. Once an overloaded processor is located, it must choose the most appropriate number of chunks to migrate. Also, underloaded processors broadcast a termination message to reduce the number of processor checking.

- **Heterogeneity parameters:**
  Load chunks (overloaded and underloaded).
- **Application type:**
  Parallel loops.
- **Algorithm strategy:**
  Initiation: Receiver initiated

  Load balancer location: Distributed, Asynchronous

  Decision making/Communication: Local/ Random, Global

  Processor/Load matching: Overloaded processors are matched by underloaded processors in a round robin way. Load transferred is a ratio of the available chuck loads, depending on the current chunks and the number of processors (chuck/(2*processors)).

The main advantage of this algorithm is that it preserves the adjacency relationships among the processors that might have to exchange borders (load chunks sent are those without any other dependency except the sender). On the other hand, this algorithm only applies to parallel loops and it assumes uniform communication cost.

3.4 Data Migration Environnent (DAME) [CCN97]

DAME aims to ext end the Single Program Multiple Data (SPMD) programming paradigm to better harness heterogeneous NOWs with time-varying workloads. A centralized master collects information at run-time on the load of the workstations involved and explicit primitives are furnished to programmers in order to activate load checking and redistribute arrays in blocks whose sizes depend on the actual capacities of each workstation. Another drawback is the proposal of a global synchronization.

- **Heterogeneity parameters:**

  Computing power available (using external monitors or the application evolution itself),

  External workload (tasks in queue)
- **Application type:**

  SPMD applications
- **Algorithm strategy:**

  Initiation: Periodic

  Load balancer location: Central

  Decision making/Communication: Global/ Random, local

  Processor/Load matching: neighborhood processors exchange the workload.

This algorithm preserves the adjacency relationships among the processors that might have to exchange borders as it exchanges workloads only among neighbors. On the other hand, it’s a central periodic algorithm, which hinders much the scalability. It also assumes uniform communication cost.

3.5 Asymmetric Load Balancing on a Heterogeneous Cluster of PCs [B99]

Addresses the techniques by which the sizes of tasks are suitable matched to the processors and memories. A static load balancing strategy is used based on specific measurements and benchmarks

- **Heterogeneity parameters:**

  Relative computing power (using external benchmark applications),

- **Application type:**

  Data decomposed regular problems.

- **Algorithm strategy:**

  This algorithm implements a static load balancing strategy. It uses the previous benchmark results to partition the problem among them.

This strategy uses a classic static load-balancing algorithm. The main contribution is reducing the time taken to obtain the benchmark results. The static load balancing is more suitable for dedicated systems.

3.6 GR Protocol [LL96]

This protocol proposes an adaptive load-balancing algorithm (GR.batch) for heterogeneous distributed systems subject to different classes of tasks with different processing requirements. The key to the algorithm is to transfer a suitable amount of processing workload from queues of senders to receivers, which is determined dynamically.

- **Heterogeneity parameters:**

  Relative processor speed,

  Processing requirement for task,

  Task size,

  Arrival rate of task.

- **Application type:**

  Tasks eligible for relocation, with a known service (processing) demand.

- **Algorithm strategy:**

  Initiation: Receiver/Sender initiated

LB Location: Distributed, Asynchronous

Decision making/Communication: Local/ Random, Global

Processor/Load matching: Polls others until it finds a match. Workload is then negotiated between the sender and receiver.

The performance of the algorithm has only been evaluated using simulations. The results have shown the adaptive behavior of the algorithm towards heterogeneous systems. This algorithm assumes the heterogeneity lies only in the processing power of the different workstations. Also, the processing requirement (service demand) of each task must be known in advance.

3.7 Compile-time Scheduling Algorithms [CZL97]

Proposes a simple yet comprehensive model for use in compiling for a network of processors, and develop compiler algorithms for generating optimal and near-optimal schedules of loops for load balancing, communication optimizations, network contention, and memory heterogeneity.

- **Heterogeneity parameters:**

  Relative processor speed,

  Resident memory size,

  Start up time for messages,

  Message transfer rate,

  Number of iterations.

- **Application type:**

  Independent parallel loops on dedicated NOW for SPMD/ Master-Slave model of computation.

- **Algorithm strategy:**

  Compiler algorithms for generating optimal and near optimal schedules of loops were developed with specific cost functions.

This is one of the very few strategies that considered heterogeneity in many aspects. It incorporated most of the heterogeneous parameters. However, this strategy assumes a dedicated system. Also, it is application dependent, which means for different applications the relative processor power and resident memory size must be calibrated. This strategy is classified as a static algorithm.

Table 3.1 summarizes the different algorithms. It is clearly observed that most studies on heterogeneous network of workstations consider only the relative processing power of the workstations as the only factor for heterogeneity. In practice, heterogeneous network of workstations may contain different workstations with different memory size and network connections. In fact large computational applications like computational fluid dynamics CFD are limited by the memory size. Thus, it is important to consider all heterogeneous parameters when considering HNOW.

Algorithm	**Heterogeneity parameters**			**Application**	**Load balancer algorithm**	
	Processor	Memory	Network		Pros	Cons
AHS [DCG93]	Yes	Yes	No	Whole independent programs	Simple algorithm	- Estimating the execution time of each program on each machine - Central Scheduler
SASH [HLA95]	Yes	No	Yes	Independent tasks	Overlaps scheduling with execution	- Estimating the execution time of a task on a certain processor - Central Scheduler - Dedicated system
SUPPLE [OP97]	Yes	No	No	Parallel loops	- Distributed with asynchronous execution - Preserves adjacency relationships	- Broadcasting termination messages
DAME [CCN97]	Yes	No	No	SPMD Applications	- Preserves adjacency relationships	- Periodic - Central
[B99]	Yes	No	No	Data decomposed regular problems	- Suitable for dedicated systems	- Static
GR protocol [LL96]	Yes	No	No	Tasks eligible for relocation	- Distributed, Asynchronous	- Service demand of each task must be known in advance
Compile-time [CZL97]	Yes	Yes	Yes	Independent parallel loops	- Incorporates all heterogeneous parameters	- Compile time only which is suitable for dedicated systems only

Table 3.1: Summary of different load balancing algorithms for HNOW

Chapter 4

PROBLEM DEFINITION

4.1 Problem Statement

The problem considered in this research is defined as follows:

"Designing a scalable dynamic load-balancing algorithm for pipelined SPMD applications on non-dedicated heterogeneous network of workstations (HNOW)"

4.2 Problem Significance

We will discuss the different aspects of the problem statement and its significance in the following sections.

4.2.1 Non-Dedicated Heterogeneous Network of Workstations

Network based distributed computing has become a low cost alternative to dedicated parallel supercomputer systems. As these systems are widely available in academic and industrial environments, it is becoming increasingly popular to use these resources. To benefit from the maximum computation power, it is necessary to use all available resources, namely old machines in addition to more recent ones. Thus, a scalable load-balancing algorithm is required to harness the computing power potential of this heterogeneous network of workstations.

No load-balancing technique is suitable for all the different application types. Instead one studies the different attributes and programming model of one's application and develops a load-balancing algorithm accordingly. We have chosen the SPMD programming model as it has been widely used in parallel programming, due to the ease of designing a program that consists of a single code running on different processors. Moreover, data decomposition is a natural approach for the design of parallel algorithms for many problems [Mat96]. We will discuss in details the different SPMD attributes in the following section.

4.2.2 SPMD applications

By definition [Qui94], SPMD applications are suitable for implementation on the network of workstations model. This model provides attractive scalability in terms of computational power and memory size.

SPMD involves a number of parameters namely the computation time, communication time and the communication pattern. The computation to communication ratio is called the granularity ratio, which could dictate the decision to parallelize and the parallelization method used.

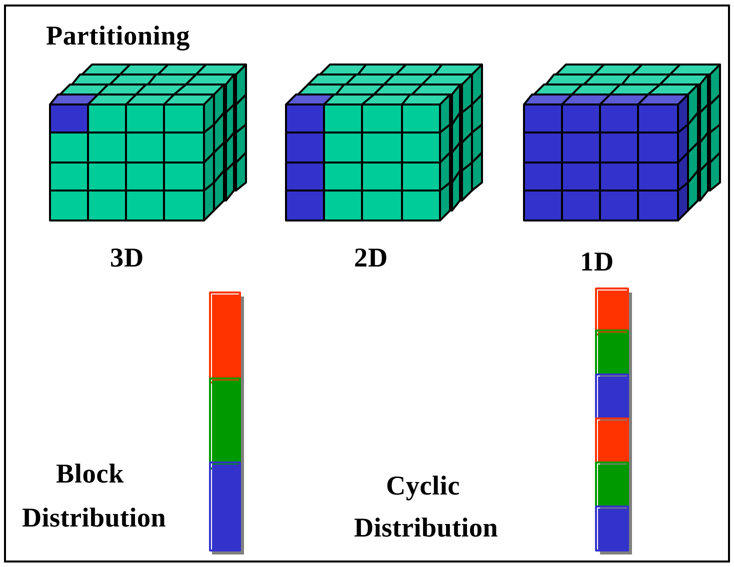


Figure 4.1: Data partitioning and distribution

SPMD also involves partitioning of data (data domain decomposition) and distributing it to different workstations. Depending on the problem a suitable partitioning that increases the granularity ratio is desired [OA201]. Figure 4.1 illustrates a number of possible partitioning and distribution options.

SPMD applications usually involve running a simulation for a large number of simulation steps (iterations); we will refer to that as SPMD simulation steps. Another aspect that we should consider is the regularity of the computations and the communication pattern between different workstations. For example, although each workstation executes the same program and has the same domain size, each may have different number of computations like molecular dynamics simulations. Also, each application has its own communication pattern to exchange different boundary data.

In this study, we are concerned with regular domain computations and uniform communications applications. In fact, a large number of simulation applications belong to this class of applications like computational fluid dynamics (CFD). Other examples of SPMD applications are simulations like ship wake simulations [OA101], Heat transfer simulations, weather simulations, and image processing applications [MG97].

4.2.3 Scalable Dynamic Load-Balancing Algorithm

SPMD applications are scalable by nature, as the same program is being executed on different workstations but with different data sets. Thus, in order to take full advantage of the heterogeneous network of workstations, it is necessary to design a scalable load-balancing algorithm that does not hinder the SPMD scalability.

From the previous discussions, it becomes imperative to design a scalable dynamic load-balancing algorithm that can cope to the non-dedicated heterogeneous workstations in the cluster and ensure that each workstation is assigned a fair workload proportional to its capabilities.

4.3 Formal Problem Description

We consider an environment of a set of autonomous non-dedicated workstations connected by a communication network. The system is represented by an undirected graph $G=(V,E)$, where V is the set of workstations labeled from 1 through P, and $E \subseteq V \times V$ is the set of edges. Each edge $(i,j) \in E$ corresponds to the communication link between processors i and j. At time t node $v_i \in V$ has a processing power $F_i(t) \in \mathbb{R}$, available memory $M_{f_i}(t) \in \mathbb{R}$ and a workload of $W_i(t) \in \mathbb{R}$. At a certain time t, each node takes time (iteration time) $L_i(t) \in \mathbb{R}$ to execute the workload with its current resources. The goal is to redistribute the total workload among the workstations such that if G is not changed for some finite time A, then a global balance can be achieved. In other words, the goal is to redistribute W_i

proportional to the iteration time L_i. In addition, the algorithm used to redistribute the total workload should be scalable with the number of workstations.

4.4 Problem Assumptions

The problem assumes the execution of pipelined SPMD application on a heterogeneous network of workstations with the following characteristics:

- Computational intensive SPMD applications, which involves a large number of iterations (much larger than the number of workstations) like heat transfer simulations, weather simulations and computational fluid dynamics (CFD) simulations (sections 5.1). In these applications, the problem is usually divided into a grid of data points. The simulation progresses in time with each iteration.
- the main data domain is uniform, which means that each data point requires the same amount of computations.
- each processing unit has the same executable program, and is assigned a continuous part of the main data domain,
- neighboring workstations communicate regularly to exchange boundary variables,
- pipelined in the sense that each workstation can't begin its execution before it receives the boundary variables from the neighbors,
- there is no synchronization required between the different processing units.

4.5 Problem Parameters

We clearly identify the different parameters that will be used throughout the literature for describing the problem.

4.5.1 Pipelined SPMD Application

The pipelined SPMD application is divided among the workstations such that:

- The smallest computation unit is called a *datapoint*.
- Each workstation i is assigned a number of continuous datapoints W_i.

- The application requires l loops (iterations) to complete, where the number of loops is much bigger than the number of workstations.

4.5.2 Heterogeneous Network of Workstations Parameters

The HNOW consists of P different workstations; each workstation has processing power of F datapoints per second, free memory M_f datapoints, and swap access time of M_a datapoints per second. The workstations are connected to each other through a network with bandwidth of BW datapoints per second; the bandwidth may differ between different workstations.

The total execution time is given by:

$$T_{Total} = \sum_{l} (\underset{i \in [1..P]}{MAX} (T_{comp}[i] + T_{mem}[i] + T_{comm}[i] + T_o[i] + T_{bal}[i])) \qquad (4.1)$$

Where T_{comp} is the computational time given by:

$$T_{comp} = \frac{\text{workload per processor}}{\text{Processor speed}} = \frac{W}{F} \qquad (4.2)$$

- T_{mem} is the extra time taken using the swap memory given by the product of both the datapoints in the swap memory and the retrieval time:

$$T_{mem} = M_s \times T_r \qquad (4.3)$$

- T_{comm} is the time taken in exchanging boundary data points, given by:

$$T_{comm} = \text{Latency} + \frac{\text{Datapoints exchanged}}{\text{Transfer rate}}$$

$$T_{comm} = L + \frac{LE}{BW} \qquad (4.4)$$

- T_o: is the extra time introduced by the parallelization scheme, which includes: parallelism coding overhead, synchronization overhead, and load imbalance overhead.

- T_{bal}: is the extra time experienced when using a load balancing algorithm, which includes: extra computations added by the load balancing algorithm, extra load exchanged for balancing, and miscellaneous messages used to exchange load status.

The main goal is to execute the SPMD application in minimal time. In other words to minimize the total execution time T_{total} .

It is not hard to see that minimizing the above equation to obtain the optimum solution involves gathering all the different parameters from all the workstations to one central processor. This central processor should execute a certain algorithm to minimize the total execution time and then send to each workstation its required load distribution with its neighbors. Obviously, this is not a scalable technique. In the following section, we prove that this problem is actually an NP complete problem and thus there exists no optimal solution that can be implemented.

4.6 Problem NP-Completeness

The task of distributing the total workload among a cluster of heterogeneous workstations seems very hard. Actually, we will prove that it is an NP-Complete problem. We will consider the case of distributing the workload on a dedicated cluster of workstations, i.e., a static case.

4.6.1 Problem Description

We consider a dedicated cluster of workstation composed of a finite set of heterogeneous workstations interconnected by a communication network. The underlying parallel computation is performed on a large data set that is divided among the workstations. We represent the system as follows:

- Set of processors $P = \{p_1, p_2, \dots p_m\}$, each processor is defined by the following two tuple, $p_i = (f_i, m_i), i \in [1, m]$ where f_i is the processor speed and m_i is the memory available for processor p_i .
- Set of data domains $D = \{d_1, d_2, \dots d_n\}$, each represents a segment of the whole computational data domain,
- Set of communication time $C = \{c_{ij} : i, j \in [1 \dots m]\}$, which represents the communication time between the processors p_i and p_j .

- Set of tasks $T = \{t_1, t_2, \dots t_n\}$, each task is defined by the tuple $t_i = \left(d_i, \vec{b_i}\right), i \in [1 \dots n]$, where d_i is the computational data domain, and $\vec{b_i}$ is the communication requirement from data domain d_i to the other domains.
- Length $l\left(t_j, p_i, \vec{c_j}\right) \in \mathbb{Z}^+, i \in [1 \dots m], j \in [1 \dots n]$, which represents the time taken for a task t_j on processor p_i, including the communication time $\vec{c_j}$ required to communicate with the other processors.

4.6.2 Problem Statement:

We will refer to our problem as NOW scheduling defined as follows:

INSTANCE

Set of Tasks T, number $m \in \mathbb{Z}^+$ of processors, length $l\left(t_j, p_i, \vec{c_j}\right) \in \mathbb{Z}^+$ for each task t_j on processor p_i with communication time requirement $\vec{c_j}$, $i \in [1 \dots m], j \in [1 \dots n]$.

PROBLEM

An m-processor schedule for T,i.e., a function $f : T \rightarrow [1 \dots m]$, that minimizes the total execution time given by $\max\limits_{i \in [1 \dots m]} \sum\limits_{f(t_j) = i} l\left(t_j, p_i, \vec{c_j}\right)$.

We can rewrite the problem as a decision problem as follows:

QUESTION

Is there an m-processor schedule for T,i.e., a function $f : T \rightarrow [1 \dots m]$ that meets the overall deadline $D \in \mathbb{Z}^+$, such that the total execution time $\max\limits_{i \in [1 \dots m]} \sum\limits_{f(t_j) = i} l\left(t_j, p_i, \vec{c_j}\right) \le D$?

Before discussing the complexity of this problem, let us refer to a similar problem “Minimum Multiprocessor Scheduling”.

INSTANCE

Set of Tasks T, number $m \in \mathbb{Z}^+$ of processors, length $l\left(t_j, p_i\right) \in \mathbb{Z}^+$ for each task $t_j \in T$ on processor p_i, $i \in [1 \ldots m]$, $j \in [1 \ldots n]$.

PROBLEM

An m-processor schedule for T, i.e., a function $f : T \rightarrow [1 \ldots m]$, that minimizes the total execution time given by $\max_{i \in [1 \ldots m]} \sum_{f(t_j)=i} l\left(t_j, p_i\right)$.

QUESTION

Is there an m-processor schedule for T,i.e., a function $f : T \rightarrow [1 \ldots m]$ that meets the overall deadline $D \in \mathbb{Z}^+$, such that the total execution time $\max_{i \in [1 \ldots m]} \sum_{f(t_j)=i} l\left(t_j, p_i\right) \leq D$?

This problem has been proven to be an NP-Complete problem [LST90]. We will reduce this problem to our problem. The reduction process involves creating a polynomial time algorithm R which transforms the inputs of the multiprocessor scheduling to equivalent inputs of our problem.

First, let us assume that we have an algorithm for the NOW scheduling problem that answers the scheduling question. We will then construct a polynomial algorithm that reduces the multiprocessor scheduling problem inputs to the NOW scheduling problem inputs.

The inputs to the multiprocessor scheduling are given by $< T, P >$, where T is the set of tasks and P is the set of processors, each processor p_i has a processing power of f_i and infinite memory. While the inputs for our problem are $< T, P, M, C >$ where T is the set of tasks and P is the set of processors, each processor p_i has a processing power of f_i and memory of $m_i \in M$, and C is the set of communication times between each two processors.

Construct the following function R, by simply adding the memory and communication parameters to the inputs. The memory is initialized to a very large number, while the communication is initialized to zero.

$$R(\langle T, P \rangle) = \langle T, P, M = \infty, C = 0 \rangle$$

Accordingly, we can then use the NOW scheduling problem to answer the question of the multiprocessor scheduling problem. That means if the NOW scheduling algorithm answered "yes" there is a solution then the answer will be "yes" to the multiprocessor scheduling problem too and vice versa.

That means that the multiprocessor scheduling is no harder than the NOW scheduling. Since the multiprocessor scheduling is NP-Complete, then it follows that the NOW scheduling is also an NP-Complete problem and there exists no optimal algorithm to solve it.

It is worth noting that in our analysis we only considered a dedicated cluster of heterogeneous workstations, i.e., the cluster resources dedicated for the application are static, which is simpler than the non-dedicated version.

Chapter 5

PIPELINED APPLICATIONS: IMPLEMENTATION AND ANALYSIS

In this chapter, we will review two different types of pipelined applications and demonstrate the advantages of using pipelining as a parallel paradigm. Each application has been implemented on a different kind of underlying hardware architecture: a cluster of workstations and a supercomputer. We will discuss the details of each implementation, analyze its performance, and show how to extract the different parameters required for our simulations. These parameters include both hardware and application parameters.

The first section discusses a CFD application implemented on a cluster of workstations, while the second section discusses a real-time distributed dispatcher implemented on a supercomputer.

5.1 Domain Decomposition of Large Eddy Simulations of Ship Wakes[1,2]

Simulation of turbulent fluctuations in ship wakes is one of the complex applications of large eddy simulations in computational fluid dynamics (CFD). Ship wakes simulation requires extensive computations and large amounts of computer resources. The accuracy of ship wake prediction is limited by the memory of the workstation.

In this section, we present the parallel implementation of large eddy simulations (LES) of a flat plate wake using domain decomposition technique for a cluster of workstation environment. We present the results of the implementation executed on a cluster of workstations and show how the pipeline paradigm scales up with the number of workstations.

[1] This work was funded by the DoD research grant No. N000014-98-1-0611, monitored by the Office of Naval Research to WVU

[2] The implementation is a joint work with the department of mechanical and aerospace engineering under the supervision of Prof. Ismail Celik. Special thanks to Andrei Smirnov and Shoaping Shi.

5.1.1 Introduction

Algorithmic improvements and faster machines, particularly parallel machines, provide the opportunity for effectively using large-eddy simulations (LES) for the problems of practical importance. The main objective in this paper is to predict the turbulent flow development in a ship wake using cluster computing and investigate its scalability using the domain decomposition technique.

This requires analysis of the ship wake flow, development of an efficient and accurate simulation of the in the ship wakes by refining the large eddy simulation methodology, setting up the computational domain, analysis of the computer resources, and analysis of the results.

5.1.1.1 Related work

Direct Numerical Simulation (DNS) of turbulence requires many CPU days even months and Gigabytes of memory. These requirements limit most DNS to using supercomputers, available at supercomputer centers. With the rapid development and low cost of PCs, PC clusters are evaluated as a viable low-cost option for scientific computing. A number of studies have been made to evaluate the cluster capabilities. One of the recent studies [KEB99] presented a comprehensive overall evaluation of the applicability of PC/Linux clusters for DNS of turbulence. Low and high-end PC clusters were compared to existing supercomputers, both at kernel and application level, to evaluate the CPU and network capabilities. The research concluded that parallel simulations using Ethernet-based networks indicate inefficiency in communications above four processors. Internal timings indicate that the bottleneck is due to group communications (e.g. MPI all-to-all).

Another recent research [OB99] was concerned about the different programming paradigms. These are message passing (MPI), shared memory (CC-NUMA), and multithreading. A dynamically adapting, unstructured mesh application was used to evaluate the different paradigms. The research concluded that the multithreading offers the highest scalability, the message passing offers the best portability while the shared memory offers the simplest coding.

Using the previous results, we preferred to use MPI for portability and pipelining the different computational domains in order to avoid group communication and synchronization delays.

5.1.1.2 Simulation of turbulent flows using LES

Turbulent forced convection occurs in many important technological applications, such as mechanical, aerospace, electrical, computer, chemical, and nuclear engineers and in flows of interest to meteorologists and earth scientists. The three-dimensional unsteady Navier-Stokes (NS) equations are known to govern such flows. Direct numerical simulation (DNS), which involves numerically solving the full unsteady NS equations, is currently limited to only the simplest flow geometries and low Reynolds numbers. Alternatively, the Reynolds averaged Navier-Stokes (RANS) equations, obtained from time averaging the unsteady NS equations, require much less computational resources and are used successfully to compute many flows of practical importance. However, the turbulence models used in conjunction with the RANS equations are not applicable to a wide range of flow geometries and are unsuccessful for many turbulent flow situations.

Large eddy simulation (LES) is a compromise between DNS and RANS. LES relies on the fact that small scales of turbulent motion are nearly isotropic and independent of the geometry, whereas the large scales of turbulent motion are mostly anisotropic and vary from flow to flow. The small-scale motion, which is mostly a function of the amount of energy that must be dissipated and therefore more universal, is filtered out of the governing equations and modeled with a subgrid scale (SGS) model. The large-scale motion is computed directly by numerically solving the three-dimensional, time dependent filtered NS equations. Although LES is not computationally expensive as DNS, it still requires large amounts of computer resources.

5.1.1.3 Significance

Through the different experiments, we were able to confirm:

- The capability of pipelined domain decomposition technique to adopt the cluster computing potentials and scale up with the number of workstations,
- The importance of granularity on the scalability,
- The ability to predict the execution time and number of workstations required for the ship wake simulation using a small-scaled pilot simulation.

In the following sections, we will briefly describe the methodology used in LES simulations. Then, discuss in details the cluster environment used and present the domain decomposition strategy. Finally, we will present the implementation results and analyze its scalability.

5.1.2 Methodology

Large-eddy simulation of the turbulent wake is an expensive but useful approach, since it allows the detailed study of the flow in which rapid stream-wise adjustment is present. For this reason, the near-wake region is very difficult to model theoretically. The simulation of a turbulent wake, especially the ship wake, requires a refinement in the LES technique. The flow is spatially developing, inhomogeneous along all three dimensions and strongly influenced by the interaction with the free surface. This creates the need for time-dependent turbulent inflow and outflow boundary conditions.

In this study, an LES code originally developed by Zang and Street [ZS92] and modified by Shi and Celik [SSC01], boundary fitted grid technique is used in which a single-connected curved domain can be transferred into rectangular domain with unit cubic cells. The Navier-Stokes solver itself is built on regular shaped grid and, consequently, it is efficient and easily modified if necessary. A global second order accurate scheme in both temporal and spatial directions is applied. The governing equations are discretized by using finite volume method. With the exception of the convective terms, all the spatial derivatives are approximated with second-order central differences. The convective term in the momentum equation is discretized using QUICK scheme or central differencing. Because there is no explicit relation to solve for the pressure in time, the fractional step method in conjunction with the projection method is applied to solve the incompressible Navier-Stokes equation. Multigrid technique is used to solve the pressure Poisson equation. Although in LES, explicit schemes are preferable, some implicitness, e.g. via Crank-Nicolson time splitting, can be introduced if stability is an issue. In this code, time advancement is semi-implicit, using an explicit Adams-Bashforth scheme for the convective and source terms and an implicit Crank-Nicolson scheme for the diffusive terms.

There are many different kinds of subgrid-scale (SGS) models used to capture the unresolved (small) subgrid-scale motion. A basic and widely used SGS model is the Smagorinsky eddy-viscosity model. We [SSC00] have applied Smagorinsky model, dynamic viscosity and dynamic mixing model in our study of wake flow, such as wake of bluff body, wake of flat plate and wake of a ship. None of them could predict all the turbulence features very well. On the other hand, the results without any subgrid-scale model but with some degree of numerical dissipation, using, for instance the QUICK discretization scheme, could provide reasonable predictions and are much more attractive from computational viewpoint and stability considerations. The detail information including the governing equations and the subgrid-scale models can be found in the references [SCS99] and [SSC00].

5.1.3 Computer Resources

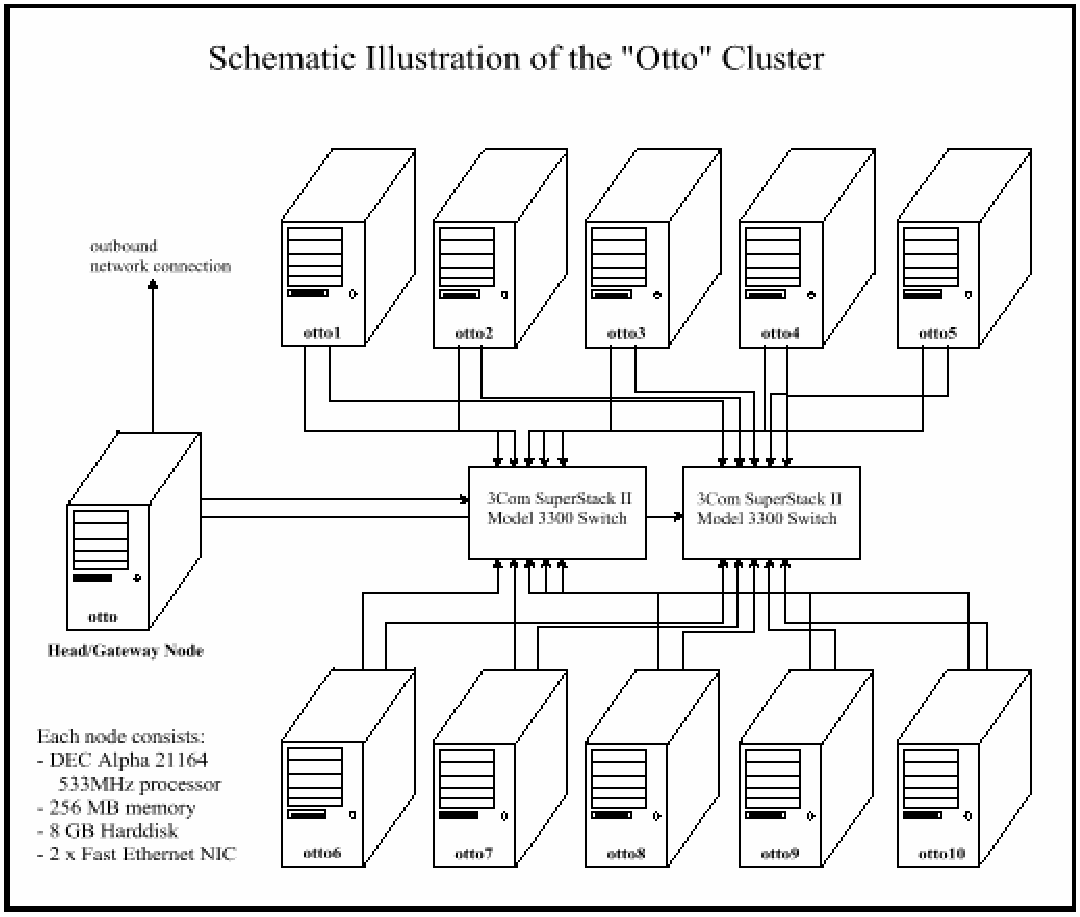
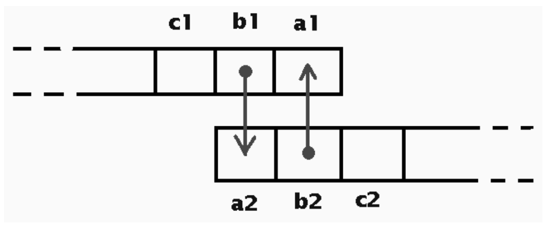

Schematic Illustration of the "Otto" Cluster
outbound network connection
otto1
otto2
otto3
otto4
otto5
3Com SuperStack II Model 3300 Switch
3Com SuperStack II Model 3300 Switch
otto
Head/Gateway Node
Each node consists:
- DEC Alpha 21164 533MHz processor
- 256 MB memory
- 8 GB Harddisk
- 2 x Fast Ethernet NIC
otto6
otto7
otto8
otto9
otto10

Figure 5.1: CFD lab's cluster [SS01]

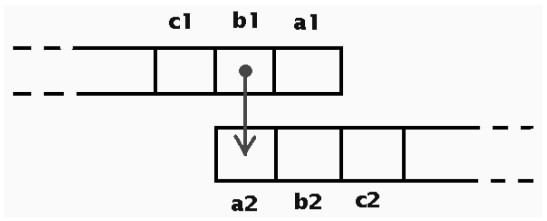
The experiments were implemented on a Beowulf cluster [BWF] of 10 DEC-Alpha workstations at the CFD laboratory of WVU. The cluster is running Linux as a software operating system, interconnected by a private high-speed network as seen in figure 5.1. We implemented message passing parallel programming model using MPI [GLS94], [MPCH] protocol for message passing.

5.1.4 Domain Decomposition Strategy

The conventional domain decomposition technique [Hwg93] for elliptic problems is realized through a two-way exchange of data at the boundaries of the domains [Smn92] and [DGP84] as illustrated in figure 5.2(a) for a one-dimensional problem. This guarantees the physical integrity of the solution and the convergence to the corresponding single domain case. However, this strategy may carry an excessive communication overhead for three-dimensional CFD simulations. If the problem is parabolic in one of the spatial directions one can employ a more efficient one-way communication approach illustrated in figure 5.2(b). This may reduce communication overhead considerably especially when non-blocking send-routines are used.



(a) Two-way decomposition



(b) One-way decomposition

Figure 5.2: Decomposition strategy: a,b,c – cells of the grid in different computational domains

Considering these factors, we implemented the first parallel version of the LES code using one-way data exchange. This approach is valid for ship-wake applications since the flow in most of the ship-wake region has a parabolic character. Even though small re-circulation zones exist in the proximity of ship's stern, they do not stretch far enough and can be contained entirely within the first sub-domain.

To exploit the parabolic nature of the solution the computational domain of the whole wake should be subdivided into Nd sub-domains by cutting it with planes normal to the ship-velocity. Figure 5.3 illustrates a typical layout of two sub-domains. Data exchange between the domains occurs at the plane EH. As can be seen this implementation of the domain-decomposition technique is rather simple and straightforward. It also carries minimum communication overhead and is suitable for ship-wake applications.

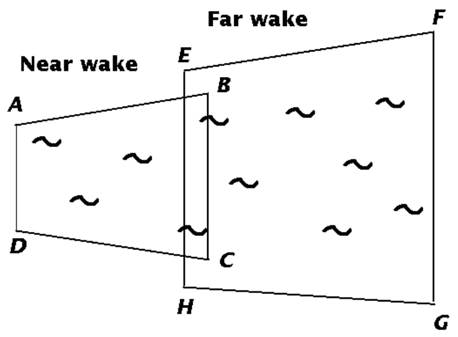


Figure 5.3: Wake decomposition

The drawback of the scheme is the necessity to provide additional outlet boundary conditions for each domain, which can alter the character of the flow close to the domain outlet, as compared to the two-way coupled case.

5.1.4.1 Assumptions

- identical workstations in a cluster using MPI.
- SPMD (Single Program Multiple Data) model is used, in which the same program code runs on all the workstations, while the data domain is partitioned among them.
- The domain used is a three dimensional domain.

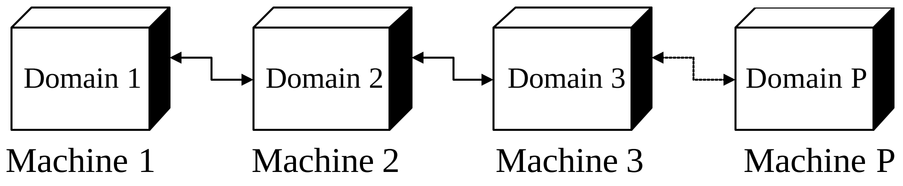


Figure 5.4: Domain decomposition of the application

The application is considered pipelined in the sense that the task cannot proceed before it exchanges messages with the proceeding task of an adjacent domain as shown in Figure 5.4.

5.1.4.2 Scalability Analysis

A computer system, including all its hardware and software resources, is called scalable if it can scale up (i.e., improve its resources) to accommodate ever-increasing performance and functionality demand and/or scale down (i.e., decrease its resources) to reduce cost. To exploit the power of scalable parallel computers, the application programs must also be scalable [HX98].

The speedup can be used as a measure to the scalability. The speedup is defined as the ratio between the execution time of an application on a parallel system and the best-known serial algorithm.

$$S = \frac{T_s}{T_P} \qquad (5.1)$$

Where:

- T_s is the execution time taken using the serial algorithm,
- T_P is the execution time taken in case of the parallel system.

Since the memory resources limit the accuracy of the ship wake, we use the fixed memory model for our analysis [SN93]:

- **Cluster case:**

The total execution time:

$$T_P = (\text{number of cycles}) \times (\text{time for one pipeline cycle})$$
$$T_P = (L + P - 1)(T_{comp} + T_{comm})$$

Where:

- L : is the number of instances (time steps),
- P : is the number of workstations (processors),
- T_{comp} : is the computation time of one instance in a workstation,
- T_{comm} : is the communication time to exchange variables.

- **Serial case:**

The total execution time taken to process the same data:

$$T_s = (\text{total number of instances}) \times (\text{time of instance})$$
$$T_s = (LP)T_{comp}$$

$$\therefore S(L,P) = \frac{LP}{(L+P-1)\left(1+\frac{T_{comm}}{T_{comp}}\right)} \qquad (5.2)$$

The ratio $\frac{T_{comm}}{T_{comp}}$ is called the granule size, the bigger the granule size the better the speedup.

For ideal cases $L \gg P$. Then:

$$S(L,P) = \frac{P}{\left(1+\frac{T_{comm}}{T_{comp}}\right)} \qquad (5.3)$$

For the system to be scalable, it should perform the same or better as the number of processors and/or problem size increases. The efficiency is used to measure this aspect.

$$\text{Efficiency} = \frac{\text{Speed up}}{\text{no. of processors}}$$

$$E(L,P) = \frac{L}{(L+P-1)\left(1+\frac{T_{comm}}{T_{comp}}\right)} \qquad (5.4)$$

The above equation suggests that for small L the scalability is worse as we increase the number of processors. For large L, the only factor that affects the scalability is the granule size.

- **The effect of granularity on the speedup:**

According to equation 5.2, it is obvious that as the granularity increases, the better the speedup. The granule size can be chosen by the amount of data assigned to the workstation. It should be noted that the granule size has a maximum value limited by the memory of the processor. Therefore, care should be taken in choosing the granule size.

It may seem that for a fixed size problem that by increasing the processing elements the speedup will increase linearly. This is only valid as long as the granule size remains constant. Actually as we increase the number of processing elements for fixed size problems, T_{comp} decreases as each processor takes a smaller load and T_{comm} may increase as network load increases with more overheads. This gives smaller granularity, which leads to lower efficiencies.

Therefore, it is important to investigate the granularity of the application by measuring a number of parameters like the network bandwidth/latency, network performance under different loads, and computation regularity.

For our LES application, equal loads were assigned to each processor as shown in figure 5.5. We concluded the following:

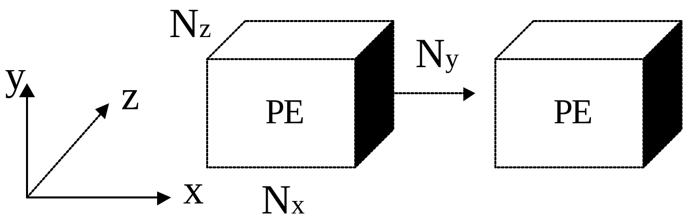


Figure 5.5: Load assignment to processing elements

- The computation is regular, i.e., $T_{calc}\,\boldsymbol{a}\,N_x N_y N_z$ where N_x, N_y and N_z are the number of grid nodes in the direction of X, Y and Z axis.
- The communication is made through a dedicated switch with only 10 workstations, so we can neglect the effect of network load.
- $T_{Comm}\,\boldsymbol{a}\,N_y N_z$, which represents the interface between the different processing elements.
- For fixed size problems, $N_x\,\boldsymbol{a}\,\frac{1}{P}$ where P is the number of processors. As we increase the number of processors, the load assigned for each processor decreases.

$\therefore \frac{T_{comm}}{T_{comp}} = GP$ where G is the granularity constant.

Substituting in equation 5.2:

$$\therefore S(L,P) = \frac{LP}{(L+P-1)(1+GP)} \qquad (5.5)$$

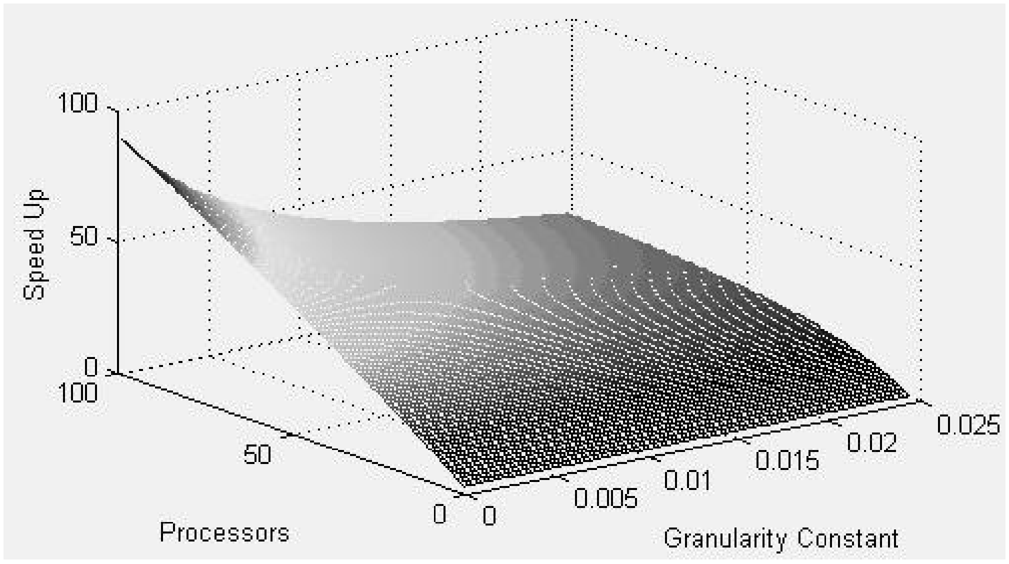


Figure 5.6: Speedup and granularity

Plotting the above equation for different values of granularity constants and different number of processors, we obtain figure 5.6. It is shown that a slight increase in granularity constant reduces the effective speedup. Actually, the granularity may even inhibit any speedup regardless of the number of processors added. Hence, care must be taken in choosing the granule size and measurements should be made to ensure this choice.

5.1.5 Simulation Results

A flat plate wake was used for demonstration and making pilot simulations. A number of different simulations were carried out with different parameters to investigate the influence of each parameter on the results.

5.1.5.1 Domain information

For flat plate wake, the domain size is 1.0m x 0.2m x 0.6m in X, Y, and Z direction respectively. A number of simulations have been made for the flat plate wake with different grid size.

Non-uniform grid is used in the different directions. Note that X represents the stream-wise direction, Y represents the vertical direction, and Z represents the span-wise direction, as shown in figure 5.7.

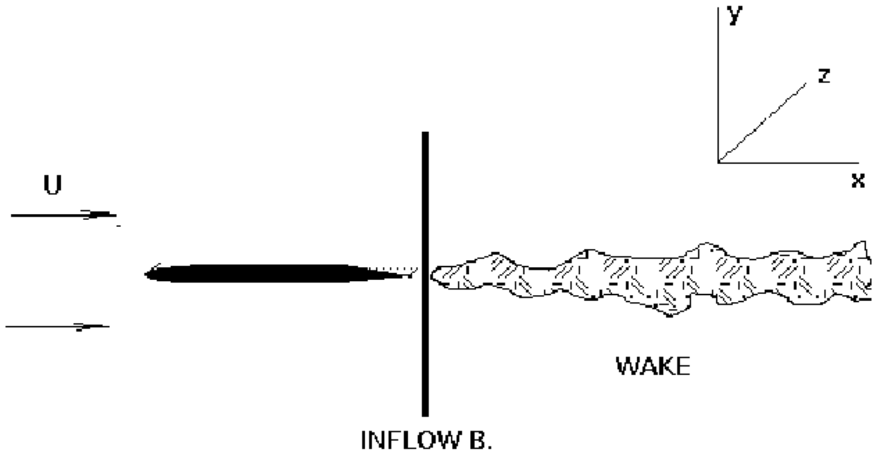


Figure 5.7: The schematic of the flat plate wake [SS01]

5.1.5.2 Boundary conditions

For both cases, inflow boundary and outflow boundary are applied in x direction. For outflow boundary, we applied convective outflow boundary, i.e.

$$\frac{\partial u}{\partial t} + U_{con} \frac{\partial u}{\partial x} = 0 \qquad (5.6)$$

Symmetry boundaries are used in Y direction and periodic boundaries are used in the span-wise direction. At the free surface, slip in X and Z directions are allowed, but the velocity component normal to the free surface is set to zero. The free surface is approximated as a flat plane without walls. For more details, see [SS01].

5.1.5.3 *Simulation test*

A 2-processor simulation was successfully conducted on two processors of the cluster using a grid size of 18x18x18 nodes for each domain. The stream-wise velocity contours is shown in figure 5.8 from [SS01]. The two domains are consistent very well. This indicates that the one-way decomposition technique is successful.

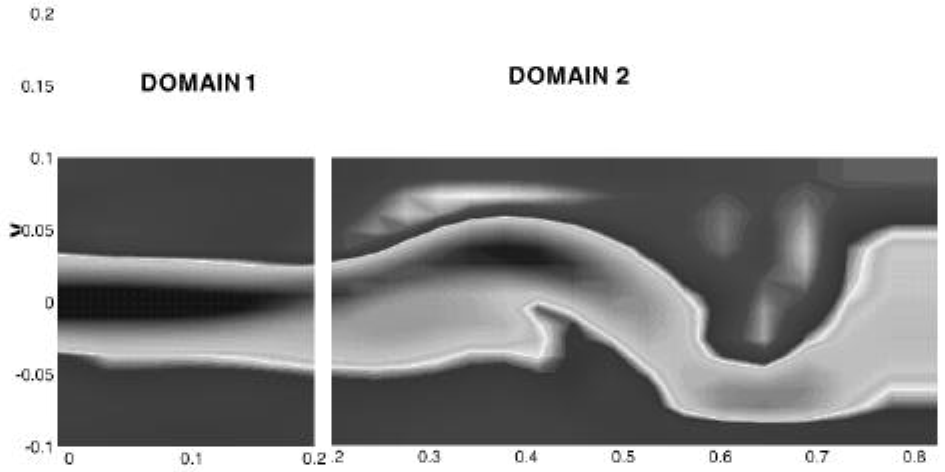


Figure 5.8:The stream-wise velocity contours of one-way decomposition scheme.

5.1.5.4 *Simulation results*

A number of different experiments were carried out on the cluster with different parameters. The parameters used for the analysis are the number of processors, the number of grid nodes per workstation (accuracy), and the total simulation time (number of iterations). The results of each experiment with various parameters are shown in the following sections.

- **Effect of the simulation time**

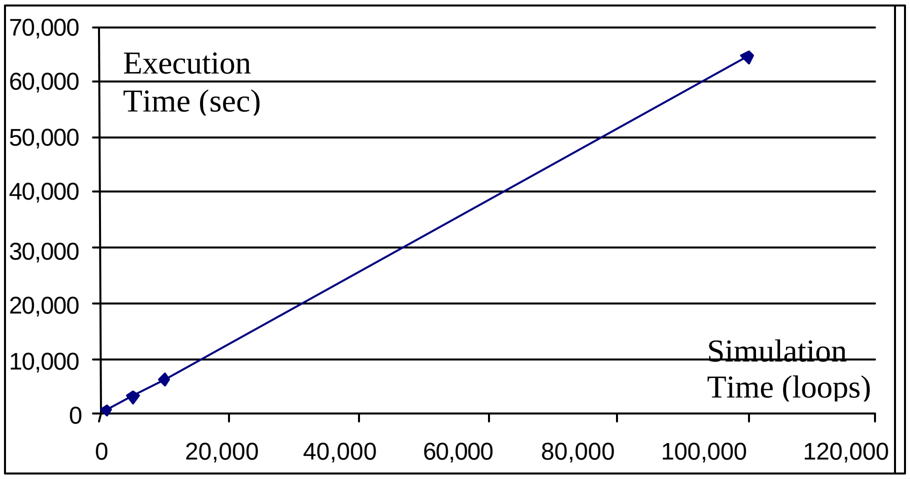


Figure 5.9: Execution time versus the simulation time

These simulations were executed on three workstations with a grid size of $18 \times 18 \times 18$ nodes each, for different simulation time (iterations). We can predict the speedup using equation 5.3, and the measured quantities of T_{comm} and T_{comp}.

$T_{Comm} = 0.0012\text{sec}$, $T_{Calc} = 0.637\text{sec}$. Then, $S \approx 2.99$.

As illustrated in figure 5.9, the execution time is nearly directly proportional to the required simulation time (number of iterations). This is due to the fact that for this case the ratio $T_{comm} / T_{comp} \ll 1$, so equation 5.3 is reduced to $S(L,P) \approx P$. This makes it easy to predict the total execution time required for long simulations using the results from a small number of processors.

- **Effect of the number of processors**

In these simulations, we increase the problem size proportional to the number of processors. In other words, the number of grid nodes for each processor is not changed when adding another processor. The number of grid nodes used for each workstation is $18 \times 18 \times 18$ with a simulation time of 5000 loops.

Figure 5.10, illustrates the speedup experienced while increasing the problem size with the number of processors. It is clearly seen that the speedup is directly proportional to the number of processors as long as the granularity remains constant.

This demonstrates one of the important features of cluster computing, which is the ability to scale up the size of the problem for the same execution time with the number of available workstations. Also, we notice that the measured speedup is directly proportional to the number of processors as predicted.

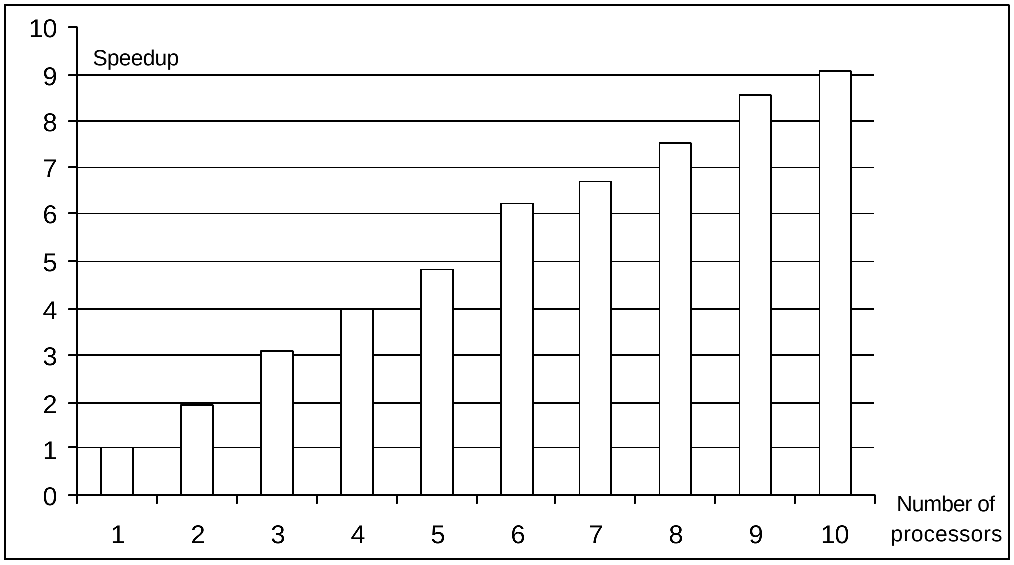


Figure 5.10: Speedup experienced while increasing the problem size with the number of processors.

- **Effect of the number of grid nodes per workstation**

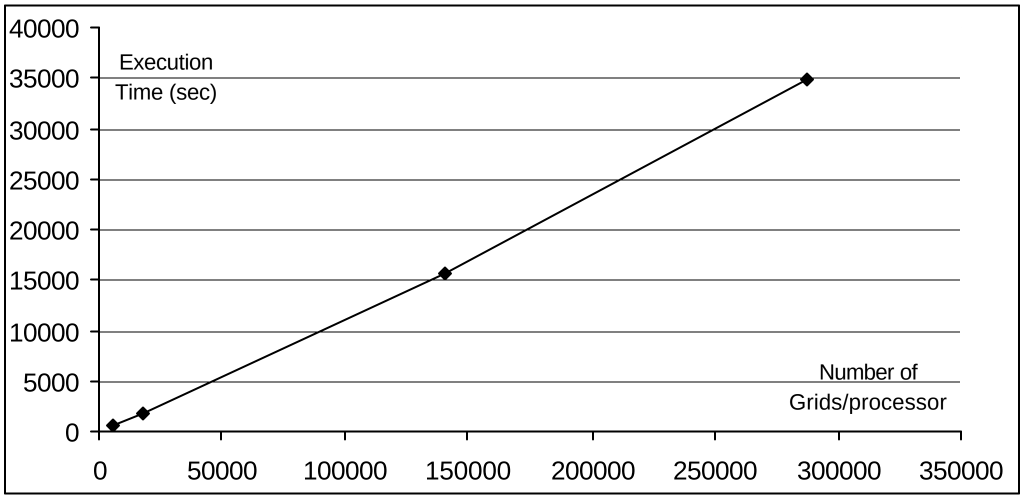


Figure 5.11: Execution time versus number of grid nodes per workstation.

To test for the effect of the number of grid nodes per workstation on the execution time, a number of experiments were carried on 4 workstations for a simulation of 1000 iteration. Figure 5.11 illustrates the results obtained, which shows that the execution time is directly proportional to the total number of grid nodes per workstation, or in other words the execution time is of order $O\left(N^3\right)$ where N is the grid nodes per dimension. Note that when the total number of grid nodes per workstation is changed other parameters like memory size and cache replacement methods could affect the results.

With the previous experiments and deducted relations, one may predict the execution time or the number of processors required using a small-scaled pilot simulation. For example, consider the execution time for $18 \times 18 \times 18$ grid nodes per workstation for 8 processors is 3254 seconds for a simulation time of 5000 simulation ticks (iterations). One can estimate the execution time for $66 \times 66 \times 66$ grid nodes per workstation, on four processors with simulation time of 1000 ticks. Using the previous relations, the estimated execution time is 32082 seconds. Actually, the estimated execution time is fairly close to the measured value, which is 34760 seconds. A number of experiments were carried out wit different parameters and compared to the estimated execution time; the relative error never exceeded 10%.

Accordingly, a small-scaled simulation is sufficient to estimate the actual execution time for a large-scaled simulation.

5.1.6 Conclusion

Beowulf clusters can provide ample and cost effective resources for high performance computing. However, it encapsulates many features that are unique, and offers hope of providing a solution to the needs of many supercomputer users. The Beowulf architecture provides a standard message-passing hardware and software environment, with a low cost of entry.

We have implemented pipelined domain decomposition technique in simulating a flat plane wake flow. One of the great merits of pipeline domain decomposition is that it permits the overlapping between communication and computation, so synchronization time can be neglected.

The domain decomposition has shown its ability to adopt the cluster computation potential and to scale up with the cluster capabilities. The numerical experimental have verified the analytic study of the scalability of the domain decomposition.

It is possible to predict the execution time and the number of processors required for a simulation by using a small pilot simulation, and the relations deducted between the grid size, simulation time and the number of processors. This is important especially when computational time is being paid for. However, one of the weaknesses of this domain decomposition technique is that the execution time is very sensitive to the slowest workstation in the pipeline, which can cause degradation in performance.

5.2 Distributed Algorithm for Partially Clairvoyant Dispatchers[3]

Real-time systems are finding use in complex and dynamic environments such as cruise controllers, life support systems, nuclear reactors, etc.,. These systems have separate components that sense, control and stabilize the environment towards achieving the mission or target. These consociate components synchronize, compute and control themselves locally

[3] The implementation is a joint work under the supervision of Dr. K. Subramani. Special thanks to K. Yellajyosula.

or have a centralized component to do the above. Distributed computing techniques improve the overall performance and reliability of large real-time systems with spread components.

Partially Clairvoyant scheduling was introduced in [Sak94] to determine the schedulability of hard real-time jobs with variable execution times. The problem of deciding the Partially Clairvoyant schedulability of a constrained set of jobs has been well studied in the literature [GPS95, Cho00, Sub03]. These algorithms determine the schedulability of the job set offline and produce a set of dispatch functions. The dispatch functions of a job depend on the start and execution times of the jobs sequenced before the job. The dispatching problem is concerned with the online computation of the time interval to start a job such that none of the constraints is violated. In certain situations, the dispatcher fails to dispatch a job, as it takes longer to compute the interval within which the job has to be dispatched; this phenomenon is called *Loss of Dispatchability*. For a job set of size n, sequential approaches using function lists suffer from two major drawbacks; $\Omega(n)$ dispatching time and the Loss of Dispatchability phenomenon. Existing approaches to this problem have been along sequential lines using stored function lists.

In this section, we implement and evaluate a distributed pipeline-dispatching algorithm for Partially Clairvoyant schedules. For a job set of size n, the algorithms have dispatch times of $O(1)$ per job. All the processors execute jobs assigned to them and compute the dispatch functions in a certain defined order.

5.2.1 Introduction

Real-time systems are characterized by deadlines, dependencies between jobs and parameter variability; execution time is one such parameter. The execution time of a job can vary due to input dependent loops, caching and compiler-architecture mapping of the machine as explained in [Sub02]. Another factor varying the execution time is the alteration of the clock frequency by power aware processors. Transmeta's LongRun, AMD's PowerNow, or Intel's SpeedStep technologies vary the processor voltage or clock frequency to decrease the power consumed by the processors according to the system load. [AM+01] proposes to decrease energy consumption for real-time systems by adjusting processor speed and reusing the unused processor cycles.

Distributed computing is quite popular in various consumer and safety-critical applications today. The huge applications that exist today have multiple levels of processing spread-out in various domains. In such applications, local control is preferred with little data moving between the levels. These applications have intelligent online algorithms that are invoked in response to the fluctuations within the environments. Consider a few examples of distributed systems that need to respond in small durations when presented with unpredictable circumstances:

- Clusters of independent robots are being developed to achieve missions in hostile environments like surveying landscape and searching for survivors [RG+02]. The robots control their own motion, communicate with each other, and complete jobs distributed among them to complete the mission. The motion of a robot requires complex modeling and has to consider various kinematics equations, which require different computing times [YYM01, HCF03]. This motivates the requirement of online controllers, which would control the actions of the robot and maintain the deadlines across them. Since the environment is dynamic, we cannot use any online strategy for controlling the system. [RS+00] presents the hardware and software components of a robotic team that survey a landscape communicating with the central robot.
- An automobile cruise control maintains the speed of the car by coordinating and monitoring the actions of different components of the engine such as fuel injection, braking, transmission, etc.,. New cars have adaptive shifting algorithms, modifying shift points based on road conditions, weather, and the driver's individual habits. The cruise control system can vary the car acceleration according to the exact speed of the car provided by the Anti-lock Braking System. These systems require variable times to compute the required torque to drive the car at the speed safely. A 7-series BMW has 63 microprocessors while a Mercedes S-class has 65 microprocessors.

For job-sets of size n, [GPS95] and [Cho00] propose $\Omega(n)$ dispatching algorithms for Partially Clairvoyant schedules. These dispatch algorithms may result in the phenomenon called "Loss of Dispatchability" due to the linear dispatch time. [TL99] proposes linear time, online algorithms that schedule firm aperiodic, hard sporadic and periodic jobs in priority based real-time systems with a complexity of $q(n)$. [Sub00] proposes a parallel algorithm with

$O(1)$ dispatch time per job to eliminate Loss of Dispatchability. This algorithm requires n processors, uses $O(n)$ space on each processor, and provides a tradeoff between the computing time and resources required, i.e., the constraints are met by increasing the resources to compute the interval during which the job can be dispatched. Hard real-time systems require reliability of the system at any cost [SP92, SS+98].

In this section, we use the number of the processors as an input parameter much less than the number of jobs and propose an algorithm with a $O(1)$ dispatch time. We explore the dispatchability of schedules for different job-sets with different timing constraints and show that scaling up the number of processors would successfully dispatch some non-dispatchable schedules. This work marks the distribution of the jobs of a Partially Clairvoyant schedule across processors using a simple heuristic and demonstrates communication between the processors and synchronization of job execution across the processors. We present results observed on dispatching schedules of different sizes with varying number of processors.

5.2.2 Problem Statement

5.2.2.1 Job model

Let $J = \{J_1, J_2, \ldots, J_n\}$ be a set of non-preemptive, ordered hard real-time jobs. We assume that job execution starts at time $t = 0$.

5.2.2.2 Constraint model

The constraints on the jobs are described by the following equation:

$$A.[\vec{s}\ \vec{e}]^T \leq \vec{b},\ \vec{e} \in E \qquad (5.7)$$

where,

- A is an $m \times 2n$ rational matrix; unless explicitly stated otherwise, we assume that the constraint set comprises of standard constraints between two jobs. Constraints express the relationships between the start times or finish times of jobs.

- E is an axis-parallel rectangle aph represented by:

$$E = [l_1, u_1] \times [l_2, u_2] \times \ldots [l_n, u_n] \qquad (5.8)$$

The aph E models the fact that the execution time of job J_i can assume any value in the range $[l_i, u_i]$, i.e., it is not constant.

- $\vec{s} = [s_1, s_2, \ldots, s_n]$ is the start time vector of the jobs, and
- $\vec{e} = [e_1, e_2, \ldots, e_n] \in E$ is the execution time vector of the jobs.

5.2.2.3 Query model

Suppose that job $J_a, 1 \leq a \leq n$ has to be dispatched. We assume that the dispatcher has access to the start times $\{s_1, s_2, \ldots s_{a-1}\}$ and the execution times $\{e_1, e_2, \ldots e_{a-1}\}$ of the jobs $\{J_1, J_2, \ldots J_{a-1}\}$.

Definition 5-1

A Partially Clairvoyant Schedule of an ordered set of jobs, in a scheduling window, is a vector $\vec{s} = [s_1, s_2, \ldots, s_n]$, where each $s_i, 1 \leq i \leq n$, is a function of the start time and execution time variables of jobs sequenced prior to job J_i.

Definition 5-2

A Partially Clairvoyant Schedule of $\vec{s}$ for the constraint system 1 is said to be feasible, if for all sequences $b_{seq} = \langle s'_1, e'_1, s'_2, e'_2, \ldots, s'_n, e'_n \rangle$, where s'_i is chosen as per $\vec{s}$ and $e'_i \in [l_i, u_i]$, we have $A.[\vec{s'}\ \vec{e'}]^T \leq \vec{b}$, where s'_i and e'_i are numeric vectors, corresponding to the sequence b_{seq}.

The discussion above directs us to the following formulation of the schedulability query:

$$\exists s_1 \forall e_1 \in [l_1, u_1] \exists s_2 \forall e_2 \in [l_2, u_2] \ldots \exists s_n \forall e_n \in [l_n, u_n] A.[\vec{s'}\ \vec{e'}]^T \leq \vec{b}\,?$$

The combination of the Job model, Constraint model and the Query model constitutes a scheduling problem specification within the E-T-C scheduling framework [Sub02].

Definition 5-3

A feasible Partially Clairvoyant schedule is said to be dispatchable on a machine M, if for every job J_i, M can start executing J_i such that none of the constraints are violated.

Definition 5-4

A safety interval for a job is the time interval during which the job can be started without violating any of the constraints imposed on it.

In this research, we are concerned with the dispatching problem, i.e., how to compute the safety intervals of the jobs, such that the jobs can be dispatched safely within the proper time intervals, assuming that a Partially Clairvoyant schedule was obtained from the above query.

5.2.3 Motivation and Related Work

[Sub02] proposes the E-T-C framework to formalize problems in real-time systems, which takes into account the variability of execution time, complex relationships between jobs and clairvoyance of the system. [Sak94] introduced Partially Clairvoyant scheduling to reduce the inflexibility of static scheduling in hard real-time systems. Partially Clairvoyant scheduling is explained in detail in [GPS95, Sub03] and [Sub00]. [GPS95] proposes a sequential online dispatching algorithm for the schedule generated using the algorithm in [Sak94]. The algorithm stores lists of dispatch functions and has dispatch time proportional to the number of jobs. The computing overhead of the online dispatcher may cause constraint violation, i.e., the time after computing the safety interval (l_b, r_b) exceeds r_b. The phenomenon by which a job cannot be dispatched is called Loss of Dispatchability. [Sub00] proposes a parallel online algorithm for eliminating Loss of Dispatchability for Partially Clairvoyant schedules.

The original single controller algorithm proposed in [Sub00] assumes that there are as many processors as the number of jobs n. The jobs are executed on a central processor, which then broadcasts the start and execution time of the completed job to the other processors. The n supporting processors receive the start and execution times of a job J_k and update the safety intervals, by relaxing the 4 constraints between the job completed and the job assigned to it. The satellite processor k sends the safety interval of job J_{k+1}. This algorithm has $O(1)$ dispatch time per job and uses $O(n)$ space per processor.

5.2.4 Computer Resources

Experiments were conducted on Lemieux system at the National Science Foundation (NSF) Terascale computing system at the Pittsburgh Supercomputing Center (PSC). Lemieux comprises 750 Compaq Alphaserver ES-45 nodes and two separate front end nodes. Each computational node contains four 1GHz processors SMP with 4 Gbytes of memory and runs the Tru64 Unix operating system. Nodes are connected using a quadrics interconnection network.

The quadrics network has two building blocks, a programmable network interface called Elan and a low-latency high bandwidth communication switch called Elite. The Elan network interface links the high-performance, multi-stage Quadrics network to the nodes. The Elan also provides substantial local processing power to implement high-level message-passing protocols, such as MPI, in addition to generating and accepting packets to and from the network. The Elite switch provides 8 bidirectional links supporting two virtual channels in each direction, an internal 16×8 full crossbar switch and a bandwidth of 400 MB/s with a latency of $35ns$. We used MPI libraries in C to implement the dispatcher.

5.2.5 Experiment Design

The parameters used to generate the test cases are:

- Number of jobs n : The number of jobs in the schedule
- Execution time (l,u): The lower and upper limit of the execution time of the jobs.
- Spacing time (p,q): The time interval $[p,q]$ in which the next job would begin.
- Number of constraints E : The number of standard constraints between jobs.

5.2.5.1 Generation of Partially Clairvoyant schedules

We specify the number of jobs n , the number of constraints E , the execution time $[l,u]$ and the spacing time $[p,q]$. We also specify a random seed for generating the constraints. The generating algorithm GA does as follows:

- For each job, GA generates two numbers between l and $u\,(u > l)$, which are bounds for the execution time of the job.
- Between every job J_i and $J_{i+1}\,(1 \leq i \leq n)$, GA generates standard constraints of the form $s_i + e_i \leq s_{i+1}$ and $s_{i+1} \leq s_i + e_i + c$ where c is a random number between the p and q . The generator generates at least $2n$ constraints.
- If the number of constraints $E > 2n$, then constraints are generated between the two jobs at random such that a Partially Clairvoyant schedule exists.

5.2.5.2 Schedule execution

The dispatcher takes as input the number of jobs, execution time periods, a random seed, and the dispatch functions. The dispatch functions are stored in a two-dimensional triangular array. Arrays are maintained to store the start time, execution time, and execution time periods of the jobs.

During execution, the control passes from one processor to the other with every job. In order to avoid clock synchronization and drift problems of the processors in our implementation, we also send the current time from one processor to the other with the start and execution times.

The communication time is measured by dividing the sum of time required to send and receive a message to the next processor by 2. This approximately simulates the time that would be required by the next processor to receive a message. In case the receiving processor is still updating constraints, then the waiting time is automatically added to the communication time, which agrees with the other case that relies on the processors' clock.

The pipeline approach alternates between different stages of updating and execution. At first, a processor executes a job and sends the start and execution times to the next processor. After sending the current time, the processors update the safety intervals of the remaining jobs assigned to them. The other processors relay the start and executing times before updating the safety intervals of the jobs assigned to them. Figure 5.12 illustrates the expected outcome compared to the serial implementation.

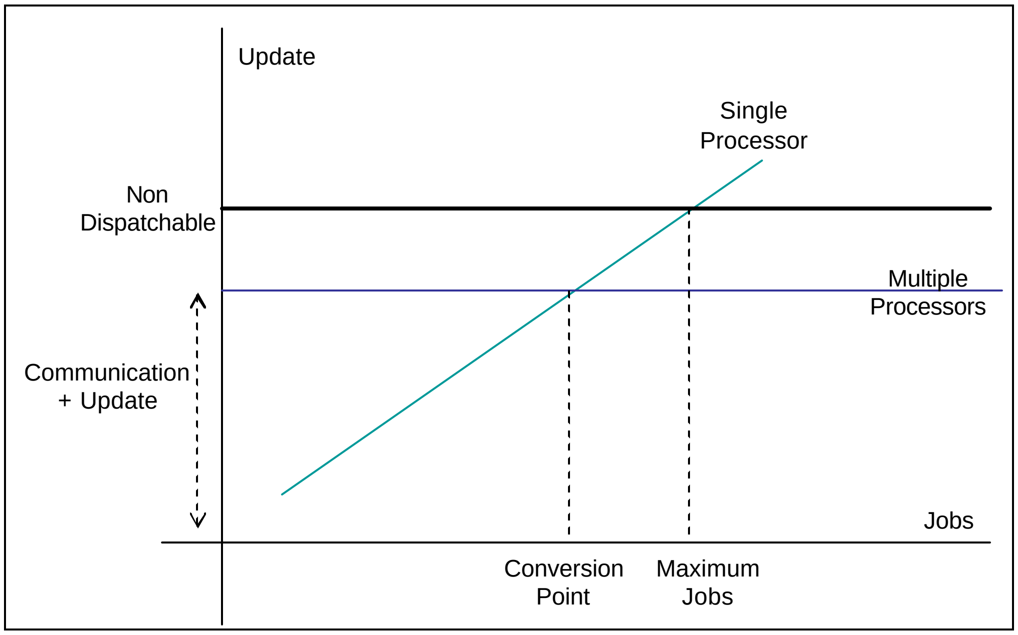


Figure 5.12: Single and Parallel implementation performance with respect to the number of jobs

5.2.6 Empirical Analysis

For our hardware architecture, we noticed that we have two stages of communication; intra-node communication and inter-node communication. Intra-node communication is communication between processors of the same node through the ELAN interface, while inter-node communication is communication between processors on different nodes through the Quadrics interconnect network. Thus, we performed two sets of experiments for each implementation of the dispatchers. In the first set of experiments, all processors are chosen from the least number of nodes containing them while the second set of experiments chooses one processor per node resulting in inter-node communication only.

During our tests, we observed serious overshoots with the communication time that includes waiting time for the updates. These overshoots are probably due to uncontrolled traffic over the network or other operating system jobs, where the response time depends on the network load between the nodes. We performed many experiments and observed the communication time between the different nodes. We noticed that the normal communication time is limited to an interval of time and the overshoots occur further than ten times the length of this interval as seen in figure 5.13.

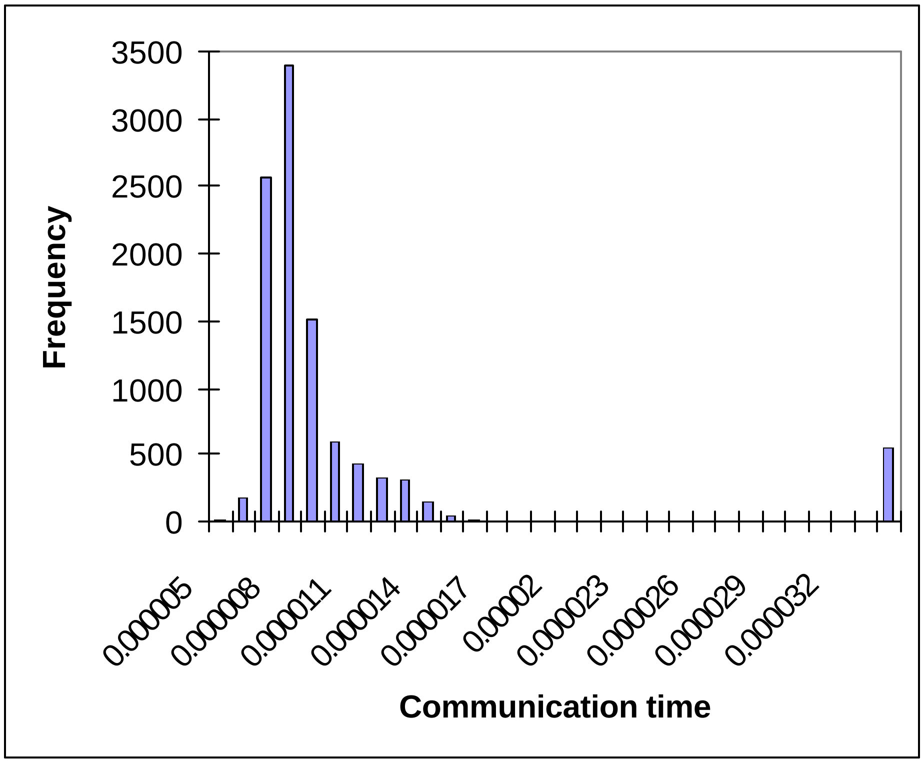


Figure 5.13: Histogram of the communication time observed in our experiments

Accordingly, we neglected theses overshoots by checking if the observed communication time is greater than 10 times the previous communication time. These overshoots can be safely neglected, as real-time systems require dedicated machines with predictable performance.

5.2.6.1 Using all processors on a node (packed node)

We conducted experiments to investigate the dispatchability of the job sets with different number of processors. In these experiments, nodes are allocated such that all the processors of the same node are used where applicable. For example, to conduct an experiment with 9 processors, 3 nodes are allocated; all 8 processors of the 2 nodes and 1 processor of the third node are used.

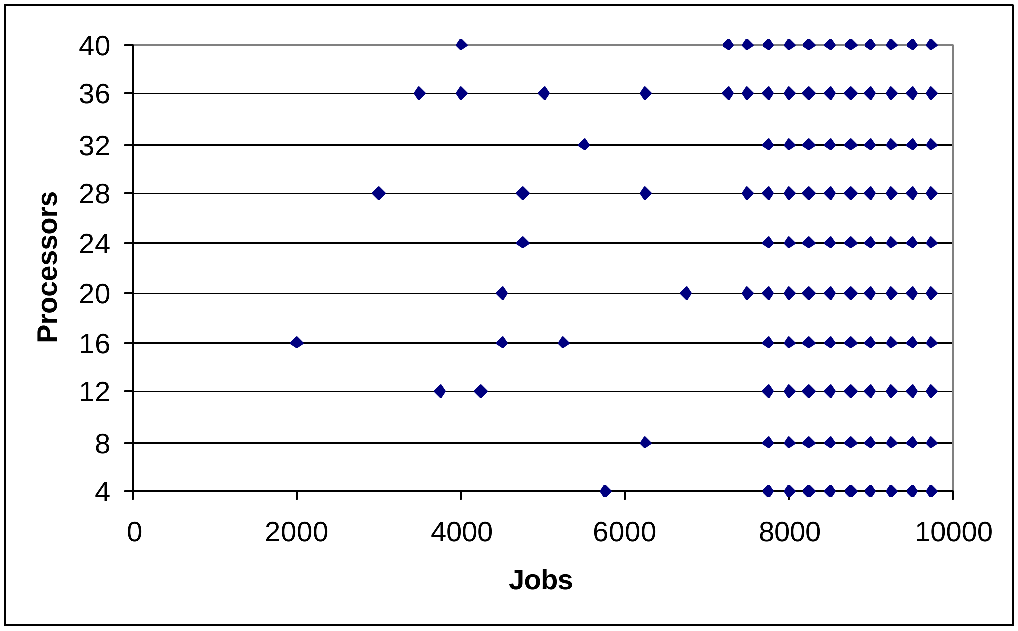


Figure 5.14: Undispatchable jobs for packed nodes

As illustrated in figure 5.14, each processor set dispatches different job sets. There is no observable relation between the number of processors and job sets other than that all the processors failed to dispatch job sets larger than 8000 compared to 5000 for the serial implementation. In addition, we observe that increasing the number of computing nodes does not increase the dispatchability sets. We conclude that, multiple processors can dispatch more job sets than the serial implementation and each processor set dispatches different job sets.

5.2.6.2 Using one processor per node

In these experiments, we eliminate the effect of intra-node communication by allocating one processor per node. For example, to conduct an experiment with nine processors, we allocate nine processors on nine nodes. Figure 5.15 illustrates the job sets that are not dispatchable by the processor sets.

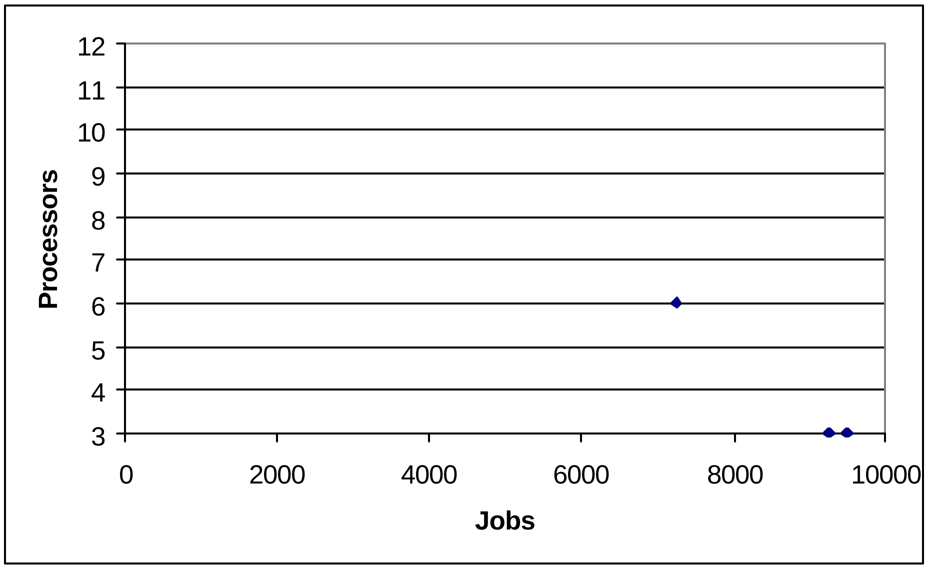


Figure 5.15: Undispatchable jobs while using one processor per node

It is clear that this implementation of the dispatcher is superior to the previous implementation as all of the job sets are dispatched except for a few cases. We conclude that by eliminating the intra-node communication, the processor sets are able to dispatch more job sets. We account this observation to the increased network congestion caused at the ELAN interface by communication requests produced by processors of the same node.

5.2.7 Conclusion

We implemented a distributed dispatcher using pipelined approach. Our results show the superiority of distributed dispatching over the uniprocessor dispatching. We showed that for every schedule, there would be a processor set which dispatches the schedule successfully.

Our tests show that choosing a processor per node is better than multiple processors per node. This increases the length of the connection paths between processors through switches but the load on the ELAN switches decreases. The ELAN switch that uses a shared memory to communicate data between processors in a node becomes a bottleneck when all the processors send data to each other. Thus, it is important to know the characteristics of the underlying hardware before attempting to write a parallel program.

Chapter 6

HNOW MODELLING

In order to design load-balancing algorithms for a heterogeneous network of workstations (HNOW), a number of parameters need to be well defined. These parameters should cover the heterogeneity of the network of workstations, the applications considered and the characteristics of the required load-balancing algorithm. In this chapter, we will discuss these different parameters, and then we will discuss the different approaches to model the HNOW specifically theoretical and simulation modeling. Finally, we will present our HNOW model.

6.1 HNOW Measurable Parameters

HNOW measurable parameters are divided into cluster heterogeneity parameters and HNOW performance parameters. Cluster heterogeneity parameters are used to describe the structure of the HNOW while the performance parameters are used to measure the performance of the HNOW for a certain application.

6.1.1 Cluster Heterogeneity Parameters

The sources of heterogeneity in a network of workstations can be divided into three main categories: processor, memory, and network parameters.

6.1.1.1 Processor Parameters

In a fully detailed processor model, we would need to consider the speed of a processor in terms of the number of floating-point operations per second, and the number of integer operations per second. Multiple instructions and instruction pipelining would further complicate the model. Thus, a number of more simplified parameters that have shown their effectiveness may be considered as follows:

- ***Processor speed:*** some researches [LL96] just considered the processor speed only and used benchmarks to obtain the speed index.

- ***Relative processor speed:*** most researches use the "relative processor speed" defined as the ratio of the time taken to execute a sample of the application on the processor in consideration, with respect to the time taken on a base processor. This measurement incorporates most of the workstation elements from processor speed to cache and memory. It should be noted that the relative processor speed is application dependent.
- ***Processor load:*** some other researches use the workload observed at time of load balancing on the processors as an indication of the workstation computational power [DCG93]. There are two main alternatives for estimating this value: by means of external functions *(active methods)* or by using the application itself *(passive methods).* This latter is an ideal solution that aims at avoiding extra overheads caused by the active methods. Although active methods are more time wasteful, they guarantee transparency to the programmer and a more accurate estimate to the actual workload.

6.1.1.2 Memory Parameters

The required memory required by the application and the actual available memory should be considered in scheduling the computations and data. Usually the total amount of memory in the cluster limits the data size considered by numerical scientific applications like weather modeling and computational dynamics. In HNOW, the amount of physical memory for each machine may be different which requires special consideration when scheduling workloads.

A closer inspection to the memory, we find that the total memory is divided into physical "RAM" memory and "swap" memory. RAM is high-speed random access memory, while the swap memory uses the slower hard disk drive as an extension to the RAM. Most of the researches use the term total memory and free memory without specifying which type is included in their experiments. Obviously, the performance degrades as more swap memory is used.

It should be noted that the access time of the RAM ranges from (5–50 ns) from the SRAM to (60–100 ns) of the DRAM while the swap memory ranges from (10-20 ms) for seek time besides data transfer time (5-40 MB/sec).

Many researches as in [OP97], [CCN97], [LL96], [B99], and [HLA95] assume infinite memory in their models, which is an invalid assumption for our HNOW model.

6.1.1.3 Network Parameters

For a heterogeneous network of workstations, we have to consider the cost of communication between linked (logically linked by the application) machines. We must consider both the network latency and bandwidth.

- ***Network latency:*** This is one of the primary concerns for heterogeneous systems. High latency can make communication extremely expensive, and restrict the scalability of the system.
- ***Network bandwidth:*** With different interconnection networks, the network heterogeneity can become a significant factor in the parallel performance of applications. Bandwidth can even be a bottleneck, especially for Ethernet LAN.

Accordingly, the network parameters may be summarized into two parameters: the startup time (independent of message size) and the actual time spent for sending the message (proportional to the size of the message).

6.1.2 HNOW Performance Parameters

Performance parameters are important in order to evaluate parallel programs. The most common performance parameters are the speedup and efficiency. We define the speedup, efficiency, and related variables as follows.

- **Number of processors** P

P is the number of processors involved in the HNOW. The performance measured needs to be referred to the number of processors used. Good algorithms should be able to scale with the number of processors.

- **Sequential execution time** T_s

T_s is the time taken by a by a single workstation to execute the application. In a heterogeneous network, we will consider the minimum execution time (T_{sp}) achieved by a single workstation P of the HNOW for the calculation of the speedup.

i.e. $T_s = Min\left(T_{sp}\right)$

- **Parallel execution time** T_p

T_p is the time taken by the HNOW to execute the parallel version of the application.

- **Speedup** $S(p)$

The speedup is the ratio between the serial execution time and the parallel execution time. It indicates the degree of speed gain in a parallel computation. Most researches use this parameter only as an acceptable measure for performance.

i.e., $S(p) = \frac{T_s}{T_p}$

- **Efficiency** $E(p)$

The efficiency is the ratio between the speedup and the number of workstations. It measures the useful portion of the total work performed by the processors. Efficiency drops as overhead of parallel processing grows. The lowest efficiency corresponds to the entire program being executed sequentially on a single processor.

$$E(p) = \frac{S(p)}{p}$$

6.2 Theoretical Model of Parallel Computers

Several theoretical models were introduced to facilitate the study of the behavior of algorithms applicable to parallel computers. We will summarize these different models in the following and discuss if its suitability in modeling a heterogeneous network of workstations.

6.2.1 PRAM Model

Figure 6.1 illustrates the parallel random access machine (PRAM) model, which is a multiprocessor system with shared memory and zero synchronization and no memory access overhead.

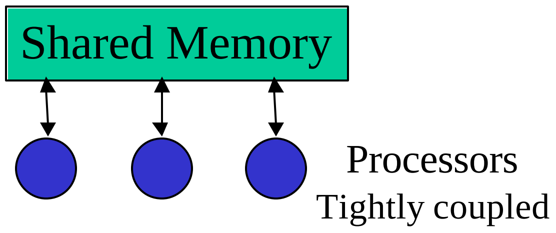


Figure 6.1: PRAM model

There are four variants of the PRAM depending on how the memory reads/writes are handled.

- **EREW-PRAM** (Exclusive Read Exclusive Write): This model forbids more than one processor to read or write the same memory cell simultaneously.
- **CREW-PRAM** (Concurrent Read Exclusive Write): This model forbids more than one processor to write the same memory cell simultaneously, but concurrent reads to the same memory cell is allowed.
- **ERCW-PRAM** (Exclusive Read Concurrent Write): This allows exclusive read or concurrent writes to the same memory cell.
- **CRCW-PRAM** (Concurrent Read Concurrent Write): Concurrent read or write to the same memory cell.

The conflicting writes are resolved by one of the following:

- **Arbitrary:** Any one of the values written may remain and all the other are ignored
- **Minimum:** The value written by the processor with the minimum index will remain
- **Priority:** The values written are combined using some function as summation or maximum.

Certainly, this model is suitable for shared memory parallel computers only.

6.2.2 BSP Model

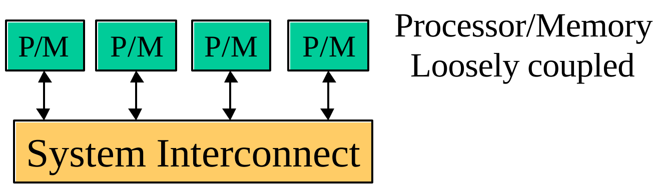


Figure 6.2: BSP model

The bulk synchronous parallel (BSP) [LV90] was introduced to overcome the shortcomings of the PRAM model, while keeping its simplicity. It consists of a set of n processor/memory pairs that are interconnected by a communication network as shown in Figure 6.2. Also, it has a synchronizer, which performs barrier synchronization.

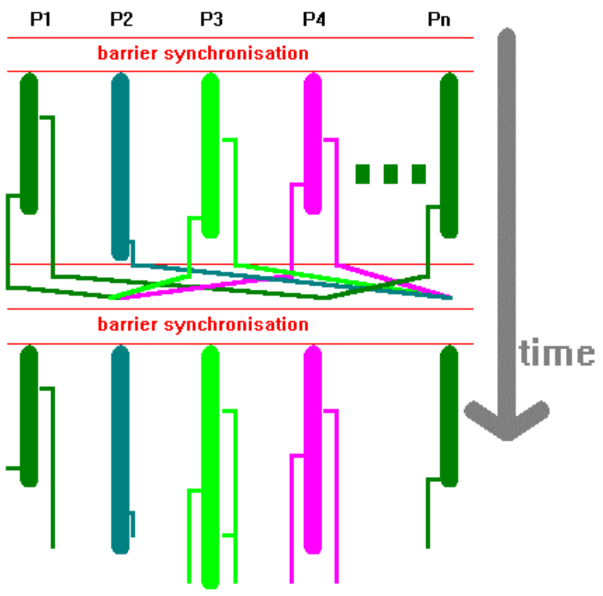


Figure 6.3: BSP execution

The essence of the BSP approach to parallel programming is the notion of the *superstep*, in which communication and synchronization are completely decoupled. A BSP program is simply one, which proceeds in phases, with the necessary global communications taking place between the phases as shown in Figure 6.3.

A BSP computation consists of a sequence of parallel supersteps, where each superstep is a sequence of steps carried out on local data, followed by a barrier synchronization at which point any non-local data accesses take effect. Requests for non-local data, or to update non-local data locations, can be made during a superstep but are not guaranteed to have completed until the synchronization at superstep end. Such requests are non-blocking; they do not hold up computation.

The BSP model is more realistic than the PRAM model as it accounts for different overheads listed as follows:

- Load imbalance, with w as the maximum computational time taken by a processor.
- Synchronization overhead, which has a lower bound of the communication network latency l.
- Communication overhead gh, where h represents the maximum number of messages that can be sent and received by each processor in each superstep, and g is a constant decided by the machine platform.
- The time for the superstep is estimated by the sum $w + gh + l$.

The BSP model is a realistic model that facilitates the time-complexity analysis of parallel algorithms. However, it's mainly used for homogeneous cluster calculations. Also, the BSP assumes a special hardware support to synchronize all processors at the end of the superstep. The synchronization hardware may not be available on many parallel machines. Most existing parallel machines use messages for synchronization, which has a different model than the one used by BSP. Furthermore, BSP model does not support overlapping computations with communications.

6.2.3 LogP Model

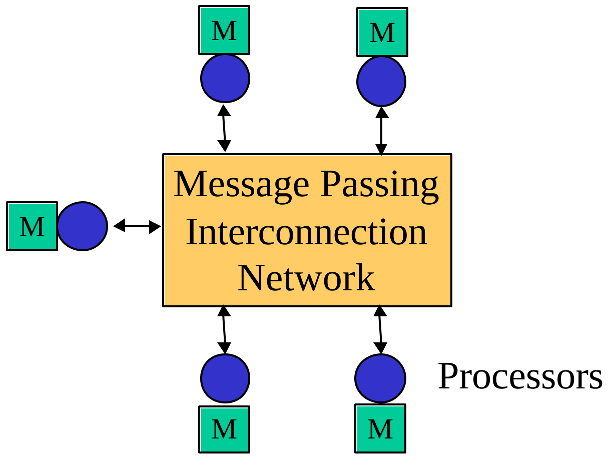


Figure 6.4:LogP model

The LogP model [CKS93] reflects the convergence of parallel machines towards systems formed by a collection of complete computers, each consisting of a microprocessor, cache, and large DRAM memory, connected by a communication network as shown in Figure 6.4.

The LogP model for parallel computation models communication performance through the use of four parameters: the communication latency L, overhead o, bandwidth g and the number of processors P. Communication is modeled by point-to-point messages of some fixed short size.

LogP model represents a more realistic model for network of workstations than the other models. However, it uses only fixed message size and a large memory. Also, it does not incorporate most of the parameters needed for the heterogeneous workstations.

A number of other models like {[GMR94], [ACS89], [MMT95], [MNV94]} derived from the discussed models were suggested, but none of them captures the heterogeneity parameters.

It should be noted that theoretical models and the choice of their parameters is a compromise between faithfully capturing the execution characteristics of real machines and providing a reasonable framework for algorithm analysis and design. No small set of parameters can describe all machines completely. On the other hand, analysis of interesting algorithms is difficult with a large set of parameters. Accordingly, most researches (refer to related work) use simulation or experimental studies.

It should be noted that in our study we are interested in dynamic behavior as well as the static performance of load balancing algorithms. These theoretical models are used only to provide static performance of an algorithm on a certain parallel computing system. Consequently, we choose to simulate the HNOW environment with simulation models as presented in the next section.

6.3 HNOW Simulation Model

We built a discrete event simulation model of HNOW. We used OMNeT++ [OMNT] that provides the simulation environment and programmed the model using visual C++. In the next section, we will list the different features of OMNeT++ and discuss how we used it in our simulations.

6.3.1 What is OMNeT++?

OMNeT++ is an object-oriented modular discrete event simulator. The name itself stands for Objective Modular Network Testbed in C++. The simulator can be used for modeling:

- communication protocols,
- computer networks and traffic modeling,
- multi-processor and distributed systems,
- any other system where the discrete event approach is suitable.

6.3.2 Modeling Concepts

An OMNeT++ model consists of hierarchically nested modules, which communicate with messages. Modules that contain submodules are termed compound modules, as opposed simple modules, which are at the lowest level of the module hierarchy. Simple modules contain the algorithms in the model. The user implements the simple modules in C++, using the OMNeT++ simulation class library.

Modules communicate by exchanging messages. In an actual simulation, messages can represent frames or packets in a computer network, jobs, or customers in a queuing network or other types of mobile entities. Messages can contain arbitrarily complex data structures. Simple modules can send messages either directly to their destination or along a predefined path, through gates and connections. Due to the hierarchical structure of the model, messages typically travel through a series of connections.

OMNeT++ supports a process-style description method for describing activities. During simulation execution, simple module functions appear to run in parallel, because they are implemented as co-routines. Co-routines were chosen because they allow an intuitive description of the algorithm and they can serve as a good basis for implementing other description methods like state-transition diagrams or Petri nets.

6.3.3 Building and Running Simulations

An OMNeT++ model consists of the following parts:

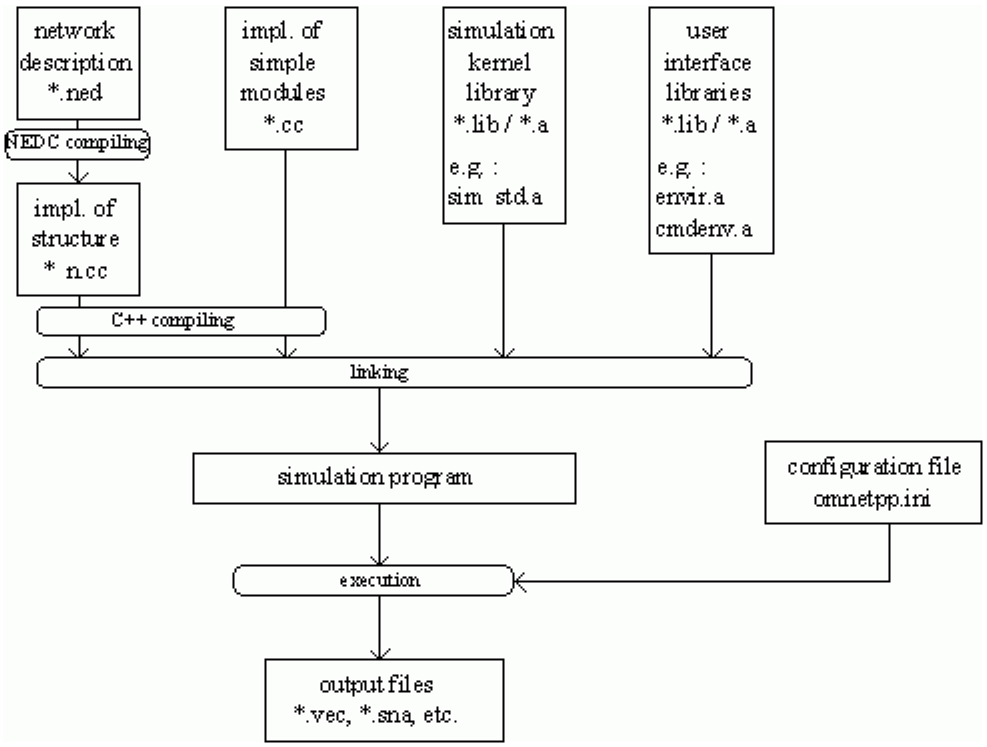
- NED language topology description(s), which describe the module structure with parameters, gates etc.
- Simple modules sources: These are C++ files.

The simulation system provides the following components:

- Simulation kernel: This contains the code that manages the simulation and the simulation class library.
- User interfaces: OMNeT++ user interfaces are used with simulation execution, to facilitate debugging, demonstration, or batch execution of simulations.

The simulation executable is a standalone program; thus, it can be run on other machines without OMNeT++ or the model files being present. When the program is started, it reads in a configuration file (usually called omnetpp.ini); it contains settings that control how the simulation is run, values for model parameters, etc. The configuration file can also prescribe several simulation runs; in the simplest case, they will be executed by the simulation program one after another.

The output of the simulation is written into data files: output vector files, output scalar files, and possibly the user's own output files. OMNeT++ provides a GUI tool named Plove to view and plot the contents of output vector files. This process is summarized in figure 6.5


network description *.ned
NEDC compiling
impl. of structure *_n.cc
impl. of simple modules *.cc
simulation kernel library *.lib / *.a
e.g. : sim_std.a
user interface libraries *.lib / *.a
e.g. : envir.a cmdenv.a
C++ compiling
linking
simulation program
configuration file omnetpp.ini
execution
output files *.vec, *.sna, etc.
Figure 6.5: Building and running a simulation

6.3.4 Simulation Model

In order to construct a simulation model for the HNOW, we need to identify the following:

- Input variables that define the system,
- Output variables that define the performance measures,
- Mathematical/logical relationship between the inputs and outputs.

In the following sections we will discuss each of these in details.

6.3.4.1 Input variables that define the system

To simplify our model, we define “datapoint” as our main measuring unit. A data point is the smallest calculation unit of the SPMD. It represents the grid point or an array element, according to the application. Each data point is characterized by requiring the same number of operations and the same storage space.

Without loss of generality, we may relate the processor speed and memory to the datapoints. For example, the processor can execute 2000 datapoint operations/sec (DPOPS) or its memory can hold up to 5000 datapoints (DP). Also, reverting back to the standard units is easy, just by measuring the data point size and the number of Mflops needed. Therefore, in our simulations and calculations we will be using this application unit "DP". We summarize the notation of all the parameters in the following table:

Terminology	**Notation**	**Unit**	**Application Unit**
Processor speed	F	MHz	Datapoint operations /s
Workload	W	MB	Datapoints (DP)
Free memory	M_f	MB	Datapoints (DP)
Swap memory	M_s	MB	Datapoints (DP)
Memory Access time (swap)	M_a	Sec	Sec
Latency	L	Sec	Sec
Bandwidth	BW	MB/sec	Datapoints /s

Table 6.1: HNOW metrics expressed in datapoints

6.3.4.2 Output variables that define the performance measures

There are two different sets of performance parameters: parameters that measure the HNOW performance and parameters that measure the performance of the load-balancing algorithms.

- HNOW performance parameters

The most common is the scalability measured by the speedup of an application with the number of workstations as discussed before.

- Load-balancing algorithms performance parameters

The most common performance parameters are:

Convergence: the number of steps required to reach steady state from an imbalanced state.

Load Exchange: total number of extra data points exchanged to reach steady state.

Balancing Overheads: this includes both extra time required for the computation for the load-balancing algorithm and the extra messages exchanged for the balancing algorithm

6.3.4.3 Relationship between the inputs and outputs

The nature of applications considered in this research assumes the following:

- Computational intensive applications, which involves a large number of iterations (much larger than the number of processing units),
- Large data domains (more than that one processing unit can handle),
- The main data domain's shape is regular,
- The main data domain is uniform, which means that each datapoint requires the same amount of computations.

The parallel programming paradigm implemented is a pipelined SPMD in which:

- Each processing unit has the same executable program,
- Each processing unit is assigned a continuous part of the main data domain,
- Neighboring processing units communicate regularly to exchange boundary variables,
- Pipelined in the sense that each processing unit can't begin its execution before it receives the boundary variables from the neighbors,
- There is no synchronization between the different processing units.

The network of workstation used is a non-dedicated heterogeneous cluster of workstations.

Chapter 7

DYNAMIC LOAD BALANCING ALGORITHM OUTLINE

7.1 Dynamic Load Balancing Algorithm (DLAH) for HNOW Overview

The DLAH algorithm is based on the diffusion technique in which neighboring (logically connected by the application) workstations communicate with each other and exchange workloads to eliminate any load imbalance. Diffusion technique has a number of properties listed as follows:

- Load balancing algorithm is distributed as there is no central scheduler,
- No synchronization is required as each workstation may be triggered independently depending on its current state,
- Decision making is mainly based on local information exchange which yields less communication cost,
- Uniform communication between the neighboring workstations, which supports the pipelined SPMD applications.

It seems that the diffusion technique may not converge towards a globally balanced system, but it has been proven mathematically [CLZ99] that the execution of a diffusive load-balancing policy nullifies any load imbalance in a system. The results were derived on massively parallel architectures and implemented algorithms demonstrated their scalability and robustness. We extend the diffusive policy to incorporate the HNOW parameters.

The main features that we consider essential for the DLAH algorithm are summarized in the following points:

- Dynamic to accommodate for the non-dedicated cluster nature.
- Scalable with the number of workstations in the cluster.
- Preserve the relationship adjacency by shifting workloads between adjacent workstations only.

We use the taxonomy defined in section 2.5 to classify the DLAH algorithm:

- *Initiation:* Sender initiated by the overloaded workstation
- *Load balancer Location:* Distributed, Asynchronous
- Decision making: Local
- *Communication:* Uniform, Local
- *Processor/Load matching:* The overloaded processor sends load-packets to its neighbors until its own load drops to a specific threshold or the average load.

The main performance parameters we will focus on are the convergence rate and the extra load exchanged.

A detailed description and analysis of the algorithm is presented in the following sections.

7.2 Convergence of the Diffusive Policy

The diffusion policy relies on neighboring workstations that communicate with each other and exchange workloads to eliminate any load imbalance between them. It has been proved [CLZ99] that this policy drives the whole system to a global balanced state. We will discuss the proof in details in this section.

For this analysis, the workstation memory is considered infinity and the total workload assigned for the network of workstations remains constant throughout the whole execution, i.e. workloads are neither created nor destroyed but rather moved around the system. Also, the execution time is directly proportional to the workload assigned, thus it is equivalent to analyze the system with the workload or the execution time.

Let i denote the *i-th* workstation in the cluster. Let its loop execution time at time t be $L_i(t)$

The average system execution time at time t is:

$$m(t) = \frac{1}{P}\sum_{i=1}^{P} L_i(t) \tag{7.1}$$

One simple measure of the system imbalance is the variance of the loop execution time of the workstations given by.

$$s^2(t) = \frac{\sum_{i=1}^{P} \left(L_i(t) - m(t) \right)^2}{P} \tag{7.2}$$

We will state the main theorem and discuss the proof in details.

Theorem 7-1

The execution of a diffusive load balancing policy nullifies any load imbalance in the system, i.e.,

$$\forall e > 0, \exists T > 0 \text{ such that } s^2(T) = \frac{\sum_{i=1}^{p} \left(L_i(T) - m(T) \right)^2}{P} < e \tag{7.3}$$

To prove theorem 7-1, we need to use theorem 7-2.

Theorem 7-2

Reducing the local variance of loop executions in a domain D of workstations, by exchanging workload among themselves, reduces the global variance of the system.

Proof:

Using equations 6.3 and 6.4, we can express the variance of the loop time execution as follows:

$$\begin{aligned} s^2(t) &= \frac{\sum_{i=1}^{P}\left(L_i(t)-m(t)\right)^2}{P} \\ &= \frac{\sum_{i=1}^{P} L_i^2(t) - 2m(t)\sum_{i=1}^{P} L_i(t) + Pm^2(t)}{P} \\ &= \frac{\sum_{i=1}^{P} L_i^2(t) - 2Pm^2(t) + Pm^2(t)}{P} \\ &= \frac{1}{P}\sum_{i=1}^{P} L_i^2(t) - m^2(t) \end{aligned} \quad (7.4)$$

Assume after time Δt, the loop execution time variance changes due to the act of the load-balancing algorithm in domain D only, as shown in the following equation:

$$\begin{aligned} s^2(t+\Delta t) &= \frac{\sum_{i=1}^{P}\left(L_i(t+\Delta t)-m(t+\Delta t)\right)^2}{P} \\ &= \frac{1}{P}\sum_{i=1}^{P} L_i^2(t+\Delta t) - m^2(t+\Delta t) \end{aligned} \quad (7.5)$$

Therefore, the change in variance can be expressed as:

$$\begin{aligned} s^2(t+\Delta t, t) &= s^2(t+\Delta t) - s^2(t) \\ &= \frac{1}{P}\sum_{i=1}^{P} L_i^2(t+\Delta t) - \frac{1}{P}\sum_{i=1}^{P} L_i^2(t) - \left(m^2(t+\Delta t) - m^2(t)\right) \end{aligned} \quad (7.6)$$

As mentioned before, the workload is not created nor destroyed but moved from one workstation to another, thus the average remains constant, i.e., $m(t+\Delta t) = m(t)$.

Since the workloads of the workstations outside of the domain did not change, i.e., their loop execution time did not change. Then we may rewrite the change in the variance as follows:

$$
\begin{aligned}
\Delta s^{2}(t+\Delta t,t) &= \frac{\sum_{i\in D} L_i^2(t+\Delta t)+\sum_{i\notin D} L_i^2(t+\Delta t)-\left(\sum_{i\in D} L_i^2(t)+\sum_{i\notin D} L_i^2(t)\right)}{P}\\
&= \frac{\sum_{i\in D} L_i^2(t+\Delta t)-\sum_{i\in D} L_i^2(t)}{P}\\
&= \frac{N_D s_D^2(t+\Delta t)+N_D m_D^2(t+\Delta t)-N_D s_D^2(t)-N_D m_D^2(t)}{P}\\
&= \frac{N_D \Delta s_D^2(t+\Delta t,t)+N_D\left(m_D^2(t+\Delta t)-m_D^2(t)\right)}{P}\\
&= \frac{N_D}{P}\Delta s_D^2(t+\Delta t,t)+\frac{N_D}{P}\left(m_D^2(t+\Delta t)-m_D^2(t)\right)
\end{aligned}
\tag{7.7}
$$

where N_D is the number of workstations in the domain of interest.

Since the workloads are exchanged in domain D only, then the average has not been changed. Therefore, the above equation is reduced to:

$$\Delta s^{2}(t+\Delta t,t)=\frac{N_D}{P}\Delta s_D^2(t+\Delta t,t) \tag{7.8}$$

From the above equation, we can see that the global variance depends directly on the change of the variance of the domain only. That means if the load-balancing algorithm is able to decrease the variance of the local domain then that will directly decrease the global variance and accordingly contribute to the global balance.

The result of this lemma assumes that the load-balancing algorithm acts only in one domain at a time. This result can be generalized to show that the load-balancing algorithm can act on several disconnected domains in parallel to achieve a global balance.

In the next section, we will prove theorem 7-1.

Proof of theorem 7-1:

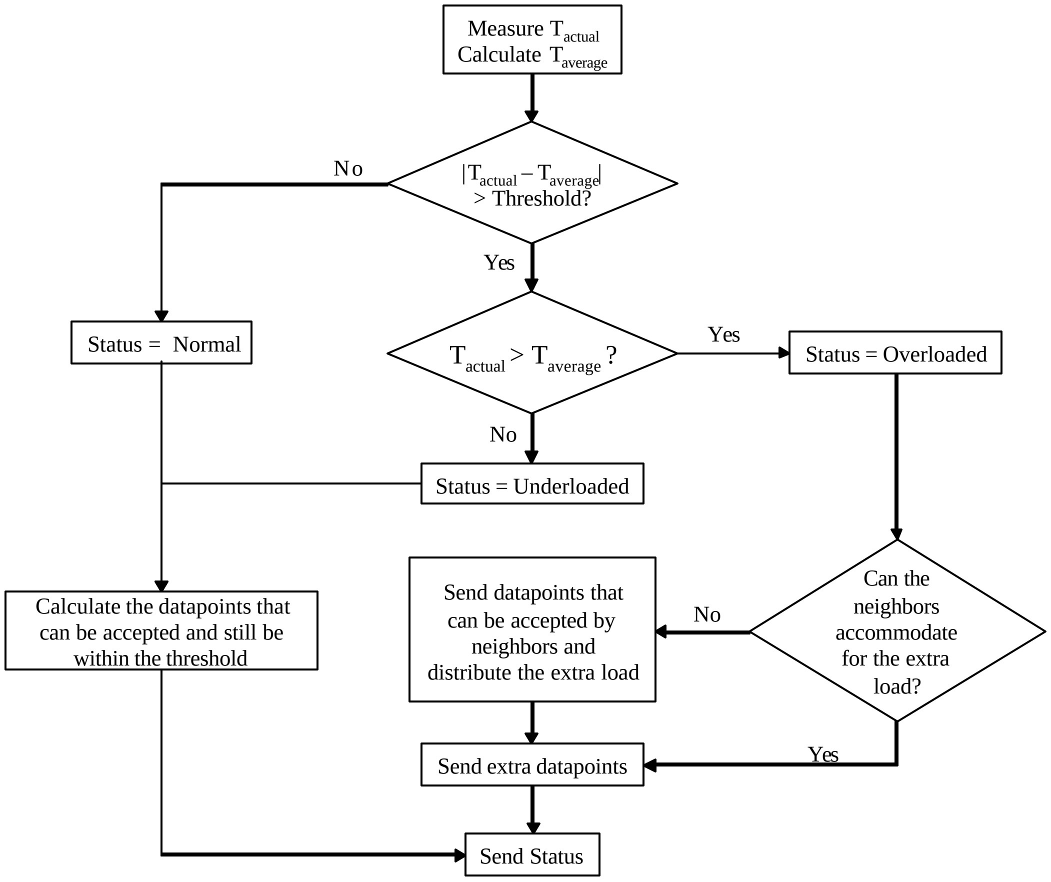
Let us assume that the cluster of workstations is divided into domains, each domain overlaps with the neighboring domains. The load-balancing algorithm can operate on any number of domains at the same time. The load-balancing algorithm realizes any imbalance and works to diminish the local variance.

The load-balancing algorithm will only stop at a time T when each domain is balanced. At time $T - \Delta t$, the system was imbalanced with a variance of e. The above setup guarantees that at least one of the domains is imbalanced. Using the result of theorem 7-2, the diffusive policy reduces the local variance of each domain beyond e. Thus, when the load-balancing algorithm stops, the global variance will be reduced to a value beyond e.

7.3 DLAH Algorithm

Figure 7.1 presents a flowchart diagram of the DLAH algorithm. Before beginning the procedure, the execution time of a predefined number of iterations (loops) is measured. This execution time measured includes all the HNOW factors as it includes the calculation time, communication time and various overheads such as paging, caching, and operating system overheads. The load balancer then classifies the workstations into three categories: *Overloaded*, *Normal* and *Underloaded* workstations.

The load balancer algorithm will only be invoked if the workstation status is overloaded. Then, it checks its underloaded neighbors to see how much extra data points they can accept before there status changes to normal. Accordingly, if they can accommodate the extra points, the load balancer algorithm sends the datapoints that just make it changes its status from overloaded to normal. Otherwise, the overloaded workstation sends the datapoints, which can be accepted by the neighbors, and averages its extra load among the underloaded and normal workstations.



Measure T_{actual} Calculate $T_{average}$
$|T_{actual} – T_{average}|$ > Threshold?
No
Yes
Status = Normal
$T_{actual} > T_{average}$?
Yes
Status = Overloaded
No
Status = Underloaded
Calculate the datapoints that can be accepted and still be within the threshold
Send datapoints that can be accepted by neighbors and distribute the extra load
No
Can the neighbors accommodate for the extra load?
Send extra datapoints
Yes
Send Status

Figure 7.1: DLAH algorithm

It is worth noting that the most critical part of the algorithm is the calculation of the load to be exchanged. If the load calculation does not consider HNOW parameters, a *bouncing* effect may result. The bouncing effect is characterized by sending data points to a neighboring workstation. This neighboring workstation would take more execution time than expected (due to neglecting the HNOW parameters) and in turn will return to the sender some or all the workload received and this cycle repeats as seen in figure 7.2. Actually, in some cases this bouncing effect can prevent any convergence to the global balance.

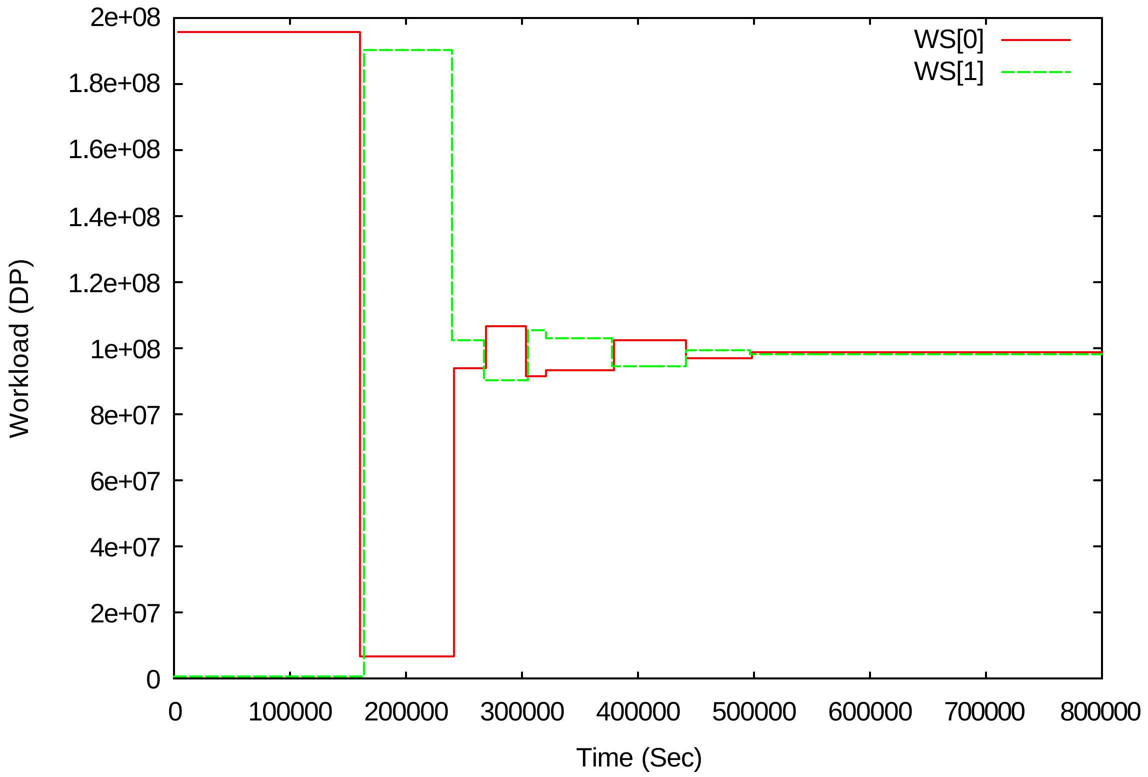


Figure 7.2: Workload exchanged between two workstations with a load-balancing algorithm designed for a homogenous network

The load-balancing algorithm is governed by the following equations:

- **Deciding the status:**

Overloaded: $T_{actual} - T_{average} > T_{threshold}$

Normal: $|T_{actual} - T_{average}| < T_{threshold}$

Underloaded: $T_{average} - T_{actual} > T_{threshold}$

Where:

T_{actual} : is the average time needed to complete an iteration.

$T_{average}$: is the local average execution time for the workstation and its neighbors.

$T_{theshold}$: is user-defined according to the application and execution environment. It determines the accepted toleration between the execution times of the workstations. This may be an explicit value or just a percentage from the average time.

- **Calculating the extra data points that can be accepted by the underloaded and normal workstations**

Extra execution time = extra execution time for added data points + extra time for points added in swap memory + extra boundary points added.

$$T_{new} - T_{actual} = \frac{DP}{F} + (DP - M_f) \times M_a + \frac{K \times DP}{BW} \qquad (7.9)$$

Where:

DP: is the extra workload added to the workstation,

T_{new}-T_{actual}: is the extra execution time added because of the extra workload,

F: is the workstation processing power,

DP-M_f: is the extra data points added to the swap memory,

M_a : is the memory access time for the swap memory,

BW : the network bandwidth,

K : is a constant, which determines the extra boundary data points, resulting from the extra workload exchange.

Conversely, we can calculate the number of extra data points needed for a workstation to change its status from underloaded to normal as shown below.

$$DP = \frac{(T_{threshold} - T_{actual}) + M_f \times M_a}{1/F + M_a + K/BW} \qquad (7.10)$$

The DLAH algorithm calculates the reduction of the execution time gained by each neighbor. Once it finds that the neighbors can change the workstation status from overloaded to normal, it issues the corresponding send requests. Otherwise, it will proceed to the averaging phase as detailed in the next section.

- **Averaging the load among underloaded and normal neighbor workstations**

In this phase, when the neighbors can't change the status of the workload from overloaded to normal, the overloaded workstation will first send the datapoints that the neighbors can accommodate then it distributes the remaining load among themselves.

It should be noted that the algorithm calculation does not depend on the total number of the workstations in the cluster, but on the number of neighbors. The number of neighbors is usually a constant for each application depending on the data decomposition; it usually ranges from two in one dimensional data decomposition to six for three-dimensional data decomposition. Accordingly, the calculation time for the algorithm is constant with respect to the total number of workstations, thus it is scalable.

7.4 DLAH Analysis and Bounds

In this section, we will present an analytical bound that will provide an approximation to the number of steps required to reach the balanced state by using the DLAH algorithm. The analysis is performed on a homogeneous network of workstations.

To derive an upper bound for the number of steps required to reach balanced state we will use theorem 14 from [San96].

Theorem 7-3

The worst case balancing time for diffusion for on the linear array with diffusion parameter $a = 1/2$ is in $O\left(P^2\right)$, where P is the number of workstations.

Where a determines the ratio of the extra workload exchanged.

From this theorem, we state our theorem:

Theorem 7-4

The worst case balancing time for diffusion for a one-dimensional pipelined application that averages its workload with its neighbor is in $O\left(Steps^2\right)$.

Where $Steps \geq \log_2 \dfrac{(m-1)(1-c)}{2c}$, if $Steps < P$

and $Steps = P$, if $Steps \geq P$,

m is the ratio of the workload of the workstation with the maximum workload to that of the workstation of minimum workload,

$c \in [0,1]$ is the threshold ratio which determines the stable region, which is a ratio from the local average. Thus the stable region is $(1 \pm c) \times Local\ \ Average$. $c = 0$ means that there is no stable region and strict balance is required while $c = 1$ means that no balance is required.

Proof:

We are going to proof that we do not need all the workstations to participate in the load imbalance for all the cases and determine the balance time from the number of workstations required to balance the system.

Let us consider the worst case of a homogeneous cluster of P workstations with a one-dimensional pipelined application. The first processor has a load of mL, while all the other processors have a load of L as shown in figure 7.3. DLAH algorithm will operate until the workload of each workstation is within the threshold range $T_{threshold}$.

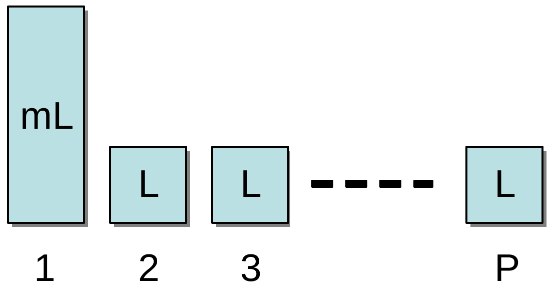


Figure 7.3: DLAH Analysis

The local average is calculated from the neighbors' workloads only, while the threshold is calculated as a percentage from the local average as shown in the following equations.

$$T_{average} = \sum_{i \in N} L_i$$
$$T_{Threshold} = (1 + c)T_{average}, \quad c \in [0,1] \tag{7.11}$$

When $c = 0$, that means that it is required to strictly balance the workstation to the local average. While $c = 1$, means that the workstation can tolerate an imbalance equal to twice the local average.

At step 1:

The first workstation is overloaded. It will compare its current workload with the local average and will invoke the DLAH algorithm if it is more than the threshold, as shown in the following equation.

$$\begin{aligned} mL &> T_{Threshold} \\ mL &> (1+c)T_{Average} \end{aligned} \qquad (7.12)$$

Accordingly, the first workstation will send to its neighbor a load to decrease its current workload below the threshold. If the load sent makes the second workstation's workload above the threshold, then the two workstations average their workloads. For simplicity, we will consider the extra load is large enough and that all the workstation will average its workload with its underloaded neighbors. Thus, both workstations will have an equal workload after the load exchange with a value, as shown in the following equation:

$$\frac{mL + L}{2} = \frac{L}{2}(m+1) \qquad (7.13)$$

At step 2:

The first workstation is normal; however, the second workstation is now overloaded. The second workstation will compare its current workload with the local average and invoke the DLAH algorithm if it is more than the threshold, as shown in the following equation.

$$\begin{aligned} \frac{L}{2}(m+1) &> T_{threshold} \\ \frac{L}{2}(m+1) &> (1+c)\frac{\frac{L}{2}(m+1)+L}{2} \end{aligned} \qquad (7.14)$$

Accordingly, the second workstation will average its load with the third workstation, so each workstation will now have a load of:

$$\frac{\frac{L}{2}(m+1)+L}{2}=\frac{L}{4}(m+3) \qquad (7.15)$$

At step 3:

The third workstation is now overloaded. The procedure will be repeated as above. Accordingly, the third workstation will average its workload with the forth workstation, as shown in the following equation.

$$\frac{\frac{L}{4}(m+3)+L}{2}=\frac{L}{8}(m+7) \qquad (7.16)$$

At step i :

The i^{th} will have a workload of:

$$\frac{L}{2^i}\left(m-1+2^i\right) \qquad (7.17)$$

The workload propagation will stop when the value of the workload decreases below the threshold, as shown in the following equations:

$$\frac{L}{2^i}\left(m-1+2^i\right) \le T_{Threshold}$$
$$\frac{L}{2^i}\left(m-1+2^i\right) \le (1+c)T_{Average}$$
$$\frac{L}{2^i}\left(m-1+2^i\right) \le (1+c)\left(\frac{\frac{L}{2^i}\left(m-1+2^i\right)+L}{2}\right)$$
$$\left(\frac{m-1}{2^i}+1\right) \le (1+c)\left(\frac{m-1}{2^{i+1}}+1\right) \qquad (7.18)$$
$$\frac{m-1}{2^i}-\frac{(1+c)(m-1)}{2.2^i} \le c$$
$$2^i \ge \frac{2(m-1)-(1+c)(m-1)}{2c}$$
$$2^i \ge \frac{(m-1)(1-c)}{2c}$$

Therefore, the number of steps required to propagate the extra workload from the first workstation to the other workstations is:

$$Steps_i \ge \log_2 \frac{(m-1)(1-c)}{2c} \qquad (7.19)$$

Checking the above equation with the two extremes:

When $c=0$, the threshold is equal to the local average time. This means a strict balance is required in which all the workstations have to have a workload equal to the local average. Thus, the extra workload of the first workstation has to be distributed equally among all the workstations in the cluster. In this case, the number of workstations required to propagate the extra workload of the first workstation indicated by the above equation is infinity. Of course in our case, the networks of workstations is not infinity but limited by the number of workstations P .

When$c=1$, the threshold is equal to twice the local average time. This means we have a very relaxed balancing condition. Thus for workstation i , we have the following equation:

$$\frac{L}{2^i}\left(m-1+2^i\right) \le (1+1)\left(\frac{\frac{L}{2^i}\left(m-1+2^i\right)+L}{2}\right) \qquad (7.20)$$

The equation is true for all i. Therefore, no kind of balanced action needs to be taken. Checking the number of steps given by the equation, it would give us negative infinity, which practically means no need to take any balance actions.

So far, we have just traced the number of steps required to propagate the extra workload from the first workstation to the remainder of the network. However, during this operation the second workstation will send a part of its workload to its next neighbor. This will probably cause an imbalance between the first and second workstation and another propagation sequence will start. The propagation sequences will continue until all the workstations reach a steady state. However, in the following stages, the algorithm will need much less workstations to propagate the workload as the workload difference is much less now.

Since the maximum number of workstations required to balance the system is $Steps$ with a maximum of P, then the worst case balancing time is in the order of $O\left(Steps^2\right)$.

Figure 7.4 depicts the relation between the threshold ratio and the upper bound on the number of steps required to reach balance state. As the threshold ratio increases the number of steps decreases exponentially, then it remains nearly constant for a while and finally decreases slowly at the end. It is worth noting that careful consideration should be taken when choosing the threshold ratio, as we can see that it can considerably increase the number of steps needed to reach the steady state. We recommend that pilot experiments are to be conducted in which the execution time variance is measured and thus have a better judgment in choosing the threshold value.

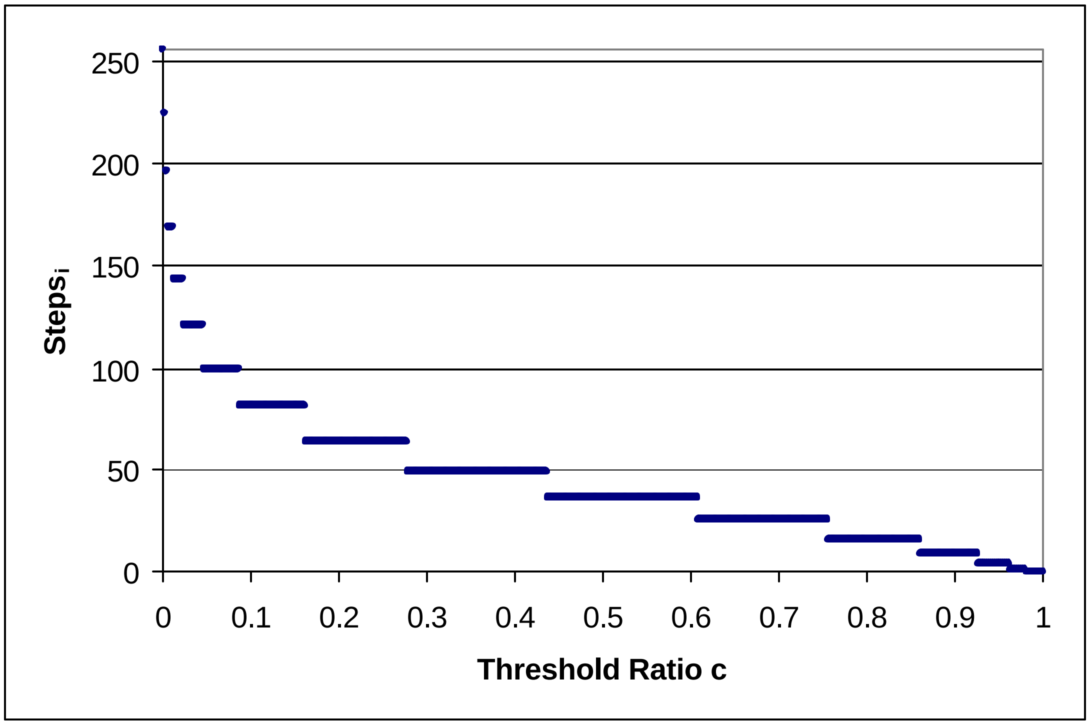


Figure 7.4: Relation between the threshold ratio and the upper bound of the number of steps required to balance

Chapter 8

SIMULATIONS

Extensive simulations were conducted to target the following objectives: verify and validate the simulation model, investigate the performance of the DLAH algorithm and compare the performance of the DLAH algorithm to other dynamic load balancing algorithms. These simulations are discussed in details in the following sections.

8.1 Simulation Model Verification and Validation

Verification is concerned with determining if the simulation computer program is working as intended while validation is concerned with determining how closely the simulation model represents the actual system; the following were some of the verification and validation procedures preformed:

- The HNOW simulation model was coded and debugged in steps; in an incremental fashion.
- An interactive debugger was used to verify that each program path is correct.
- A *trace* in which all the model parameters and state variables were printed out and compared to hand calculations to confirm that the simulation program is operating as intended.
- In several cases, the simulation model was run under simplified assumptions with deterministic inputs and the outputs were compared to the computed results. The results were computed from equation 4.1 by calculating the loop time for the slowest workstation and multiplying it in the number of iterations. Both results matched since there was not any randomness introduced to the model.
- The simulation environment provided an animation of the HNOW simulation model in which messages could be easily traced and verified.

- The input probability distributions were verified that they were correctly generated. This was achieved by plotting a histogram of the inputs and comparing it to the probability distribution.
- The simulation model results were checked for reasonableness. For example, figure 8.1 illustrates how the execution time of the workstation jumps when the RAM becomes full and becomes proportional to the datapoints in the swap memory.

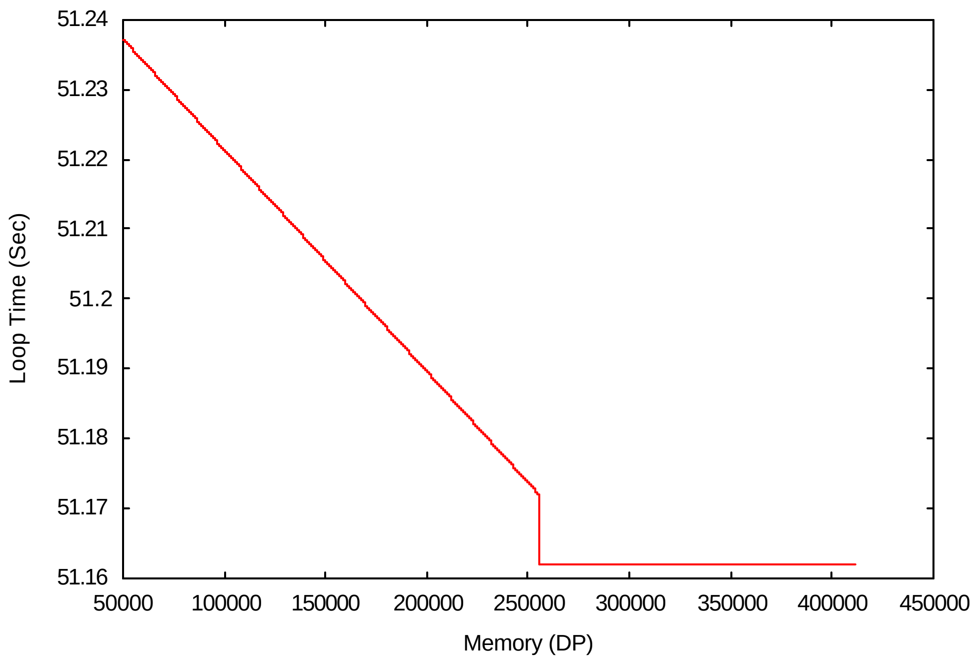


Figure 8.1: Loop time vs. memory available

It is worth noting that it is generally impossible to validate a simulation model completely, since some part of the actual system may not actually exist. Thus, a simulation model of a complex system can only be an approximation to the actual system [LK91].

8.2 DLAH Performance

In this section, we will study the performance of the DLAH algorithm. First, we will validate the DLAH algorithm by conducting simulation on homogeneous network of workstations and comparing it to the analytical bounds. Then, we will examine the sensitivity of the DLAH algorithm to each of the HNOW parameters. Finally, we will study the performance for a complete HNOW and check its scalability. The following table 8.1 describes the parameters used for the simulations:

PP	Processing power with a default of 100,000 DP/S. Processing power of each workstation has a uniform distribution around its mean with a range of $PP \pm 0.1 \times PP$.
$Memory$	Memory (RAM) size available to the application. Default is 10,000,000 DP
$Disk\_Access$	Hard disk access (virtual memory) with a default of 2,100,000 DP/S and a latency of $10ms$.
$Network$	The Network bandwidth has a default of 10,000,000 DP/S which corresponds to 10 MB/S and a latency of $1ms$.
$workload$	Number of datapoints assigned to a workstation. Default is 9,800,000 DP, which occupies the default memory size.
$Boundary$	Boundary datapoints required to communicate between calculation phases. Default is 200,000 DP from both left and right neighbors.
$Threshold$	Threshold level determines the stable region. Workstations with loop time more than the threshold level are overloaded while those below the threshold level are underloaded. Default value is 0.3 of the local average loop time.

Table 8.1 Simulation parameters

8.2.1 Homogenous Network of Workstations

The following sections are organized by stating the objective of each set of simulations, the simulation parameters, expected results and the actual results.

8.2.1.1 Validate the $DLAH$ algorithm to the analytical performance bounds

- **Simulation objective**

Validate the DLAH algorithm by comparing it to the analytical performance bounds.

- **Simulation parameters**

Two cases were conducted; the first case a homogeneous network of 100 workstations with one workstation overloaded with 10 times the workload of the other workstations, the second case a network of 10 workstations with one workstation 50 times the other workstations. Simulations were performed with different threshold levels.

- **Expected outcome**

The graph of the number of steps obtained from the simulations should resemble that of the analytical performance bounds.

The number of steps is in $O\left(Steps^2\right)$

Where $Steps \geq \log_2 \frac{(m-1)(1-c)}{2c}$, if $Steps < P$

and $Steps = P$, if $Steps \geq P$,

$m = \frac{Workload_{overloaded}}{Workload_{normal}}$ is the is the ratio of the workload of the overloaded workstation to that of the normal workstation,

$c \in [0,1]$ is the threshold ratio which determines the stable region, which is a ratio from the local average. Thus the stable region is $(1 \pm c) \times Local \;\; Average$. $c = 0$ means that there is no stable region and strict balance is required while $c = 1$ means that no balance is required.

- **Results**

Figure 8.2 and figure 8.3 illustrate the relation between decreasing the threshold ratio c and the number of steps required to reach the balanced state. As the threshold ratio c decreases, the stable region decreases, thus it will need more steps to reach the balanced state. The DLAH algorithm performance is similar to the analytical performance bounds.

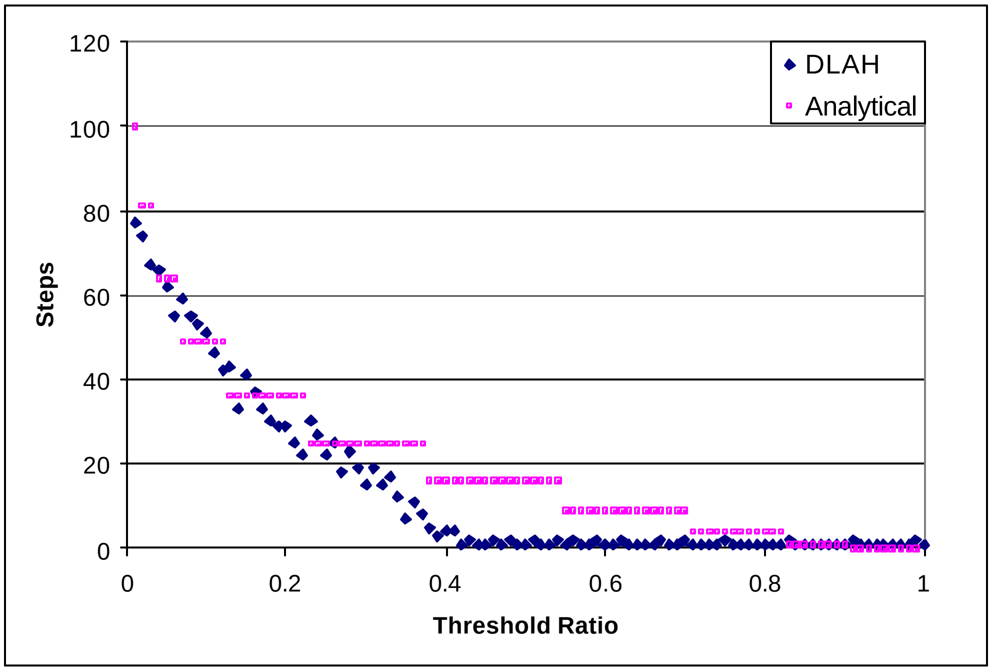


Figure 8.2:Number of steps obtained from the simulation vs. analytical performance bounds with $Steps_i < P$

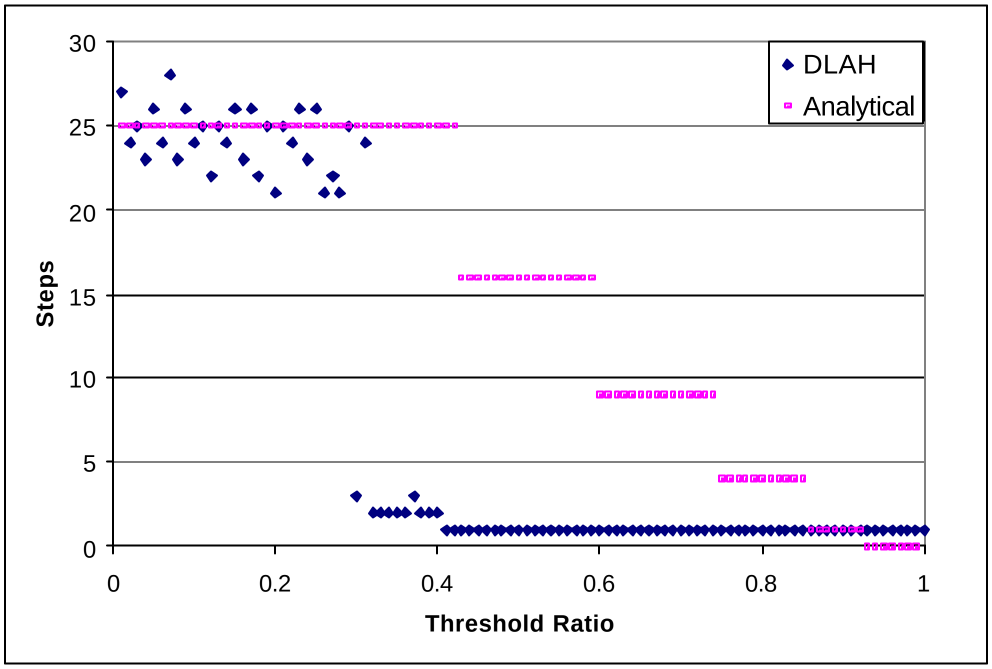


Figure 8.3: Number of steps obtained from the simulation vs. analytical performance bounds with $Steps_i > P$ for some cases

8.2.1.2 Validate the DLAH algorithm

- **Simulation objective**

Validate the DLAH algorithm, since it is easier to track the performance for a homogeneous network of workstations.

- **Simulation parameters**

Random workloads are assigned to 5 identical workstations with a uniform distribution ranging from $workload/10$ to $workload \times 10$.

Processing power of each workstation has a uniform distribution around the mean ranging from $PP \pm 0.1 \times PP$

- **Expected outcome**

All the workstations will eventually have the corresponding workload within the desired threshold.

- **Results**

Figure 8.4 illustrates how the DLAH reacts to load imbalance in a homogeneous network of workstation. The loop time, which is the time taken to execute one loop is plotted with respect to the simulation time. The algorithm was able to reduce the loop time difference between the workstations. In order to check that the DLAH algorithm reduced the difference to the desired threshold we use a smoothed version. Figure 8.5 is a smoothed version in which every 200 samples are replaced by its average, this allows us to roughly calculate the loop time difference and check if it falls below the threshold level.

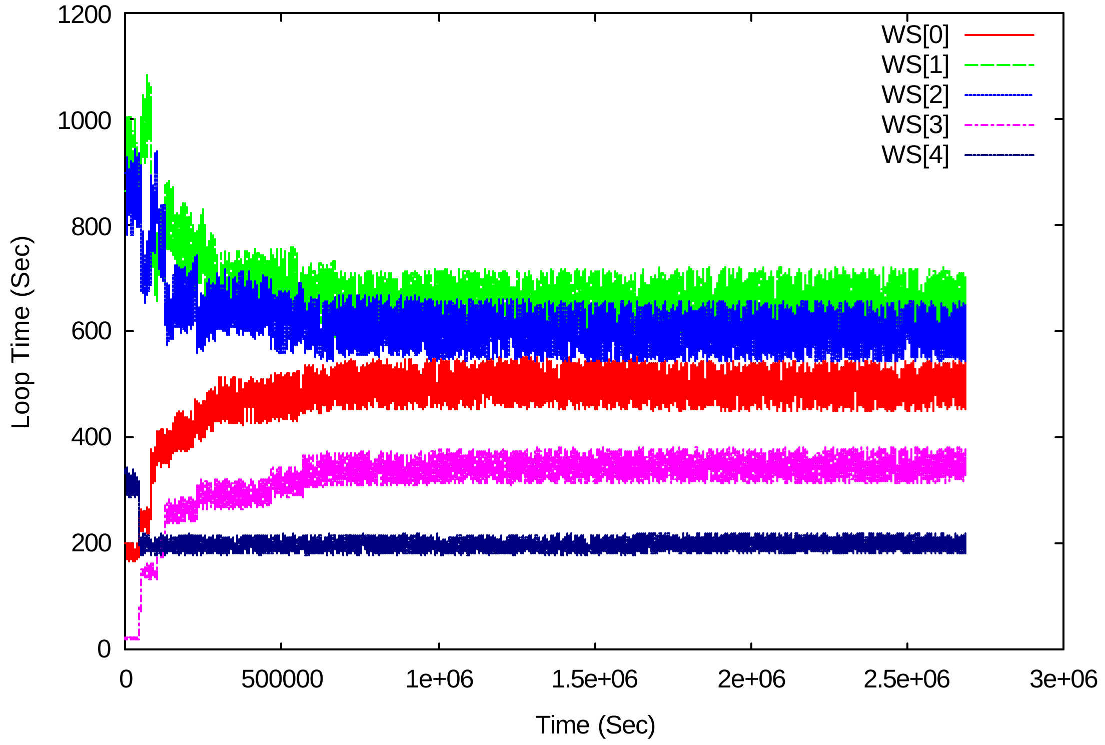


Figure 8.4: Loop time of a homogeneous network of 5 workstations

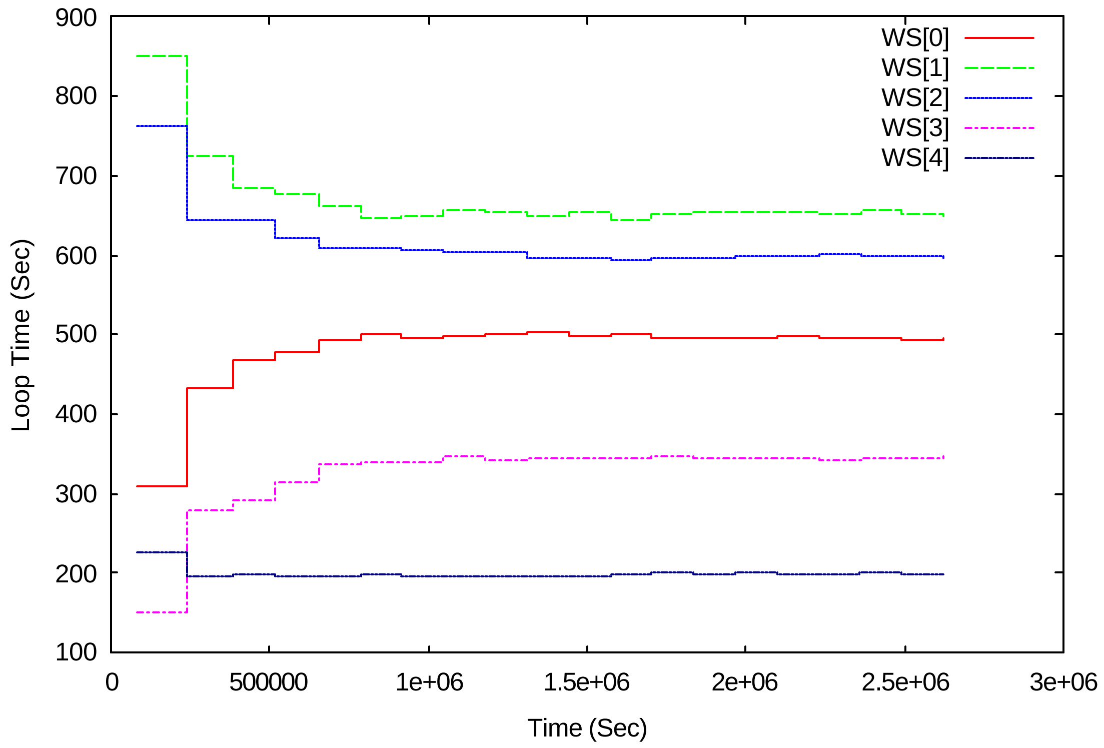


Figure 8.5: Smoothed loop time of a homogeneous network of 5 workstations

DLAH's heuristic is based on the eliminating the slower workstations. Checking the slowest workstation in this simulation WS[1], it has a loop time of 650 sec while its neighbors WS[0] and WS[2] are 500 sec and 600 sec respectively. The local average of WS[1] is 550 sec, which makes it overloaded with respect to its neighbors. The threshold level is $1.3 \times 550 = 715$ sec. Thus, WS[1] is operating within its threshold level. The same argument has been repeated to the other workstations to confirm the validity of the algorithm. In figure 8.6, we plot the workload of each workstation to the simulation time. The distribution of the workload with time resembles the loop time, as this is a homogeneous network of workstations.

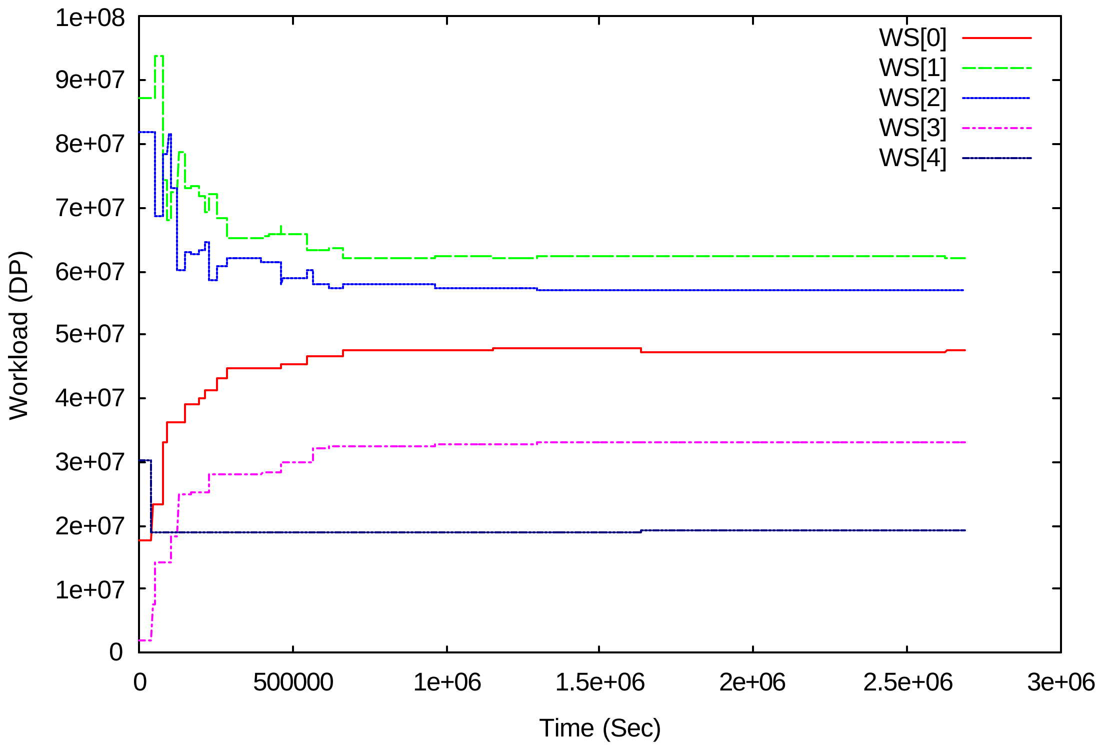


Figure 8.6: Workload assignment for a homogeneous network of 5 workstations.

8.2.1.3 DLAH performance for a homogeneous network of workstations with different workloads

- **Simulation objective**

Investigate the DLAH performance for a homogeneous network of workstations with different workloads.

- **Simulation parameters**

Each simulation, workstation WS[0] was assigned a different workload ranging from $workload/10$ to $workload \times 5$.

Processing power of each workstation has a uniform distribution around the mean ranging from $PP \pm 0.1 \times PP$

- **Expected outcome**

All the workstations will eventually have the corresponding workload within the desired threshold hence they will all have loop time within the desired threshold.

- **Results**

Figure 8.8 tracks the performance of the HNOW to a homogeneous network of workstations with different workloads. The loop time is plotted with respect to the simulation time. The DLAH algorithm was able to successfully balance WS[0] with its neighbors WS[1] and WS[4].

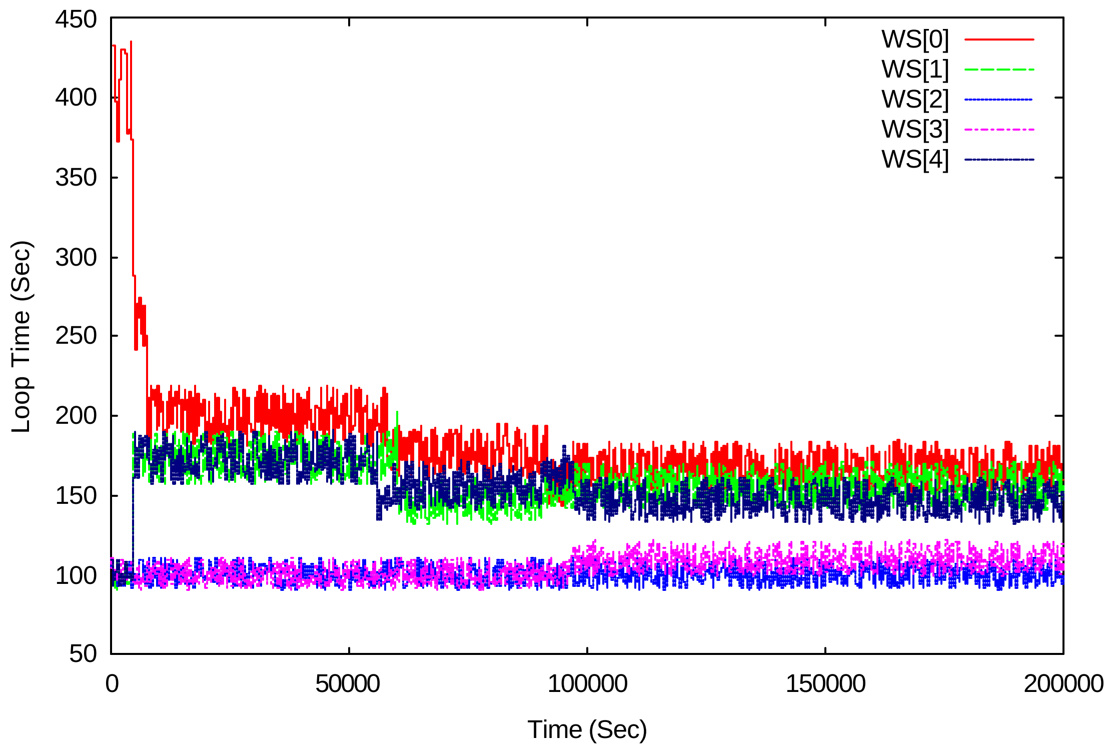


Figure 8.7: Loop time of a homogeneous network with workstation where WS[0] has been assigned a workload 4 times its neighbors.

In order to study the performance of DLAH to different workloads, we changed the workload for WS[0] only; starting from 0.1 times the normal workload to 5 times the normal workload. Figure 8.8 and figure 8.9 illustrate the performance of the DLAH algorithm. It is worth noting that as the workload increases; it takes more time for the workstation to balance itself with its neighbors, as it has longer calculations (loop time) before it begins its balancing phase.

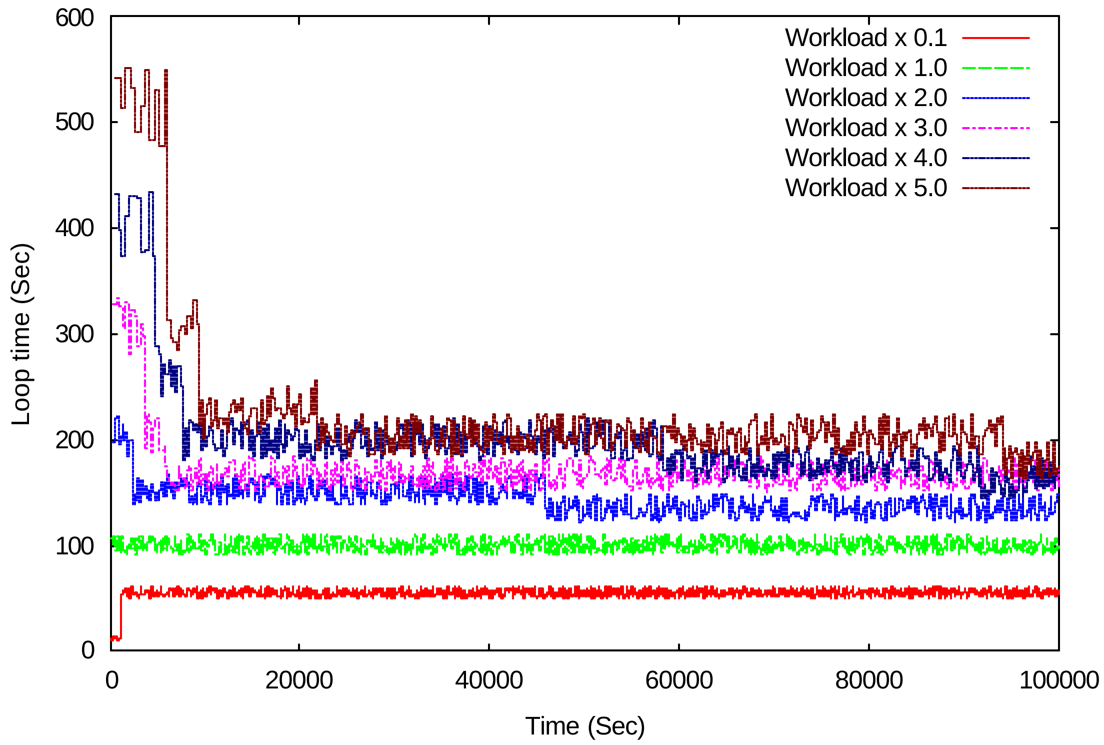


Figure 8.8: Workstation WS[0] loop time with different workloads.

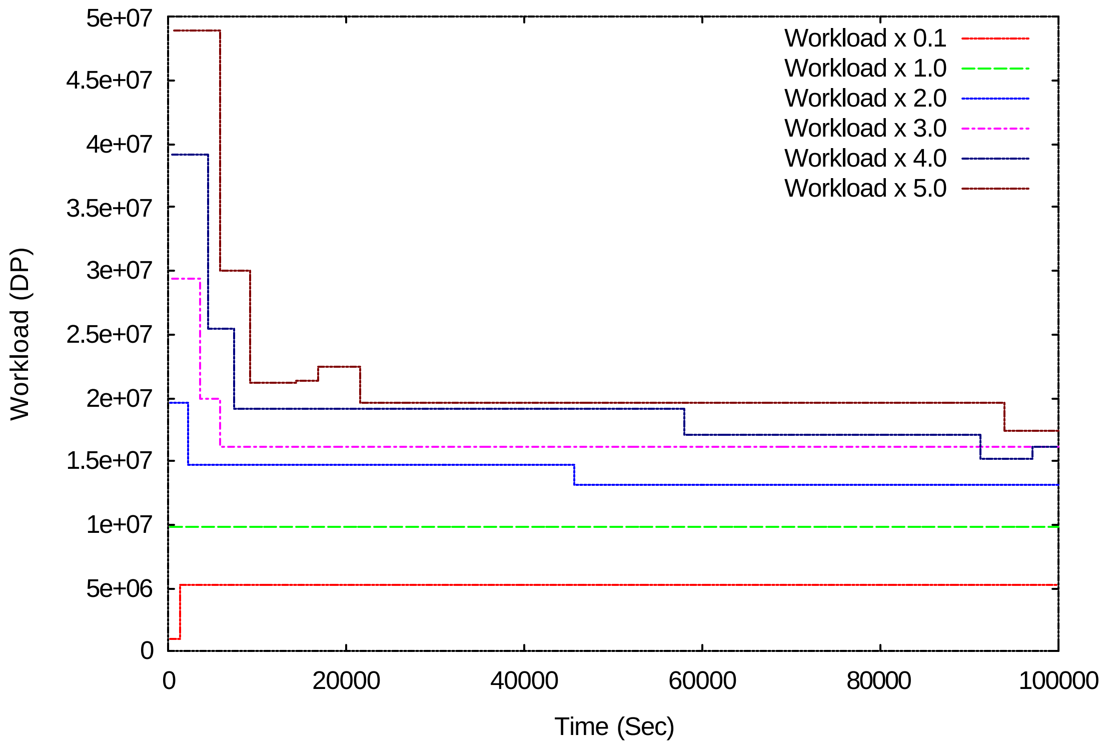


Figure 8.9 Workstation WS[0] workload distribution with time

The previous simulations have been conducted on a homogeneous network of workstations to validate the performance of the DLAH algorithm, as it is much easier to predict the flow of the extra workloads. The next section studies the sensitivity of the algorithm to the different HNOW parameters and measures its performance in each case.

8.2.2 HNOW with Different Processing Power

- **Simulation objective**

Investigate the DLAH performance for an HNOW of different processors.

- **Simulation parameters**

HNOW of 20 workstations with different processing power ranging from $PP/0.1$ to $PP/5$.

Processing power of each workstation has a uniform distribution around its mean ranging from $PP \pm 0.1 \times PP$

- **Expected outcome**

Each workstation will have a workload corresponding to its processing power within the threshold limit.

- **Results**

Figure 8.10 presents how the DLAH redistributes the workload with time while

figure 8.11 shows the corresponding loop time of the workstations. The loop time is smoothed such that each 20 samples are replaced with their mean so that tracking the loop times would be easier. the figures show that it takes about one or two steps to reach the balanced state.

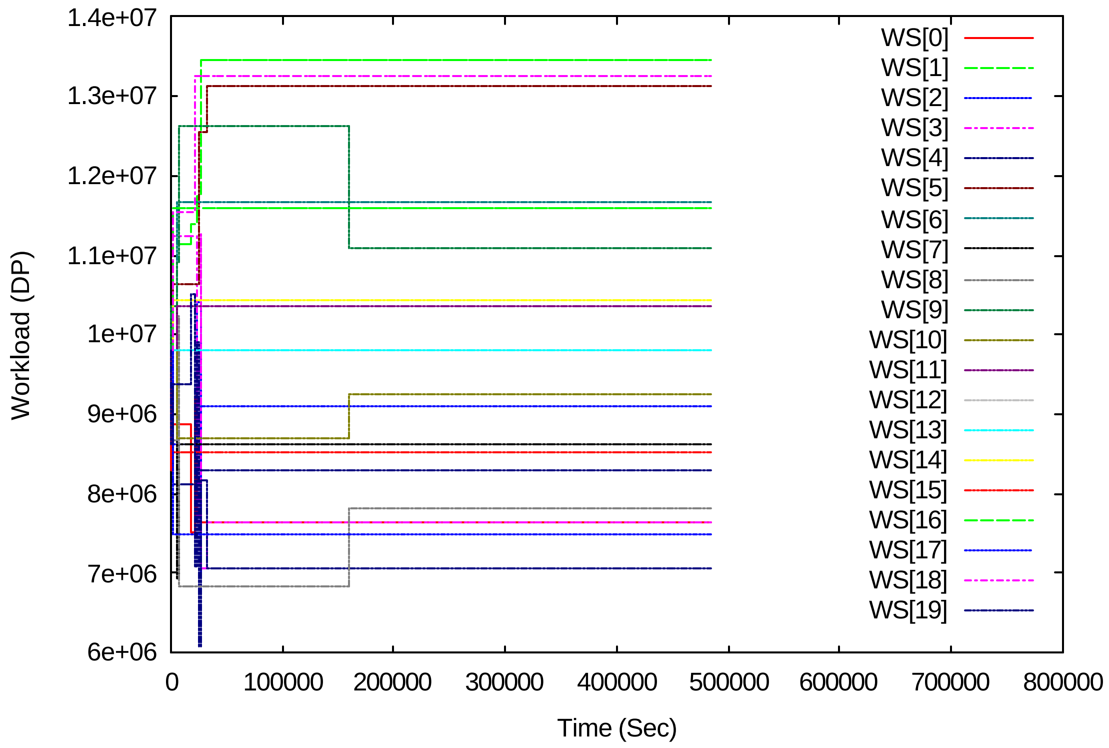


Figure 8.10: Workload distribution of an HNOW of different processing power.

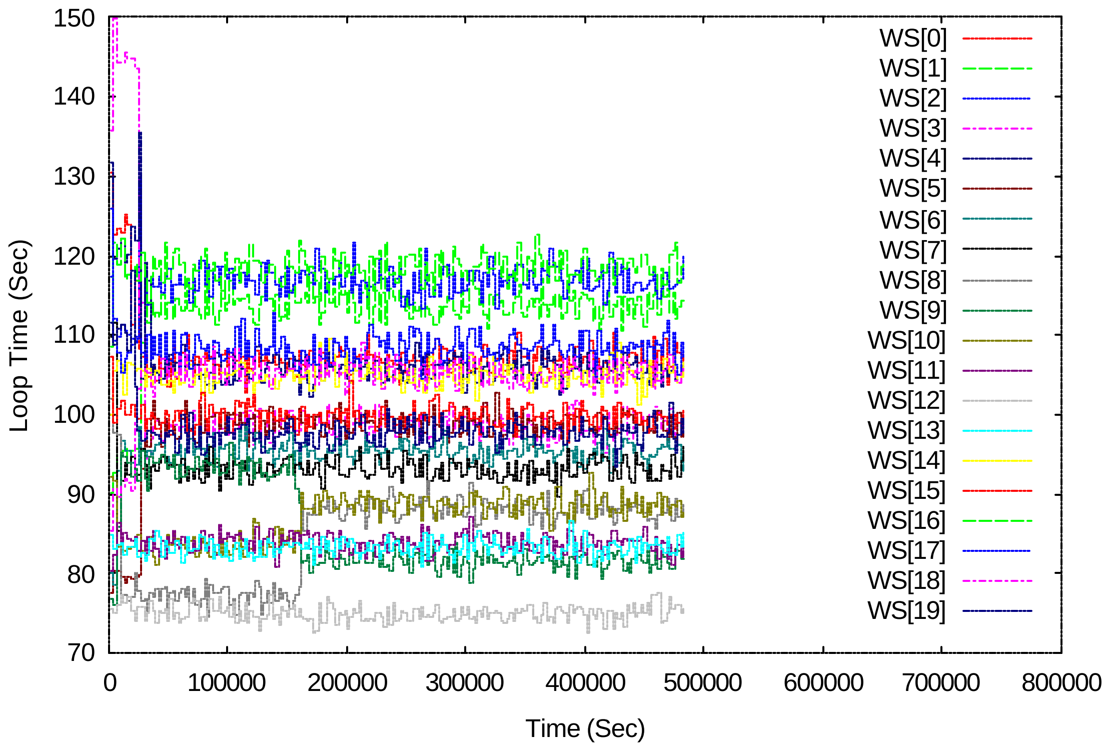


Figure 8.11: Smoothed loop time of an HNOW of different processing power.

We conducted a series of simulations and calculated the average number of steps required to reach balance state, as seen in table 8.2. Thus, we can conclude that with 90% confidence that the average number of steps 0.83 is in the interval [0.78, 0.89]. On the other hand, figure 8.12, which is a histogram of the number of steps encountered for balance, indicates that about 50% of the cases did not need any balance actions and that 90% of the cases needed two steps or less.

Average number of steps	Standard deviation	95% Confidence interval	*90% Confidence interval*
0.834	*0.977*	*0.058*	*0.048*

Table 8.2: Confidence interval for an HNOW of workstations of different processing power.

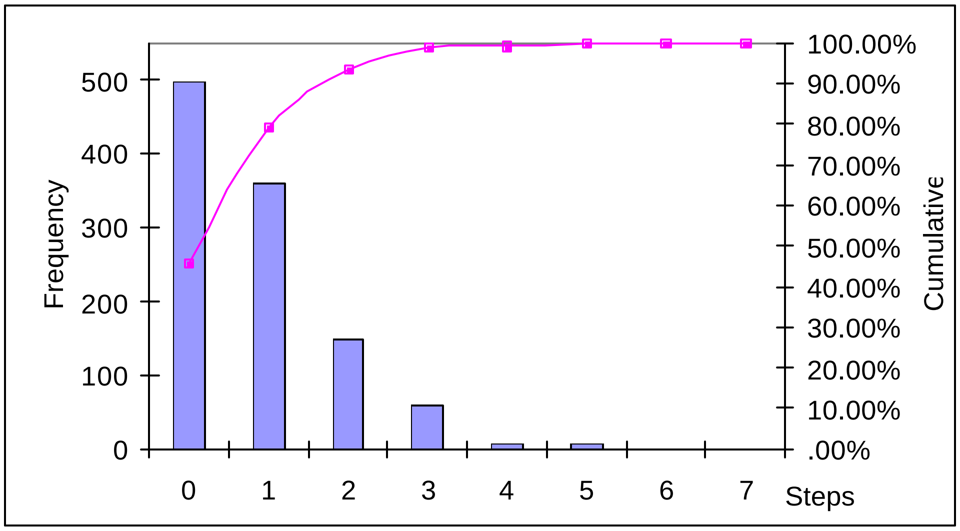


Figure 8.12: Histogram of the steps required for balance

In order to verify that the results obtained from the simulations are reliable enough, we plotted the graph of the 95% confidence interval with the number of simulations conducted as shown in figure 8.13. The confidence interval decreases as we increase the number of simulations, this observation validates both the HNOW simulation model and the DLAH algorithm. Also, from the graph we notice that we only need about 400 simulations to get a confidence interval less than 0.1.

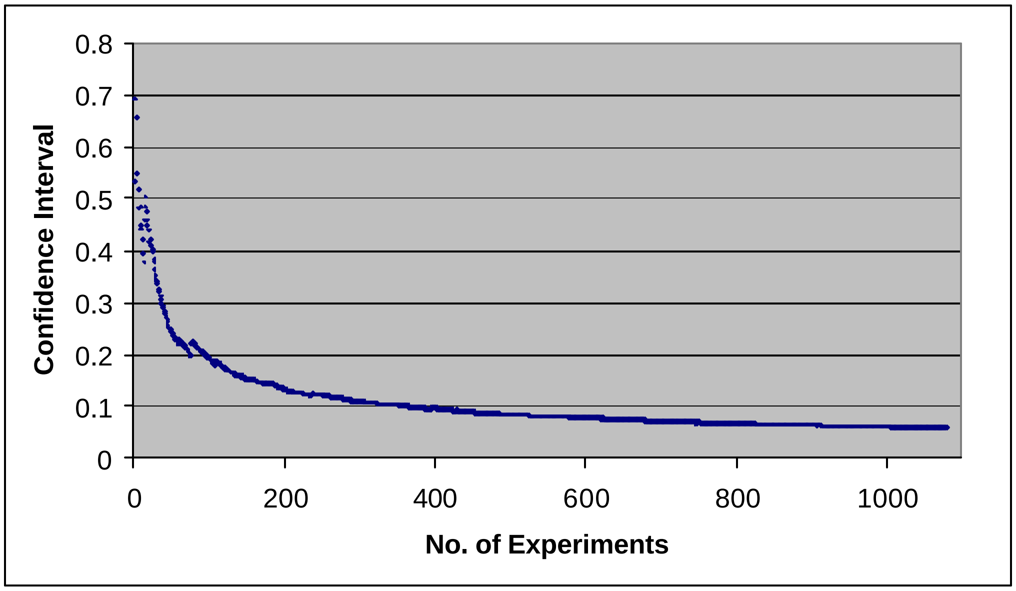


Figure 8.13: 95% confidence interval vs. number of simulations

In order to study the sensitivity of the DLAH algorithm to the processing power difference, a series of simulations have been conducted on an HNOW in which only workstation WS[0] has different processing power ranging from 0.2 to 10 times the normal processing power. Figure 8.14 illustrates the performance of the algorithm to different processing power. The DLAH is able to balance the workstations for the desired threshold setting, but it is not easy to conclude its relation to the number of steps required to reach the balanced state. Therefore, we conducted a series of simulations summarized in figure 8.15; we plotted the number of steps required to reach the balance state versus the processing power, which is a percentage of the default processing power. We can now conclude that the less the processing power, the more steps are required to reach the balance state.

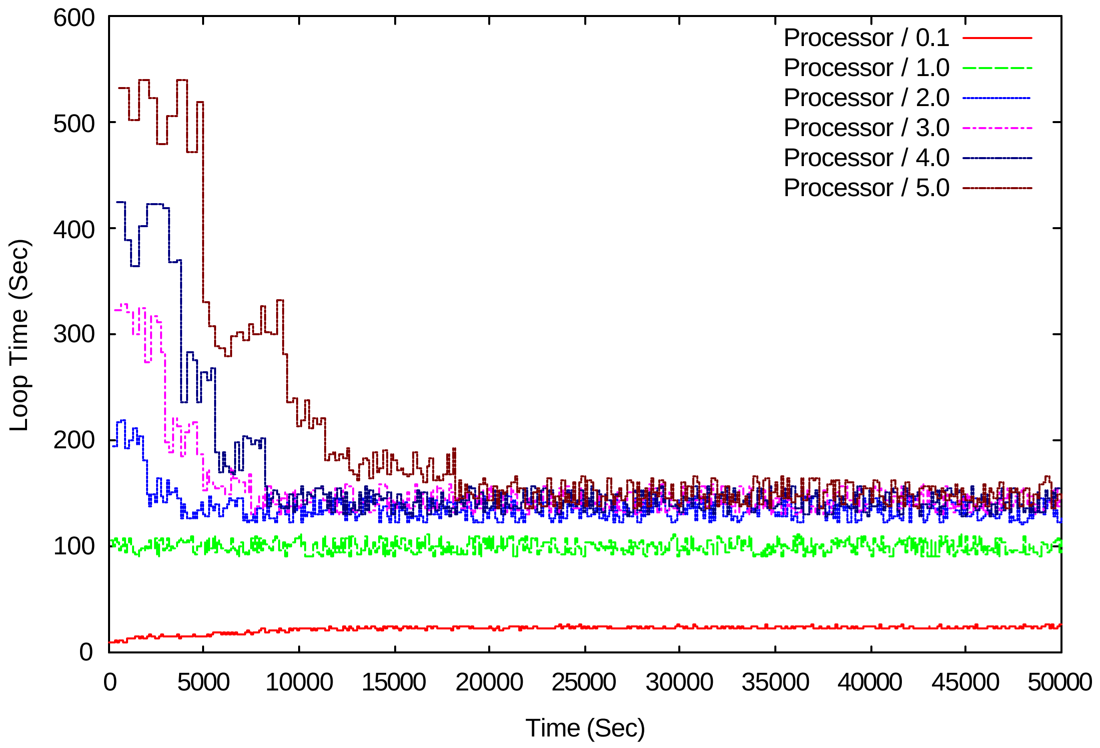


Figure 8.14:Workstation WS[0] with different processing power

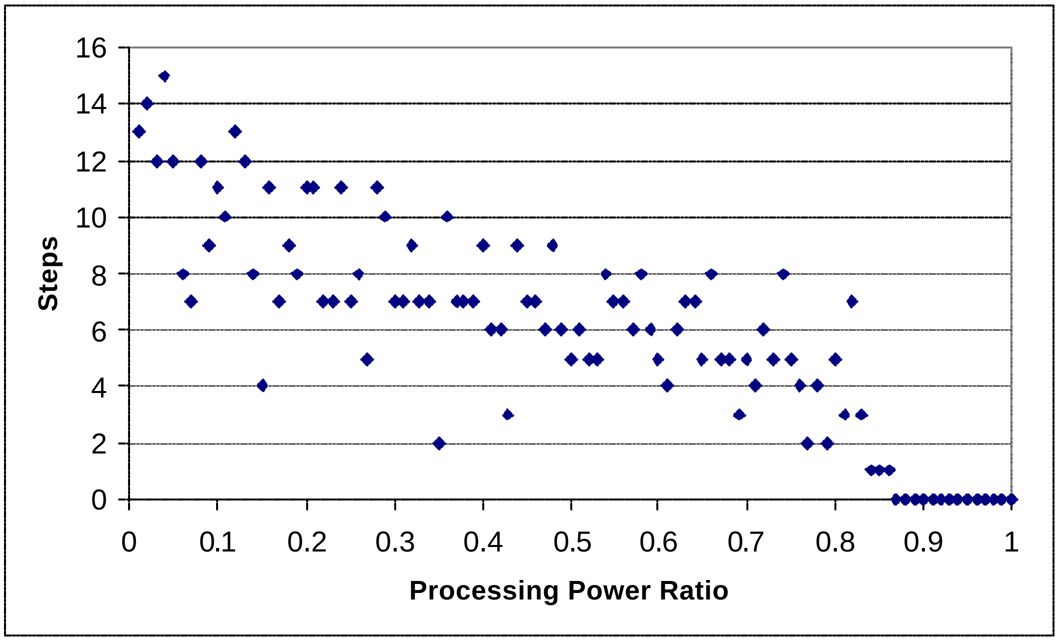


Figure 8.15: Number of steps vs. processing power ratio

8.2.3 HNOW with Different Memory Capacities

- **Simulation objective**

Investigate the DLAH performance for an HNOW of different memory capacities.

- **Simulation parameters**

HNOW of 20 workstations with different memory capacities, such that some workstations will have to use their swap memory.

To show the effect of the memory capacity on the performance, the total memory access time has to be comparable to the total loop time. Thus, we have increased the workload to twice its default value.

- **Expected outcome**

Each workstation will have a workload such that the total loop time for each workstation will be within the threshold range with its neighbors.

- **Results**

Figure 8.16 and figure 8.17 present both the loop time and the workload distribution with the simulation time. For this configuration, the DLAH was able to achieve balance in just one step as seen from the workload distribution with time. Table 8.3 shows the average number of steps required to reach the balance state.

Average number of steps	Standard deviation	95% Confidence interval	*90% Confidence interval*
0.067	*0.251*	*0.014*	*0.012*

Table 8.3: Average number of steps for an HNOW with different memory capacities.

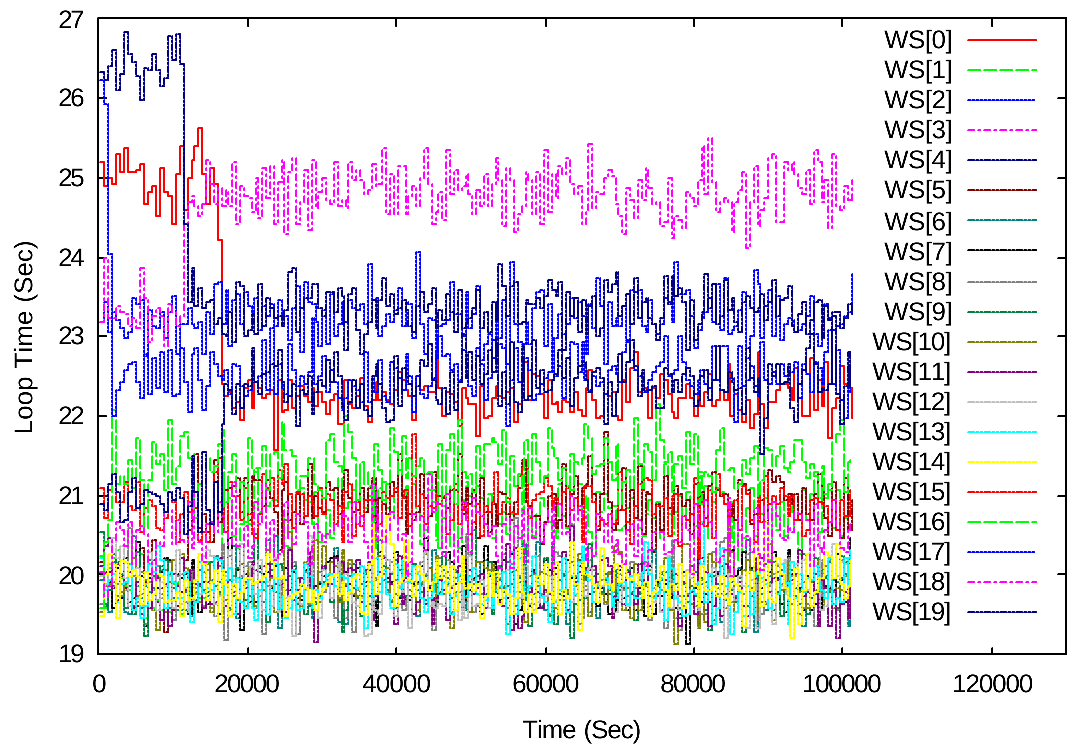


Figure 8.16: Loop time of an HNOW of 20 workstations of different memory capacities

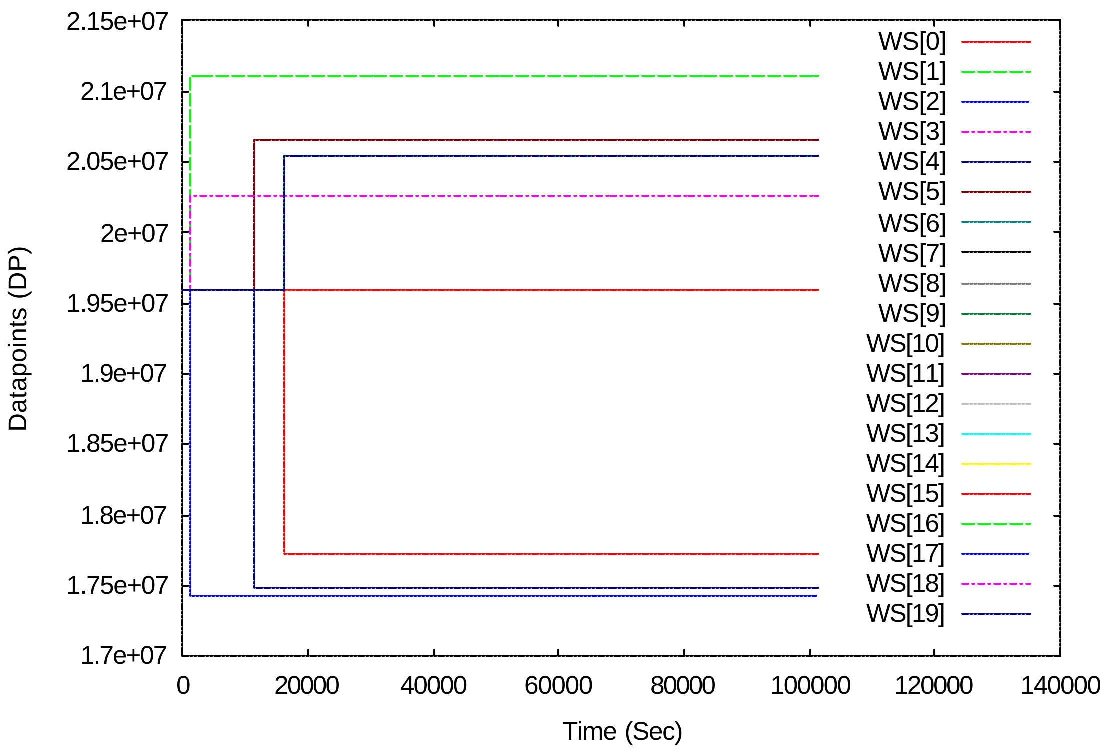


Figure 8.17: Workload distribution of an HNOW with different memory capacities.

We studied the sensitivity of the algorithm to the memory capacity by changing the memory capacity to workstation WS[0] only and plotted the result as shown in figure 8.18. The figure plots the memory capacity ratio, which is the ratio of the memory capacity of workstation WS[0] to the memory capacity of the default memory capacity. As the available memory decreases, the effect of the virtual memory increases. The number of steps increases steadily with the decrease of the available memory.

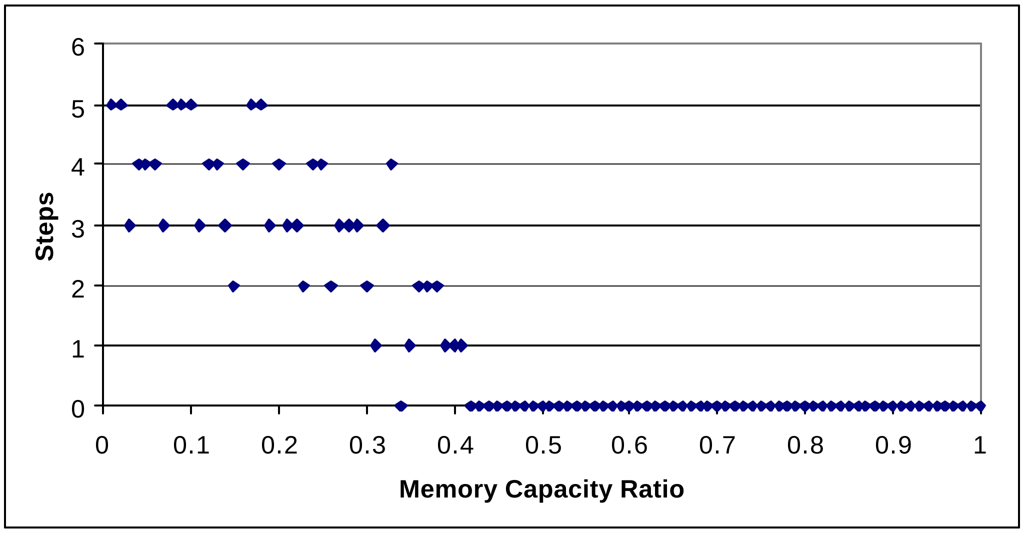


Figure 8.18: Number of steps vs. memory capacity ratio

8.2.4 HNOW with Different Network Parameters

- **Simulation objective**

Investigate the DLAH performance for workstations of different network parameters.

- **Simulation parameters**

HNOW of 20 workstations with different network parameters.

- **Expected outcome**

The workstation with slower network connections will send some of its workload to its neighbors such that the total loop time will be within the threshold limits.

- **Results**

In figure 8.19 we plotted the loop execution time to the simulation time for one of the simulations in which each workstation had different network parameters. The slow workstations from the figure; they are characterized by having a loop time peak when they transfer their extra workloads to its neighbors. In this set of simulations, all of the overloaded workstations did not take more than one step to get balanced. Table 8.4 reports the average number of steps it required to reach balanced state.

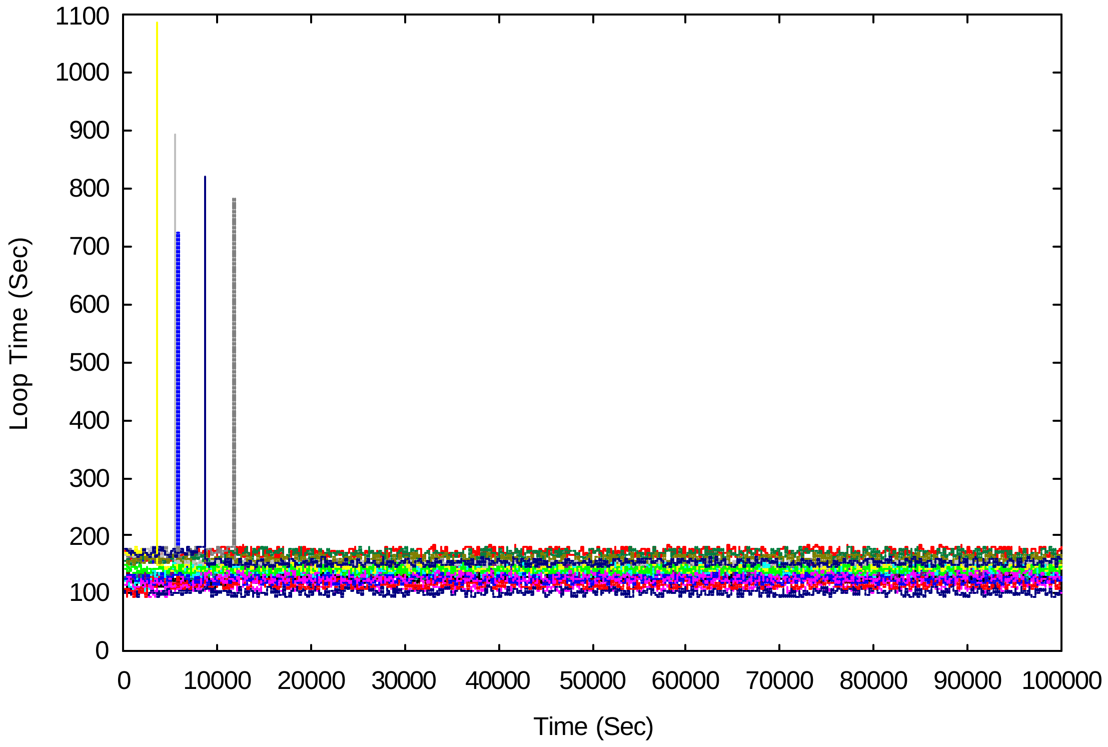


Figure 8.19: Loop time for an HNOW with different network parameters.

Average number of steps	Standard deviation	95% Confidence interval	*90% Confidence interval*
0.2194444	*0.470413*	*0.028055334*	*0.023544791*

Table 8.4: Average number of steps for an HNOW with different network parameters.

To study the sensitivity of the algorithm to the network parameters like the previous cases, we conduct a series of simulations with different network parameters assigned to one workstation WS[0]. Figure 8.20 and figure 8.21 confirm that the workstations need only one-step to get balanced irrespective of how slow the network connection is and obviously, the workstation with slower network connection takes a longer loop time to send its workload.

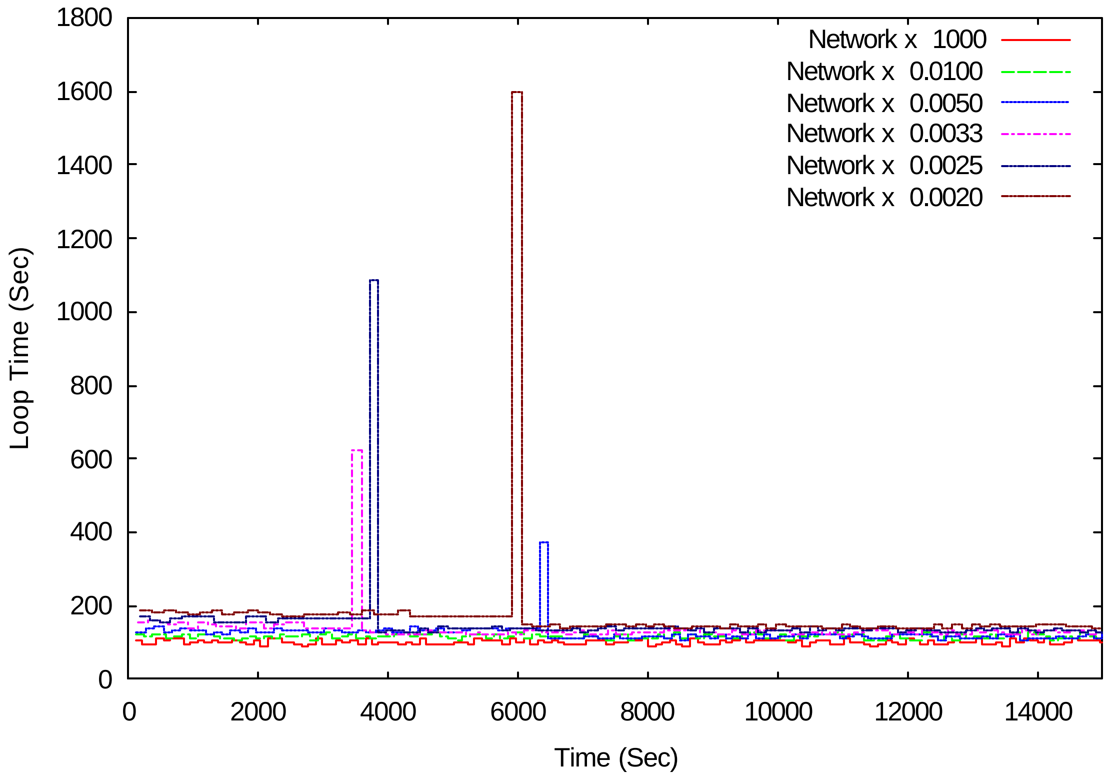


Figure 8.20: Loop time for workstation WS[0] with different network parameters.

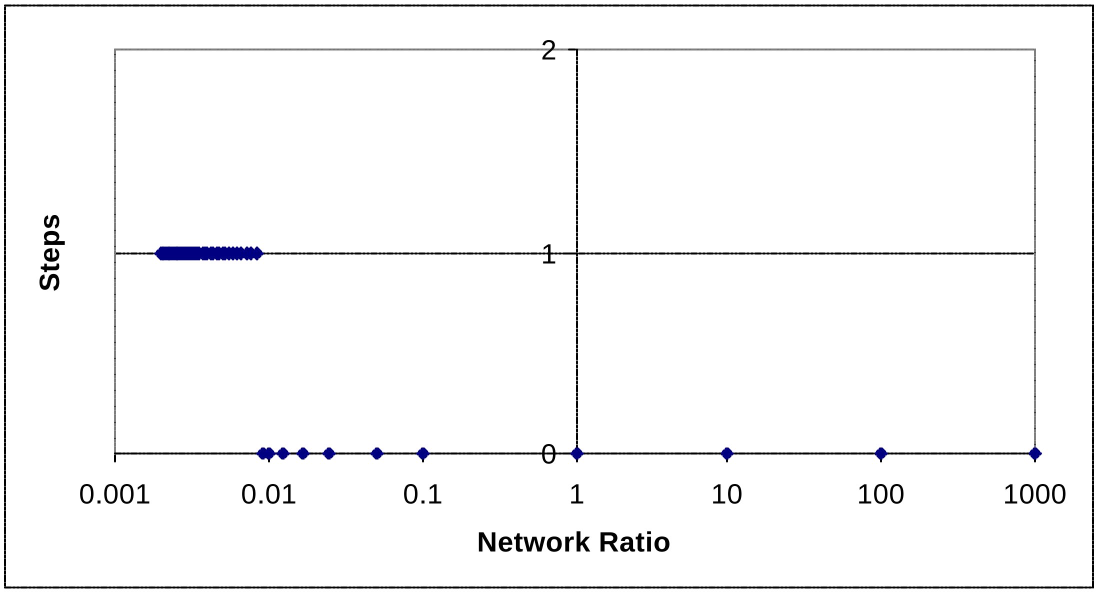


Figure 8.21: Number of steps vs. network ratio

8.2.5 Complete HNOW

- **Simulation objective**

In this section, we study the overall DLAH performance for a complete HNOW in which all the workstations differ in all the parameters.

- **Simulation parameters**

HNOW of workstations with different parameters.

- **Expected outcome**

The number of steps should be within the same range, irrespective the random assignment of the parameters.

- **Results**

Figure 8.22 displays a histogram of the number of steps used to balance one workstation in the complete HNOW, while table 8.5 illustrates the average number of steps encountered from 1080 simulations conducted with different random seeds.

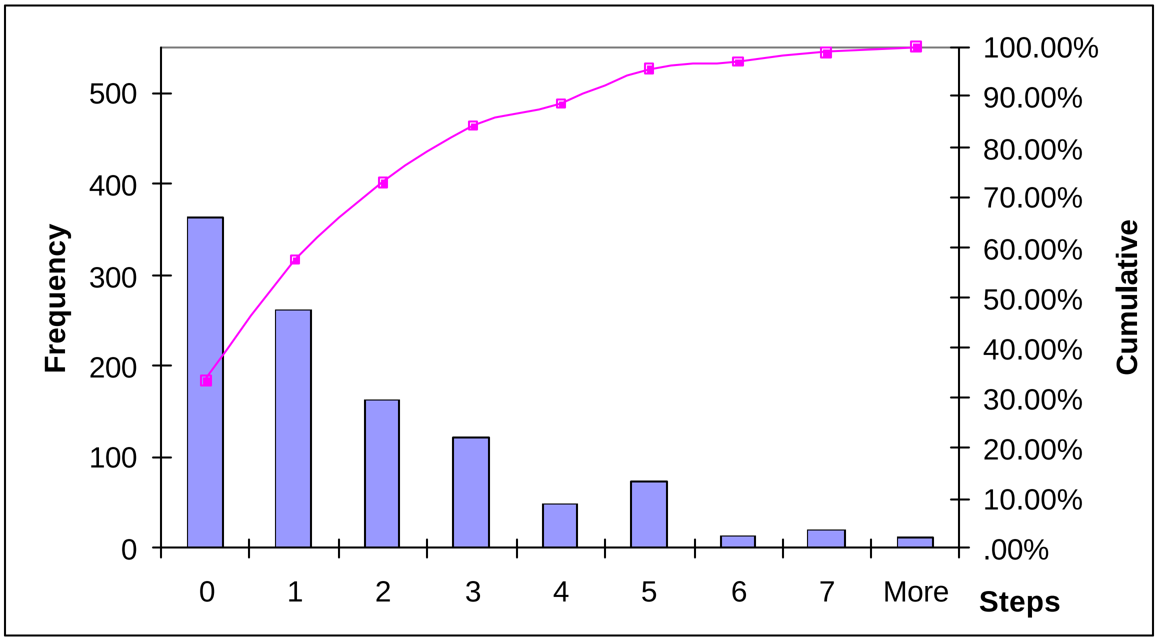


Figure 8.22: Histogram of the number of steps required to reach balance state.

Average number of steps	Standard deviation	95% Confidence interval	*90% Confidence interval*
1.723	*1.914*	*0.114*	*0.095*

Table 8.5: Average number of steps for a complete HNOW.

The complete HNOW simulations have been repeated with different number of workstations. All of the simulations yielded nearly the same results as seen in figure 8.23. Each average number of steps is plotted with its confidence interval versus the number of workstations.

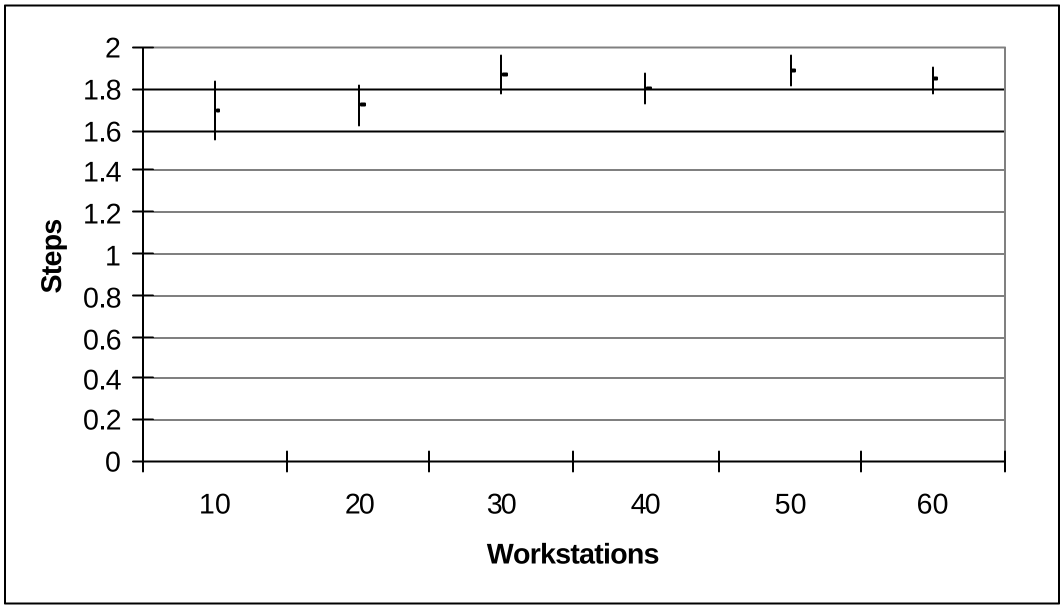


Figure 8.23: Average number of steps vs. number of workstations.

8.3 DLAH Influencing Factors

In the previous section, the DLAH performance was studied for different HNOW parameters. However, there are other parameters that may influence the DLAH performance, which need further investigation. These factors are the workload difference between neighbors, the threshold limit, and the scalability of the algorithm. We will investigate each factor in the following sections.

8.3.1 The Effect of the Workload

Simulations have been conducted to study the effect of the workload difference between neighbors on the DLAH performance. We used a homogeneous network of workstation with one workstation being assigned a different workload.

In figure 8.24, we plotted the workload ratio, which is the ratio between the workload of workstation WS[0] to the default workload, and the number of steps required to reach balance state.. As the workload difference increases, it takes more steps to reach the balance state, this also agrees with the analytical performance bounds.

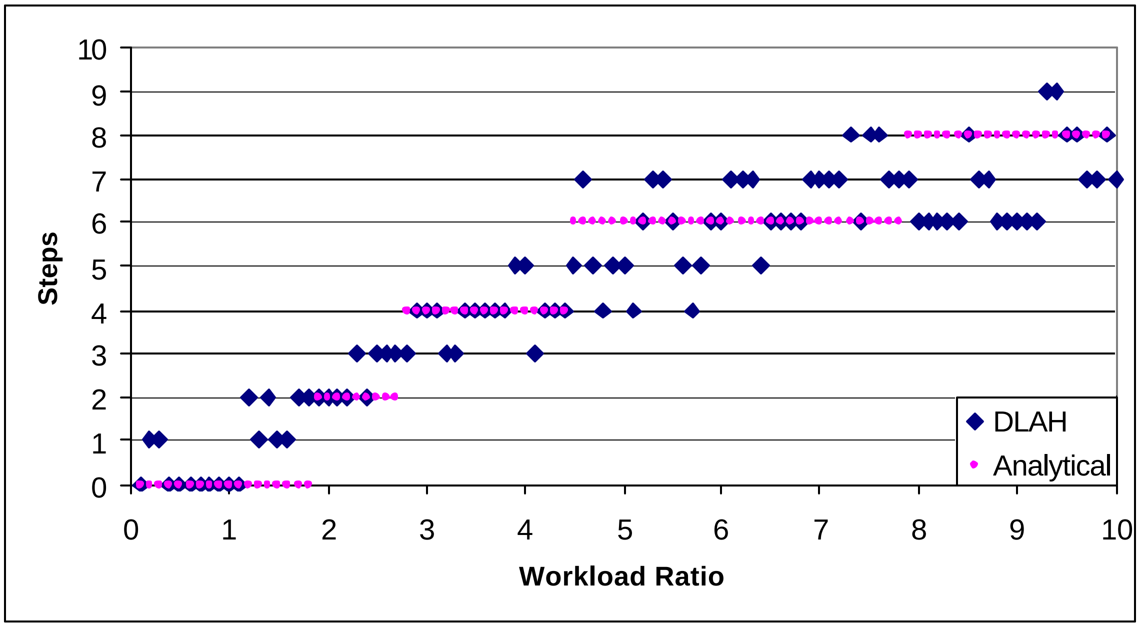


Figure 8.24: The effect of the workload difference between neighbors

8.3.2 The Effect of the Threshold Level

As mentioned previously, the threshold ratio determines the stable region, which is a ratio from the local average. Thus the stable region is $(1 \pm c) \times Local \;\; Average$. $c = 0$ means that there is no stable region and strict balance is required while $c = 1$ means that no balance is required. Figure 8.25 depicts the relationship between the threshold ratio and the number of steps required for balance. Obviously, the less the threshold the more responsive the algorithm will be to the changes, but the more overheads it requires keeping it in balance. Furthermore, when the threshold drops below the loop time variance, the algorithm will keep running until the end of the simulation.

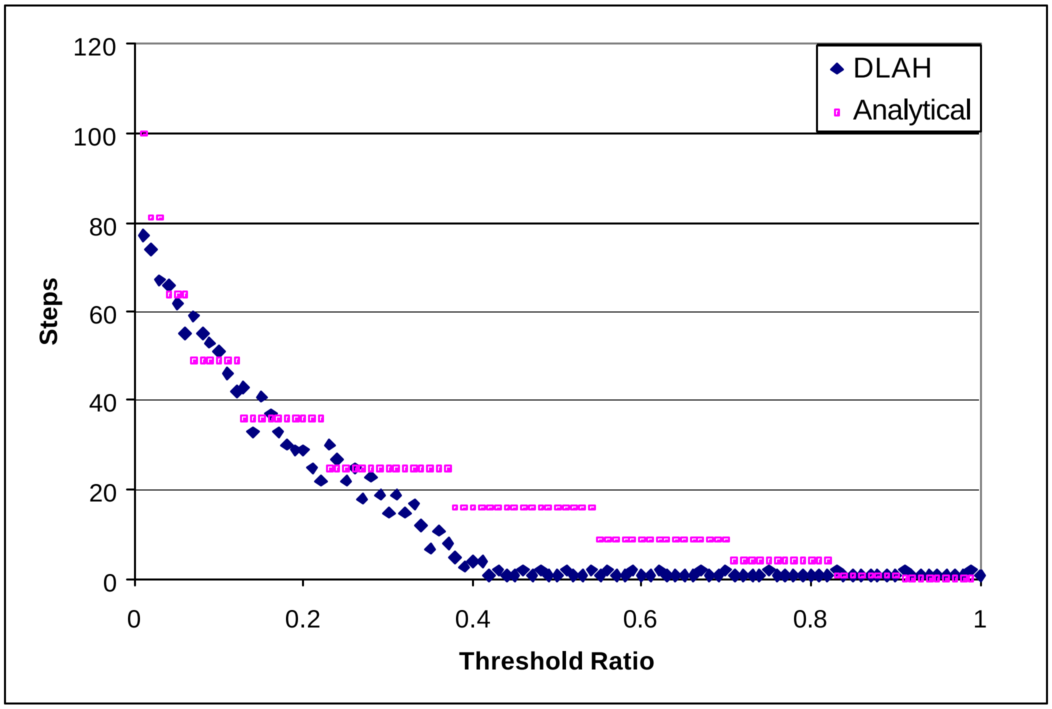


Figure 8.25: The effect of the threshold level

8.3.3 Scalability of the DLAH Algorithm

One of the main attributes that we set as a goal while designing the DLAH algorithm is to make sure that the algorithm is scalable with the number of workstations. As discussed previously, the DLAH algorithm uses the information of its neighbors only to determine its current status and accordingly exchange different workloads to achieve balance. Therefore, the DLAH should be scalable with the number of workstations as it only depends on the number of neighbors and not the total number of workstations in the network. We have conducted several simulations to confirm that theory. Figure 8.26 confirms that the DLAH algorithm is scalable; the figure shows the average number of steps required for balancing with its confidence interval versus the number of workstations in the network. The average number of steps is nearly constant throughout the whole simulations.

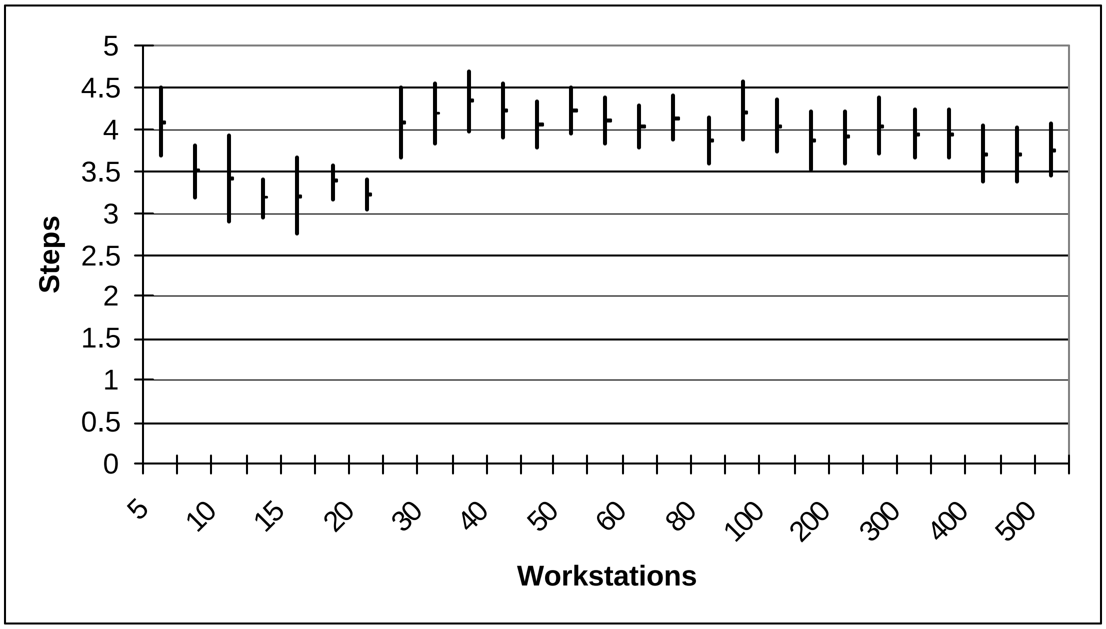


Figure 8.26: Average number of steps vs. the number of workstations

8.4 DLAH Compared to Other Dynamic Load Balancing Algorithms

In this section, we compared the performance of the DLAH algorithm to other diffusion algorithms. We compared its performance to a diffusion-oriented algorithm for a homogeneous network of workstations. In addition, we compared it to another diffusion algorithm for an HNOW that takes into account the processor heterogeneity only.

We conducted two sets of simulations. In the first set of simulations, random values were assigned to each workstation and then we measured the average number of steps and load exchanged it takes to reach a steady state. We conducted the second set of simulations on a non-dedicated homogeneous network of workstations. The non-dedication is simulated by reducing the available resources to a random value at a certain time, then releasing back the resources after some time. Both simulations are discussed in details in the following sections.

8.4.1 DLAH Compared to Other Diffusion Algorithms on an HNOW

In this set of simulations, we assigned random variables to each workstation. We conducted several simulations with different number of workstations. Figure 8.27 compares the average number of steps required to balance the HNOW system. DLAH shows slightly better performance over the other diffusive algorithms and the processor only algorithm shows a.

slightly better performance than the homogeneous algorithm. However, figure 8.28 indicates that the datapoints exchanged to achieve the balance is much less using the DLAH algorithm. This shows that the DLAH algorithm is able to tune into the HNOW parameters and exchange just exact the datapoints required to balance the system

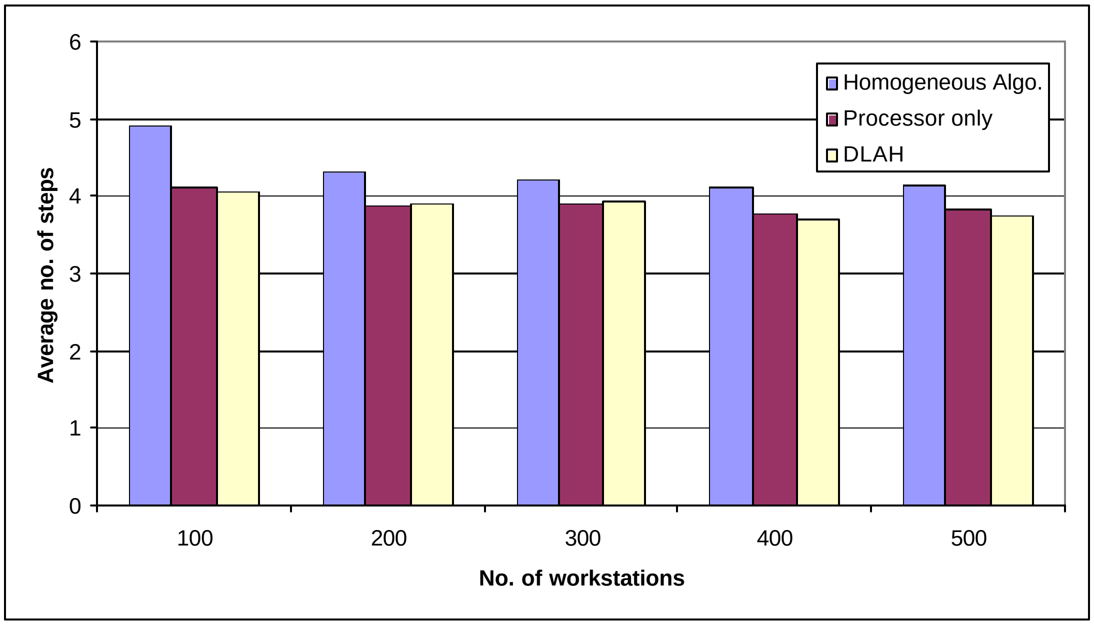


Figure 8.27: Comparison between the average numbers of steps required for balance

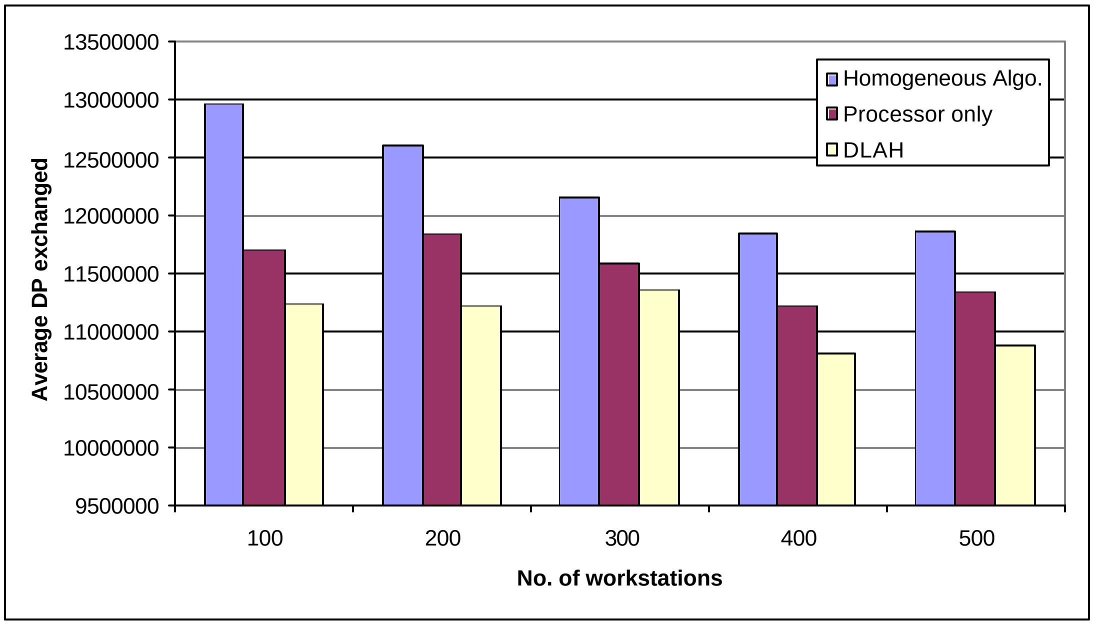


Figure 8.28: Comparison between the average DP exchanged.

8.4.2 DLAH Compared to Other Diffusion Algorithms on a Non-Dedicated Homogeneous Network of Workstations.

It seems obvious that the DLAH algorithm should perform better than the other algorithms that do not consider all the HNOW parameters. Consequently, we conducted another set of simulations on a homogeneous network of workstation to compare the DLAH's performance.

This set of simulations was conducted on a non-dedicated homogeneous network of workstations. A step of disturbance was introduced at the first workstation by reducing its resources to a set value at loop 100, then releasing them back at loop 600. The simulations were conducted on a network of 10 workstations. Table 8.6 shows that the DLAH has a slightly better performance regarding the average number of steps required to cope with the disturbance step. However, table 8.7 shows that the DLAH needs much less datapoints (about half) to be exchanged to achieve the balance.

Figure 8.29 shows an example of the reaction of each algorithm to the step disturbance. Both the homogeneous and processor only have the same response, while the DLAH exchanged less load to reach the steady state.

Algorithm	*Average number of steps*	*Standard deviation*	*95% Confidence interval*	90% Confidence interval
Homogeneous Algo.	*1.532*	*2.149*	*0.133*	0.111
Processor only	*1.458*	*2.0170*	*0.125*	0.104
DLAH	*1.417*	*2.416*	*0.149*	*0.125*

Table 8.6: Average number of steps

Algorithm	*Average number DP exchanged*	*Standard deviation*	*95% Confidence interval*	90% Confidence interval
Homogeneous Algo.	*10194117*	*20273108*	*1256516*	1054502
Processor only	*10179673*	*20261533*	*1255799*	1053900
DLAH	*5730649*	*13268132*	*822351*	*690139*

Table 8.7: Average number of datapoints exchanged

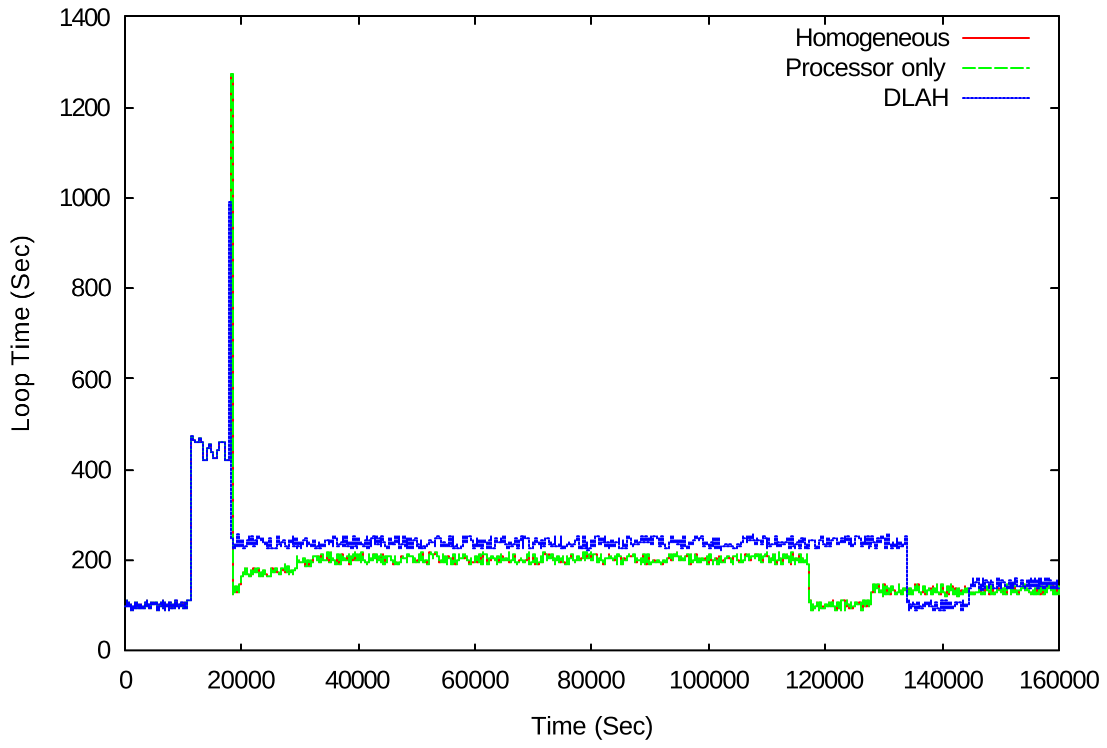


Figure 8.29:The Algorithms reactions to a step disturbance

8.5 Summary of Results

In this chapter, we applied the DLAH algorithm to a simulation model we developed for an HNOW. We conducted several types of simulations to study the DLAH algorithm. These simulations are divided into four different sections.

In the first section, we conducted a set of simulations to validate the DLAH algorithm. This set of simulations was conducted on a homogeneous network of workstations, as the simulations results are predictable in this case, thus, validation can take place. The simulations included:

- Compare the results of the DLAH with different threshold ratio to the analytical performance bounds. Results showed that the DLAH follows the analytical bounds.
- Track the execution of the DLAH algorithm on a homogeneous network with random workloads assigned to the workstations. The results showed that the DLAH is working as expected; hand calculations were performed on the results to validate the algorithm.

- Track the performance of the DLAH on a homogeneous network with one workstation being assigned different workloads. All the workstations eventually were balanced within the threshold range.

In the second section, we studied the sensitivity the DLAH algorithm to the HNOW parameters. The simulations were conducted on an HNOW, which included:

- HNOW with different processing power. The number of steps increased as the processing power mismatch increased. We also plotted the confidence interval with the number of simulations conducted. We found that the confidence interval decreases with increasing number of simulations; that is yet another validation of the simulation model.
- HNOW with different memory capacities. The number of steps increased as the memory capacity mismatch increases.
- HNOW with different network capacities. In this case, DLAH needed only step to eliminate any load imbalance due to network connections. It does not make any difference how much the network mismatch is. The slow network connections are characterized by having a loop execution time impulse when they send their extra workload to their neighbors.
- Complete HNOW. A series of simulations were conducted with different number of workstations being assigned random HNOW parameters. All of these simulations needed nearly the same number of steps to reach global balance.

In the third section, we studied the performance of the DLAH algorithm to influencing factors. The simulations included:

- The effect of the workload. We studied the effect of workload mismatch on the DLAH performance in more details. The number of steps required for balance increases with the workload mismatch. The number of steps is bound by the analytical performance bound.
- The effect of the threshold ratio. In these simulations, we plotted the result of changing the threshold ration with the number of steps required to reach balance. The number of steps increased as the threshold decreases. The results show that they obey the analytical

bounds too. It is worth to note that as the threshold ratio decreases the DLAH becomes more responsive to imbalance, but also takes more steps to reach a balanced state.

- Scalability of the DLAH. One of the important factors that we needed to check is that we do not want the load-balancing algorithm to inhibit the scalability of the pipelined SPMD application. Simulations were conducted on networks ranging in size from 5 to 500 workstations. The output results were nearly the same; number of steps needed to reach global balance was nearly the same. This confirms that our algorithm is scalable with the number of workstations.

In the last section, we compared the performance of the DLAH algorithm to other related algorithms. The simulations included:

- Complete HNOW. Although the number of steps required to reach a balanced state was close to the other algorithms, the DLAH was able to achieve balance with much less load exchanged. This simulation was repeated for different number of workstations and the results were almost the same each time.
- Non-dedicated homogeneous network of workstations. We also compared the performance of the DLAH to the related algorithms on a non-dedicated homogeneous network simulated by a pulse of disturbance. The number of steps to achieve balance was practically the same. However, the DLAH was superior in that it was able to achieve balance with much less load exchanged nearly half the others.

Chapter 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions and Contributions

Throughout this work, we designed DLAH, which is a scalable dynamic load-balancing algorithm for pipelined SPMD applications on HNOW. During this process, we formally defined the load-balancing problem and proved it to be an NP-Complete problem. We identified the different HNOW parameters and proposed a general taxonomy for load-balancing algorithms. Accordingly, we designed DLAH and deduced its analytical bounds.

We used measurements from two case studies to build our simulation model of the HNOW. We then implemented the DLAH on the simulation model and conducted four different sets of simulations.

The first set is concerned with validating the DLAH algorithm. These simulations were conducted on a homogenous network of workstation. They included comparing the results with the analytical performance bounds and hand calculations.

The second set of simulations studied the sensitivity of the DLAH algorithm to the HNOW parameters, summarized in the following points.

- As the processing power mismatch increases between the workstations, the number of steps required to reach the steady state increases.
- As the available memory mismatch increases between the workstations, the number of steps required to reach the steady state increases.
- As for the network mismatch, the DLAH needed only step to eliminate any load imbalance due to network connections. It does not make any difference how much the network mismatch is. The slow network connections are characterized by having a loop execution time impulse when they send their extra workload to their neighbors.

In the third set of simulations, we studied the performance of the DLAH algorithm to influencing factors, which are the workload mismatch, threshold ratio and the scalability.

- The number of steps required for balance increases with the workload mismatch. The number of steps conforms to the analytical performance bound.
- The number of steps increases as the threshold ratio decreases. This also agrees with the analytical bounds. It is worth noting that as the threshold ratio decreases the DLAH becomes more responsive to the imbalance, but also takes more steps to reach a balanced state.
- The DLAH algorithm is scalable with the number of workstations, as it only depends on the neighbors to obtain its status.

In the last set of simulations, we compared the performance of DLAH to other load-balancing algorithms. Although the simulations showed that the DLAH takes slightly less steps to reach the balanced state in average, it does however achieve it with much less load exchanged (nearly half as much). This is because the DLAH is able to tap into the HNOW parameters and suppress the bouncing effect.

We would like to point out that the process of designing DLAH is as important as the DLAH algorithm itself. The process is summarized in the following points:

- Comprehend the different HNOW parameters in the intended network.
- Understand the application and determine the best suitable parallel programming paradigm.
- Determine the essential features for the load-balancing algorithms besides the main purpose of it, which is balancing itself. These features include scalability, responsiveness, least overheads, simplest implementation, etc.
- Design the load-balancing algorithm accordingly using the proposed taxonomy (Figure 2.3).
- Derive analytical performance bounds to be able to verify the performance of the new algorithm.

- Conduct pilot simulations and measure the performance using the different performance parameters.

9.2 Future Work

As old clusters are updated with new workstations, the need for a good load-balancing algorithm or scheduler increases (in order to take full advantage of the HNOW). As mentioned before, there is no load-balancing algorithm fit for all types of applications. We have only tackled one kind of applications, the pipelined SPMD. Other parallel paradigms need to be investigated like phase parallel, divide and conquer, and work pool.

The analytical performance bounds for diffusive strategies derived in this research are based on a homogeneous network of workstations. Deriving analytical performance bounds for HNOW will allow us to validate load-balancing algorithms for HNOW besides the homogeneous network of workstations.

In this research, we considered the common case in which the workstations communicate only two of its data domain sides. Actually, depending on the domain decomposition, the workstation may communicate all its six data domain sides. Although the DLAH algorithm is scalable with the number of workstations, but the overall performance may be affected when increasing the communication sides. Further investigation is required to study the performance.

In our simulations, we made our best effort to capture the parameters of a real network of workstation. However, there are many other parameters like cache memory, word length, operating system, etc., that we did not consider in our model. Implementing the DLAH on a real pipelined SPMD application and comparing its performance to the simulations will give us an indication of how close the simulation model is to the real HNOW and how well does the DLAH perform accordingly.

In addition, the load-balancing problem is still an NP-Complete problem; the field is open to all kind of algorithms and heuristics especially for the HNOW.

One of the interesting research topics that we recommend is building a theoretical model that captures the different HNOW features, which could provide us with quick static performance measures.

REFERENCES

[ACS89] A. Aggarwal, A. K. Chandra, and M. Snir. "On communication latency in PRAM computations," In Proceedings of the first ACM Symp. on Parallel Algorithms and Architectures, pages 11-21, June 1989.

[AM+01] H. Aydin, R. Melhem, D. Mosse, and P. Meja-Alvarez. "Dynamic and aggressive scheduling techniques for power-aware real-time systems," In the 22nd IEEE Real-Time Systems Symposium (RTSS'01), pages 95-105, Washington - Brussels - Tokyo, December 2001.

[B99] C.A. Bohn. "Asymmetric load balancing on a heterogeneous cluster of PCs," MSCE Thesis, AFIT/GE/ENG/99M-02, Graduate School of Engineering, Air Force Institute of Technology (AETC), Wright Patterson AFB OH, March 1999.

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

[CCN97] M. Colajanni, M. Cermele and G. Necci. "Dynamic Load Balancing of Distributed SPMD Computations with Explicit Message-Passing," In Proceedings of the IEEE Workshop on Heterogeneous Computing, pages 2-16, 1997.

[Cho00] Seonho Choi. "Dynamic time-based scheduling for hard real-time systems," Journal of Real-Time Systems, 2000.

[CK98] T. L. Casavant, and J. G. Kuhl. "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," IEEE Trans. on Soft. Eng. 14, pages 141-154, 1998.

[CKS93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. "LogP: Towards a realistic model of parallel computation," In Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993.

[CLZ99] A. Corradi, L. Leonardi, and F. Zambonelli. "Diffusive load-balancing policies for dynamic applications," IEEE Concurrency Parallel, Distributed and Mobile Computing, pages 22-31, January-March 1999.

[CZL97] Michal Cierniak, Mohammed Javeed Zaki, and Wei Li. "Compile-time Scheduling Algorithms for Heterogeneous Network of Workstations," The Computer Journal , special issue on Automatic Loop Parallelization, Vol. 40, No. 6, December 1997.

[DCG93] Dietz, H. G., Cohen, W.E. and Grant, B. K. "Would You Run it Here... or There?" Automatic Heterogeneous Supercomputing, International Conference on Parallel Processing, Volume II: Software, pages 217-221, 1993.

[DFM95] Jean-Luc Dekeyser, Cyril Fonlupt, and Philippe Marquet. "Analysis of synchronous dynamic load balancing algorithms," Parallel Computing: State-of-the Art Perspective (ParCo'95), volume 11 of Advances in Parallel Computing, pages 455-462, Gent, Belgium, September 1995.

[DGP84] Dihn, QQ.V., Glowinski, R. and Periaux, J.. "Solving Elliptic Problems by Domain Decomposition Methods with Applications," Elliptic Problem Solvers II, Academic Press, New York, 1984.

[FMD98] C. Fonlupt, P. Marquet, and J. Dekeyser. "Data-parallel load-balancing strategies," Parallel Computing 24, pages 1665-1684, 1998.

[GJ79] Michael R. Garey, David S. Johnson. "Computers and intractability: a guide to the theory of NP-completeness," New York, W.H. Freeman, 1979.

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. "Using MPI: Portable Parallel Programming the Message-Passing Interface," The MIT Press, Cambridge, MA, first ed., 1994.

[GPS95] R. Gerber, W. Pugh, and M. Saksena. "Parametric Dispatching of Hard Real-Time Tasks," IEEE Transactions on Computers, 1995.

[GMR94] P.B. Gibbons, Y. Matias, and V. Ramachandran. "The Queue-Read Queue-Write PRAM model: Accounting for contention in parallel algorithms," SIAM Journal on Computing, 1997.

[HCF03] Karin Hgstedt, Larry Carter, and Jeanne Ferrante. "On the parallel execution time of tiled loops," IEEE Trans. on Parallel and Distributed Computing, March 2003.

[Hil85] W.D. Hillis. "The Connection Machine," MIT press, 1985.

[Hwg93] Kai Hwang. "Advanced Computer Architecture: Parallelism, Scalability, Programmability," MIT Press, 1993.

[HX98] Kai Hwang, Zhiwei Xu. "Scalable Parallel Computing," McGraw-Hill, 1998.

[KEB99] G-S Karamanos, C. Evangelinos, R.C. Boes, R.M. Kirby and G.E. Karniadakis. "Direct Numerical Simulation of Turbulence with a PC_Linux Cluster Fact or Fiction," Proceedings of Supercomputing conference SC99, Oregon, Portland, November 1999.

[LK91] Averill M. Law and W. David Kelton. "Simulation Modeling and Analysis," McGraw-Hill, 1991.

[LL96] C. Lu and S. Lau. "An Adaptive Load Balancing Algorithm for Heterogeneous Distributed Systems with Multiple Task Classes," In Proceedings of the 16th International Conference on Distributed Computing Systems, pages 629-636, 1996.

[LST90] J. K. Lenstra and D. B. Shmoys and É. Tardos. "Approximation Algorithms for Scheduling Unrelated Parallel Machines," Mathematical Programming: Series A, Volume 46(3), pages 259-271, 1990.

[LV90] L. G. Valiant. "A bridging model for parallel computation," Communications of the ACM, 33(8):103-111, August 1990.

[Mat96] T. G. Mattson. "Scientific computation," Parallel and Distributed Computing Handbook (A. Y. Zomaya, editor), Series on Computer Engineering, pages 981-1002, McGraw-Hill, 1996.

[MG97] N.A. Moga and M. Gabbouj. "Parallel Image Component Labeling with Watershed Transformation," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 19 No. 5, pages 441-450, 1997.

[MMT95] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. "Models of parallel computation: A survey and synthesis," In Proceedings of the 28th Hawaii International Conference on System Sciences, pages II: 61--70, January 1995.

[MNV94] Y. Mansour, N. Nisan, and U. Vishkin. "Trade-offs between communication throughput and parallel time," In Proceedings of the 26th ACM Symp. on Theory of Computing, pages 372-381, 1994.

[MPCH] MPICH A Portable Implementation of MPI: http://www-unix.mcs.anl.gov/mpi, 2000.

[OA02] A. Osman , H. Ammar. "Dynamic Load Balancing Strategies for Parallel Computers," International Symposium on Parallel and Distributed Computing (ISPDC), Romania, July 2002.

[OA101] A. Osman , H. Ammar, A. Smirnov, S. Shi, and I. Celik. "Domain Decomposition Analysis of Large Eddy Simulations of Ship Wakes," IASTED International Conference, Modeling and Simulation MS'2001, May 2001.

[OA201] A. Osman , H. Ammar, A. Smirnov, S. Shi, and I. Celik. "Scalability Analysis and Domain Decomposition of Large Eddy Simulations of Ship Wakes," ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), June 2001.

[OB99] L. Oliker, R. Biswas. "Parallelization of a Dynamic Unstructured Application using Three Leading Paradigms," Proceedings of Supercomputing conference SC99, Oregon, Portland, November 1999.

[OMNT] OMNeT++, discrete event network simulation tool, http://whale.hit.bme.hu/omnetpp, 2002.

[OP97] S. Orlando and R. Perego. "Scheduling Data-Parallel Computations on Heterogeneous and Time-Shared Environments," Technical Report TR-16/97, Dip. di Mat. Appl. ed Informatica, Universit`a di Venezia, Sept. 1997.

[PFK93] C. Powley, C. Ferguson, and R. E. Korf. "Depth-first heuristic search on a SIMD machine," Artificial Intelligence, 60:199-242, 1993.

[PRR03] Plastino, A., Ribeiro, C. C. and Rodriguez, N. R. "Developing SPMD applications with load balancing," Parallel Computing 29, pp: 743-766, 2003.

[Qui94] M. J. Quinn. "Parallel Computing - Theory and Practice," Mc.Graw Hill, 1994.

[RG+02] Paul E. Rybski, Maria Gini, Dean F. Hougen, Sascha A. Stoeter, and Nikolaos Papanikolopoulos. "A distributed surveillance task using miniature robots," In Maria Gini, Toru Ishida, Cristiano Castel-franchi, and W. Lewis Johnson, editors, Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02), pages 1393-1394. ACM Press, July 2002.

[RS+00] Paul E. Rybski, Sascha A. Stoeter, Michael D. Erickson, Maria Gini, Dean F. Hougen, and Nikolaos P. Papanikolopoulos. "A Team of Robotic Agents for Surveillance," In Carles Sierra, Maria Gini, and Jeffrey S. Rosenschein, editors, Proceedings of the Fourth International Conference on Autonomous Agents, pages 9-16, Barcelona, Catalonia, Spain, June 2000.

[Sak94] Manas Saksena, "Parametric Scheduling in Hard Real-Time Systems," PhD thesis, University of Maryland, College Park, June 1994.

[San96] P. Sanders. "On the efficiency of nearest neighbor load balancing for random loads," In Parcella 96, VII. International Workshop on Parallel Processing by Cellular Automata and Arrays, pages 120-127, Berlin, 1996.

[SCS99] Shi, S., Celik, I., Smirnov, A. "Comparison of different numerical schemes and sub-scale models in large-eddy simulation," Boston, Massachusetts, 1999.

[Smn92] Simon, H.D., "Parallel Computational Fluid Dynamics: Implementations and Results Using Parallel Computers," MIT Press, Cambridge, 1992.

[SP92] Michael Schiebe and Saskia Pferrer. "Real-Time Systems Engineering and Applications," volume 1. Kluwer Academic Publishers, 1992.

[SN93] X. H. Sun and L. Ni. "Scalability Problems and Memory-Bounded Speedup," Journal of Parallel and Distributed Computing, Vol. 19, pages 27-37, September 1993.

[SS01] Shi, Shaoping. "Large-Eddy Simulation of Ship Wakes," Dissertation, West Virginia University, 2001.

[SS94] R. Subramanian and I. Scherson. "An Analysis of Diffusive Load Balancing," Proceedings of Sixth Annual ACM Symposium on Parallel Algorithms and Architectures, pages 220-225, June 1994.

[SS+98] John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. "Deadline Scheduling for Real-Time Systems," Kluwer Academic Publishers, 1998.

[SSC00] Smirnov, A., Shi, S., Celik, I.. "Large Eddy Simulations of Particle-laden Turbulent Wakes Using a Random Flow Generation Technique," In ONR 2000 Free Surface Turbulence and Bubbly Flows Workshop, pages 13.1-13.7, California Institute of Technology, Pasadena, CA, 2000.

[SSC01] Smirnov, A., Shi, S., Celik, I.. "Random Flow Generation Technique for Large Eddy Simulations and Particle-Dynamics Modeling," Trans. ASME, Journal of Fluids Engineering, Vol. 123, pages 359-371, 2001.

[Sub00] K. Subramani. "Duality in the Parametric Polytope and its Applications to a Scheduling Problem," PhD thesis, University of Maryland, College Park, August 2000.

[Sub02] K. Subramani. "A specification framework for real-time scheduling," In W.I. Grosky and F. Plasil, editors, Proceedings of the 29th Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM), volume 2540 of Lecture Notes in Computer Science, pages 195-207. Springer-Verlag, November 2002.

[Sub03] K. Subramani. “An analysis of partially clairvoyant scheduling,” Journal of Mathematical Modeling and Algorithms, 2003. Conference version available in Proceedings of the 8th International Conference on High-Performance Computing (Hi-PC), Lecture Notes in Computer Science, volume 2228, pages 36-46, Springer-Verlag.

[TL99] Michael E. Thomadakis and Jyh-Charn S. Liu. “On the efficient scheduling of non-periodic tasks in hard real-time systems,” In Proceeding of the 20th IEEE Real Time Systems Symp., Phoenix, AZ, December 1999.

[VS90] V. A. Saletore. “A distributive and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks,” Proceedings of the 5th Distributed Memory Conference, pages 995-999, April 1990.

[WA99] Barry Wilkinson and Michael Allen. “Parallel Programming,” Prentice Hall, 1999.

[WLR93] M.H. Willebeek-LeMair and A.P. Reeves. “Strategies for dynamic load balancing on highly parallel computers,” IEEE Trans. on parallel and distributed systems, vol. 4, No. 9, Sept. 1993.

[YYM01] Yang.S.X, Guangfeng Yuan, and Meng M., “Real-time collision-free path planning and tracking control of a non-holonomic mobile robot using a biologically inspired approach,” In Proceedings of Computational Intelligence in Robotics and Automation, pages 113-118. IEEE Computer Society, 2001.

[ZLP96] M. J. Zaki, W. Li and S. Parthasarathy. “Customized dynamic load balancing for a network of workstations,” Proceedings of the 5th IEEE Int. Symp., pp. 282-291, HPDC, 1996.

[ZS92] Zang, Y. and Street, R., “Program to solve 3-D Navier-Stokes Equation on a Composite Grid,” Stanford University, 1992.

Curriculum Vitae

Ashraf M. Osman

Ph.D. Computer Engineering

Lane Department of Computer Science
& Electrical Engineering,
West Virginia University,
Morgantown, WV 26505-6109
Office phone: (304) 293-0405, ext. 2537

Home address: 1200C Van Voorhis Rd.,
Morgantown, WV 26505
Home phone: (304) 599-9062
Email: osman@csee.wvu.edu
http://www.csee.wvu.edu/~osman

Summary of qualifications

- Nine years of experience in academic and industrial environments.
- Ability to communicate and interact with large groups of diverse backgrounds and to work independently or in teams.
- Oriented towards achieving research goals and experienced with both experimental and simulation-based research environments.
- Developed and taught classes to undergraduates, graduates and professionals in class sizes ranging from 2 to 80 persons.
- Developed, deployed and managed several commercial software applications.

Education

Aug. 1999 – Dec. 2003
Ph.D. in Computer Engineering, Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV.
Thesis subject: *Designing a scalable dynamic load-balancing algorithm for pipelined single program multiple data applications on a non-dedicated heterogeneous network of workstations.*
GPA: 4.0/4.0

Sept. 1994 – May 1999
M. Sc. Computer Engineering, Cairo University, Cairo, Egypt.
Thesis: *Adaptive synchronization for real-time multimedia applications.*
GPA: 3.8/4.0

Sept. 1989 – June 1994
B. Sc. Electrical Engineering, Cairo University, Cairo, Egypt.
Graduation project: *Multimedia-based archiving system documenting tourist sites in Cairo.*
GPA: 3.9/4.0

Publications

1. **A. Osman**, H. Ammar. "DLAH: a scalable dynamic load-balancing algorithm for pipelined single program multiple data applications," journal paper to be submitted.
2. **A. Osman**. "Designing a scalable dynamic load-balancing algorithm for pipelined single program multiple data applications on a non-dedicated heterogeneous network of workstations," PhD Dissertation, West Virginia University, December 2003.
3. **A. Osman**, H. Ammar. "Dynamic Load Balancing Strategies," International Symposium on Parallel and Distributed Computing (ISPDC), Iasi, Romania, July 2002.
4. **A. Osman**, H. Ammar, A. Smirnov, S. Shi, and I. Celik. "Scalability Analysis and Domain Decomposition of Large Eddy Simulations of Ship Wakes," ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), June 2001.
5. **A. Osman**, H. Ammar, A. Smirnov, S. Shi, and I. Celik. "Domain Decomposition Analysis of Large Eddy Simulations of Ship Wakes," IASTED International Conference, Modeling and Simulation MS'2001, pp 327:112, May 2001.
6. H. Arafa and H. Ammar, and **A. Osman**, "An Adaptive Dynamic Scheduling Technique for Parallel Loops on Shared Memory Multiprocessor Systems," in Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems (PDCS-2000), Las Vegas, Nevada, August 2000.
7. **A. Osman,** A. Darwish and S. Shaheen, "Adaptive synchronization for real-time multimedia applications," SPIE Proceedings, Multimedia Systems and Applications, Vol. 3528, p. 202-213, Boston, MA, USA, 1999.

Academic Experience

Aug. 2001 – present
Graduate Teaching Assistant,
West Virginia University, Morgantown, WV.
Developed, taught, and graded assignments and exams for graduate and undergraduate courses (Object oriented programming in C++, Introduction to data structures using C++ and standard template library). Developed course web sites at http://www.csee.wvu.edu/~osman/.

Jun 2000 – July 2000
Instructed a course in Embedded Real-time Systems,
The National Energy Technology Laboratory (NETL), Morgantown, WV.
Participated in developing and teaching a course in "Embedded Real-time Systems," including a final interfacing project on an SBC (Single Board Computer). The course was offered to NETL staff.

Aug. 1999 – Aug. 2001
Graduate Research Assistant,
West Virginia University, Morgantown, WV.
Participated in a multi-disciplinary research project focused on: Parallelizing a Computational Fluid Dynamics problem of "Large Eddy Simulations of Ship Wakes" onto a cluster of workstations using C and Fortran. Research was funded by the DoD and monitored by the Office of Naval Research to WVU.

Jan. 1995 – Aug. 1999
Teaching Assistant,
Cairo University – Fayoum Campus, Egypt.

Prepared lectures for discussion sections, graded assignments and administered laboratory sessions for undergraduate courses.
Courses developed and taught:

- Visual Basic programming language: developed course, lab assignments and final exams and project. Taught this course for 80 undergraduate students including labs.
- Fortran Programming language: developed course, assignments and taught lab sessions.

Courses taught:
Logic design, Microprocessors, Matlab, Electrical engineering labs.

Industrial Experience

March 1999 – Aug. 1999
Information Technology Officer
United Nations Education, Science and Culture Organization (UNESCO),
Cairo Office, Egypt.

- Provided technical support in the implementation of the UNESCO programme for Upgrading Science and Engineering Education (USEE).
- Maintained the main web-site for the office, including designing the required databases, reports and web-entry forms found at: http://unesco-cairo.org.
- Assisted in organizing workshops.

June 1994 – Jan. 1999
Systems Engineer,
Cairo Information Technology and Engineering (CITE) company,
Cairo, Egypt.
Job responsibilities included development, deployment, training, and support for computerized payroll system and telephone billing system.

Honors and Activities

Egyptian Student Organization (2003)	President, WVU.
Egyptian Student Organization (2001 – 2002)	Secretary, WVU.
TRUTH student Organization (2001 – 2002)	President, WVU.
Distinction Award (1994)	Faculty of Engineering, Cairo University.
Academic Excellence Award for 4 consecutive years (1990 – 1994)	Faculty of Engineering, Cairo University.

Research Interests

Generally, computer engineering field and specifically: parallel computing, load balancing algorithms, simulation and modeling, performance analysis, software engineering, reliability and fault tolerance, and real-time multimedia traffic.