Dynamic load balancing of parallel road traffic simulation.

Damian Igbe

School of Electronics and Computer Science

Dynamic Load Balancing of Parallel Road Traffic Simulation

Damian Igbe

January 2010

A thesis submitted in partial fulfilment of the
requirements of the University of Westminster
for the degree of Doctor of Philosophy

To my late brother: Sunday Umele Igbe.

Abstract

The objective of this research was to investigate, develop and evaluate dynamic load-balancing strategies for parallel execution of microscopic road traffic simulations. Urban road traffic simulation presents irregular, and dynamically varying distributed computational load for a parallel processor system. The dynamic nature of road traffic simulation systems lead to uneven load distribution during simulation, even for a system that starts off with even load distributions. Load balancing is a potential way of achieving improved performance by reallocating work from highly loaded processors to lightly loaded processors leading to a reduction in the overall computational time. In dynamic load balancing, workloads are adjusted continually or periodically throughout the computation. In this thesis load balancing strategies were evaluated and some load balancing policies developed. A load index and a profitability determination algorithms were developed. These were used to enhance two load balancing algorithms. One of the algorithms exhibits local communications and distributed load evaluation between the neighbour partitions (diffusion algorithm) and the other algorithm exhibits both local and global communications while the decision making is centralized (*MaS* algorithm). The enhanced algorithms were implemented and synthesized with a research parallel traffic simulation. The performance of the research parallel traffic simulator, optimized with the two modified dynamic load balancing strategies were studied.

Acknowledgement

My first and foremost thanks to Professor Stephen Winter, Dr Stephen Ijaha and late Dr Nasser Kalantery for the offer of CPC PhD scholarships. This funding, which was provided through the OSSA project, afforded me the opportunity to register for a PhD programme.

My special thanks and appreciation to my Director of Studies, late Dr Nasser Kalantery for all his guidance, patience, and instructions in the ways of quality research. Unfortunately, Nasser died during my write-up period but I would for ever be grateful to him for his steadfast support in my pursuit of PhD. Also, I am deeply grateful to Professor Steve Winter for his wealth of experience, with which he guided me throughout this long journey and for offering good advices at critical points along the way. My special thanks to Dr. Steve Ijaha, for his valuable contributions, support and encouragement while he was in the supervisory team. Steve had to leave for greater assignments before the completion of my PhD but his support is deeply appreciated. To the three of my supervisors, I would like to say a big thank you from the bottom of my heart.

My informal advisors and mentors from the start of my career till date are invaluable and deeply appreciated. The list includes Mr Yemi Raymond, Mr Chris Emejuru, Mr Lanre Ajayi, Dr. Thierry Delaitre and Mr David Garcia.

Many of my friends have contributed in significant ways to the completion of my PhD. Their moral support and encouragement add significantly to the quality of my life. These include Hilary Ode, Jonathan Obele, Emmanuel Mendy, Tony

Okeze, Bolaji Coker, Taiwo Adekaunsi, Marta Kozwoska etc. But guys, I was fed up with the question of 'when are you finishing? -). Also, the support and help of my friends and colleagues in the university, Haifeng Xie and Noam Weingertein are really appreciated.

Finally, let me express special thanks to my family for their help and support in several ways. I am indebted to my wife, whom I married in 2008, for standing by me and for her help in editing the thesis. I am indebted to my sister Victoria Okponya, who laid the foundation for my studies. Reserving the best for the last, my gratitude to Almighty God for His guidance and protection, for the gift of dedication and perseverance, for always paving ways and bringing people to assist me, whenever help is needed, in the journey of my earthly life. To Him is the final Glory!

London, January 2010.

Damian Igbe.

Contents

List of Figures

List of Tables

Abbreviations

DLB	Dynamic load balancing
MaS	Master-Slave algorithm
MB-noDLB	Parallel simulation with manually balanced network and no DLB
Sim-noDLB	Parallel simulation with uneven network and no DLB
Sim-DLB	Parallel simulation with uneven network and DLB
ILD	Initial load distribution
ILE	Initial load evenness
FLD	Final load distribution
FLE	Final load evenness
LDAD	Load distribution after DLB
LEAD	Load evenness after DLB
UoW	University of Westminster
CPC	Center for Parallel Computing
BCA	Basic Centralized Algorithm
ECA	Enhanced Centralized Algorithm
BDA	Basic Diffusion Algorithm
PD	Profitability Determination
EDAG	Enhanced Diffusion Algorithm with Global PD
RTLI	Response Time Load Index
VLI	Vehicles Load Index

Chapter 1

Introduction

1.1. Research Aim and Motivations

The efficient use of distributed memory parallel computing systems require the computational load to be evenly distributed across the computers. However, for some classes of applications (e.g. parallel road traffic simulation, molecular dynamic simulation, etc) the load distribution on different compute-nodes often vary over a long simulation run, leading to a system that is unevenly distributed. The load imbalance is caused by the dynamic nature of the applications as simulation objects (e.g. vehicles in traffic simulation) move between the network partitions and hence across the computers. In such a dynamic situation, even for an initially balanced system, the system becomes unbalanced after some time.

Load imbalance leads to poor system performance because it creates a scenario whereby some compute-nodes are relatively idle, wasting processing cycles, while other compute-nodes remain extremely busy with computations. It would be ideal if the busy processors could share out their workload with the idle processors in order to improve the run-time of the application and hence the efficiency of the system.

Therefore, in applications where load density varies across the system, load balancing may be essential for good performance [117][16][106]. Load balancing aims to balance the load between the compute-nodes and keep them equally busy. This allows compute-nodes to finish computation at approximately the same time. Algorithms for analyzing and calculating the amount of load to be redistributed are called dynamic load balancing algorithms. Early research in load balancing started as far back as 1970 and therefore considerable literatures exists. However with the evolving nature of the distributed systems platforms, some of the research issues already addressed are being readdressed. For example, in recent times, distributed computing platform has extended from the cluster to the Grid. This is an advantage to researchers as the Grid means more computational power for complex applications. However, the heterogenous nature of the Grid for example(as against the previously dominant homogenous distributed clusters) presents challenging research in areas such as the *load index* of dynamic load balancing. Load index is critical because if not accurately quantified, load balancing can not be effective. In a heterogenous environment where compute-nodes vary in CPU power, memory, network backbone, etc, load index must be able to measure these parameters effectively.

The characteristics of any application under considerations also present new design challenges that may lead to further research questions. It means that some existing algorithms can be customized and re-used in some applications, while for others, new designs may be needed. In this regard, studying and understanding the application to be optimized is the first step in selecting an algorithm to be customized.

In [94] the authors said that until now research and development in dynamic load balancing area have been focused on the identification and evaluation of efficient policies for load information distribution, job transfers, and migration decision-making. The authors also said that, current and future research emphasis will be on the development of efficient policies for load information measurement and distribution and placement decision-making, because these areas cause most of the overhead of dynamic load balancing. The authors also reported that profitability analysis is an important area of research because although dynamic load balancing incurs overhead due to task transfer operations, a task transfer should not take place unless the benefits of the relocation outweigh its overhead.

In view of the foregoing, this thesis concentrated mainly on the following areas:

- design of a load index algorithm and a profitability determination algorithm suitable for parallel traffic simulations and
- design of a centralized load balancing algorithm with global and local communication characteristics, optimized with the load index and profitability determination algorithms that are designed in this thesis.

It is observed that the design or the implementation of any load balancing algorithm is application specific. Since the research in this thesis spans two main fields of parallel traffic simulations and dynamic load balancing, two different approaches can be adopted to the research:

1. Design a general purpose load balancing algorithm and use the parallel traffic simulation to test the performance of the algorithm or
2. Study the application to be optimized (i.e. parallel traffic simulation) and based on its properties, design a load balancing algorithm for its optimization.

This research adopted the second method. All the research issues were addressed with reference to traffic simulations as this is the application under investigation. While the differences between the 2 approaches may be subtle, it is important to understand the approach adopted in this research. In other words, while the algorithms designed in this research may be suitable to other classes of applications, they were designed primarily with traffic simulation in mind.

1.2. Introduction to Road Traffic Modelling and Simulation

A simulation model is a representation of a physical system in terms of a set of states and events [66][39][22]. Performing a simulation is to mimic the occurrence of events as they evolve in time and recording their effects on the represented system state. Computer modelling and simulation is used in many scientific fields of study (e.g. particle physics, molecular dynamics [5], quantum chemistry, meteorology, fluid dynamics, etc). In transportation engineering, urban traffic analysis is a problem whose complexity requires the use of tools based on computer simulations.

Urban transportation system presents many important challenges that have large impacts on modern societies. Some of these problems are issues concerning safety, delays due to congestion, pollution from vehicle emissions, and vulnerability to human error and vehicle accidents. Traffic simulation offers the potential to make significant contributions in alleviating these problems thereby making urban road network more safe, efficient, and environmentally friendly. Traffic simulations are used in *research, planning, training,* and *development of traffic systems* [19]. Some of the advantages [67] of traffic simulation are enumerated below:

1. *Research* - Traffic simulation and modelling offers the ability to study the feasibility of systems and the benefits expected from their operation by validating the proposed system, assessing their expected impacts and providing the basis for sound cost benefit analysis. It also offers the ability to study and predict unanticipated future events and traffic flow.

2. *Traffic designs* - Traffic simulations are used in various design stages:

    - *Design analysis* - Design procedures are iterative and the designer is often faced with frequent design refinement. Design process is simplified by use of simulations whereby the simulation model is used to provide detailed statistics that can form the basis for identifying design flaws and limitations.
    - *Alternate designs* - Traffic simulation is used to evaluate the suitability and correctness of different designs. By asking the '*what-if* ' questions, traffic engineers can design a range of experimental scenarios on different models, explore the conditions, before applying them to real life situations.
    - *Testing new designs* - Once a design has been approved, traffic simulation can be further applied to test the performance of different geometric designs before committing resources to construction. In this way, failures and hence overall costs of constructions can be reduced.

3. Training - Simulation is used as a real-time laboratory to train operators of traffic management centres. The simulation model, after being interfaced with real-time traffic control equipments and telematics, acts as an environment for real world surveillance, communication and traffic equipments.

4. Planning road safety - Simulation models can be used to recreate accident scenarios. This provides indispensable tools to build safer vehicles and roadways.

Traffic simulation models can be classified as continuous or discrete simulations. Continuous simulation models the state changes of the system as it progresses over time while discrete simulation models changes only at specific points in time. Traffic simulations are typically dynamic in nature due to the continuous movement of vehicles and other traffic elements in the simulation. However urban transportation systems are modelled as discrete simulations by asserting that their states change only at certain points in time. In this way, it is possible to simulate traffic using computer systems, which are discrete in nature. Generally, systems that change states continually such as the movement of vehicles and pedestrians in a simulation are better modelled in discrete time [67] if it is desired to capture their activities on a second by second basis.

Discrete simulation models can be generally classified as: Discrete time and discrete event simulations [100]. In discrete time simulation, time is segmented or discretized into a succession of time intervals and within each time interval, the simulation model computes the activities, which changes the states of selected system elements. Discrete event simulation models systems that are idle most of the time, or that changes state only at certain periods of simulation. For example, the instantaneous changes in the states of traffic lights (green, yellow, and red) can be modelled as discrete events according to the colours of the traffic light. In this way, a considerable time is saved by recording only the state changes rather then modelling the signal at each simulation time (as done in discrete time simulation). Discrete time simulations usually involve more detailed specifications and are therefore more expensive to simulate than discrete event systems.

There are three main approaches to traffic simulation models, classified according to the level of details, which represent the system being studied. These are referred to as *Macroscopic, Microscopic and Mesoscopic* [14][93][87].

In macroscopic method of traffic simulation, all traffic information is aggregated into traffic streams with no distinction between the individual entities in the simulation. For example traffic flow may be represented by some scalar values of flow rate, flow density and flow speed [15]. The macroscopic approach is helpful in predicting the traffic flow and analysing the traffic density and is easier and less costly to develop, execute and maintain. It is mostly useful for analysing a large section of simulation. The disadvantage is that they lack the detailed calibration of real life situations. They are mostly appropriate if:

- The desired results are not subject to microscopic details
- The available model development time and resources are limited
- And the scale of the application cannot accommodate the higher execution time of the microscopic model

By contrast, in microscopic simulation, the aggregate flow emerges from the interaction of individual entities. While macroscopic simulations describe simulation entities and their activities and interactions at a lower level of details, microscopic models describe both the system objects and their interactions to higher details. Microscopic traffic models demand that the individual entities such as cars, driver behaviour, roads, traffic junctions, traffic lights etc. are considered separately for further analysis. The basic components of a microscopic traffic simulation models are:

- A comprehensive representation of the road network topology and a detailed and explicit reproduction of traffic control plans
- A detailed modelling of individual vehicle/driver's behaviour
- And the interaction between the different simulation elements

Microscopic models are useful tools in modelling traffic interactions to greater details. They are useful in detecting, analysing and understanding a large range of traffic problems. These make them an ideal choice for traffic planners, traffic engineers and consultants. With the detailed model of a microscopic traffic simulator, the traffic professionals can analyse a section of road network to greater details and be able to deduce reasonable conclusions. Microscopic traffic simulations include *Paramics [15], Transims [78], OSSA [28][29], Hipertrans [56] [55] and Aimsums [1].* Similarly, Madcity, the research traffic simulator is a microscopic discrete time simulation [53] [54].

The main disadvantages of microscopic models are that they are costly to develop, execute and maintain, when compared to macroscopic models. While these models possess the potential to be more accurate than the macroscopic models, the potential may not always be realized due to the complexity of their logic and the larger number of parameters that need to be calibrated. Hence, large-scale microscopic models lead to simulators that are slow when executed on single compute-node systems due to the high level of modelling details involved. A way of speeding up execution is to run the simulation on a parallel processing platform comprising more than one compute-node.

In the last decade, several research efforts have been investigating the integration of the good aspects of microscopic and macroscopic traffic models into a single model called mesoscopic simulation model. In this model, the traffic entities are defined at an abstract level of detail (macroscopic), but their behaviour and interactions are described at a greater level of detail (microscopic). For example, vehicles are grouped into packets and these are simulated through the road network. The packet of vehicles act as one entity and its speed on each road is derived from a speed-density at the moment of entry. The number of vehicles per kilometer per lane defines the density of a road. A speed-density function relates the speed of vehicles on the road to the density. If there is a lot of traffic on the road, the density is high and the speed-density function will give a low speed to the vehicles, whereas a low density will result in high speeds.

1.3. Dynamic Load Balancing (DLB) in Parallel Traffic Simulations

As stated earlier, this research aims to optimize traffic simulation so it is the starting point in the review. The review is done with the aim of finding research problems in dynamic load balancing of parallel traffic simulation. Following the review is the critical analysis of the dynamic load balancing algorithms used. The problems identified in the review are further explored in dynamic load balancing research with a view to finding the solutions.

Transims traffic simulation implemented a Local Decision Local Migration (LDLM) method [73] of load balancing in [88]. This is similar to diffusive method of load balancing. In Transims, since the performance of a subnetwork only weakly depends on the *number of vehicles* in the grid, a value proportional to the number of grids handled by a segment was used as a measure of its computational load. To estimate the load, each compute-node used wall-clock timers to monitor the time needed to execute one time-step of the local subnetwork. The compute-node keeps track of the time spent on different tasks such as: execution time, idle time, graphics time, boundary time and work time [88]. In addition, each compute-node computes the estimated load of the residing sub-net l and a performance p. These weights are computed by simply summing over the actual road lengths.

At the same time a compute-node receives equivalent information from its neighbors in intervals of t_{lb} time-steps, which is used to compute a performance value $p_i = l_i/t_i$. Processor i stores those values for a certain period of time $t_{monitor}$, from which it retrieves the minimum ${p_i}^{min}$ as its safe performance value. A safe workload for this processor is thus $L_i = l_i/{p_i}^{min}$, which has the unit of time period and is used to determine the load differences with respect to its neighbors.

Tibaut Andrej et al. implemented a parallel traffic simulation with an algorithm for adaptive load balancing [98]. Load balancing is achieved by static distribution or dynamic redistribution of simulation load. Vehicles are the simulation loads used here. The algorithm checks the number of vehicles on each participating compute-node and finds out the compute-nodes with the maximum, and the minimum number of vehicles. This information tells the algorithm about the source (maximum number of vehicles) and the destination (minimum number of vehicles) of load relocation.

The algorithm is a master-slave global algorithm in which a compute-node designated as the decision-maker takes the decisions. The decision-maker system receives regular updates from the slaves to inform it of the states of the systems. Based on this information, it identifies the most loaded compute-node and the least loaded compute-node. If the load difference between these two computers diverge much from the calculated average value, the system is considered unbalanced. Based on this information, it initiates the calculation to relocate the right amount of load between the two in order to balance the load.

Cynthia B. Lee et al. presented a parallel traffic simulator which is also optimized with a global load balancing algorithm in [27]. The load on the processor was defined as the number of cars it is responsible for. The authors claim that using a simple load heuristic such as the number of cars is sufficient, especially in a homogenous computing platform such as the cluster. The authors selected a global method of algorithm as against a distributed approach (e.g. diffusion) for the reason that, for a very uneven data, it may take longer to reach a satisfactory distribution of the load if a distributed approach is used. The performance of the parallel simulator, enhanced with global dynamic load balancing was analysed and its performance compared with the simulator without load balancing. In one experiment the performance impact of load balancing on a balanced load was reported. In this experiment, it was reported that the system with the load balancing performed worse that the system without load balancing. It said that this is expected because in this case, load balancing presents an additional overhead for a system that was balanced. Secondly, it presented results on 3 compute-nodes where the load was unbalanced. The paper reported that on 2 compute-nodes, there was no performance gain as a result of load balancing. On 3 compute-nodes however, the system with load balancing performed worse that the system without load balancing. This shows that as the number of compute-nodes increases, the load balancing algorithm shows rather poor performance than the system without load balancing. This obviously shows that the algorithm is non-scalable.

In [25] Antonio Coradi et al. presented an approach adopting a ‘parallel objects' environment to let users drive object allocation in parallel/distributed architecture. A set of high level directives permits users to specify the allocation needs of application objects. This approach was tested with a parallel simulator. Again, the number of vehicles was used as a measure of load. The experiments were conducted with about 500 vehicles to compare different levels of load balancing in different levels of user directives. In each experiment, load balancing was performed. In addition, the experiments ranges from the case of no user intervention to cases where the user has influence on specific parameters such as vehicle and street. Generally, it was concluded that the tool notably improved the performance, even without any user-level directive. The paper reports that the allocation policy pursues a load balancing goal adopting a local *diffusive algorithm*: load information is exchanged within a neighborhood and allocation decisions are limited within the neighborhood. Load balancing is triggered either by the creation of a new object /activity or by changes in the system state (i.e. load level changes).

1.4. Critical Analysis and Conclusions of DLB in Parallel Traffic Simulations

All the parallel traffic simulators reviewed above employed dynamic load balancing in order to optimize the application. However, they all employed different methods of load balancing. Two of the reviewed simulators employed centralized methods [27] [98] while the other two employed distributed methods [88] [25] of load balancing.

Centralized algorithms are simple to implement and have the advantage of making better load balancing decisions because they have a good view of the entire system. However, centralized algorithms often suffer scalability problem. This is because as the number of compute-nodes increases, the communication overhead increases linearly as the compute-nodes all try to communicate with the central decision making compute-node. They also suffer from single point of failure which means that if the decision making compute-node goes down, the whole algorithm collapses.

Distributed algorithms on the other hand make poor load balancing decisions since each compute-node is only aware of its neighbours and hence make decisions based on local information. They also suffer what is called *termination detection*– this is the situation in which it is difficult to reach global balanced decision since there is no process overseeing the overall activities of the algorithm. However, communication overhead is less compared to the centralized algorithms and since communication overhead presents serious performance bottleneck, distributed algorithms are a favourite of many applications. It is interesting to note that since these simulators did not report on their performances, there are no experimental comparisons to support the theoretical advantages and disadvantages of both the centralized and distributed algorithms with respect to traffic simulators. The discussions in this respect are based on the general perceptions about the different classes of these algorithms as reported in load balancing literatures. This thesis, by comparing distributed and centralized algorithm, will present some results in this regard.

Three of the algorithms [27] [98][25] used *vehicles* as load index. A simple load index such as this is efficient since the overhead is less compared to when a composite load index is used. However, it also has a disadvantage because it implies that none of the systems takes into consideration the underlying properties of the hardware platform used for the simulations. While this may be suitable for homogeneous platforms, it is not suitable for heterogenous platforms where the hardware characteristics of the compute-nodes differ considerably. That is, using vehicles as load index is not adaptive to the hardware platform and therefore not an accurate measure of load. This is a major shortcoming that this research aims to address. A load index is designed to consider the properties of the underlying hardware resources in addition to the load of the application. It is aimed at heterogenous clusters and the grid environment. The following are the conclusions of the summary:

- The load index used by most of the algorithms is the number of vehicles in the simulation. These are not adaptive to hardware platforms and therefore especially not suitable to a heterogenous platform such as the grid.
- None of the algorithms employs profitability determination. However, this is a critical part of a load balancing algorithm. Since every invocation of load balancing algorithm incurs a non-negligible overhead, it is better to minimize this process by not completing all the stages except it is profitable to do so. Profitability determination ensures that simulation proceeds to carry out the remaining stages of load balancing *if and only if* it is profitable to do so.
- The algorithms use different methods of load calculation. In some algorithms, this is centralized in one compute-node (centralized algorithms) while in some other algorithms, this is distributed among the participating compute-nodes (distributed algorithms). This implies that the algorithms also use different communication patterns. The centralized algorithms use a mix of both global and local communications while the distributed algorithms mainly use local communications. However, none of the algorithms were described in detail and also, there is no indication of which of these methods give better performance in practice.

From the discussions above, there are no specific requirements for load balancing of parallel road traffic simulations that are different from other applications. However, in domain decomposition method of traffic simulations, this research observes that a global (direct) method of load relocation between the most loaded (maximum) and the least loaded (minimum) compute-nodes as implemented in [98] may not be the best method. This is because in this way, the boundary partitions are likely to increase which in turn increases the communication overhead. This research adopted a local method of load migration in which load '*diffuses*' from the most loaded compute-node to the least loaded compute-node through its neighbours. In this way the communication boundaries are preserved and this is capable of increasing the performance of the system by decreasing the communication overhead. Global and local methods of load relocation are explained in detail in section 3.10.3.

1.5. Outline of the Thesis

This thesis is organized into five chapters.

Chapter 1: Introduction
Chapter one starts with research aims and motivations. It also discusses the research background with introduction to road traffic modelling and a review of dynamic load balancing in parallel traffic research. The chapter concludes with a critical analysis and conclusions of DLB in parallel traffic simulations.

Chapter 2: Literature Review of Load Balancing Research
Chapter two discusses the dynamic load balancing problem. It discusses the taxonomy and literature review of the main contributions to knowledge–literature review of dynamic load balancing, load index and profitability determination.

Chapter 3: Design and Implementation of Dynamic Load Balancing Algorithms
Chapter three discusses the research framework and the design considerations necessary in any parallel simulation application. This was followed by the design of a centralized load balancing algorithm called (*MaS*), a design of a load index and a design of profitability determination. This chapter also discusses the parallelization of the research traffic simulator using the concept of Lane Cut Points (LCP). The implementation of dynamic load balancing algorithms were also presented.

Chapter 4: Experimental Results and Discussions
Chapter four presents the experimental results obtained from the experiments

and the discussion of the results. It shows that load evenness is a factor affecting performance. It studied the behaviour of the new designs and compared with existing strategies. It also studied the overhead, quality of load balance and scalability of the algorithms and in all cases, comparing the performances of the two algorithms.

Chapter 5: Contributions to knowledge and Final Remarks

Finally, chapter five discusses the contributions to knowledge, conclusions and possible future direction for this research project.

Chapter 2

The Load Balancing Problem and Literature Review

2.1. Chapter Overview

In a parallel processing platform consisting of multiple processes, each process is responsible for the computation of one or more operations as defined by the parallel application. A task is defined as the smallest unit of concurrency performed by the parallel program while a process is defined as an abstract software entity executing its assigned tasks on a compute-node [115].

Generally, there are 2 basic steps to programming a parallel computer. The first stage is decomposing the problem into tasks and the second is the subsequent assignment of the tasks to processes. The first stage is often called partitioning while the second stage is about mapping the processes to compute-nodes that are available to the computation. The two steps have similar optimization objectives, that is, to balance the load and minimize communication between the processes/compute-nodes [115]. The above 2 stages of *partitioning* and *mapping* for the purpose of achieving their respective optimization objectives constitutes the *Load balancing problem*. In this thesis, load balancing is discussed as an application optimization problem.

2.2. Taxonomy of Dynamic Load Balancing

There are many classifications of load balancing algorithms and therefore so many taxonomies exists [17][85]. The taxonomy adopted in this thesis is presented in Fig. 2.1 and its discussions is taken from [85] and presented below. The Madcity traffic simulator is an SPMD parallel application, therefore, this taxonomy is chosen because it is tailored for SPMD applications. Also, the discussions of its constituent components addresses the design issues in this thesis.

2.2.1. Time of Distribution

The criterion of partitioning data between participating compute-nodes and the instant of time in which this partitioning takes place is defined by the *distribution policy (DP)* of a load balancing algorithm for SPMD applications. Based on this definition, the load balancing policy is said to be static if data is partitioned before any task is executed and dynamic if the distribution of tasks to compute-nodes occurs along the execution of the application, or if the initial distribution of tasks can be dynamically modified. [40][111][119] presents many examples of dynamic load balancing policies.

In static load balancing, workloads are distributed to compute-nodes at compile time, the algorithms being based on the knowledge of the system predicted by models. Static load balancing algorithms attempt to predict the program execution behaviour at compile time, partition the tasks into smaller

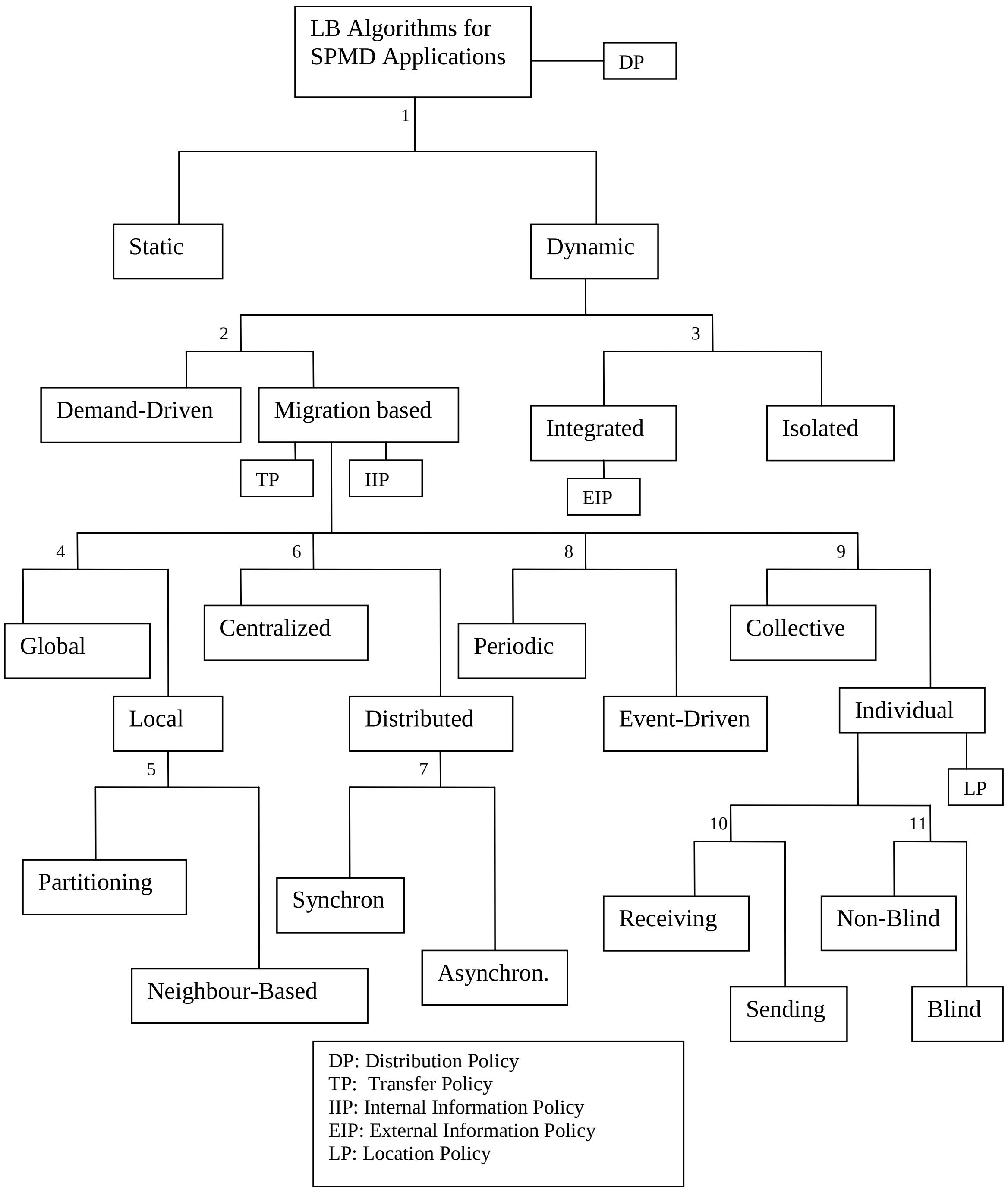


Figure 2.1: Taxonomy of load balancing algorithms for SPMD applications

subtasks (with the aim of reducing communication delays), and then allocate the subtasks to the compute-nodes. The major advantage of static load balancing is that the overhead of the scheduling process is incurred at compile time, resulting in a reduction in the overall execution time, compared to the dynamic load balancing algorithms in which the overhead is incurred at run-time [94].

Optimal solutions to static load balancing are generally NP-hard [7][8]. However, under certain assumptions about processes' behaviour and/or characteristics of the system, there exists some theoretical results on optimal static assignments [9][23][80][81][89]. Heuristics approaches, which searches for good solutions, simply by some rule of thumb [10][21][71] are a common practice because they are simple and fast. Many approximate and heuristic approaches have been proposed [44]. Examples of heuristics are:

- Minimize communication by finding dependent processes and assigning them to the same compute-node.
- Minimize execution time by assigning equal load distributions to computing compute-nodes.
- Give higher priority to tasks that may be bottlenecks in the execution of an application.

Two important, general-purpose heuristics are simulated annealing [62], and variable depth local search originally developed by Kernighan and Lin (K-L) [60][61]. Some methods specific to the mapping of unstructured meshes include recursive coordinate bisection (RCB), and recursive spectral bisection (RSB)[2]. For the partitioning of generic (random) graphs, the 'best' heuristics are simulated annealing and K-L. However, for unstructured meshes, K-L is substantially better than simulated annealing, and is also much faster [75]. Static partitioning can be often challenging. To ease the challenges of manual partitioning, software tools have been developed. These tools are referred to as graph partitioners. The algorithms use a graph model of the computation, while applying graph partitioning techniques to divide the graph among the processors. Examples of graph partitioner are ParMETIS [91][13] and JOSTLE [101][102].

The major problem with the graph partitioners is that they are general purpose in design and therefore may not work ‘out-of-the-box’ and customizing them may require so much effort as it involves delving into the code to know the software interfaces. For some researchers, a better method is to design routines specific to the application under study. This also has the advantage of being faster than the general purpose ones as the data structures and other software details are tied to the specific application.

The work on static load balancing generated considerable interest in load balancing research, but mainly suffered the following drawbacks [121]:

- Because static balancing is done at a preprocessing stage, static algorithms cannot respond to short-term load fluctuations that may occur at run-time. As a result, the performance improvement potential of load balancing is not fully realized.
- Static algorithms often assume two much job information to be implemented effectively. Even when the information is available, intensive computation may be involved in obtaining the optimum schedule.
- Most static partitioning tools are too expensive or difficult to parallelize and have no notion of incrementality [47]. These make them generally unsuitable for adaptation to dynamic load balancing algorithms.
- Most static algorithms are specific to applications and not suitable to all classes of applications

These disadvantages led to the research on dynamic load balancing, in which the current system load is considered in determining job relocations. Because the static method attempts to correct the load imbalance only once, it is not suitable to a dynamically varying computational load with unpredictable behaviour. To improve the performance of such applications require the redistribution of work in a time-varying, dynamic nature. Examples of such applications are found in computational mechanics (such as particle simulations) and also in the field of road traffic simulations. For example, the dynamic and unpredictable vehicle behaviour in parallel simulation of road traffic. A road traffic simulation program simulates the dynamic movement of vehicles and its interactions with the simulation objects such as traffic lights, pedestrians, etc for a period of time. As vehicles move between different network partitions, the computational requirements of the processes may change from time to time. Since the processes need to be synchronized at the end of each simulation step, an imbalanced workload distribution will cause a severe penalty for some processes within the step. To improve parallel efficiency, processes' workloads have to be redistributed periodically at run-time [115].

Similarly, particle simulation calculates the interactions among the atoms for a period of time. At each time step, the simulation calculates the forces between atoms, calculates the energy of the whole structure and also the movements of atoms. Assuming that each process of the program is responsible for simulating a portion of the system domain, atoms tend to move around the system domain, thereby changing the computational requirements of the processes.

In general, dynamic algorithms are significantly more complex to implement than static ones. While static partitioning tools often run in sequential pre-processing step, dynamic load balancing must be performed on the parallel platform. The time and memory consumed by the partitioning algorithm takes time away from the simulation. The system also requires additional code to determine when it is advantageous to redistribute the data so that its performance can be monitored. Furthermore, when data is moved between the compute-nodes, data structures must be reconstructed. All these issues add additional complexity to the parallel application code and hence more sophisticated than the sequential code [47].

Despite the run-time overhead of dynamic load balancing algorithms they have the potential to outperform static algorithms. Dynamic load balancing algorithms have been proven beneficial in many parallel and distributed systems. By aiming to equalize the workload among compute-nodes and minimize inter processor communication costs, dynamic load balancing can significantly improve the performance of a distributed system.

As the overheads of dynamic load balancing algorithms at run-time are non negligible, in practice, it is not always profitable to aim at a global balanced state. Sometimes, it pays to aim a little lower from the perfectly balanced state by relaxing the requirements of load balancing to various degrees. The allowable degrees can be set as a threshold and compared when making load balancing decisions. In such situations there is a certain tradeoff between balancing quality and run-time overhead. Some applications are only suitable for static load balancing. However, it is also common for both static and dynamic load balancing to be used in an application.

2.2.2. Distribution Policy

Dynamic load balancing algorithms are also classified based on the task distribution strategy:- *demand driven and migration based.* In demand-driven policies, after the initial distribution of some of the existing tasks between the participating compute-nodes, subsequent distribution of tasks are handled by a decision-making node, based on the behavior of the compute-nodes [41][84][85]. The decision-making compute-node allocates an initial set of tasks to each participating compute-node. As the program proceeds in execution, each compute-node requests and receives a new set of task when it finishes executing its current tasks. The set of tasks included in each request is handled by the *distribution policy* in demand-driven algorithm. The decision making compute-node can become a bottleneck especially if the rate at which tasks are requested from it becomes high. This high request usually increases as the number of participating compute-nodes increases. As a result, a solution for this scalability problem based on the use of hierarchy of coordinators is proposed in [84].

In migration-based algorithms, after the initial distribution of all the existing tasks between the participating compute-nodes, the algorithm strives to keep the load balanced by transferring tasks between the compute-nodes during execution. *Transfer policy* and *internal information policy* are always part of the migration-based algorithms. The *transfer policy* defines how much tasks are to be transferred among which compute-nodes and also the different criteria in defining the tasks migrations. Similarly, the *internal information policy (IIP)* defines the internal load index that estimates the remaining of the workload generated by the application at each compute-node. The IIP also defines the '*how*' and '*how often*' this index is measured and its value communicated to other compute-nodes [85].

The major difference between *demand-driven* and *migration-based* algorithms is that in *migration-based*, *all* the tasks are distributed at first and then the algorithm strives to balance the system as execution proceeds. In *demand-driven* *some* of the tasks are distributed at first and then as execution proceeds, more and more tasks are given to the compute-nodes on request. Also, *demand-driven* always requires a central coordinator for its execution while *migration-based* does not necessarily need a central coordinator as load migration can also be between neighbour compute-nodes as in the case of distributed algorithms.

2.2.3. External Load Indexes (EIP)

Parallel applications execute concurrently with other parallel and sequential applications in a wide range of computational environments. This leads to a variation of the availabilities of resources at different participating compute-nodes at different time instants. Load balancing policies for such environments must deal with this dynamic heterogeneity [85]. As a result of this, many dynamic load balancing policies use an *external load index*, which estimates the workload generated by the competing processes. These are called *integrated policies* in the taxonomy used here, because the policies take into consideration the SPMD application in a multiprocessing environment. *Integrated algorithms* use an *external information policy* which, in contrast to the *internal information policy*, defines the external load index as well as the method and frequency of its measurements.

Isolated policies are those algorithms that external load indexes are not inclusive in their load balancing decisions. These policies are appropriate for dedicated environments where the only executing processes are those generated by the parallel application itself [36][84][111]. There are however a few exceptions, for example, the isolated policy described in [84] is demand-driven and the external load is indirectly taken into consideration, since the rate of requests for tasks at each compute-node reflects its load.

2.2.4. The Scope of Load Balancing

In load balancing operation, the transfer policy of a *migration-based* policy defines the possible transfer sources and destination routes of tasks. Tasks transfers may take place between any two participating compute-nodes in *global policies* [111][118][40][57] whereas *local policies* usually define groups of compute-nodes, and then allow task transfers only between two compute-nodes within the same group [111][118][40].

Typically, the information policy will reflect the scope of the policy. In global strategies, each compute-node may have load information about every other compute-node, whereas in local strategies a compute-node only need to have information about other compute-nodes in its group. The scope of information is reflected in communication costs–for global policies, the cost of propagating load information across the entire system is typically high. The advantage however is that the availability of global information allows load balancing decisions to be more precise. The reverse are true for local algorithms.

2.2.5. Local Algorithms and Group definitions

In local algorithms, groups are used to classify compute-nodes. Each compute-node may belong to only one group [118] and a groups comprises a partition of the set of participating compute-nodes. Since transfers occur only within a group of compute-nodes, if no other scheme is used to allow load to be exchanged between groups, participating schemes may result in permanent unbalance.

In neighborhood-based algorithms [40][111], groups are defined by physical or logical topologies. Basically, each compute-node may exchange tasks with the compute-nodes within its neighbourhood, which may be defined by the available physical communication paths, or by the logic of the application. In this case, a compute-node may belong to more than one group, thus allowing load to migrate between groups. However, several executions of a neighborhood-based load balancing algorithm may be necessary to reach a state of global load balance, while a global strategy may allow such a state to be reached in a single execution.

2.2.6. The Structure of the Algorithm

In this taxonomy, *Migration-based* algorithms are also classified according to the location where the algorithm itself is executed. If all compute-nodes take part in the load balancing decisions and there is no central decision making compute-node, the algorithm is said to be distributed. In this case, the load balancing algorithm is itself an SPMD application, with its code replicated at all compute-nodes. [40][111][119] present examples of distributed algorithms.

Distributed Algorithms

Distributed algorithms are sometimes called the nearest-neighbour dynamic load balancing algorithms [116][32]. At each operation, they are concerned with the direction of workload migration between the nearest adjacent neighbours. By exchanging an appropriate amount of workload with the neighbours, the compute-nodes strive to enter a more balanced state. In other words, a process compares its workload with all of its neighbours and then gives away or takes in a certain amount of workload with respect to the load of each nearest neighbours. Distributed algorithms are usually in two steps; firstly, the compute-nodes decide how much load to be moved between the neighbour compute-nodes to achieve load balance and secondly, the compute-nodes select the objects required to meet the requirement in the first step.

In distributed load balancing algorithms, each system is autonomous in its load evaluation decision. This results in faster decision-making process. Also, because communication is performed within a small set of compute-nodes, distributed methods scale well with increased number of compute-nodes. Distributed methods are incremental by design, as they move objects only within a small set of compute-nodes. Since the total computational time is determined by the time required by the most loaded compute-node, a small number of iterations is usually required to reduce imbalance to an acceptable level. When global balance is required, however, many iterations of distributed method may be required to spread load from a few heavily loaded compute-nodes to the other less-loaded compute-nodes. The convergence rate to global balance is determined by the particular local algorithm used to compute the amount of work to migrate [47].

A review of major distributed algorithms is presented below:

Diffusion Algorithm

A well researched class of distributed algorithms to determine the flow of work between processors is known as diffusion. In diffusion [24][74][6][95] a process balances its workload with all of its nearest neighbours simultaneously in a load balancing operation. Cybenko [26] was the first to formally propose these class of algorithms. However, the problem was being studied by several researchers from different points of view [6][12][45][49][68] [77][114][115].

The performance of diffusion was explored in [111] and found to be superior to other distributed load balancing algorithms such as dimension exchange and gradient model. Like any other distributed algorithm, a critical issue with diffusion algorithm is its rate of convergence. Several methods have been proposed to accelerate the convergence of diffusion methods. For example, Cybenko first studied the condition of convergence of diffusion method in [26]. Parallel multilevel techniques have been used in [49][91][101][102]. Wheat et al. [110] proposed using a second-order implicit finite discretization to compute work transfers. This scheme converges to global balance in less iteration, but requires a bit more work and communication per iteration. The method of Hu and Blake [51][50] is used in several parallel decomposition packages [91][102]. Similar studies performed by Boillat [95] concluded that diffusion can converge to an equilibrium state over a number of iterations.

Gradient Load Balancing Algorithm

Gradient load balancing algorithms are explored extensively in the literature [68][77][68]. They employ a gradient map of the proximities of under-loaded compute-nodes in the system to guide the migration of tasks between overloaded and under loaded compute-nodes. The basic idea is that the under loaded compute-nodes inform other compute-nodes of their state while the overloaded compute-nodes respond by sending a portion of their load to the nearest lightly loaded compute-node in the system.

In the design of this method, a computer can exist in either of three states: *lightly loaded, averagely loaded or heavily loaded.* A computer, by comparing its load to predetermined thresholds, of "low" and "high" water marks, inform their neighbours of their status. This information is propagated to overloaded computers within a fixed radius (typically the dimensions of the network). A gradient map is then constructed to route work from overloaded to underloaded compute-nodes. A major problem with this method is the possibility of transferring too much work from the overloaded to the underloaded compute-nodes. The authors of [77] attempted to solve this problem by doing a direct transfer of work between the overloaded and underloaded compute-nodes. Before doing this, it performs a check to ensure that the underloaded compute-nodes is still underloaded before committing to the transfer. Gradient methods often lead to undesirable behaviour and it has been shown to be inferior to the diffusion method in its performance [111].

Dimension Exchange Model (DEM)

This diffusion-like algorithm is introduced in [26] and analysed further in [31][113][111][42]. It is described by an n-dimensional hypercube where computers exchange tasks with each other in each dimension. In a loop over a hypercube of dimension n, a compute-node performs load balancing with its neighbours in that dimension, i.e., with the compute-node whose compute-node number matches that of the given compute-node except in bit n. The two compute-nodes divide the sum of their loads equally among themselves. The system is completely balanced by iterating over all the dimensions of the hypercube.

Although the dimension exchange algorithm is described in terms of a hypercube, it can also be applied to other architecture such as meshes. The difference is that for non-hypercube architectures, communication is non-local as logical neighbours will not necessarily be physical neighbours. The generalized dimension exchange method [113] suggests using an edge-colouring to maintain nearest-neighbour exchanges in non-hypercube architectures; however, it requires more iterations to reach convergence. More importantly, dimension exchange may migrate work to distant compute-nodes and compute-nodes with non-contiguous data regions, increasing communication costs for the application code. This disadvantage outweighs the improved convergence of the method over other diffusion methods.

Centralized Algorithms

If load balancing calculations are performed at a single compute-node, where the compute-nodes determine the necessary task transfers and inform the processors involved in the transfers what they must do, the algorithm is classified as centralized. Examples of centralized strategies can be found in [57][119]. Although the use of a central decision-making processor may lead to a bottleneck, it is important to note that distributed strategies may require load information to be propagated to all processors, which can also lead to higher communication costs.

Centralized methods require more coordination and can incur high communication costs, as each compute-node needs to communicate with the central decision maker and vice versa. These costs tend to increase non-linearly, which is why centralized methods do not scale well. Also, because the processor designated as the central scheduler receives update of load information from all the computers in order to take decisions based on that information, centralized methods tend to create a single point of failure at the central scheduler.

A review of major centralized algorithms is presented below:

Master-Slave Algorithm

Master-slave algorithm is a class of centralized load balancing algorithms that take the form of master-slave relationship in which a dedicated compute-node(s) collects the load status of every slave, analyzes the average workload and broadcasts the average to all the slaves. The slaves then do a local negotiation and reallocation of load between its neighbours, to achieve the specified average workload, or come as close as possible to the computed average workload [46].

Master-slave algorithm contains an inherently serial component, as the master handles all the messages from the slaves, and computes the average workload. In a homogeneous environment such as the cluster in which all the compute-nodes have equal speed, the optimum way to divide the load is to have equal load between the compute-nodes. This simple strategy minimizes communication overhead between the master compute-node and the slaves, and results in the fastest processing of load. The master-slave algorithm is able to achieve this aim because it has a central view of the entire system.

Hierarchical Load Balancing Algorithms

Hierarchical methods are another class of centralized algorithms. Hierarchical methods are explored in [49][111]. In this method, computers are initially organized into two large groups, balanced between one another and then the two groups are recursively divided and load balanced. At each level, the loads of the new group are determined by having the computers grouping themselves recursively, and summing the total loads of the subgroups. The major disadvantage is that the algorithm inherently neglects to minimize the distance and volume of work transferred to achieve load balance. This is particularly true for communication intensive applications and the resulting disruption of the existing locality in task mapping may have a severe impact.

2.2.7. Synchronization

Distributed algorithms are further classified as *synchronous* or *asynchronous* algorithms.

Some examples of synchronous algorithms are described in [40][111][119]. A synchronous load balancing algorithm executes simultaneously on all participating compute-nodes. In synchronous strategy, the execution of the SPMD application contains points of synchronism, where the processors stop processing the application and switch to load balancing operation. Synchronous load balancing algorithms are a natural choice for synchronous applications, because the implementation of synchronous load balancing can benefit from the synchronism of the application itself.

Some examples of asynchronous load balancing algorithms are described in [59][119]. In contrast to synchronous algorithms, asynchronous algorithm can be executed at any time in a given compute-node with no dependency on what is being executed at the other participating compute-nodes [85].

2.2.8. Load Balancing Activation

A migration based algorithm may be activated either at a fixed frequency (periodic) or when some specific condition is identified (event-driven or adaptive).

Periodic algorithms are activated regularly and independently of the current workload distribution of the system. The interval between two activations may be dictated in three ways; by time, by the execution of a fixed number of operations, or by some other criterion [85]. The algorithms in [40][57] present load balancing strategies for synchronous applications where the load balancing algorithm is activated after a given number of iterations is executed.

The interval between two subsequent activations hugely affects the efficiency of a periodic load balancing algorithm. This is a non-trivial problem because if the interval between activations is too large, the system may remain unbalanced, with idle compute-nodes, for an intolerable period of time [85]. Similarly, a small interval may result in unnecessary activations of the algorithm, and consequent waste of processing time. Either extremes can give poor performance, but setting the right period is non-trivial.

To reduce the overhead of initiating load balancing at regular intervals, the adaptive method may be employed. The adaptive method tries to determine when it is best to balance the load. This is often triggered by either a lightly loaded compute-node (receiver initiated) or by a heavily loaded compute-node (sender initiated) based on some specified threshold values [111] or the absence of further tasks to be executed by one of the compute-nodes [119][59].

2.2.9. The Target Compute-node

Task transfers in a *migration-based* load balancing algorithm can be classified into two algorithms. *Collective algorithms* aim at balancing the load of a group of compute-nodes while *individual algorithms* aim to correct the load at a single compute-node by electing the compute-nodes with which it is to exchange load. Examples of collective algorithms are described in [40][111][119] while examples of individual algorithms are presented in [40][59][111].

In individual algorithms, a *location policy (LP)* is included as part of the transfer policy, to determine which compute-nodes must exchange load with the overloaded or under-loaded compute-node [85].

2.2.10. Transfer Direction

The direction of tasks transfer can be classified into *receiving* and *sending*. *Individual receiving strategies* find new tasks to be executed by an under-loaded compute-node. The *location policy* in this case tries to identify overloaded compute-nodes from which the underload compute-node can receive tasks. As a contrast, *sending strategies* focus on overloaded compute-nodes. The goal of the location policy is to identify under-loaded compute-nodes to which tasks can be sent. Examples of sending strategies are described in [40][111] while examples of receiving strategies are described in [59][111].

2.2.11. Location Policy

Finally, *location policies* are classified into individual *non-blind* algorithms and individual *blind* algorithms. *Non-blind* policies take into considerations the load indexes of the candidate for load exchange. External load index, internal load index, or both kinds of indexes may be taken into account. Examples of non-blind load balancing algorithms are in [40][111].

Blind algorithms, on the other hand, do not consider any load indexes. In many blind algorithms, compute-nodes are selected for workload exchange on a random or round-robin basis. Some examples of blind load balancing algorithms are described in [40][59].

2.3. Critical Analysis and Conclusions of Review of Load Balancing Algorithms

Load balancing algorithms were reviewed with bias given to those types of algorithms reviewed in parallel traffic simulation in section §1.3. Below are the summary of the findings:

1. Static load balancing algorithms: These algorithms do not suite a time-varying dynamic load situation such as traffic simulation and hence found insufficient. They are useful just before the simulation starts but over a

long period when the load becomes unbalanced, due to the dynamic nature of the traffic, the algorithm are no longer sufficient to correct the imbalance.

2. Dynamic load balancing algorithms: These are suitable to the time-varying dynamic traffic situations of parallel traffic simulation as they are capable of keeping the system continually balanced and hence capable of improving and maintaining the performance of the system. Many applicable dynamic load balancing algorithms already exist in the literature review. However, all of the algorithms would need to be modified to adapt them to any required application. The two main classes of algorithms investigated are discussed below:

- *Centralized algorithms*: These classes of algorithms are suitable for parallel traffic simulation. This is confirmed by the fact that many of the reviewed traffic simulations in section §1.3 use this method. The most widely used algorithm in this category is the master-slave algorithm. From the literature review, it is reported that the algorithms exhibit good decision-making because they are capable of 'seeing' the global view of the entire system. Their reported disadvantage is that they do not scale well as a result of various slave compute-nodes communicating with the master compute-node. However, none of the reviewed traffic simulators evaluated the performance of this method.

- *Distributed algorithms*: Many distributed algorithms exist in the literature review and suite the properties of traffic simulation. They are reported to have good scalability, which is an important feature of any load balancing algorithm. The only problem with them is that since there is no central control, achieving a global load balance would usually require several iterations and is often difficult to achieve.

A table summarizing the review of dynamic load balancing algorithms is presented in Fig. 2.2.

Load Balancing Algorithm	**Type of Algorithm**	**Advantages**	**Disadvantages**
MaS	Centralized	• Good view of the system • Easy to develop	• Non-scalability • Overhead as a result of global communications with the coordinator
Hirachical Balancing Method (HBM)	Centralized	• Reduced communicational overhead as compared to MaS • Improved scalability as compared to MaS	• Global balance involves coordination between the different groups. Quality of load balance may be less than MaS. •
Diffusion	Distributed	• less communication compared to centralized algorithms • good scalability	• restricted view of the system compared to centralized algorithms • more difficult to control compared to centralized algorithms • convergence is more difficult to achieve
Dimension Exchange	Distributed	• less communication compared to centralized algorithms • good scalability • improved convergence over diffusion	• restricted view of the system compared to centralized algorithms • more difficult to control compared to o centralized algorithms • likely increase in communication cost as a result of maintaining non-contiguous data regions.
Gradient Model	Distributed	• less communication compared to centralized algorithms • good scalability	• restricted view of the system compared to centralized algorithms • more difficult to control compared to centralized algorithms • too much work likely to be migrated from one overloaded computer to another under loaded computer

Figure 2.2: Summary of Literature Review

Reasons for Selecting the Algorithms

- From the review of parallel traffic simulation, it was observed that two of the reviewed simulators employed *centralized methods* [27] [98] while the other two employed *distributed methods* [88] [25] of load balancing. However, none of the authors described the algorithms in detail as to be reused or applied to other traffic simulation research. The reason for this may be because most algorithms are application specific.

- In the reviewed traffic simulations, there is no performance comparisons between the *centralized and distributed algorithms.* One of the aims of this research is to study and compare a *centralized algorithm* with a *distributed algorithm.* The idea is to see how the two classes of algorithms compares with each other in parallel traffic simulation research.

- Among the distributed algorithms studied, diffusion was selected because:

    1. It is widely reported to have better performances over the other distributed algorithms [108].
    2. Diffusion is also reported to be simpler to implement than the dimension exchange and the gradient model algorithms.
    3. The fact that in diffusion, each node can balance its load simultaneously with its immediate neighbour (unlike dimension exchange, for example) suites traffic simulations.

- *MaS* is selected based on the following reasons:

    1. The structure of *MaS* algorithm fits well into that of SPMD Madcity parallel traffic simulations.
    2. Compared to distributed algorithms, *MaS* algorithms are simpler to implement. This is mainly because the logical control and coordination of the simulation is designated to the master processor.

The above reasons for selecting *MaS* seems to give it an advantage over the distributed algorithms but whether these advantages are reflected in its performance is left to be studied later in the thesis. Since *MaS* algorithm has a generic structure, it is the other components of the algorithms that decides its novelty. For example, the *MaS* designed in this thesis is unique in its features (the load index and PD are new) and hence considered a new design. Its work transfer calculation, which is based on the most loaded compute-node, is also novel. The work transfer calculation supervises the transfer of load between the neighbours to ensure that load 'diffuses' from the most loaded compute-node through its neighbours to the least loaded compute-node.

Reasons for not Selecting the other Algorithms

As stated above, the main motivation for selecting *MaS* and *diffusion* algorithms is because while both of them are used in the revised parallel traffic simulations, there is no report on their performances. As such, this thesis aims to study the performances of these two algorithms. This reasons excludes the use of other algorithms which have never been used in traffic simulations. The aim is to study and know the performances of these algorithms before studying the other algorithms that have never been used in traffic simulation research.

Secondly, in the case of *distributed algorithms*, since *diffusion* is reported to have the best performance compared to *dimension exchange* and *gradient model*, the aim is to study its performance in parallel traffic simulation before investigating the other algorithms. Similarly, *MaS* fits perfectly well with SPMD parallel traffic simulation which is not the case with hierarchical algorithms. Moreover, the fact that hierarchical algorithms regroups over and over again would complicate the communication structure of the parallel simulation and maintaining the neighbour partitions would be more difficult each time the computers are regrouped.

2.4. Review of Load Indices in Dynamic Load Balancing Research

As the name indicates, load balancing is simply about balancing ‘the load’ on the distributed system. To obtain an effective load index, an accurate measure of the quantitative load on the compute-nodes need to be measured. The accurate and efficient measure of load is a key issue and a prerequisite for any load balancing algorithm. Load information is usually measured by what is called a load index. The higher the value of the index, the heavier the load on the compute-node, hence the less desirable it is to transfer jobs to it. Balancing the load means removing load from the most loaded compute-node to the less loaded compute-node[121]. Although simple in concept, a good load index that accurately reflects the system’s load, is very difficult to quantify [121]. This is mainly due to the complexity of a computer with multiple resources and the different resource demand patterns of the various jobs [121].

Load indices can be classified based on, whether it is a single index (specific) or a combination of indices (generic). Specific indices involve obtaining a single value and, according to some authors, they are useful in specific cases, imposing lower overloads [64]. Generic indices involve the joining of two or more values. These are recommended by some authors when enough information is known about the application or about the objectives of the scheduler [64][121][76][72]. Generic load indices, in addition to displaying a greater tendency to overload, are not expected to show the same quality of workload representatives as correctly utilized specific indices.

A wide variety of load indices have been explicitly or implicitly mentioned in the literature review. For example, the CPU queue length was used as the load index in [121][37][112][79]. Some other authors used the CPU utilization [64].Other possibilities include the normalized response time(defined as the ratio between the response time of a process on a loaded machine and its response time on the same machine when it is empty), the remaining processing time of all the jobs running on a host [112], the processing time accumulated by the active processes, and the total processing time of the active processes [72][112]. Functions of the above simple variables have also been used in [115][64]. In a number of studies in which reducing job response time was the objective of load balancing, the estimated response time was used as the load index [72].

In [64] Kunz evaluated the performance achieved by the scheduler when load is balanced in relation to six load indices, and found that the best index is the number of processes in the queue of the processor (*processor queue length*). In another experimental evaluations, Kunz used aggregated load indices (generic) and found that these load indices did not improve the system's performance in comparison to the linear indices (specific). He also discovered that they also caused overloading when obtaining the various linear indices that would make up the aggregated indices. These studies were all done on a homogeneous cluster [72].

Generic or composite load indexes are achieved in several ways. In [72] a mathematical relation expressing all the factors of interest is used as load index and weights assigned to the various factors. Higher weights means greater influence in the load balancing decision and this depends on the resource utilization of the application. e.g. for applications that are memory intensive, higher weights are given to memory utilization. In the relation below, the properties of the hardware platform that are of importance to the load index (LI) are the utilization of CPU ($CPU_{utilization}$), Memory ($Memory_{utilization}$), Disk ($Disk_{unitization}$) and Network ($Network_{utililization}$).

$$LI = f(CPU_{utilization}, Memory_{utilization}, Network_{unitization}, Disk_{utilization})$$

Each of the factors are characterized individually and then substituted in the equation to obtain a sum which varies between 0 and 1. The factors can be characterized by means of the operating system utilities. A threshold value is set which is used for comparing the load in different compute-nodes.

In [35], several load indices are also integrated into one load index. Length of load vector is taken as the load index and each task has a resource demand vector. For each node, when a new task is received, the load index calculates the sum of tasks' resource demand vector and node's load vector. It then selects the node whose sum of the load vector is shortest to be destination node for the task.

[35] [11] present another way to integrate different load information into single load index. The load index uses available memory amount, RAM, as threshold, and calculates the sum of all the tasks minimum memory demand, denoted as MT. When $MT < RAM$, it uses CPU queue length, L, as load index, otherwise it sets load index to critical, CT, as if CPU were overloaded so that the system refuses to accept new task. This load index could balance utilization of CPU and memory resource, to enhance system performance. But because the load index requires the task's minimum memory demand information, applicability of this load index is limited.

Studies in [97] get static weight of load information through experiments, then used weighted sum of load information to be system load index. Load index decided through experiments can obtain excellent system performance when task resource demand characteristic is stable. But load index's flexibility is restricted. If task resource demand characteristic changed, load index need to be reconstructed manually. The literature review generally recommends the adoption of a load index consisting of the average few essential characteristics of some resources over a given time frame, reducing the possibility of choosing a value that does not correspond to the system's real state.

2.4.1. *Critical Analysis and Conclusions of Review of Load indexes*

Every application has a basic primary load unit (e.g. vehicles in traffic simulation, molecules in molecular dynamics) which is internal to the application. Every other load index can be considered as external to the application (e.g. CPU queue length, memory, network bandwidth, hard disk, etc). The fundamental aim of a load balancing algorithm is to improve the application runtime and this can be done either with the internal or the external load indexes. A combination of internal and external load indexes are also possible. In such a case, weights are usually assigned to each of the indices and the weight assigned varies from application to application. For example, if a composite load index comprises of CPU and memory utilization, higher weights are given to CPU utilization in applications strictly CPU-Bound (e.g meteorological weather simulation) and higher weights given to memory utilization in applications that are strictly memory-bound (e.g 3D image rendering)[72]. In fact, if this load index is used in an application that is 100% CPU-bound, memory is assigned a weight of 0 which then makes a composite load index a specific load index. This means that the parameters that have greater influence on load index depends on the characteristics of the application under consideration.

Single load indices (specific) are simple and reported to have better performance than composite ones(generic) as the overheads of these are much less than for composite ones [64]. Of the six different load indices reviewed and analysed by Kunz in this paper, CPU queue length was reported to have the best performance.

Most of the reviewed load indexes were originally designed for homogeneous systems(e.g. clusters). In recent times, many researchers have extended the algorithms to the heterogenous systems such as the grid platform[43][63]. The algorithms are adapted to take advantage of the different heterogeneous nature of the cluster such as different CPU capacities, varying memory sizes and different network bandwidths.

What recent researchers fail to address in their algorithm design is that many of the algorithms did not include the load of the application. Most of them consider the parameters of the hardware platform but fail to address the load of the application. At the same time, a few other researchers only include the load of the application and nothing else. For example, the load index used by majority of all the reviewed traffic simulations with dynamic load balancing is vehicles. While this is simple and efficient, it can only work effectively in a homogeneous cluster. In a distributed platform such as the grid, it does not take into consideration the heterogeneous nature of the grid platform and the dynamic load of the compute-nodes. For example what happens if a compute-node that is say 70% busy had 3000 vehicles but another compute-node that is say 30% busy had 4000 vehicles? In this example, simply using vehicles as the only load index would attempt to remove 500 vehicles from the 30% compute-node to the 70% compute-node in an attempt to equalize the vehicles. If the system load is inclusive in quantifying the load however, the system would be aware that the 70% system may not be ready for more vehicles at this time.

The design in this thesis aims to consider the load of the application (such as vehicle in traffic simulation) in the design of the algorithm in addition to the system load. The system *response time* load index captures both the load of the application and the system load. This design is intended to be suitable for both homogeneous and heterogeneous systems. However, even though the designed load index is intended to work on both the homogenous and heterogeneous distributed platform, the heterogeneous characteristics of the grid would be created on the cluster(further work could test it on a real grid platform). This is suitable because in a homogenous system, the system's hardware resources may be similar but depending on the load of the individual computers, the available resources may differ. In other words, most applications and tasks in the homogeneous system are heterogeneous in nature because they consume different amounts of resources (some applications run for short or long time duration and use different percentage of the CPU time and memory). In addition, a synthetic load would be used to vary the resource usage of the cluster therefore creating a grid-like environment.

2.5. Review of Profitability Determination (PD) in Dynamic Load Balancing Research

Load balancing incurs additional run-time overhead to an already poorly performing unbalanced system. To minimize the additional overheard, a cost-benefit analysis is performed to ascertain if it is profitable to undertake load balancing. That is, if the predicted improvement in the system performance after load balancing is more than the overhead from the load balancing algorithm. A periodic load balancing without PD implies that load balancing would be attempted at every period. This is simply not efficient because load imbalance is not the only criteria for performing load balancing. The imbalance must be high enough to warrant its operation.

Many load-balancing algorithms use a version of the gain criteria from the algorithm of Kernighan and Lin (KL) [60] to select objects to transfer [47]. For each of the compute-node's objects, the gain of transferring the objects to another compute-node is computed. For example, to minimize edge-cuts, the gain can be taken as the net reduction of cut edges if the object is transferred to the new target compute-node. The set of objects with the highest total gain is selected for migration. Objects are selected until the sum of their work loads is approximately equal to the work transfer computed by the load-balancing phase. In some variants of the KL algorithm only objects on sub-domain boundaries are examined for transfer [47].

The element selection priority scheme of Wheat [109] is an example of a KL-like algorithm with uniform object and edge weights. Gain is measured by edge-cuts in the graph. All transfers of objects are one-directional, so collisions (the simultaneous swapping of adjacent objects between two compute-nodes which counteracts their individual gains) do not arise. In [30], Wheat's work is extended by weighting the edges by frequency of communication and the objects by their computational load. To reduce migration costs, high-gain objects with the largest computational loads are selected for migration [47].

In [65] a cost benefit heuristics was designed for SAMR application which provides a very conservative estimate of the amount of decrease in execution time that will occur from the redistribution of load resulting from the DLB. To determine if a global redistribution is invoked, an evaluation model is required to calculate the redistribution cost and the computational gain. The cost and gain of the algorithm are calculated thus:

Cost: The redistribution cost consists of both communicational and computational overhead. The communicational overhead includes the time to migrate workload between compute-nodes. In order to adaptively calculate communication cost, the network performance is modelled by the conventional model, that is $T_{comm} = \alpha + \beta + L$, where T_{comm} is the communication time, α is the communication latency, β is the communication transfer rate, and L is the data size in bytes. The heuristic scheme sends two messages between groups, and calculates the network performance parameters α and β. If the amount of workload needed to be redistributed is W, the communication cost would be $\alpha + \beta + W$. To estimate the computational cost, the scheme uses history information, that is, recording the computational overhead of the previous iteration. If the cost is denoted as δ, the total cost for redistribution is: Cost $=(\alpha + \beta + W) + \delta$.

Gain: The scheme predicts the computational gain by a heuristic whereby the global load redistribution is invoked when the computational gain is larger than some factor times the redistribution cost, that is, when Gain $>$ (γ) x Cost. Here, (γ) is a user-defined parameter (default is 2:0) which identifies how much the computational gain must be for the redistribution to be invoked.

In [108] the authors proposed using historical improvement of past load balancing methods as a basis for formulating the advantages of performing dynamic load balancing. The time required to load balance can be measured directly using available facilities. The expected reduction in run time due to load balancing can be estimated loosely by assuming efficiency will be increased to 100 percent or more precisely by maintaining a history of the improvement in past load balancing steps. If the expected improvement exceeds the cost of load balancing, the next stage in the load balancing process should begin. The problem with this method is that it is expensive as a result of the time required to go through all the historical data each time.

In [111], during the profitability determination phase (triggered by a compute-node's load estimate or timer expiration) a decision is made as to whether or not to invoke the load balancer. The load imbalance factor $\phi(t)$ is an estimate of the potential speedup obtainable through load balancing at time t [14]. It is defined as the difference between the maximum compute-node loads before and after load balancing, L_{max} and L_{bal}, respectively. $\phi(t) = L_{max} - L_{bal}$. A decision on whether or not to load balance is made based on the value of $\phi(t)$ relative to the balancing overhead, $L_{overhead}$, required to perform the load balancing. In general, load balancing is profitable if the savings is greater than the overhead, i.e., $\phi(t) > L_{overhead}$.

2.5.1. *Critical Analysis and Conclusions of Review of Profitability determination*

From the above revisions, most of the algorithms employ heuristics to calculate the gain of performing load balancing. Since different applications employ different heuristics to suit the properties of their application, most of these are application specific. It is also observed that some of the above reviews used calculations to determine PD. For example, in [65], separate calculations were used to estimate gain and cost, taking few network parameters into considerations. The disadvantages of such method is the additional overhead associated with too many calculations. This thesis designed a novel profitability gain algorithm/method based on the most loaded compute-node in the system since the most loaded processor determines the run-time of the application. This thesis used a methodology of doing the PD calculations off line. By setting the load threshold off line, it is expected that PD would add very minimal overhead to the overall load balancing overhead.

2.6. Chapter Summary

This chapter discussed load balancing problem and review of state of the art of dynamic load balancing algorithms, load indices and profitability determination algorithms. The chapter also presents a comprehensive taxonomy of load balancing in SPMD applications along with its discussion. Combining the information in this chapter with the review of parallel traffic simulation in chapter one, the following conclusions have been made regarding the major work of this research:

- Design a suitable load index and a profitability determination(PD) methods for parallel traffic simulation
- Design a centralized dynamic load balancing algorithm, enhanced with load index and PD.

Chapter 3

Design and Implementation of Dynamic Load Balancing Algorithms

3.1. Chapter Overview

This chapter presents the research framework and a discussion of its components. It then discusses the factors affecting the design of parallel and distributed processing such as *graph partitioning, synchronization, communication overhead* and *load balancing challenges.* The partitioned data must be distributed across the loosely coupled processing environments and processed concurrently for high performance, yet the processes require timing and scheduling coordination. Also, the logical processes need to communicate with each other. By far, a load imbalance in the overall distributed system presents the most serious factors that affect performance. This chapter discusses the above mentioned design constraints and goes further to design a *load index* and a *profitability determinant* for dynamic load balancing algorithms. These were used as building blocks in the design of a centralized master slave(MaS) dynamic load balancing algorithm and the enhancement of diffusion algorithm with the designed building blocks, to correct load imbalance in the research parallel traffic simulation.

3.2. The Research Framework

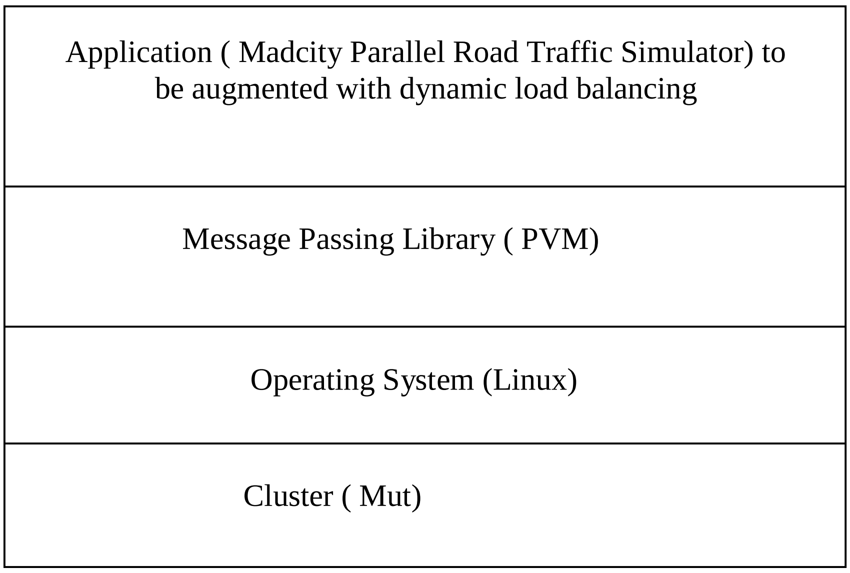

Application (Madcity Parallel Road Traffic Simulator) to be augmented with dynamic load balancing
Message Passing Library (PVM)
Operating System (Linux)
Cluster (Mut)

Figure 3.1: The Experimental Research Framework

This section presents a simplified research framework and the discussions of the different parts of the framework. The framework for the research platform is shown in Figure 3.1.

3.3. The Mut Cluster

The University of Westminster (UoW) has a 32 comute-node cluster called Mut. The head-node is a Sun V20z, 2 X AMD Opteron 250 (64) bits processor, 4GB memory, 2 x 73 GB hard disk and running Suse Linux Enterprise (SLES) 9 operating system. The 32 compute-nodes are Intel Xeon 2.8Ghz processor, 512 memory, 36Gb hard drive and running Suse Linux Enterprise 9 operating system. The compute-nodes are connected to the head node by a 10Gbps Infiniband [70][69] interconnect fabric. The cluster runs Parallel Virtual Machine (PVM) [96][3][33], the parallel programming API used for this research project.

3.4. The Carmen Cluster

Close to the end of completing the thesis, the Mut cluster was replaced with a much faster and larger 64 compute-node cluster. Most of the experiments were performed on Mut but a few others were also done on Carmen, after Mut was decommissioned. As a result, in each experiment, the hardware platform would be specified and if not specified, the default is Mut. This is important because even though the behaviour of the algorithms are identical on both clusters, the time it takes to run the experiments differ.

Carmen is a 64 compute-node cluster with Spiderman as the head-node. The head-node is an IBM x3455 series, each featuring 2x Dual core AMD Opteron 2281 Rev F 2.6GHz processors, 2x 80Gb hard drives and 4GB Memory. The 64 Compute-nodes are also IBM x3455, each featuring 2x Dual core AMD Opteron 2218 Rev 2.6GHz processors, 1x 80Gb hard drive and 4GB Memory with Infiniband adaptors fitted. The compute-nodes are running SuSe Linux Enterprise 10 operating system. Also, the compute-nodes are connected to the head-node by a 10Gbps Infiniband interconnect fabric. The cluster runs Parallel Virtual Machine (PVM) the parallel programming API used for this research project.

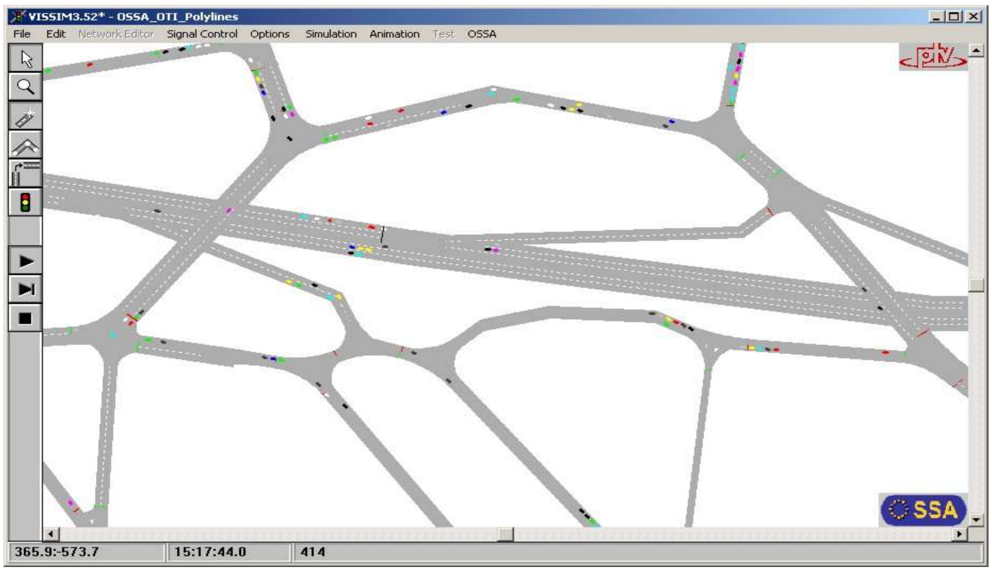


Figure 3.2: Traffic Simulation of the Hyde Network

3.5. Madcity Traffic Simulator

The research traffic simulator, Madcity, available for research purposes, is developed by the center for parallel computing (CPC) at the University of Westminster (UoW). It is a discrete-time based microscopic simulator, organized around a compound data structure that represents the road network. The road network is modelled as a collection of interconnected junctions. *Junctions* are interconnected through *roads* where each road may have multiple traffic *lanes*. The road network also contains representation of traffic control equipment (e.g. traffic lights).

Traffic is represented as a set of *vehicle objects* that use the road network. A vehicle object consists of a set of attributes that identify the vehicles and its coordinate position, speed, distance etc. This enables every vehicle to keep a record of essential vehicle data, necessary for the simulation.

At the start of simulation, vehicles are distributed randomly through the lanes. The total number of vehicles are stored in a *vehicletotal* variable. At each step of the simulation, each vehicle is moved according to a set of simple localized rules to compute its new state and new position. In this, account is taken of the vehicle's surrounding conditions. For example, proximity to a slower vehicle ahead will influence the speed of the vehicle. The ability to read the surrounding conditions is made available through the network structure. The overall complex pattern of the urban traffic emerges from the simple local actions of the individual vehicles.

Vehicles travel until they come to the end of the lanes. At this point, vehicles decide which of the outgoing lanes to join. This decision is made by a simple randomisation function. The decision to change lanes is also affected by the colour of the traffic light. As in real life, red light implies that the vehicles wait until the light turns green, which may be in the next simulation step. After the decision, vehicles are removed from their current lane and inserted into the destination lane and simulation continues. A typical simulation scenario, simulating the town of Hyde in UK is shown in Fig. 3.2.

3.5.1. Load in Madcity Traffic Simulation

To understand what constitute the load in Madcity traffic simulator, an experiment was performed with the sequential Madcity on a single compute-node. For a fixed network size, the number of vehicles were varied and in each increment, the simulation was run for about 5000 steps and the run-time taken. Figure 3.3 is the result of the experiments. As can be seen in the graph, the run-time increases linearly as the number of vehicles is increased. This shows that vehicles constitute the load in Madcity. Henceforth in this thesis, the load of the system refers to the number of vehicles in the simulation. This simplified metric allows saying in general that "if compte-node A has a load of 1600 vehicles and compute-node B has a load of 1000 vehicles, there is need to relocate 300 vehicles from compute-node A to B to achieve load balance".

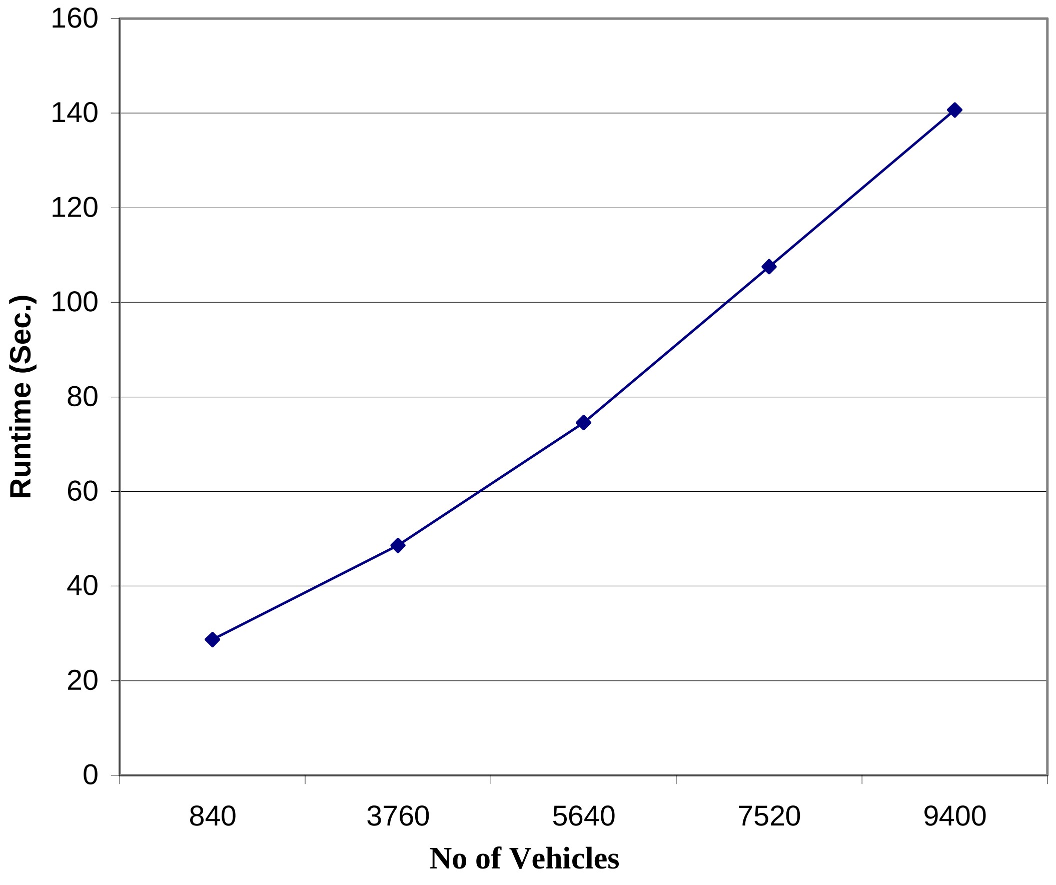


Figure 3.3: Effect of Load on the Run-time of Madcity

3.5.2. Manhattan Network Generator

The Manhattan legacy code is an application to generate inputs for the Madcity simulator: a ‘road network file’ and a ‘turn file’. The Madcity road network file is a sequence of numbers, representing the topology of a road network. The number of columns, rows, unit width and unit height can be set as input parameters to this component. The Madcity turn file describes the junction manoeuvres available in a given road network. Traffic light details are also included in this file.

3.6. Constraints in the Design of Parallel Applications

For an effective parallel application, the following design factors must be carefully considered in the design stages of the algorithm.

3.6.1. Graph Partitioning/Domain Decomposition of Parallel Applications

An important step in parallel processing is the decomposition of the problem domain and/or data into different sub-domains. The sub-domains are distributed to different compute-nodes for simultaneous parallel executions. Adjacent processes exchange information by message passing during the computation. For example, in Single Program Multiple Data (SPMD) domain decomposition, each program is replicated to all the participating compute-nodes while each of them have different data subset. SPMD applications have the structure of a master slave program in which the master process supervises and coordinates the general working of the program. The compute-nodes does the calculations and communicate with each other through message passing.

Efficient execution of simulations on distributed-memory machines requires the mapping of the data domain onto the compute-nodes with two main aims, known as the *optimal conditions* [90].

- Equalize the number of domain elements assigned to each compute-node and
- Minimize the inter-processor communication required to perform information exchange between adjacent compute-nodes.

In order to compute a mapping of a network domain onto a set of compute-nodes via graph partitioning, it is first necessary to construct the graph that models the structure of the computation. Given a weighted, undirected graph

$$G = (V, E) \tag{3.1}$$

where each vertex (V) and edge (E) has an associated weight, the n-way graph partitioning problem is to split the vertices of V into n disjoint sub-domains so as to satisfy *optimal conditions*.

In traffic simulations, the road network is modelled as a graph where road junctions are represented by the vertices and the roads by the edges connecting the junctions. Each road has a set of directed lanes because the directions of the vehicles are important. For every junction there are associated numbers of outgoing roads connecting these junctions.

A vertex exists for each road junction, and an edge exists on the graph for each road between the road junctions. Partitioning the vertices of the graphs into n disjoint sub-domains provides a mapping of the road junctions and the roads into n compute-nodes. In the situation in which the number of vehicles is proportional to the number of roads, if the partitioning is computed such that each sub-domain has the same number of roads, each compute-node will have an equal amount of work.

The total volume of communications incurred during parallel processing is proportional to the number of edges that connect vertices in different sub-domains. Partitioning should therefore be computed so as to minimize edge cuts. For n road junctions and p compute-node, there are different ways to assign n junctions to p compute-nodes, but considerations should be given to partitions that best satisfy the optimal conditions.

Consider for example the simple example of the road network topologies shown in Figures 3.4 and 3.5 where the road network has been reduced to a mesh for simplicity. In the figures, the black dots represent the road junctions while the lines represent the roads connecting the junctions. It is possible for the roads to have more than one lane connecting the junctions but for simplicity, only single lane have been represented.

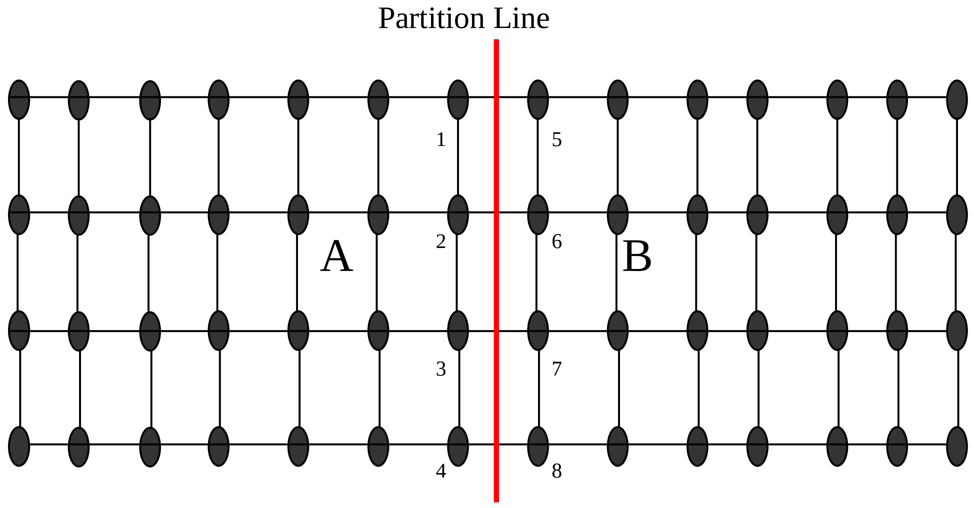


Figure 3.4: Regular Vertical Graph Partitioning

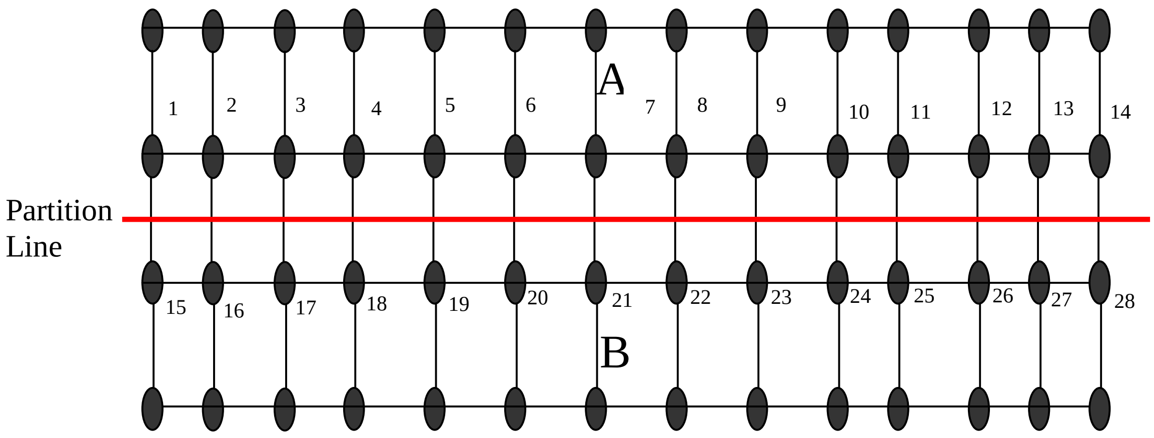


Figure 3.5: Regular Horizontal Graph Partitioning

In the figures, the road network topology has been partitioned into two partitions (sub-domain A and sub-domain B) in two different ways, each with equal load distributions (equal junctions) but different edge-cuts. In Fig. 3.4, each partition has 4 edge-cuts. During parallel computation, the compute-node corresponding to sub-domain A will need to send the data for vertices 1, 2, 3 and 4 to the compute-node corresponding to sub-domain B and likewise, the compute-node corresponding to sub-domain B will send data for vertexes 5, 6, 7 and 8 to the compute-node corresponding to sub-domain A. Similarly, in Fig.

3.5, each partition has 14 edge-cuts. The compute-node corresponding to sub-domain A will need to send the data for vertices 1, 2, 3 . . . 14 to the compute-node corresponding to sub-domain B and likewise, compute-node B will send data for vertexes 15, 16, 17 . . . 28 to compute-node A. This equals 4 units of data to be sent in Fig. 3.4 and 14 units of data to be sent in Fig. 3.5.

Considering the number of edge-cut and hence the communication required between sub-domain A and sub-domain B in both cases, it can be seen that Fig. 3.4 has a superior partitioning strategy and should be capable of producing better performance than Fig. 3.5. This is because, while Fig. 3.4 requires communication between 4 data element, Fig. 3.5 requires communication between 14 data elements.

A computational optimization is to group all the data into one communication buffer and then send the message at once. Still, the buffer required for sending a message in the second diagram would be larger than the buffer required for the first diagram, and since message latency is proportional to the size of the message, Fig. 3.4 would still perform better than Fig. 3.5.

3.6.2. Domain Replication Method of Parallel Applications

A method similar to domain decomposition is called domain replication which is a method that store the network file redundantly on all the participating compute-nodes and allocate each compute-node a different set of junctions [86]. Each compute-node simulates junctions that have the same junction ID as the process ID of the compute-node. The redundant part of the network file that is not simulated occupies a part of the memory and this is a disadvantage but its simpler to implement than domain decomposition. In this method, at load balancing, the compute-node that needs to give part of its junctions to another compute-node would do two things:

- Reassign the junctions with the process ID of the new compute-nodes and

- broadcast the new network information to all the compute-nodes so they can equally update their respective network.

Domain replication was adopted in the parallelization of Madcity traffic simulator. This is a simple and efficient method. The only disadvantage is that the size of the road network that can be used is restricted by the size of the memory which holds the entire road network on every compute-node.

3.6.3. Communication Overhead in Parallel and Distributed Systems

Parallel processing platforms are architecturally divided into shared-memory multiprocessor systems and distributed-memory multiprocessor systems. In the shared-memory architecture, all the processors have access to the same memory. The main problem with shared memory systems, however, is that they do not scale well due to contention issues resulting from every processor trying to access a single memory bus. For small numbers of processors, collisions are minimal but increase linearly as the number of processors increase.

In the distributed-memory system, each processor is an independent entity and has its own processor and memory. The compute-nodes are then connected together through a network backbone. Distributed memory systems have no memory bus problem because each processor has full bandwidth from its own local memory. Thus, it can scale to large number of processors. The size of the system is constrained only by the size of the network backbone. However, exchange of information between the processors is more difficult than in the shared memory architecture. Processors communicate by message passing, in which each processor has a private local memory in order to keep the variables and data, and thus can access local data very rapidly. If an exchange of information is needed between the processors, the processors communicate and synchronize by passing messages which are simply send and receive instructions. In order to send a message a computer has to perform the following procedures:

- Pack the message into a send buffer and send the message to the destination compute-node.
- The message travels through the network backbone and is received at the destination compute-node into the receive buffer.
- The destination compute-node unpacks the message into an allocated variable.

The disadvantage of message passing is the time delay associated with each of these phases. These delays are referred to as the communication overhead of parallel applications and constitute a major performance degradation of parallel applications. As a result, it is often said that communication is expensive.

As explained in (section §3.6.4), in Madcity, apart from the initial communication in sending the road network between the compute-nodes, communication is also performed at the end of every simulation time step. Communication at this stage involves sending the vehicles crossing from one partition to another. Since communication is expensive, it is optimized in Madcity by use of Lane Cut Points (LCP)(section §3.10.1). To optimize the communication phase, at the end of every simulation step, boundary roads and junctions are checked for every vehicle crossing from one partition to another, and put into an LCP vehicle buffer. Buffering all the vehicles and sending them at once (instead of sending the vehicles individually) helps to reduce communication overhead. Once this is done, the vehicle buffers are exchanged (i.e send and receive operations) between the compute-nodes.

3.6.4. Synchronization of Parallel and Distributed Systems

Computational synchronization is always required for data dependencies and strict ordering. When it is absolutely necessary to guarantee that operations happen in exactly a particular sequence, and in particular that communications are properly interspaced with the computations they cooperate with, one can exercise maximum degree of execution control by using mechanisms which impose strict temporal ordering on the operations they bound. This ensures that steps are taken in exactly the sequence defined, and/or with the greatest possible degree of synchronization [48][100].

In synchronous synchronization, no member of parallel computation is allowed to advance its simulation clock until all other members have acknowledged that they are done with computation for the current period. This takes the first approach to providing a global ordering of events by preventing out of order events from being generated. In this system it is impossible for inconsistencies to occur since no member performs calculations until it is sure it has the exact same information as everyone else [48].

Synchronous simulation is simple and easy to implement but there are trade-offs between simplicity and performance; the simplicity of synchronous algorithms comes with a potential cost in performance. Since the critical paths lie with the slowest compute-node at each iteration, idle time can accumulate at the other compute-nodes and the total execution time is lower bounded by the execution time of the slowest compute-node in each iteration.

Synchronous simulation cost includes computation cost, communication cost and idle time waiting for others to finish task execution. Waiting time is exacerbated by unequal load distributions in the system because simulation is lower bounded by the most loaded compute-node in the system. Compute-nodes with light loads have short execution times but will have to wait for the longest executing compute-node before all of them can proceed to the next time step. After each time step, all the compute-nodes exchange messages. A common implementation of synchronous synchronization is barrier synchronisation [48]. The concept is simply that all processes in some group must reach a certain point in execution of their code before any is allowed to proceed beyond that point [99][83]. Getting pass the barrier allows a process to deduce, locally, that the whole group has reached a common, global, state.

Barrier synchronisation consists of 3 main phases:

- *Computation phase* - performing the computational tasks associated with the application
- *Idle phase* - time between first and last compute-node in the group
- *Synchronization phase* - time to complete the barrier synchronization operation

Computation starts on all compute-nodes immediately following the barrier synchronization. During this phase, each compute-node executes all the tasks assigned to it at that iteration. At the end of the computation phase, each processor enters a barrier and waits for its completion. The idle phase is a result of variation in computation times between processors due to imbalances in workload as the algorithm progresses, multitasking other unrelated processes (background load), or processor heterogeneity. Synchronization time is determined by the communication performance of the parallel platform in completing the barrier synchronization. After the barrier synchronization completes, the processors proceed to the next iteration, repeating the cycle until the algorithm completes [38].

Madcity, the research traffic simulator is a synchronous traffic simulator that employs barrier synchronisation. Synchronisation is at the end of every simulation time step. At this stage, each compute-node does the following;

- Check the end of every boundary road and junctions and for every vehicle crossing from one partition to another, put it into a vehicle buffer. Buffering all the vehicles and sending them at once (instead of sending the vehicles individually) helps to reduce communication overhead

- Exchange (i.e send and receive operations) the vehicle buffer between the compute-nodes. This is the communication phase and its explained in more detail in section §3.6.3 above.
- Wait for every other processor to reach this stage before proceeding to the next simulation step. This is enforced by barrier synchronisation.

As explained above, the use of barrier synchronisation means that load evenness is very important for efficiency since the most loaded processor determines the overall simulation time of the Madcity simulation step. The aim of load balancing algorithm is to equalize the load between the processors and achieve better performance.

3.7. The Design of Dynamic Load Balancing Algorithms

J. Watts and S. Taylor [107] describe a typical dynamic load-balancing algorithm as follows:

```
Begin Load-balance
      EVALUATE load for each task and determine how much is needed
      for balance
         If PROFITABLE to load balance
                CALCULATE work/load transfer between computers
                SELECT loads to meet those transfers
                MIGRATE selected task to their new computers
         End if
End load-balance
```

This algorithm constitutes five main phases namely *load evaluation, profitability determination, work transfer calculation, load selection and load migration*

[107][108]. The 5 phases were used as a guide in the design considerations for the various phases of the dynamic load balancing algorithms that follows.

3.7.1. Load Evaluation Strategy

From experimental results in section (§3.5.1) it has been observed that the number of vehicles is a measure of the load in the research traffic simulation, hence its load index. This is because the run-time of the simulation is proportional to the number of vehicles. Using only vehicles as load index is simple and efficient. The problem with using vehicles as the only measure of load is that it does not give a complete indication of the total load of the compute-node in which the simulator is being run.

This may not be very important in a homogeneous system but it is particularly important in a heterogenous system where heterogeneity arise as a result of the constituent components in the compute-node and the network such as *hardware, software, interconnection network*, OS, etc. Even though the importance of these variables vary, all of them can affect load characterization. For example, whenever a system resource, such as a CPU or a hard disk, is occupied by a transaction or process, it is unavailable for processing other requests. Pending requests must wait for the resources to become available before they can complete. The higher the percentage of time that the resource is occupied, the longer each operation must wait for its turn [52].

As Figure 3.6 shows, the service time for a single resource increases dramatically as the utilization increases beyond 70 percent. For example, if a transaction requires 1 second of processing by a given resource, it can be expected to take 2 seconds on a resource at 50 percent utilization and 5 seconds on a resource at 80 percent utilization. When utilization for the resource reaches 90 percent, the transaction can be expected to take 10 seconds being serviced by that resource. As a result of this shortcoming of using vehicles load index, a response time load index was designed in this thesis and this is disscussed below:

Design of a Response Time Load Index (RTLI)

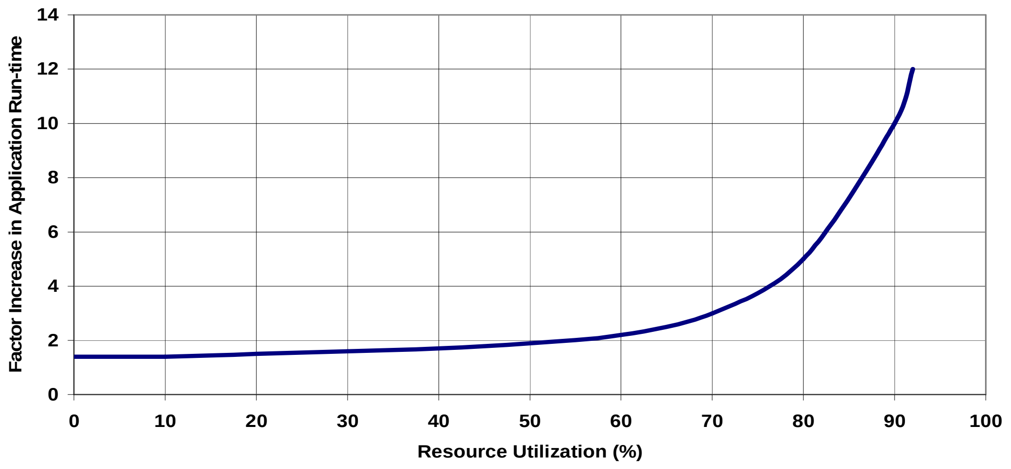


Figure 3.6: Factor Increase in Application Run-time for a Single Component/Resource as a Function of Resource Utilization

The identification of appropriate loads with the purpose of considerably increasing the utilization of idle resources is proven an important point to be considered in the design of any load balancing algorithm. It was stated in the literature review that for a load index to be effective, it must reflect accurately, or at least as closely as possible, the state of the evaluated resources [72]. Single value load indexes are simple to use (ref §2.4) but the problem is that no single factor can completely quantify how unbalanced a system is. Composite load index are better in quantifying the load of a system but the problem is that they are time consuming and since load balancing aims to incur as little overhead as possible, composite load indexes are not desirable.

As can be seen in Figure 3.6, for example, the performance of an application (e.g. Madcity) is also affected by the hardware parameters of the compute-node such as the processor utilization, etc. For effective load balancing, therefore, the load index must take into consideration both the load of the application and the hardware utilization of the parallel processing platform. That is, application run-time (RT) is a function of both the Resource Load (RL) and the Application load (AL). The response time load index is simple yet meets this requirement and gives an accurate load index to the load balancing system. In other words, RTLI implicitly measures both AL and RL.

Response time load index(RTLI) is therefore defined as the run-time per simulation time step. If the run-time per simulation time-step of a simulation is known, it can be used to predict the time to complete the simulation. Response time is therefore a measure of both the AL and RL of the system and hence its load index. However in moving load between the compute-nodes to balance the system, vehicles are relocated between the compute-nodes. The load imbalance and the load to relocate are determined in seconds, but this needs to be converted into an equivalent number of vehicles to be relocated.

The explanation of the usage of RTLI is presented in the pseudo code and the algorithm below.

Pseudo Code of the Response Time Load Index

1. At periodic DLB initiation, each compute-node measures the time per simulation step (response time load index (RTLI). Please note that RTLI measures both AL and RL). It uses the value to calculate the run-time per vehicles per step ($RT_{pervehicle}$) using equation 3.2 below:

   $$RT_{pervehicle} = \frac{RTLI}{Node_{vehicles}} \tag{3.2}$$

   where:

   $RTLI$ = response time load index and

   $Node_{vehicles}$ = total vehicles in the compute node

2. Each compute-node sends the ID and $RTLI$ to the decision making compute-node.

3. The decision making compute-node receives $RTLI$ and IDs from all the

compute-nodes. It performs work transfer calculation, using *RTLI*, it resets LB_{Advice} buffer and then stores the load relocation information for every compute node in the LB_{Advice} array. This is broadcast to all the compute-nodes. Note that it is important to reset LB_{Advice} buffer at each load balancing step because previous historical data are not used in making load balancing decisions.

4. Each compute-node receives LB_{Advice} from the decision making compute-node.

5. Each compute-node extracts, from LB_{Advice}, load for its left and right neighbours into the variables $Load_{left}$ and $Load_{right}$ respectively. Load is in seconds.

6. $Load_{left}$ and $Load_{right}$ are converted back into a number of equivalent vehicles ($vehiclestorelocate$) using equation 3.3.

$$vehiclestorelocate = \frac{Load_{relocate}}{RT_{pervehicle}} \tag{3.3}$$

   where

$$\begin{aligned} Load_{relocate} &= Load_{left} \text{ for left neighbour balance} \\ &= Load_{right} \text{ for right neighbour balance} \end{aligned}$$

7. Each compute-node uses the value in step (6) to decide on the vehicles to relocate to its neighbours in order to achieve balanced load.

8. Continue with load balancing.

Response Time Load Index (RTLI) Algorithm

Algorithm 1: The Response time Load index

```
   Input: Set of tasks and processors
   Output: Load Index
1  if Am MASTER then
2      Receive $RTLI$ and $ID$ from every participating compute-node;
3      Initialise $LB_{Advice}$ array to zero;
4      Do load transfar evaluation, using $RTLI$, and store in $LB_{Advice}$. Send $LB_{Advice}$ to
       all workers.;
5  end
6  else if am the WORKER then
7      Measure my time per sim step i.e. response time load index (RTLI);
8      Calculate $RT_{pervehicle} = RTLI/Node_{vehicles}$;
9      Send MyID and $RTLI$ to the MASTER process;
10     Receive $LB_{Advice}$ from the MASTER process;
11     Extract information for my left and right neighbours into $Load_{left}$ and $Load_{right}$
       respectively;
12     Convert the load for left and right neighbours which is specified in seconds into
       equivalent vehicles;
13     Left neighbour vehicles = $Load_{left}/RT_{pervehicle}$;
14     Right neighbour vehicles = $Load_{right}/RT_{pervehicle}$;
15     Select load to meet the requirements of my left and right neighbours;
16     Continue with load balancing;
17 end
18 Resume Normal Simulation;
```

From algorithm 1 above, RTLI was converted into vehicles, the primary load index in Madcity (see section 3.5.1). But RTLI is not only restricted to traffic simulations–it can be adapted to other simulations such as molecular simulation and just like in Madcity traffic simulation, the algorithm would also need to convert RTLI into an equivalent primary load index(e.g no of molecules).

3.7.2. Profitability Determination (PD)\Load Balancing Activation

This was discussed in section 2.2.8. Load balancing is capable of improving the performance of a system but not always. Profitability determination is the aspect of the load balancing algorithm that quantifies the potential gain in performance of an intended load balancing operation. But Can the cost of load balancing be quantified ahead of load balancing operation? This is one of the issues that this thesis addressed. The logical processes in the research parallel traffic simulator synchronize at the end of every simulation step. Synchronization points therefore provide a natural clean point at which to initiate load balancing followed by a decision to decide if the current load imbalance is worth proceeding with load balancing operation.

3.7.3. Design of a PD Based on the Most Loaded Processor

As discussed in section 3.6.4, the bottleneck in a synchronous simulation always lies with the most loaded compute-node in the system. The design in this thesis is therefore aimed at the most loaded compute-node ($Node_{mostloaded}$). The ideal load of every processor in a distributed system is the average load ($AvgLoad$) in the system. That is, for a perfectly balanced system, every compute-node must have average load. This implies that for the most loaded processor, for example, its load is given by equation .

$$Node_{mostloaded} = AvgLoad + x \tag{3.4}$$

where x is the load above average.

Ideally, x must always be redistributed because for every x above the average, some other processors are short of x either individually or collectively. But as discussed in section §3.6.3, it may not always be profitable to relocate x as a result of communication and computational overheads, except when it is above a certain value.

The real question therefore is: what is the minimum value of x in the most loaded system, which when relocated, would give performance improvement? x can be investigated by some mathematical relations and it can also be investigated by heuristic methods but those would incur additional overhead. This research used a method whereby a threshold value, called $PD_{threshold}$ is obtained prior to load balancing operation. If the threshold value is correctly determined, it would guarantee load balancing benefits in most cases. PD is therefore expressed as shown in algorithm 3.5 below.

$$if(x >= PD_{threshold})\ dynamicloadbalance() \tag{3.5}$$

where:

$x = Node_{mostloaded} - AvgLoad$

dynamicloadbalance() = Dynamic load balancing routines

$$PD_{threshold} = AvgLoad \times pd_{percentage} \tag{3.6}$$

$pd_{percentage}$ is the percentage of the average load at which load balancing is profitable. Note that $pd_{percentage}$ is determined by experimentation.

Equation 3.5 means that in the most loaded processor, if the load value above average is greater than or equal to a load threshold value, determined by a percentage of the average load, then load balancing is most likely to result in performance improvement and not otherwise.

Profitability determination is thereby defined as an estimation of load balancing gain determined by setting a load threshold as a percentage of the average load, such that when the load above average of the most loaded processor is above *this threshold value, load balancing is likely to result in improved performance.*

The threshold value is determined by experimenting with various percentages of the average value at which load balancing yields performance improvement. This is done outside load balancing operation and hence not part of the load balancing overhead. As stated earlier in this thesis, load balancing injects *additional time* into the system in an attempt to reduce its run-time. This is done in anticipation of the overall gain that is expected as a result of the additional time. An obvious way to do this is to perform some of the work off line in a preprocessing stage to the simulation. However, since load balancing is dynamic in nature, not all aspects of the load balancing algorithm can be done off line. PD is one aspect of the load balancing algorithm whose threshold value can be obtained prior to the simulation to reduce its overhead.

Pseudo Code of the Profitability Determination (PD)

The load PD algorithm is a master-slave and as such has both the master and the slave component.

The master performs the following:

1. At periodic DLB initiation, the master node receives ID and load from every participating compute-node
2. Sorts the load to determine the most loaded compute-node
3. Picks the load of the most loaded compute-node ($node_{mostloaded}$)
4. Obtains $PD_{threshold}$ and the average load from the system (AvgLoad)
5. Calculates x ($x = node_{mostloaded}$ - AvgLoad)
6. Defines an integer variable for storing PD decision ($PD_{decision}$) and initialises it to zero. By default, it is assumed that load balancing is not profitable and no values are stored to influence future decisions

7. Compares x with $PD_{threshold}$
8. If x is greater than $PD_{threshold}$, sets $PD_{decision}$ to TRUE and send $PD_{decision}$ to the compute-nodes
9. If x is less than $PD_{threshold}$, sets $PD_{decision}$ to FALSE and send $PD_{decision}$ to the compute nodes

   The slaves does the following:

10. Send ID and load to MASTER
11. Receive $PD_{decision}$ from MASTER
12. If $PD_{decision} == TRUE$ the compute-nodes perform DLB

The PD algorithm is presented below:

PD Algorithm

Algorithm 2: The PD Algorithm

```
Input: Set of tasks and processors
Output: PD Decision
1  if Am MASTER then
2      Receive ID and load from every participating compute-node;
3      Sort the load to determine the most loaded compute-node;
4      Pick the load of the most loaded compute-node ($node_{mostloaded}$);
5      Obtain $PD_{threshold}$ and the average load from the system (AvgLoad);
6      Calculate $x$ ($x = node_{mostloaded}$ - AvgLoad);
7      Define an integer variable for storing PD decision ($PD_{decision}$) and initialise it to
       zero;
8      if x $\geq PD_{threshold}$ then
9          set $PD_{decision} \leftarrow TRUE$;
10         send $PD_{decision}$ to WORKERS
11     end
12     else
13         set $PD_{decision} \leftarrow FALSE$;
14         send $PD_{decision}$ to WORKERS
15     end
16 end
17 else if am worker then
18     Send my ID and my load to MASTER;
19     Receive $PD_{decision}$ from MASTER;
20     if $PD_{decision} == TRUE$ then
21         Perform DLB;
22     end
23 end
24 Resume Normal Simulation;
```

3.7.4. Work Transfer Calculation

This step is the core of load balancing operation so it would be wise to ensure that load balancing is profitable before carrying out this operation. work transfer can either be calculated locally by the individual compute-nodes or centrally by a designated compute-node, having obtained the load from all the computers. Both strategies are investigated in this thesis. As discussed in section §3.5.1, the primary load index in Madcity is vehicles but since another load index has been designed in this thesis, load can refer to either *vehicle* or *response time*.

The aim of work transfer calculation is to correct load imbalance by estimating how much load each compute-node should give out or receive in order to maintain the average load (*AvgLoad*), which every processor should ideally have. To achieve this, *AvgLoad* is compared with the current load of a processor (*SysLoad*) to estimate how much a processor has deviated from the system average-load-per-node. That is,

Load imbalance $= SysLoad - AvgLoad.$

The Centralized (MaS) *Algorithm*

In this algorithm, the designated decision making compute-node creates an array in which it places the IDs of the compute-nodes in the order as they appear in the parallel simulation and then perform load balancing decisions on them. That is, compute-node ID 1 is the first in the array, and so on. The left and right neighbours are represented by array index of myID-1 and myID+1 respectively. Compute-node one has no myID-1 neighbour and the last compute-node has no myID+1 neighbour. Load balancing decisions are performed on this array after which the array called LB_{Advice}, is broadcasted to all the compute-nodes. Note that LB_{Advice} is initialed at the beginning of the algorithm as the values are not stored to influence future load balancing operations. The interesting thing with this algorithm is that load balance is achieved all in one time step. The algorithm is based on the most loaded processor in the system. The most loaded processor is picked, load is redistributed between the neighbours and then searched again until all the processors have been able to redistribute their loads.

In load relocation, the most loaded processor can either find the least loaded processor and then relocate its load directly to it (direct method) or it can do that in a stepwise manner through its immediate neighbours (stepwise method). Either method has its own advantages and disadvantages. For the direct method, the process of relocation is quick but the resulting road network could lead to poor performance since the general structure of the network would be disrupted such that a partition might end up having more neighbours than it had previously. This could eventually add to the complications and the communications overhead of sending vehicles from one partition to another. The stepwise method on the other hand may take longer to balance the load but the resulting road network boundaries may be preserved, leading to better performance.

The algorithm is as follows:

Algorithm 3: The Work Transfer Calculation of Centralized Algorithm

```
Input: Set of tasks and processors
Output: Load Balance Advice, $LB_{Advice}$
1  . if Am MASTER then
2       Receive ID and load from every participating compute-node;
3       Sort the load to determine the most loaded compute-node ($Node_{mostLoaded}$);
4       (in this way, the left and right neighbours is ascertained);
5       for $i \leftarrow$ to $l$ do
6           Pick $Node_{mostLoaded}$;
7           Get its neighbours;
8           Calculate $x$ ($x = Node_{mostLoaded}$ - AvgLoad);
9           Redistribute $x$ to the neighbours of $Node_{mostLoaded}$;
10          Remove $Node_{mostLoaded}$ from the list;
11          Sort the load of the remaining processors;
12      end
13 end
14 else if am worker then
15      Send ID and my load to the MASTER process;
16 end
17 Resume Normal Simulation;
```

3.7.5. Load selection and Load Migration

The *selection policy* selects the most suitable processes for transfer while the *migration policy* performs the actual reallocation of load between the participating compute-nodes [115]. The selection policy takes into account the transfer overhead, and the extra communication overhead that may be incurred as a result of the load redistribution in the load balancing operation. For example, in traffic simulations, the splitting of tightly coupled road junctions could generate high communication requirements during computer computation and consequently may outweigh the benefits of load balancing.

One way to re-balance the load is to repartition the road network using one of the static partitioning algorithms [18]. E.g parallel algorithms such as JOSTLE or ParMETIS are able to partition large mesh very rapidly [102][91]. The difficulty with this method is in preserving the initial partition boundaries and creating additional boundaries could result in higher communication overhead.

For example, ParMETIS was able to partition a mesh of the order of 1 million nodes in less than 2 seconds on 128 PEs of a Cray T3D [70]. However it is important, but difficult, to ensure that the new partitioning will be “close” to the original partitioning. Should the new partitioning deviate considerably from the old one then the cost of transferring large amounts of data will be incurred. It has been found that repartitioning is more appropriate when there has been a substantial localized refinement on the mesh [72,5]. Any of the fast parallel graph partitioning algorithms of section 2.2.1 may be employed. These include the parallel multilevel schemes.

To minimize the data movement resulting from the repartitioning, a number of techniques have been used to modify the graph partitioning algorithms, so that the new partition is as close to an existing partitioning as possible. The idea of a virtual vertex was used in [105] [34]. A virtual vertex is associated with each subdomain and is connected to each vertex in the subdomain by a virtual edge. The weight of this edge reflects the communication cost of migrating the vertex. By partitioning the combined graph, the dual objectives of minimizing edge-cut and data movement are considered at the same time. In [92], the idea of local matching was used where matchings were restricted to vertices that have the same processor. This strategy biases the multilevel graph partitioner towards an existing partitioning, thus reducing data movement resulting from the repartitioning. A re-mapping algorithm can be applied after the repartitioning to allocate the subdomains to the most appropriate processors, in order to reduce the data movement [90][82].

An alternative strategy to *repartitioning* [18] is to *migrate* [20] the excessive nodes to neighboring compute-nodes, effectively shifting the boundaries to achieve a balanced load. This approach may potentially cause less movement of data than repartitioning, although the edge-cut after the migration could possibly be larger than that given by a global repartitioning. Therefore care must be taken to keep edge-cut down when choosing the nodes to be migrated. It has been found [90] that this strategy is more suitable when the load imbalances caused by the refinement are low, or when localized high imbalances occur throughout the mesh. This is because in such cases the optimal partition will be relatively close to the initial partition. The process of migrating loads between compute-nodes to achieve load balance can be broken down into two distinctive steps [102][103]:

- Flow calculation: Each processor works out a schedule for the amount of load that should be sent to (or received from) its neighboring compute-nodes. This is referred to as a flow calculation in [102].
- Node selection: Once the flow is worked out, each compute-node decides which mesh nodes should be sent or received, to satisfy the flow as well as to minimize the edge-cut.

Load migration has been studied by a number of authors. A popular strategy is to start from the boundary nodes and gradually move to the interior of the road network, until enough nodes are marked for migration [105][104][58]. A load migration library was also developed in [20]. As a strict rule, only the boundary junctions need to be moved between the compute-nodes. Load migration is very complex and the exact details depend on the applications and the data structures used. In many adaptive computations, the amount of data associated with each mesh node may be quite large. The time for the migration of the data, and subsequent cost of updating the data structures, can dominate overall run time, specially if there is a need to re-balance the load frequently.

3.8. The Design of Centralized *(MaS)* and Distributed Algorithms

The taxonomy described in section §2.2 and the building blocks designed in section §3.7 were applied in the design in this section. Though most of the designs in section §3.7 are master-slave in structure, some of the ideas can be adapted to distributed algorithms. A combination of different strategies can be employed to yield different load balancing algorithms.

Four dynamic load balancing algorithms were investigated. Two of the designs are considered to be the basic *centralized* and basic *distributed* algorithms. *Basic* means that the structure of the algorithms are as described in the literature review of dynamic load balancing. Two other algorithms, are an extension of the basic centralized and distributed algorithms. The extended algorithms are novel in that the new designs of *profitability determination*, *load index* and *work transfer calculation* were used in the algorithms.

The motivations to extend these two algorithms are:

- To study the behaviours of the designed building blocks in two algorithms from different classifications (centralized and distributed) of algorithms.

These two algorithms are also the ones used in the reviewed traffic simulators.

- To know if any of these classes of algorithms are most suitable to parallel traffic simulations.

The distributed algorithm considered falls into the class of *diffusion* algorithms because the processors balance their workloads with their neighbours simultaneously. In the discussions that follow, *diffusion* and *distributed* are used interchangebly while *master slave (MaS)* and *centralized* are also used interchangebly.

Applying vehicles load index *(VLI)* and response time load index *(RTLI)* of section §3.7.1 to the centralized algorithm yields 2 variants of the algorithm–one with *isolated* load index, Basic Centralized Algrithm (BCA), and the other with *integrated* load index, Extended Centralized Algorithm (ECA). The former is the one most commonly used in parallel road traffic research but as explained in section §3.7.1, it lacks major features which make it unsuitable especially for heterogeneous clusters. Similarly, there are 2 variants of the distributed algorithm–one with *isolated* load index, the Basic Distributed Algorithm (BDA), and the other with *integrated* load index, the Extended Distributed Algorithm (EDA).

3.8.1. SPMD and Load Balancing Algorithms

Most of the alorithms designed below has a master and worker component because they are designed for Single Program Multiple Data (SPMD) applications. Madcity, the parallel Madcity simulator (discussed in section 3.10.1) is an SPMD application. An SPMD program uses a master-worker paradigm in which the master/controller takes the supervisory rule while the workers do the work. A distinction is made between the different sections of the program by selective *if* statements. Often, all the processes work on different part of the data while the same program resides on all the processors. SPMD parallel applications are easier to develop as only one source code is developed to run on all the processors. The control structure of an SPMD program is expressed as follows:

```
Program
    If (process is the Master/Controller) then
      /*controller program constructs*/
    else
        /*worker program constructs*/
    endif
End
```

3.8.2. Basic Centralized Algorithm (BCA)

BCA is *dynamic, centralized, migration-based, and isolated.* The first step in parallel processing is to partition the network files and distributed among the participating processors for simultaneous execution. As execution proceeds, however, vehicles transfer from one network partition to another. At the time of load balancing, the load indexes are sent by each compute-node to the decision making compute-node that executes the load balancing procedure.

The basic algorithm is also *periodic* and *individual.* The algorithm is activated periodically at a set period p. The goal of the algorithm is always to balance the load of the entire system at every invocation but before it does that, it carries out a profitability determination to access if load balancing is profitable or not. At every period, if any overloaded processor is detected and above a threshold, $PD_{threshold}$, a new distribution of road network and hence vehicles is computed.

In this strategy, the load-balancing algorithm is executed at a single predefined processor. The central decision making processor that is responsible for load balancing computes a new balanced load distribution, and then informs the other participating processors of the *load balance advice*, LB_{Advice}. The participating processors do local work transfer only between the neighbour processors.

The Pseudo Code of the Basic Centralized Algorithm (BCA)

1. At periodic DLB initiation time, each compute-node measures its vehicles load index *(VLI)*.
2. Each compute-node sends its VLI to the decision making compute-node.
3. The decision making compute-node collects ID and VLI from all the compute-nodes.
4. The decision making compute-node then initialises LB_{Advice} to zeros, calculates load relocation, using VLI, and stores the results in LB_{Advice}.
5. The decision making compute-node broadcast the result of the LB_{Advice} to all the compute-nodes.
6. Each compute-node receives LB_{Advice} from the decision making compute-node, extract information for its left and right neighbours into $Load_{left}$ and $Load_{right}$ respectively.
7. Each compute-node selects load to meet the requirements of its left and right neighbours.
8. Each compute-node migrates vehicles to the left and right neighbours.
9. Each compute-node inserts the vehicles and reconstruct the road network.

The Basic Centralized Algorithm

Algorithm 4: The Basic Centralized Algorithm (BCA)

```
   Input: Set of tasks and processors
   Output: Load Balance Advice ($LB_{Advice}$)
1  if Am MASTER then
2      Receive VLI and ID from every participating compute-node and store in an array;
3      $n$ is the total number of participating compute-nodes;
4      Sort the array to reflect the taskIDs of the compute-nodes (i.e. compute-nodes with
       taskID = 1 is the first item in the array, etc);
5      for $i \leftarrow$ to $n-1$ do
6          Sort the VLI to determine the most loaded compute-node (and determine the
           left and right neighbours);
7          Pick the most loaded compute-node ($Node_{mostloaded}$);
8          Calculate $x$ ($x = Node_{mostloaded}$ - AvgLoad);
9          Initialize $LB_{Advice}$ array with zeros and compute the load for the right and left
           neighbour of each $Node_{mostloaded}$ in the new initialized array;
10     end
11     Broadcast to all WORKERS load balance advice ($LB_{Advice}$);
12 end
13 else if am WORKER then
14     Send MyID and VLI to the MASTER process;
15     Receive $LB_{Advice}$ from MASTER ;
16     repeat search for myID until when myID is found
17     Extract information for my left and right neighbours into $Load_{left}$ and $Load_{right}$;
18     $Load_{left}$ = vehicles to relocate to left neighbour(identified by myid-1 in the
       $LB_{advice}$ array);
19     $Load_{right}$= vehicles to relocate to right neighbour(identified by myid+1 in the
       $LB_{advice}$ array);
20     Select vehicles to meet the requirements of left and right neighbours;
21     Migrate vehicles to the left and right neighbours;
22     Insert vehicles, reconstruct the road network;
23 end
24 Resume Normal Simulation;
```

3.8.3. Extended Centralized Algorithm (ECA)

ECA is *dynamic, centralized, migration-based, and integrated.* The extended centralized algorithm(ECA) differs from basic centralized algorithm(BCA) in the following:

- BCA uses isolated load index (VLI) while ECA uses integrated load index (RTLI).
- There is no PD in BCA while ECA has a PD component. Note however that when comparing the 2 algorithms, PD is sometimes used in BCA. When that is done, there would be a distinction between BCA with PD (BCA_{PD}) and BCA with no PD (BCA). In this case, (BCA_{PD}) and ECA differs only in their load index.

The Pseudo Code of ECA

1. At periodic DLB initiation, each compute-node measures the time per simulation step (response time load index, RTLI). It uses the value to calculate the run-time per vehicles per step ($RT_{pervehicle}$) using equation 3.2.
2. Each compute-node sends the ID and *RTLI* to the decision making compute-node.
3. The decision making compute-node receives *RTLI* and IDs from all the compute-nodes. It performs profitability determination.
4. If profitable to balance the load, it stores a value of 1, meaning TRUE, in $LB_{decision}$ else it stores 0, meaning FALSE. The decision making compute-node performs work transfer calculation, using *RTLI*, and stores the load relocation information for every compute node in an array called LB_{Advice}. $LB_{decision}$ and LB_{Advice} are broadcast to all the compute-nodes.

5. Each compute-node receives $LB_{decision}$ and LB_{Advice} from the decision making compute-node.

6. If $LB_{decision}$ is TRUE, each compute-node extracts, from LB_{Advice}, load for its left and right neighbours into the variables $Load_{left}$ and $Load_{right}$ respectively. Load is in seconds.

7. $Load_{left}$ and $Load_{right}$ are converted back into equivalent number of vehicles using equation 3.3.

8. Each compute-node uses the value in step (7) to decide on the load/vehicles to relocate to its neighbours in order to achieve balanced load.

9. Each compute-node performs the actual load migration (if there is load to migrate) to the neighbours.

10. The new network structure is broadcast to all the compute-nodes to update their road network information.

The Extended Centralized Algorithm (ECA)

Algorithm 5: The Extended Centralized Algorithm (ECA)

```
   Input: Set of tasks and processors
   Output: Balanced Load
 1 if Am MASTER then
 2     Receive $RTLI$ and $ID$ from every participating compute-node;
 3     $n$ is the total number of participating compute-nodes;
 4     Sort the array to reflect the taskIDs of the compute-nodes (i.e compute-node with taskID = 1 is
       the first item in the array, etc);
 5     Find the average load (AvgLoad) and read $PD_{threshold}$;
 6     Sort the load to determine the most loaded processor ($Node_{mostLoaded}$);
 7     Initialize $LB_{Advice}$ array with zeros;
 8     Set $PD_{decision}$ = FALSE;
 9     for $i \leftarrow$ to $n - 1$ do
10         Calculate $x$ ($x = Node_{mostLoaded}$ - AvgLoad);
11         if x $\geq PD_{threshold}$ then
12             Set $PD_{decision}$ = TRUE;
13             Sort the array to determine the most loaded processor;
14             Pick $Node_{mostLoaded}$;
15             Compute the load for $Node_{mostLoaded}$'s right and left neighbours, store in $LB_{Advice}$;
16             Broadcast to all WORKERS $PD_{decision}$ and $LB_{Advice}$;
17         else
18             Broadcast to all WORKERS $PD_{decision}$ and $LB_{Advice}$;
19         end
20     end
21 end
22 else if am the WORKER then
23     Measure my time per sim step i.e response time load index (RTLI);
24     Calculate $RT_{pervehicle} = RTLI/n$;
25     Send MyID and $RTLI$ to the MASTER process;
26     Receive $PD_{decision}$ and $LB_{Advice}$ from MASTER ;
27     if $PD_{decision}$ == TRUE then
28         repeat search for myID until when myId is found
29         Extract load for my left and right neighbours into $Load_{left}$ and $Load_{right}$;
30         Left neighbour vehicles = $Load_{left}/RT_{pervehicle}$;
31         Right neighbour vehicles = $Load_{right}/RT_{pervehicle}$;
32         Select vehicls to meet the requirements of my left and right neighbours;
33         Migrate vehicles to the left and right neighbours;
34         Insert vehicles into the road network;
35         Update the road network structure;
36     end
37 end
38 Resume Normal Simulation;
```

3.8.4. The Basic Diffusion Algorithm (BDA)

The algorithm is *local, neighbourhood-based, distributed, synchronous, periodic, individual, and non-blind.* In BDA, all compute-nodes simultaneously compute the new load distribution and so no need to be relayed between the participating compute-nodes. The notion of neighbourhood is defined by each compute-node, according to the network partitions that are shared between any 2 or more compute-nodes.

The algorithm is also *dynamic, migration-based, and isolated.* Vehicles are initially distributed among the participating compute-nodes. At the beginning of the simulation, the SPMD master process estimates the average vehicles, *AvgLoad*, that should be on each compute-node and stores this in a variable accessible to all the worker nodes. Also, every compute-node keeps a record of the total vehicles in its partition in $Node_{vehicles}$ and this variable is updated at the end of every simulation time-step. On initiation of load balancing routines, each compute-node obtains its current VLI from $Node_{vehicles}$ as well as the average load from *AvgLoad.* Comparing these two values, it estimates how much should be given to its neighbours or receive from its neighbours. The basic idea is: using the current total number of vehicles and the expected average vehicles per compute-node, calculate the difference between these two values. The difference represents the deviation from the ideal load (i.e. the surplus or the deficit). A negative value indicates deficit and a positive value indicates overload. How under loaded or overloaded depends on its deviation from *AvgLoad.* This value has to be compensated either by sending or receiving an equivalent number of vehicles between the neighbours. Work transfer calculation operation is performed until an optimal load distribution is estimated.

The Work Transfer Calculation of Distributed algorithm

In a distributed algorithm, because there is no central supervision, the most important aspects that work transfer calculation need to consider are the following:

- Direction of load flow
- Load thrashing. i.e. a situation whereby a processor relocates load back to the processor that just relocates load to it. This can result in an endless load relocation between two respective neighbours and must be avoided.

The Algorithm for the distributed work transfer calculation is shown below. This algorithm is executed by every participating processor. The *if* selection statements are such as to prevent load thrashing. Load relocation between the neighbours is carefully considered by each compute-node.

Algorithm 6: Work Transfer Calculation in Distributed Algorithm

```
Input: Set of tasks and processors
Output: Load calculation
1  Read my load index and AvgLoad from the variables;
2  Calculate $x$ = My Load index - AvgLoad;
3  if No left Neighbour then
4      Load flow is right;
5  end
6  else if No right Neighbour then
7      Load flow is left;
8  end
9  else if Both left and right neighbours then
10     if I have received load from left then
11         Load flow is right
12     end
13     else if I have received load from right then
14         Load flow is left
15     end
16     else
17         Pick the least load and then flow in that direction;
18         Determine the number of processors before the least load (N);
19         for $i = 1$ to $N$ do
20             Perform stepwise load redistribution;
21         end
22     end
23 end
24 Resume Normal Simulation;
```

The Pseudo Code of the Basic Diffusion Algorithm (BDA)

1. On periodic invocation of dynamic load balancing operation every compute-node participating in the parallel simulation estimates its vehicles load index (VLI).

2. Every compute-node compares its VLI with the average vehicles, *AvgLoad*,

and calculates x which is the difference between VLI and *AvgLoad*. x is sent to neighbours.

3. Each compute-node collects x from the neighbours and analyses load imbalance. It then decides how much load should be moved between its respective local neighbours to achieve balance. This step is iterative until load is balanced.

4. Vehicles are then selected for relocation. Vehicles selection strategy is as follows:

    - Collect the boundary nodes and assign them to an array.
    - Within the boundary nodes, estimate which vehicles, when selected best correct the load imbalance.
    - Select the corresponding junctions of the vehicles for relocation.

5. Load Migration Strategy

    Each compute-node performs the actual migration by reassigning the selected junctions with the IDs of the destination processor. The vehicles are then sent to the respective neighbours. The new boundary relocation information is then sent to all the neighbours to update their road network.

6. The vehicles are received by the neighbours and inserted back into the road network in the new partition.

All of the strategies are local i.e. all the load balancing decisions are performed in the individual computers.

The Basic Diffusion Algorithm (BDA)

Algorithm 7: The Basic Diffusion Algorithm (BDA)

```
Input: Set of tasks and processors
Output: Load Balancing Decision
1  if Am MASTER then
2      Do nothing
3  end
4  else if am worker then
5      Read the average load (AvgLoad) of the system;
6      Read the vehicles load index (VLI);
7      for $i \leftarrow$ to $l$ do
8          Initialize send/receive buffers for left ($load_{left}$) and right ($load_{right}$) neighbours;
9          Calculate $x$ ($x$ = VLI - AvgLoad) and store $x$ in $load_{left}$ and $load_{right}$ ;
10         Send $load_{left}$ and $load_{right}$ buffer to my $left_{neighbour}$ and $right_{neighbour}$
           respectively;
11         Receive buffer from my $left_{neighbour}$ and $right_{neighbour}$;
12         Analyse the load by comparing my $x$ with the $x$ of left and the right neighbours
           and determine how $x$ should be redistributed to neighbours;
13         Send result of analysis to my $left_{neighbour}$ and $right_{neighbour}$;
14         Receive analysis result from my $left_{neighbour}$ and $right_{neighbour}$;
15         Update my load with left and right neighbour load;
16     end
17     Migrate vehicles to the left and right neighbours;
18     Insert vehicles into the road network;
19     Update the road network structure with new partition IDs;
20     Send the new network structure to the neighbours to update their network;
21 end
22 Resume Normal Simulation;
```

3.8.5. The Extended Diffusion Algorithm (EDA)

The basic diffusion algorithm was extended with global profitability determination and response time load index (RTLI) designed in this thesis. The basic Diffusion algorithm is entirely local such that each compute-node is only aware of the information of its neighbours. As a result, it is observed that in a synchronous diffusive distributed algorithm, a local profitability determination is a redundant feature. That is, the feature plays very little role in improving the performance of the system and most often it could lead to performance degradation. This is because in synchronous diffusion algorithms, barrier synchronization is implemented using blocking send/receive messages, meaning that every send must have a matching receive. The problem with this method is that since PD decisions are made between local neighbours, there is a possibility that some groups could 'think' it profitable to perform load balancing while some other groups could 'think' the opposite. In such a situation, there would be a deadlock when the compute-nodes that went ahead with load balancing try to communicate with the neighbours that did not engage in load balancing at this time.

The implications of this to PD is that every compute-node goes through the local PD routines but waits at the synchronisation points to receive from the neighbours before proceeding beyond that point. Hence, PD is lower bounded by the slowest machine or the most loaded processor in the system leading to increased overheads. It is perhaps better for the compute-nodes to go ahead with load balancing at each initiation without incurring the PD overheads.

In global PD, however, the decision making compute-node gathers information from all the compute-nodes and then makes a quick decision as whether to proceed with load balancing or not. This is then broadcast to all the compute-nodes. If it is profitable to load balance, every further action by the compute-nodes from here is between the local neighbours. Hence, only the PD aspect of the algorithm is global.

Another advantage of global PD is that since the response time load index depends on the average load of all the compute-nodes (Algorithm 1 of section § 3.7.1), it cannot work well in a fully distributed system where the compute-nodes only have knowledge of their neighbours. Global PD makes the usage of response time in a distributed algorithm possible by calculating the average load and storing it in a variable for later use.

Pseudo Code of the Extended Diffusion Algorithm (EDA)

1. On periodic invocation of the dynamic load balancing operation every compute-node participating in the parallel simulation reads its response time load index *(RTLI)*. It then uses the equation 3.2 to calculate $RT_{Vehicle}$
2. Each compute-node sends *RTLI* to the decision making compute-node.
3. The decision making compute-node collects the RTLI and the IDs from all the compute-nodes, performs a cost-benefit analysis and broadcast the $PD_{decision}$ to all the workers.
4. The workers collect $PD_{decision}$ from the decision making compute-node and if profitable to load balance goes ahead with load balancing or exit otherwise.
5. Every compute-node obtains the average load, *AvgTime*, calculated by PD above.
6. Every compute-node compares its *RTLI* with *AvgTime* and calculates x
7. It packages x and send to neighbours. It also receives from neighbours and determine how much load should be moved between its respective local neighbours to achieve balance. This step is iterative and should terminate when load evenness is achieved. Once completed, the processors convert time into vehicles by using the relation in equation 3.3.
8. Load Selection Strategy
    - Collect the boundary nodes and assign them to an array.
    - Within the boundary nodes, estimate which load, when selected best correct the load imbalance.

- Select the appropriate junctions for migration.

9. Load Migration Strategy

   Performs the actual migration by reassigning the selected junctions with the IDs of the destination processor. The boundary relocation information is then sent to all the neighbours.

Extended Diffusion Algorithm (EDA)

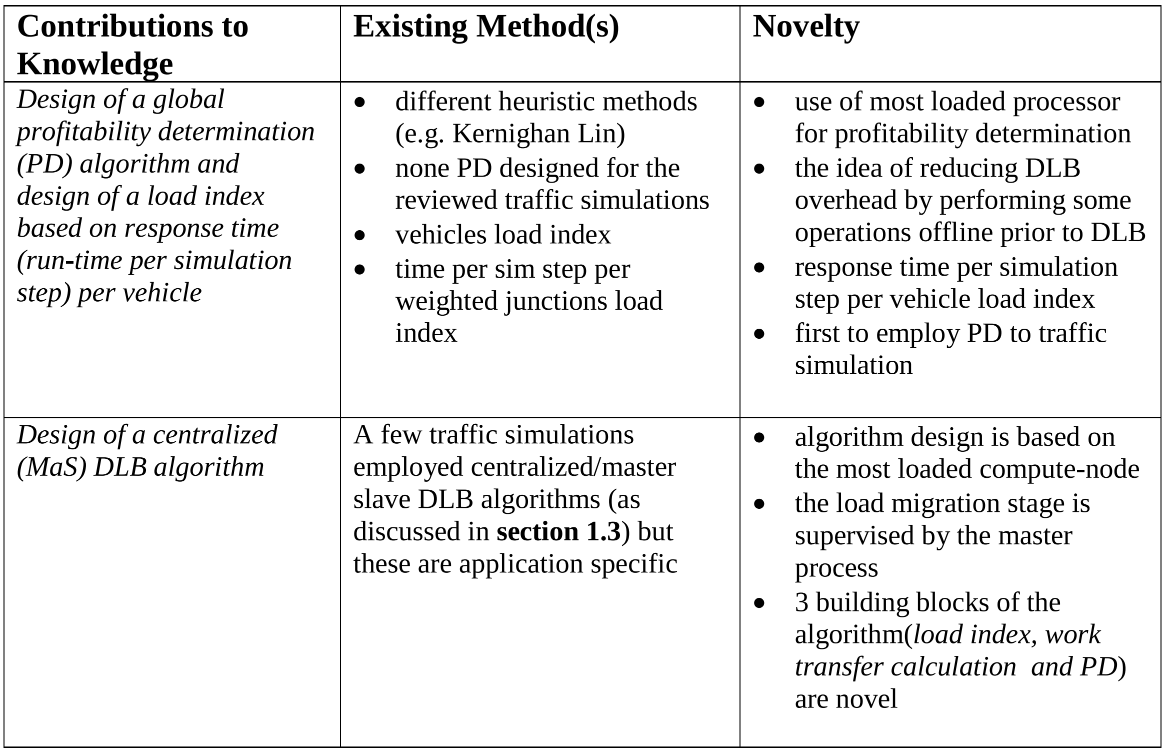
Algorithm 8: Extended Diffusion Algorithm with global PD (EDAG)

```
    Input: Set of tasks and processors
    Output: Load Balancing Decision
 1  if Am MASTER then
 2      Receive RTLI and ID from every participating compute-node;
 3      Set $PD_{decision}$ = FALSE;
 4      for $i \leftarrow$ to $l$ do
 5          Pick the most loaded compute-node ($Node_{mostloaded}$);
 6      end
 7      Calculate the average load, AvgTime. Obtain $PD_{threshold}$ from the system and calculate $x =$
        $node_{mostloaded}$ - AvgTime;
 8      if x $\geq PD_{threshold}$ then
 9          Set $PD_{decision}$ = TRUE;
10          Broadcast $PD_{decision}$ to all WORKERS;
11      end
12  end
13  else if am worker then
14      Send RTLI and ID to the MASTER process;
15      Receive $PD_{decision}$ from MASTER ;
16      if $PD_{decision} == TRUE$ then
17          Obtain the average load (AvgTime) calculated by PD above;
18          $RT_{pervehicle} = \frac{RTLI}{node_{vehicles}}$;
19          for $i \leftarrow$ to $l$ do
20              Initialize send/receive buffers for left ($load_{left}$) and right ($load_{right}$) neighbours;
21              Calculate $x$ = RTLI - AvgTime and store $x$ in $load_{left}$ and $load_{right}$ ;
22              Send $load_{left}$ and $load_{right}$ buffer to $left_{neighbour}$ and $right_{neighbour}$ respectively;
23              Receive buffer from $left_{neighbour}$ and $right_{neighbour}$;
24              Analyse the load by comparing $x$ with the $x$ of left and the right neighbours;
25              Determine how $x$ should be redistributed to neighbours;
26              Send analysis result to $left_{neighbour}$ and $right_{neighbour}$;
27              Receive analysis result from $left_{neighbour}$ and $right_{neighbour}$;
28              Update RTLI, $load_{left}$ and $load_{right}$ ;
29          end
30          Convert $load_{left}$ into vehicles (Left neighbour vehicles = $load_{left}/RT_{pervehicle}$);
31          Convert $load_{right}$ into vehicles (Right neighbour vehicles = $load_{right}/RT_{pervehicle}$);
32          Select load to meet the requirements of my left and right neighbours;
33          Migrate vehicles to the left and right neighbours;
34          Insert vehicles into the road network;
35          Update the road network structure with new partition IDs;
36          Send the new network structure to the neighbours to update their network;
37      end
38  end
39  Resume Normal Simulation;
```

The Extended Diffusion Algorithm with global PD (EDAG) differs from Basic Diffusion Algorithm(BDA) in the following:

- BDA uses isolated load index (VLI) while EDAG uses integrated load index (RTLI).
- There is no PD in BDA while EDAG has a PD component. Note however that when comparing the 2 algorithms, PD is sometimes used in BDA. When that is done, there would be a distinction between BDA with PD (BDA_{PD}) and BDA with no PD (BDA). In this case, (BDA_{PD}) and EDAG differs only in their load index.

3.9. Summary of Contributions to Knowledge



Contributions to Knowledge	**Existing Method(s)**	**Novelty**
Design of a global profitability determination (PD) algorithm and design of a load index based on response time (run-time per simulation step) per vehicle	• different heuristic methods (e.g. Kernighan Lin) • none PD designed for the reviewed traffic simulations • vehicles load index • time per sim step per weighted junctions load index	• use of most loaded processor for profitability determination • the idea of reducing DLB overhead by performing some operations offline prior to DLB • response time per simulation step per vehicle load index • first to employ PD to traffic simulation
Design of a centralized (MaS) DLB algorithm	A few traffic simulations employed centralized/master slave DLB algorithms (as discussed in **section 1.3**) but these are application specific	• algorithm design is based on the most loaded compute-node • the load migration stage is supervised by the master process • 3 building blocks of the algorithm(*load index, work transfer calculation and PD*) are novel

Figure 3.7: Summary of contributions to knowledge

A table summarizing the contributions to knowledge are presented in Fig. 3.7.

In the design of load index, the method adopted in [88] also considers the wall-clock timing in its load estimation but it is different from this design in that:- in this algorithm, time per sim-step is used with vehicles whereas in [88] time per sim-step is used with assigning each vertex with half of the weights of the incident edges. It does this by timing the simulation step of execution and combines this with the load of the simulation, estimated by summing the weights of the network vertices, to determine the load index. In this case, number of vehicles is not directly used as the load of the system. This is because the authors observed that the simulation speed of the simulator weakly depends on vehicles. For the design in this thesis, time is converted back to vehicles because this is the basic unit of load in Madcity. Here, response time is directly related to vehicles as indicated in section §3.5.1.

3.10. Implementation of the Algorithms

This section presents the actual implementations of the parallelization of Madcity traffic simulator and the dynamic load balancing algorithms using the designs in the previous section.

3.10.1. Parallelization of Madcity Traffic Simulator

The parallelization of Madcity traffic simulation was essential to study the behaviour of the dynamic load balancing algorithm/methods that were designed. The design and implementation of the parallel Madcity was the first major part of this research project. After the parallelization, several experiments were conducted to test the working of the parallel application. Like any software engineering project, the design of the parallel traffic simulation was first carried out and this was followed by the implementation. These are discussed in the following sections.

Domain Decomposition

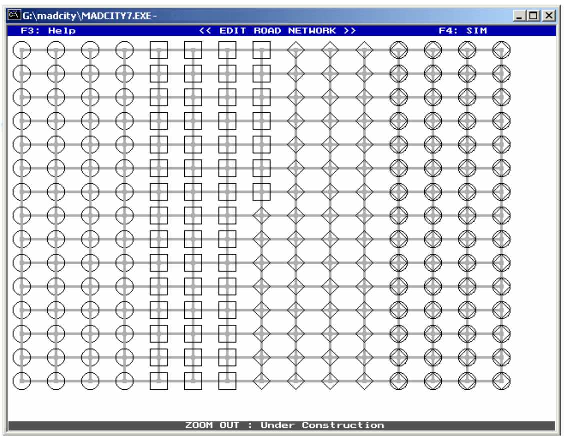


Figure 3.8: Illustration of Partitioned Network Through Domain Decomposition

An important step in parallel processing is the decomposition of the problem domain or the data into different sub-domains [120]. The sub-domains are distributed to the compute-nodes on the cluster, for simultaneous parallel executions. A consideration here is to keep the partition sizes as even as possible.

In Fig. 3.8, the road network topology has been partitioned into four sub-networks as indicated by the different shapes of the junctions comprising the road network. Here, junctions of similar shapes belong to the same partition and hence will be simulated on the same compute-node during parallel simulation. This implies that the road network will be simulated on four compute-nodes.

A critical issue in the partitioning of the road network is the number of links spanning different partitions. In decomposing a road network into multiple sub-networks, traffic continuity is potentially disrupted. During run-time, when a vehicle arrives at the boundary point in a lane, where one sub-network ends and another one begins, the vehicle must be transferred seamlessly to the computing processor where the vehicle can continue its journey. This vehicle transfer must be coordinated in time and space such that the integrity of the simulation remains intact.

Also, the transfer of vehicles from one partition to another involves message passing communication, which is expensive. It would be useful to ensure that the number of links adjoining the partitions are minimized. The difficulty with this is that minimising the number of adjoining links between partitions often lead to load imbalance, another poor performance factor. Irrespective of the number of adjoining links, it would be useful to have a mechanism whereby all the vehicles going into the same partition at every synchronisation points are buffered and sent at once. This is achieved by the concept of Lane Cut Point (LCP), a concept developed with this thesis's director of studies, Nasser Kalantery, and presented in [53][54] . The LCP interface in conjunction with parallel coordination must meet these requirements. LCP helps to handle all the complications and time delay associated with transferring vehicles between neighbour partitions, by buffering the vehicles before sending them to the destination processor.

Lane Cut Points (LCPs)

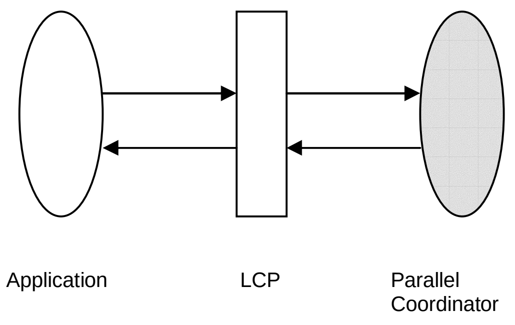


Figure 3.9: Illustration of Lane Cut Point (LCP) Concept

Once the road network topology is partitioned for concurrent sub-simulations, two major run-time issues must be dealt with; communications and synchronisation. Assuming that these two issues are resolved, a coordinated execution of the whole simulation will be achieved. However, organizing and implementing synchronisation and communication requires expertise that an application programmer may not necessarily possess. It would be useful to have a mechanism whereby the parallel coordination concerns could be separated from application development issues, such that, two different sets of entities could meet across a common interface and yet be able to work independently, each in their own familiar area. The concept of LCPs was developed to serve such a purpose. LCP is a data structure, which encapsulates vehicle data at the partition edges and discrete-time synchronisation is achieved using LCP. The LCP concept is depicted in Fig. 3.9.

Advantages of LCP-based coordination

There are many benefits of the use of LCP concept.

1. The LCP standard interfaces provide a facility whereby decomposition of a simulation is developed and debugged on a single processor environment and hence complexities of debugging in a parallel environment are significantly reduced. It is necessary here to differentiate between a sequential simulator and the simulator making use of the LCP interface objects but running on one compute-node, for testing and debugging purposes.

   Use of the LCP library helps to develop an n = 1 system that has all the features of parallel simulator. Thus whenever vehicles, which are meant for the neighbour partitions are leaving a partition, these vehicles enter the LCPs and are then removed from the LCPs at the end of every simulation step. In this way, the behaviour of the parallelized simulator can be analysed in the n=1 version and the transition from the n=1 to the n=k (k>=2) parallel development effort is therefore minimized since most of the possible errors can be debugged in the n=1 version.

2. LCPs give the advantage of separating the application layers from the communications/coordination layers that is required for the parallel version of the simulator. LCPs provide an open interface such that a given simulation application could be coupled with alternative communication layer and vice versa, such that different simulations could use the same

LCP interface for parallelization purpose. To accomplish this objective, the simulator developers need to develop their applications to conform to the interface definitions published in the LCP documentation.

3. Also, use of LCPs can reduce the communication overhead associated with parallel programming. Instead of sending individual vehicles across the partitions, vehicles meant for a particular partition can be buffered in the LCP and all the vehicles transferred at once to their respective partitions. Communication overhead, which has a negative effect on the performance of a parallel system, can therefore be minimized by use of LCPs.

4. LCP suits the method of microscopic traffic simulations. At certain intervals, all the compute-nodes involved in parallel simulations need to synchronize with each other so as to maintain the continuity of the simulation. Also, the processors sometimes need to exchange the information of vehicle positions especially if the trace of the simulations needs to be combined into a single file for visualizations. Though the use of LCPs means that every vehicle has to be monitored to know when it needs to get into the neighbour partitions, it rather fits into the method of microscopic simulation where the behaviour of individual vehicle is important. LCPs fit into this simulation model because synchronisation takes place at the end of every simulation step, whereby vehicles are always checked to determine their current position. At this point, the vehicles are also checked to determine whether the vehicles should go into or out of an LCP.

5. Finally, LCP technology is not difficult to implement and so individual developers can easily adapt it to their programs. The fewer the number of LCPs however, the better the performance of the simulator. For maximum efficiency, the point at which the network partitions are made should be to minimize the number of lanes crossing the partition boundaries. This reduces the number of LCPs in the network and hence minimizes the

overhead associated with processing the vehicles in the LCPs at the end of every simulation step.

The Parallel Traffic Simulation Algorithm

The programming model used is SPMD (Single Program Multiple Data). This means that the same program image is available on all of the compute-nodes but each of compute-node has a different data set (i.e. the different domains of the partitioned road network).

The parallel program has three different phases. The first stage is the initialisation. At the initialisation, the master process reads the network graph and partition identifier from the 'network file', uses partition identifiers to find the positions of LCPs and create the LCPs, and then sends the data to the processes. Each process receives the network data, performs initialisations by creating the road network from the network data and creating vehicles on every road.

Following initialisation, simulation proceeds as a sequence of a predefined number of steps. Each step consists of two distinct phases: local execution phase and communication phase. In the local execution phase, each process simulates vehicles in its own partition and at the end of every simulation step, vehicles leaving a partition are passed to the LCPs. Then begins the communication phase, in which the processes check for vehicles in the LCPs, remove the vehicles in the LCPs (if any) and send them across to their destination processes where they are received and inserted into the appropriate roads. Vehicles need to be received at the destination process before the next simulation step.

Details of the parallelization strategy and the LCP concepts have been published and presented in two international conferences of the IEEE DSRT (Distributed Simulations and Real Time applications) [53] and DAPSYS (Distributed and parallel systems) [54].

The algorithm for the parallel simulator is presented below:

Algorithm 9: SPMD Structure of Parallel Traffic Simulation

```
   Input: Set of tasks and processors
   Output: SPMD Parallel Madcity
1  if Am MASTER then
2      Read the road network topology data from the network file;
3      Insert the LCPs on the road network boundaries between two partitions ;
4      Identify the total number of processes in the parallel simulations;
5      Send the road network to all the compute-nodes;
6      if trace == on then
7          Receive trace from all the workers;
8          Assemble the traces into one trace file;
9      end
10 end
11 else if am worker then
12     Receive the road network from the MASTER process;
13     Construct the road network topology;
14     Initialize lcp buffers;
15     Simstep is total number of simulation steps Insert the vehicles on the roads;
16     for i ← to Simstep do
17         Perform simulation on part of the sub-network by calculating the new distances of the
           vehicles at every step and moving the vehicles along the roads;
18         if vehicles are at the end of lane then
19             Remove the vehicles and put them in LCP buffer;
20         end
21         Send the LCP buffer to the neighbours;
22         Receive the LCP buffer from the neighbours;
23         Insert the vehicles (if any) in the LCP buffer into their respective roads;
24         if trace == on then
25             Record the simulation trace for the vehicles;
26             Send the simulation trace to the MASTER process;
27         end
28         Synchronize with the MASTER process at the end of every simulation step;
29     end
30 end
```

3.10.2. Synthesis of Parallel Traffic Simulator with Dynamic Load Balancing Algorithms

The parallelization of Madcity traffic simulation provides a suitable platform to *synthesize, test, study and compare the behaviour* of the proposed dynamic load balancing algorithms/methods. This section presents the implementations of the two dynamic load balancing algorithms according to the designs in the last chapter. The designs of these algorithms were motivated by the properties of traffic simulations, and the implementation was for parallel Madcity traffic simulations.

3.10.3. Synthesis of the Centralized(MaS) Algorithm

Load Evaluation

Measuring the vehicles load index in Madcity is fairly straight forward because each compute-node updates the local vehicle counter variable, $Node_{vehicles}$ at the end of every simulation step. Also, every junction keeps track of its total number of vehicles in the data structure and at the end of every simulation step, the number of vehicles in every junction is updated. This means that the load balancing routines do not incur extra overheard to obtain the load in the simulation. For response time load index, it can only be measured dynamically as the program runs.

In the load evaluation stage of the DLB, each processor needs to be aware of its load and also the average load of the entire simulation. These are recorded at the beginning of the simulation so the compute-nodes just have to read the values. Load balancing decisions are made with these two values. A compute-node is either a giver or receptor of load, depending on whether its load is greater or less than the average load.

Load Balance Initiation/Profitability determinations

Load balancing is initiated at specified load balancing periods according to a variable called *periodic*. If *periodic* is set at 2000, for example, it means that load balancing will be called at every 2000 simulation time steps. This value can be altered to vary the period of load balancing. On initiation, the master processor waits to receive the current state of the load from all the compute-nodes. It stores this information in an array that details the process id, and the current value of the load information received. It then analyses the load and decides if its worth carrying out the load balancing at this stage. If it is not profitable to load balance at this stage, normal simulation resumes.

Work Transfer Calculation

The assumption here is that equal load does not necessarily mean that all the partitions must have exact number of vehicles. A trade off in the number of vehicles will be made for better communication structure. The detailed algorithm on how the master process calculates the work transfer that should be reallocated between the different processors was presented in the design section.

Having collected all the information from the compute-nodes in step 1 and storing them in an array, the master process sends the received array to function *analyse_the_data()*. This function receives the array and then sorts the array in ascending order starting from the information for the first compute-node to the last compute-node. This sorting is necessary because in step one, the processors do not necessarily communicate with the master processor in sequential order. Also, this is very important because in the calculation of load, the algorithm refers to the left neighbour by the array index to the left (*myindex-1*) and the right neighbour by the array index to the right (*myindex+1*).

With this information, the function analyses the load to reallocate using Algorithm 3. For each of the compute-node, it subtracts the *AvgLoad* from the current vehicle value and stores this information into another array called the *inbalinfo* (*i.e. information required for balancing the load*), along with the ID of the compute-nodes. Using this array information, it goes through the algorithm to determine how much each compute-node needs to give out to its left and right neighbours. It Creates a new empty array known as *balinfo* and initialize all the values to zero. This array is used to store all the load balance advice that would be broadcast to all the compute processors. The final array, LB_{Advice} to be broadcast to all the compute-nodes is of the form in Fig. 3.10. This array is then broadcast to all the compute-nodes.

Processor ID	Vehicles for left neighbour	Vehicles for the right neighbour	This order repeats for all the processors …	

Figure 3.10: Illustration of Load Balance Advice Buffer

Load Selection

Network partitioning in the experimental parallel simulation is junction based, each junction belonging to a given sub-domain. Junctions with identical partition IDs constitute a sub-network. Roads do not have explicit partition identification but the junction data structure maintains a list of its entire exit links. Since every compute-node has the entire simulation road network (ref. section §3.6.2), moving a vehicle from one partition to another means reassigning the associated network junctions with its new partition ID, communicating this information with all the other compute-nodes to update their network information, and moving the vehicles in the roads to their new destinations. After this the new boundary buffers are recreated. The question then is which of the junctions are to be moved to satisfy load imbalance? Since each junction keeps a record of its number of vehicles, selecting a junction for transfer is deciding how many of the junctions satisfy the work transfer calculation. The individual processors do this locally.

Each of the compute-nodes receives the broadcast array from the master process and search for its process ID in the array. Once this is found, it uses it to identify its load balance advice, which comprise of the vehicles to give to the neighbours or receive from the neighbours. Each compute-node then calls function *analyse_the_network()* to process the best junctions to be moved to the neighbours. To select a junction for transfer, one needs to identify the effects that the load redistribution would have on the overall balance of the system and especially the resulting communication pattern after the load redistribution. Increased network links between two partitions means increase in communication overhead between the processors as discussed in §3.6.3. It is therefore important to identify the effects the network repartitioning would have on the overall network structure after the load balancing stage.

Load Migration

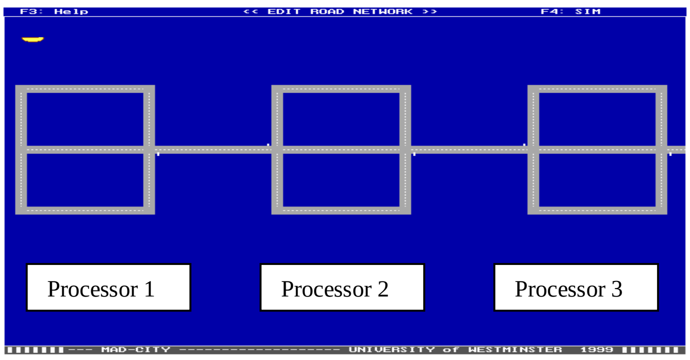


Figure 3.11: Illustration of Network Boundary Migrations

Having reassigned the junctions to their new partition IDs, the next step is the actual migration of the vehicles to all the neighbours. The compute-nodes send the vehicles needed for balance to the neighbours. The neighbours receive the information and unpack the vehicles into their respective lanes. As explained in section §3.6.2, since the parallel simulation uses domain replication, each compute-node has a copy of the whole network file and hence junctions are not physically moved from one compute-node to another. Moving a junction is the same as reassigning the junction's partition ID and sending the new network partitions information to all the participating compute nodes. This method further eliminates the communication overhead associated with physically moving junction's data structures between migrating domains.

Boundary migration suits the properties of traffic simulations because by migrating the boundary junctions, the destination (route plans) of the vehicles are not disrupted. As a strict rule, only the boundary junctions need to be moved between the compute-nodes. Migration from the boundaries eliminates 'orphan junctions', a situation whereby a relocated junction shares boundaries with the wrong partitions. Load balancing is just a way of providing more resources for the execution of the simulation by relieving the heavily loaded processors. To preserve the communication structure, each processor must first of all compile the list of all the boundary junctions between the neighbouring partitions. In Madcity traffic simulation, like in real life road traffic situations, vehicles are associated with roads and roads to junctions. Therefore, migrating junctions (i.e. assigning new partition IDs to junctions), enable the junctions, and roads to belong to the new partition domains. The new road network information is then communicated to all the other processors to update their network information. But there is also the need to move the vehicles across the partitions and insert them in the appropriate roads.

Migration of vehicles take place locally between neighbouring network partitions. This is so that a vehicle can maintain its journey when it crosses to the neighbouring partitions. Consider, for example, the simple network shown in Figure 3.11. If each of the boxes represent a road network on each processor, vehicle movements can only occur between the adjacent partitions say from processor 1 to processor 2 or from processor 2 to processor 3 and vice versa. Similarly, if processor 3 needs to reallocate some of its load to processor 1, it has to first give the excess load to processor 2 who will then give the load to processor 1.

This method is particularly important because in load balancing of traffic simulation schemes, the idea is not to disrupt the movement of vehicles from their intended destinations but simply to provide enough resources for the simulation. Boundary migration achieves this aim by reducing the load of the overloaded compute-node without disrupting the movement of the vehicles.

3.10.4. Synthesis of the Diffusion Algorithm

The *profitability determination, load selection and load migration* strategies are very similar to that of *Centralized Algorithm.* The basic difference is in the work transfer calculations and this is presented below:

Work Transfer Calculation

This step is entirely local. Each processor keeps record of two variables called Avg_{load}, which is the ideal number of vehicles that each compute-node should have to maintain load balance and the current load of the compute-node. On invocation of the load balancing operation, the compute-nodes take the difference (i.e. *Load index* - Avg_{Load}) between these two variables, and depending on the sign of the difference, decide if to give out or receive from its neighbours. Specifically, a negative value means that the compute-node contains less than average vehicles and therefore needs to expect from its neighbours while a positive value means that the compute-node has got vehicles in excess of the expected average and therefore needs to give out the excess to its neighbours. The interesting thing is that if a compute-node does not have enough to give out, it does not need to process anything. Rather, it should wait to receive whatever the neighbors have to give it. The algorithm is iterative until an optimal solution is achieved.

3.11. Chapter Summary

This chapter presented the parallelization of Madcity traffic simulation using the concept of Lane Cut Point (LCP). This was followed by the synthesis of 4 dynamic load balancing algorithms with the parallel traffic simulation. From the implementation point of view, the following conclusions were made about the algorithms:

1. *MaS* algorithm is easier to design and implement than the diffusion. This is especially true in terms of the communication structure of the algorithms. For diffusion, designing the communication structure is a bit more challenging because since there is no master controller, a communication deadlock can easily arise.

2. From Madcity parallel simulation point of view, both are easy to invoke as the synchronization points provide a natural clean point to initiate load balancing. However, in MaS, program control and coordination is handled by one master process and therefore simpler. Program control and coordination is more difficult in diffusion since control is handled individually. Program control and coordination need to be handled correctly otherwise the program could result in communication deadlock.

3. The designs follow the 5 phases of a generic load balancing algorithm. One of the algorithms, whose work transfer calculation is central is called *MaS* to denote master slave. Also designed is a diffusion algorithm with a local work transfer calculation.

Chapter 4

Experimental Results and Discussions

4.1. Chapter Overview

In this chapter, dynamic load balancing (DLB) was studied using the implementations of the algorithms described in the previous chapter. Following the design and implementations of the algorithms, experiments were designed to study the algorithms. It is interesting to note that a load index and a profitability determination algorithms can only be tested within a full dynamic load balancing algorithm and not individually.

The objectives of this study are:

- To study the effect of load unevenness on parallel road traffic simulation
- To compare the performances of the simulation synthesized with DLBs and the simulation without DLB
- To study the performances of the BCA, ECA, BDA, EDAG algorithms and

compare them with each other to ascertain the degree of performances of the new algorithms.

- To study the scalability of the algorithms in terms of:
  DLB overhead, Run-time and Quality of load balance

The experiments seek to answer such questions as:

- How does simulations with PD compare to simulations without PD on various compute-node configurations for *MaS*?
- How does simulations with PD compare to simulations without PD on various compute-node configurations for diffusion?
- How does vehicles load index compare with response time load index for *MaS*? In other words, how does BCA compare to ECA?
- How does vehicles load index compare with response time load index for diffusion? In other words, how does BDA compare to EDAG?
- How does vehicles load index of *MaS* compare with vehicle load index of diffusion? In other words, how does BCA compare to BDA?
- How does response time load index of *MaS* compare with response time load index of diffusion? In other words, how does ECA compare to EDA?

4.2. Definition of Performance Metrics

To understand the discussion of the experimental results, it would be useful to define the performance metrics that are used in the discussions. The parallel programs are evaluated with the following parameters:

Load index (L_i)*:* is used to characterize the load on the whole system. Previous measurement studies show that the run-time of the simulation is proportional to the number of vehicles, hence, suggests itself as a good load index. The run-time of the simulation is also affected by the size of the network, that is, the total number of junctions and roads in the network, but the major factor is the number of vehicles in the simulation. Since a load index was designed in this thesis, the *load of a compute-node* refers to either its *total number of vehicles* in the simulation or the response time of a simulation time step.

Speedup (S): This is defined as the ratio of the time taken by the execution of the sequential program (t_s) to that of the parallel program (t_p). That is,

$$S = \frac{t_s}{t_p} \tag{4.1}$$

For n processors, the maximum theoretically achievable speedup is n.

Efficiency (E): Another important parallel processor measure is efficiency, defined as the ratio of speedup (S) to the number of processors (n). It is a measure of the time that the processors spend on the actual computation. That is,

$$E = \frac{S}{n} \tag{4.2}$$

The maximum theoretically achievable efficiency is 100%. Actually, available parallelism (p) within an application may be smaller than the number of processors (*n)* used. In such cases, no matter how large n is, the maximum speedup cannot exceed p. Theoretically achievable maximum speedup and efficiency is rarely achieved in practice. This is because some costs are involved in realizing parallel processing. The first reason is that the inherent parallelism of the application constrains the maximum potential speedup: some applications are better suited to parallel processing than others and some applications are not suitable for parallelism at all. The other reasons were discussed in section §3.6.

*Run-time (*t*):* Here, run-time is defined as the total time it takes the application to complete its execution. The time at the beginning and end of the simulation are recorded. Then the start time was subtracted from the end time to give the simulation run-time. One of the overall aims of load balancing is to improve the run-time of the parallel simulation, and hence the run-time of the application serves as a good performance index.

Load Evenness Index (σ): It is important to characterize load evenness because this is used to compare the effectiveness of the load balancing algorithms, later in the thesis. The question is, how is load evenness or load unevenness measured? Technically speaking, the load of a system is fully balanced if the load on every compute-node is equal to the average load in the system. It can therefore be deduced that the closer the load of every compute-node to the average load in the system, the better balanced is the system. The reverse is also true, that is, the greater the difference between the average load and the load of each of the compute-nodes, the greater the load unevenness.

In this thesis, load evenness is defined using the mathematical formula provided in [24]. It is measured by the relative standard deviation (σ) of the load of all the N system compute-nodes, normalized to the average system load (μ). Mathematically, this is expressed as follows:

$$\sigma = \frac{1}{\mu}\sqrt{\frac{\sum_{i=1}^{N}(L_i - \mu)^2}{N}} \tag{4.3}$$

Where (L_i) is the load index of the ith system compute-node and (μ) is defined as:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} L_i \tag{4.4}$$

From this equation, the load of the system is evenly balanced if σ is 0 and the degree of unevenness increases as σ increases to infinity. To measure load evenness, first, the road network is partitioned and secondly, the load distributions of the various partitions are used to calculate σ, in the above formulas.

4.3. The Experimental Road Network

The road network used for most of the experiments is called the *Manhattan road network*. This is a regular grid-like mesh network comprising of junctions and roads. The size of the road networks may differ but the characteristic of the networks remain the same. The advantage of using this type of road network is that the Manhattan road network generator (discussed in section §3.5.2) can be used to produce sufficiently big road networks of this kind which are suitable for the parallel programming experiments conducted in this thesis.

Madcity can simulate real life road network such as the map of London, for example, but realistic road networks are not used simply because of the difficulty in obtaining the road network data from the respective bodies that own the Geographical Information Survey (GIS) data, and also the difficulty in converting the data into the acceptable Madcity format. Because of this, the only real road network data that has been prepared for Madcity is the Hyde network shown in Figure 3.2. This is the road network for simulating the town of Hyde in Greater Manchester, UK, that was obtained in a collaborative research project. The network is sufficient for sequential experiments but does not present enough load for parallel experiments.

For the experiments in this thesis, it does not make any difference between using synthetic or real road networks. All that is required is a road network that is capable of delivering enough load to experiment with the dynamic load balancing algorithms designed in this thesis. The road network that was used most often can accommodate a total of 25280 vehicles given an average of 5056 vehicles per compute-node on a 5-node configuration. The road networks were constructed with various degrees of load evenness ranging from $\sigma = 0.27$ to $\sigma = 1.65$.

4.4. The Experimental Design Methodology

The parallel simulator was evaluated through a series of experiments in which the run-time of the serial and parallel versions of the program were measured. Most of these experiments were carried out on the Mut cluster (ref section §3.3) and others on Carmen cluster (ref section §3.4), using the Manhattan road network, a regular grid-like mesh road network. This is a linear road network consisting of junctions. Junctions are connected with each other by a minimum of 2 roads and maximum of 4 roads. The partitioning is either linearly vertical or linearly horizontal. Linear vertical partitioning is illustrated in Fig. 3.4 and linear horizontal partitioning is illustrated in Fig. 3.5. The linear partitioning implies that every partition has 2 neighbours. Traditionally, in the Madcity simulation, each road has one vehicle at the beginning of the simulation but as many vehicles as the road can contain can be put on the road, if desired. This has a direct effect of increasing the load of the simulation.

The mesh-like nature of the Manhattan network is coupled with a synthetic traffic pattern where vehicle routes are sampled from a normalized random number generator. The experiments were performed on 1 compute-node (the sequential Madcity version) and the parallel Madcity version on varying number of compute-nodes. An important part of the experimental design is the preparation of the road networks. A graph partitioner utility (ref section §3.5.2) that is written for this purpose was used. The program takes arguments such as the desired size of the network and the number of partitions and then partitions the graph accordingly. The summary of the experimental procedures are:

- Use a program called the Manhattan network generator to draw and partition the road network into sub-networks. This is a web-based Java program that takes as input the size of the road network, the number of partitions, the nature of the partitions (horizontal or vertical) and produces a well partitioned '.net' file that can be executed on different compute-nodes on the cluster.

- Configure PVM parallel environment on the Cluster with sufficient number of compute-nodes required for the execution of the parallel program. The number of compute-nodes configured equals the number of partitions in the road network to ensure a one-one mapping of partition to compute-node.
- Run the parallel program and on completion, record the execution time/mean response time of the program.

4.5. The Experimental Results and Discussions

Experiments were performed to study the behaviours of the parallel simulator and the load balancing algorithms under a variety of conditions. First, the effect of partition load evenness on the performance of the parallel simulation was studied. Then the performance improvement achieved by the load balancing algorithms were studied by comparing parallel simulation with and without load balancing algorithms. Profitability determination (PD) was then studied by comparing simulations with and without PD for both *MaS* and diffusion algorithms. The performances of load indexes were also studied by comparing vehicles load index with response time load index. And finally, the scaling efficiency of the algorithms were studied.

It was first important to establish a performance baseline of the parallel simulator without the DLB in order to compare with the results of the DLBs. Also studied was the performance of a manually balanced network. In each of the experiments therefore, 3 measurements were taken:

1. The run-time of the manually balanced network (*MB-noDLB*)
2. The run-time of the parallel simulator without DLB (*Sim-noDLB*)
3. The run-time of the parallel simulatior with DLB (*Sim-DLB*)

In every experiment, the vehicle distributions before and after the load balancing operation were recorded. The manually balanced network (MB-noDLB) gives an indicative measure of the performance of the dynamic load balancing algorithms since the simulator starts with a network that is balanced right from inception. It is assumed that in comparing the above 3 networks, the MB-noDLB network would give the best performance, followed by Sim-DLB and then Sim-noDLB. The expected performance graph in terms of the run-time is illustrated in Fig. 4.1. Note that these are only estimated values and not based on experimental results.

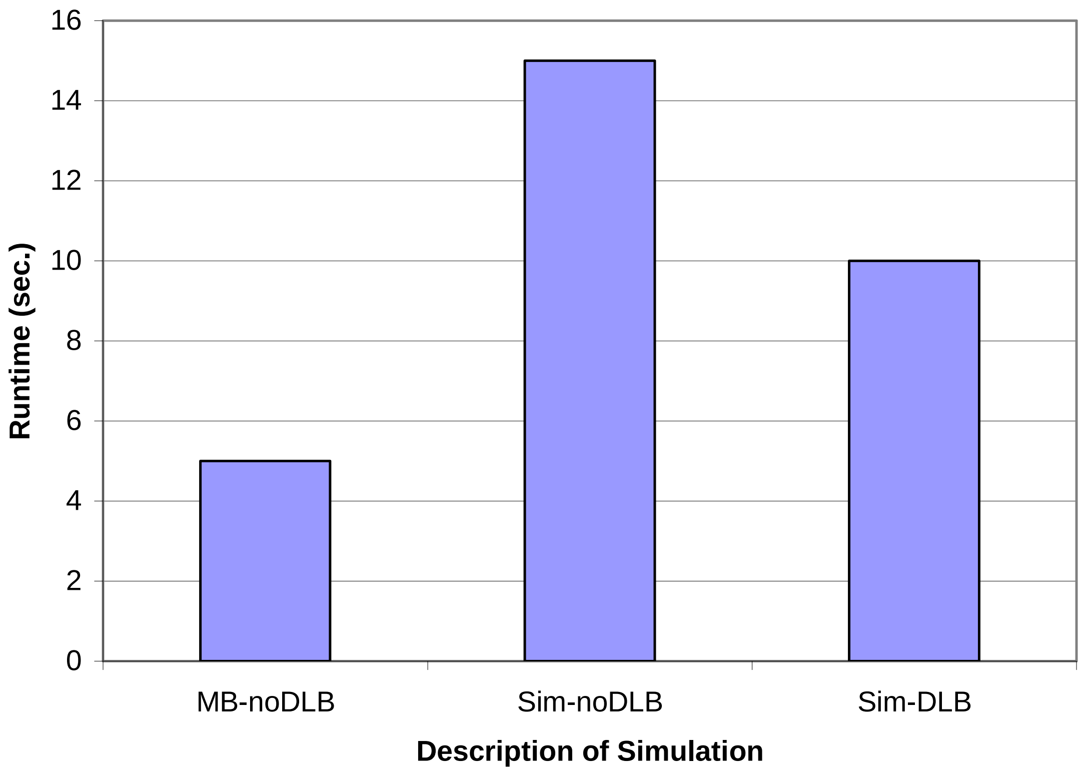


Figure 4.1: Expected Simulation Run-time of Different Simulation Scenarios (estimated values)

The expectations are that the run-time of the dynamic load-balancing algorithm would be close to that of MB-noDLB algorithm. However, the two may not be the same because of dynamic load balancing overheads, which is not present in the static case.

Synthetic static network files were created with arbitrary load imbalances ranging from 10% to 70% imbalances. That is, the experiments start with initially unbalanced networks. The experiments were designed with a simplified network model in which individual imbalances arise, one after another, with each imbalance confined to a single computer. The aim is to study the performances of the algorithms under varied network imbalance scenarios.

The run-time and vehicle distributions before and after every DLB experiment were measured so as to calculate the quality of load balance. Quality of load balance is the measure of the 'balancedness' of the network. It is helpful for calculating the % improvement of the load imbalance after DLB operations. Quality of load balance was discussed further in section §4.2.

4.5.1. Effect of Load Evenness on Parallel Traffic Simulation

The purpose of this experiment is to study the effect of load evenness on the performance of parallel traffic simulation. The measures of performance here are *run-time, speedup* and *efficiency.* Various road networks were created using Manhattan network generator tool, described in section §3.5.2. The load on individual compute-nodes were manually adjusted to reflect load imbalance. For example, in experiment 1 of table 4.2, node 1 has 430 vehicles, node 2 has 580 vehicles, node 3 has 580 vehicles, node 4 has 580 vehicles and node 5 has 6230 vehicles. Different load situations were also created for experiments 2 to 8 as indicated in the table. Degrees of load evenness were calculated using the relative standard deviation method of equation 4.3 in section §4.2.

In these experiments, the road network with the highest degree of evenness has a standard deviation of 0.04. Note that a perfectly even network has a theoretical standard deviation of 0. However, the best achievable road network in the experiments has a standard deviation of 0.04. This is because of performance design constraints, such as minimizing partition edges and hence communication

No of Vehicles	No of Junctions	No of Roads	No of partitions	Time steps
8400	225	840	5	5000

Table 4.1: Description of the Road Network

No of Expts	Node 1 (vehicles)	Node 2 (vehicles)	Node 3 (vehicles)	Node 4 (vehicles)	Node 5 (vehicles)	Load Evenness (σ)
1	430	580	580	580	6230	1.35
2	430	580	580	1160	5650	1.19
3	430	580	580	1740	5070	1.04
4	430	580	580	3480	3330	0.84
5	430	580	1740	2900	2750	0.62
6	1010	1160	1160	2320	2750	0.42
7	1010	1740	1740	1740	2170	0.22
8	1590	1740	1590	1740	1740	0.04

Table 4.2: Calculation of Load Evenness for Different Experiments

overhead, discussed in §3.6. Similarly, the network with the smallest degree of load evenness has a standard deviation of 1.35 (These are shown in table 4.2). The lower limit of $\sigma = 0.04$ and upper limit of $\sigma = 1.35$ are due to practical constraints. All experiments were performed on a 5 compute-node network with 5 partitions on the Mut cluster. The full network description is shown in Table 4.1 and the load distributions for each experiment are shown in table 4.2.

Analysis of Experimental Results

The experimental results are presented in Fig. 4.2 and 4.3. Fig. 4.2 shows the variation in simulation run-time plotted against load evenness. The results show that as load evenness (σ) increases, simulation run-time increases approximately linear. The best achievable run-time is when σ= 0.04. However, interpolating the line graph on a straight line cuts the y-axis at run-time = 25 secs (see fig. 4.2), representing the case of $\sigma = 0$, the theoretical perfectly even network. The difference of 3 seconds between the interpolated value (25 secs) and the real value (28 secs) reveals at a glance that load evenness affects the performance of the experiment. Also, the case of $\sigma = 0$ is the best run time value that the load balancing algorithms could aim to achieve but in reality, it is impossible due to

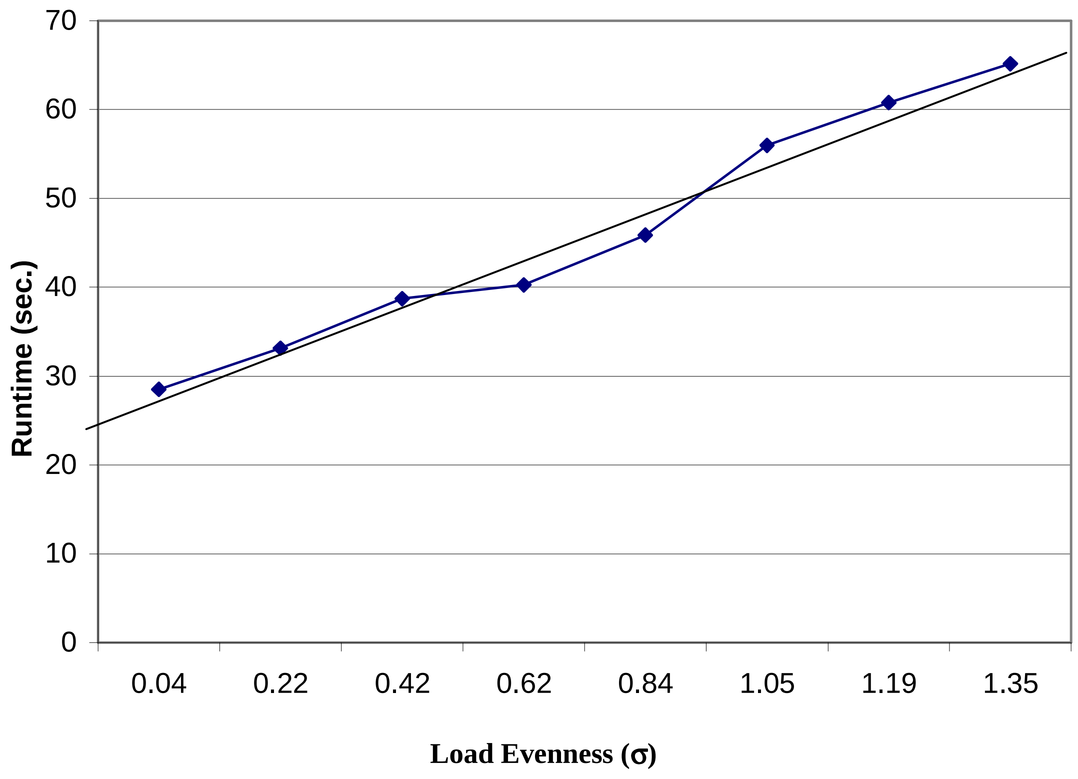


Figure 4.2: Effect of Load Evenness on Parallel Simulation Runtime.

inherent parallel simulation overheads.

Fig. 4.3 shows corresponding speedup against load evenness. These results clearly show that load unevenness significantly reduces the performance of parallel traffic simulations. For example, on 5 compute-nodes, the theoretical maximum speedup is 5 but a speedup of 2.7 is achieved on the 0.04 load evenness and 1.4 on 1.35 load evenness, representing parallel computational efficiency of 54% and 34% respectively. A significant 20 percentage-point performance difference corresponds to nearly a 60% worsening compared with the almost perfectly even network. Two questions arise from these results.

1. Why is the speedup of evenly partitioned network better than the speedup of the uneven network? and

2. Why is the theoretical maximum speedup not achieved in the almost

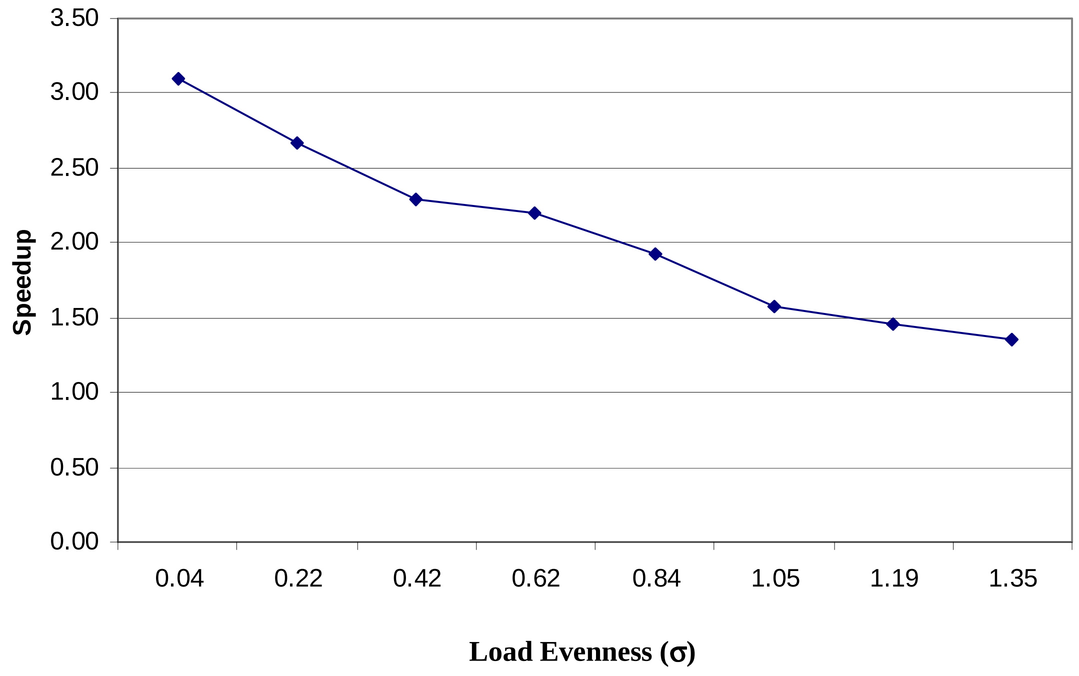


Figure 4.3: Effect of Load Evenness on Parallel Simulation Speedup

perfectly even network?

The answer to the first question is that in the implementation of the parallel simulation, global barrier synchronization is employed (as described in section §3.6.4) and all the processes wait at each global synchronization point until the last compute-node has reached the point. In a system with an uneven load, the wait time of the processors is likely to be longer than when the load is even. The last processor to reach the synchronization point is the one with the biggest load; the greater the load evenness, therefore, the longer the wait time at the synchronization point.

On the second question, the performance shortfall is due to the communication and computation overheads on parallel performance as discussed in section §3.6.3. The best case of 54% parallel efficiency has an overhead of 46%, which is rather large on 5 compute-nodes, but the question of whether this trend continues in bigger networks and larger compute-nodes will be investigated in later experiments.

Conclusion

In conclusion, for a network of 5 compute-nodes, load evenness generally affects the performance of parallel traffic simulations. The greater the degree of load unevenness, the worse the performance of the system. For example, the network with $(\sigma) = 0.04$ gives a significant 20 percentage-point improvement over the network with $(\sigma) = 1.35$. These results are indicative of a need to investigate the corrective effect of load balancing algorithms in parallel simulation on 5 compute-nodes. In later experiments the investigations will be explored on bigger compute-nodes.

4.5.2. Profitability Determination Experiments

Since the overheard of dynamic load balancing is its major disadvantage, reducing the overhead would be beneficial to the overall performance of the system. A method is hereby proposed whereby some aspects of the load balancing algorithm such as the PD is performed off line.

As discussed in section 3.7.3, $pd_{threshold}$ is dependent on such factors as:

- size of the road network
- number of compute-nodes used
- and degree of load evenness

As a result of the above factors, for every road network that would be used in the experiments, the first step is to determine $pd_{threshold}$ in equation 3.5 and also since $pd_{percentage}$ in equation 3.6 is not known, it has to be determined by experimentation. Experiments were performed on Carmen cluster using different

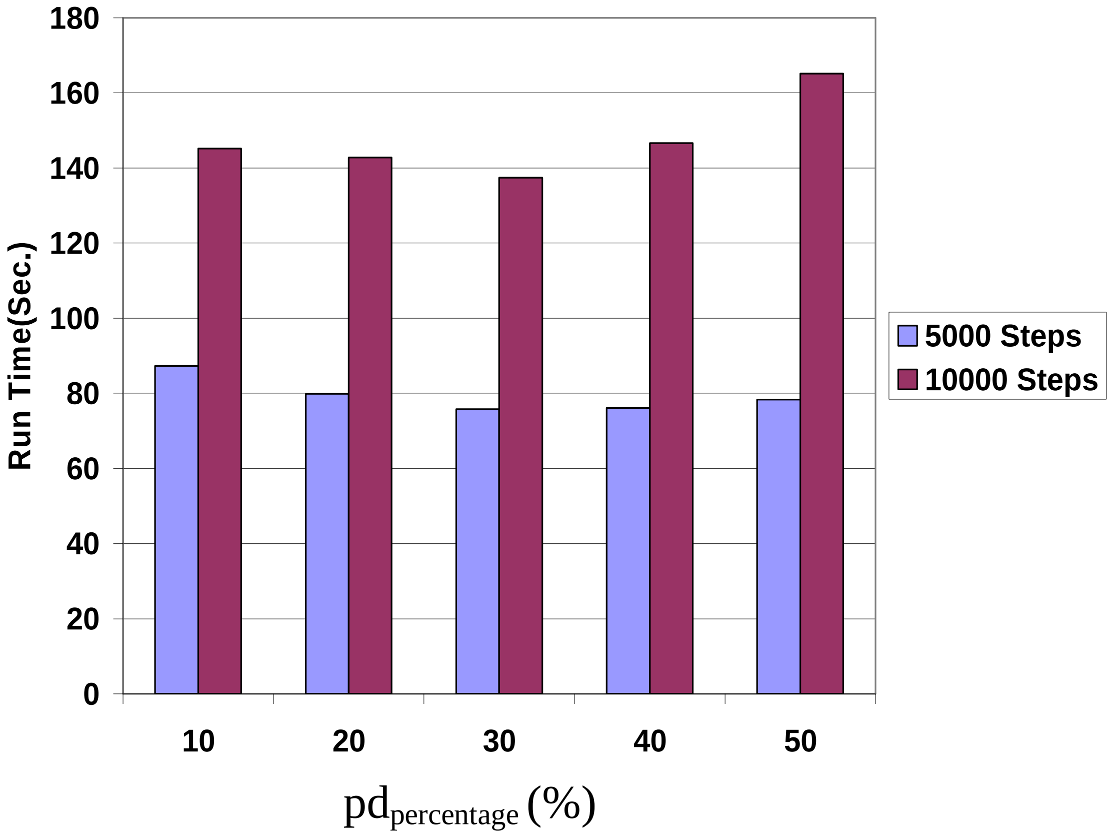


Figure 4.4: Finding $pd_{percentage}$ for *MaS* algorithms on 5 compute-nodes.

values of $pd_{percentage}$ whereby $pd_{percentage}$ was varied between 10% and 50% (i.e, 10, 20, 30, 40, 50) as shown in Fig. 4.4. The experiments were conducted for two different time steps and it was observed that the run-time reduced as $pd_{percentage}$ increases from 10% to 30%, after which it started to rise again. This means that for this road network, run-time is lowest at $pd_{percentage} = 30\%$ for both time steps. In order words, 30% of load above the average load ($pd_{percentage}$) gives the best performance for this road network. x is measured by using equation 3.4 where $AvgLoad$ is subtracted from the load of the most loaded compute-node (i.e $x = node_{mostloaded} - AvgLoad$).

The PD relation of equation 3.5 therefore becomes:

Algorithm 10: The Pofitability Determination(PD)Algorithm

```
   Input: Set of tasks and compute-nodes
   Output: Profitability Determination
1  if $x \geq AvgLoad \times 0.3$ then
2      perform DLB
3  end
```

Effect of Period of Load Balancing and PD on the Performance of Load Balancing Algorithms

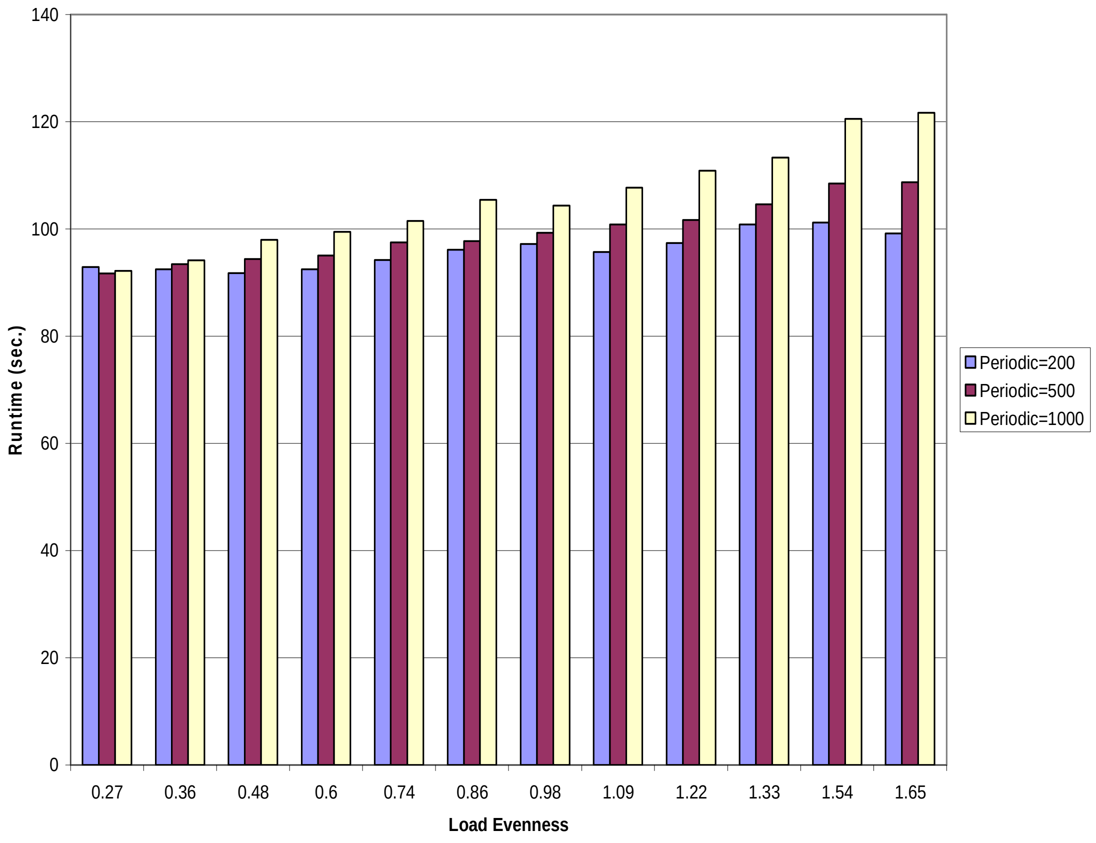


Figure 4.5: Comparing different periodic values.

A critical parameter in any periodic load balancing algorithm is setting the '*right*' period. In figure 4.5, three experiments were performed with three different periodic values of load balancing (200, 500 and 1000). The value 200, for example, means that load balancing is performed every 200 time steps. For these experiments, the results are best at period = 200 and worst at period = 1000. The results show that as the period of load balancing is made bigger, the run-time of the simulation increases. This is because longer period means that load imbalance is prolonged before correction and hence its adverse affect on the performance of the system. For this network, simulation period performs best at 200 but would differ from network to network. Just like in the case of PD, this value would be influenced by such factors as;

- size of the road network
- number of compute-nodes used
- the total simulation time steps
- degree of load evenness

This means that no single periodic value is appropriate for all road networks and situations. The value differs for every road network setup, but there exist an optimum period that gives the best results. This increases the controllable factors in performing periodic load balancing experiments. Apart from setting the right value of PD threshold, for example, the period also need to be determined prior to experiments.

The interesting observation in this regards is that PD helps in eliminating the importance of setting the right period. This is because no matter how short or long the period of load balancing is, PD ensures that load balancing is performed only if it is advantageous to perform load balancing.

Comparing Simulation with PD and Simulation without PD of MaS and Diffusion Algorithms

In Fig. 4.6, a simulation with profitability determination (PD) was compared with another simulation in which the PD was turned off, using VLI, for a *MaS* and

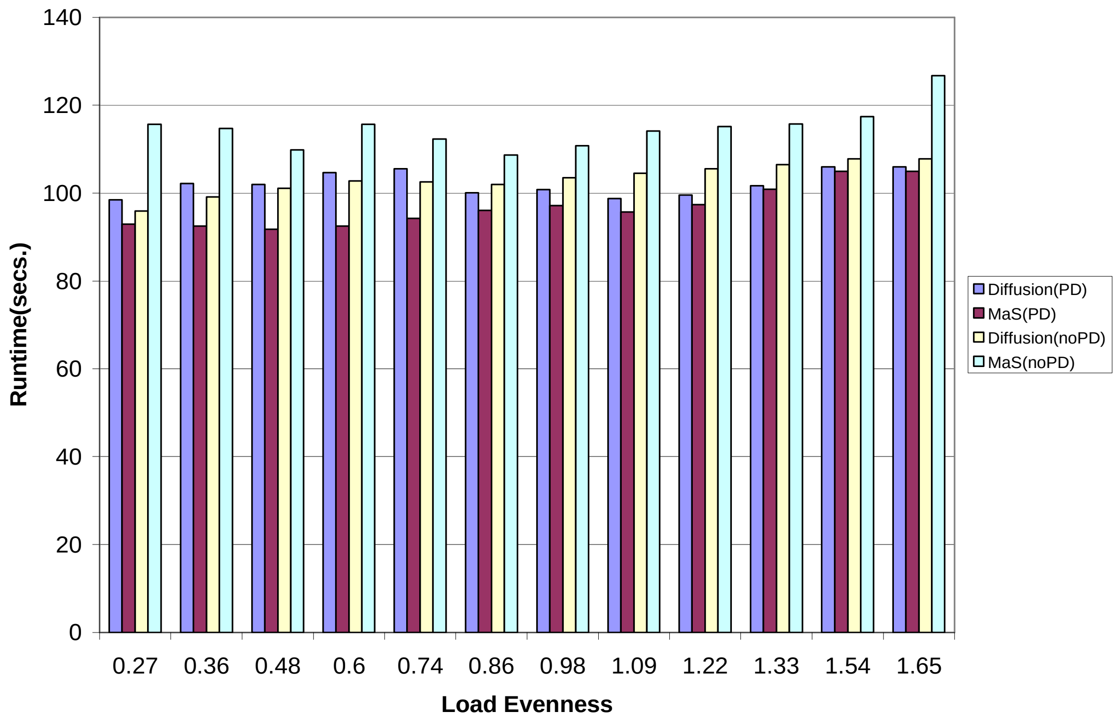


Figure 4.6: Comparing simulation with PD and simulation without PD of *MaS* and Diffusion Algorithms.

diffusion algorithms. For a simulation of 5000 time steps, periodic load balancing was performed every 200 time steps.

Four(4) comparisons are made from Fig. 4.6 as discussed below:

1. *MaS with PD and MaS with no PD (MaS(PD)and MaS(noPD))*

   It can be seen in the figure that the simulation with PD performs better than simulation without PD for all the experiments considered. This is expected because when PD is turned off, the simulation balances the load at every 200 time steps when dynamic load balancing routines are called. This kind of simulation lacks the intelligence to know if it is profitable to balance the load or not. As a result, load balancing overheads are incurred at every periodic load balancing operation. The simulation with PD has the intelligence to determine when it is profitable to load balance or not. As a result, it attempts to balance the load only when it is profitable hence reducing some load balancing overhead.

2. *Diffusion with PD and Diffusion with no PD(Diffusion(PD)and Diffusion(noPD))*

   It can be seen in the figure that the simulation with PD performs better than simulation without PD for all the experiments considered. However, the difference between the two is not as huge as the the case of *MaS* above. While PD helps in reducing the runtime of the simulation, PD added another overhead to the Diffusion and as such the overall improvement is not as good as in the case of *MaS*.

3. *MaS with PD and Diffusion with PD (MaS(PD) and Diffusion(PD)) Algorithms*

   In comparing *MaS(PD)* and Diffusion(PD), *MaS(PD)* performs better than Diffusion(PD) in all situation. This shows that global PD fits perfectly into the architecture of *MaS*. In the case of diffusion algorithm, it is like adding an additional overhead to the overall runtime of the algorithm and hence performance is lower than *MaS*.

4. *MaS with noPD and Diffusion with noPD(MaS(noPD) and Diffusion(noPD)) Algorithms*

   Comparing the runtime of *MaS (noPD)* and *Diffusion (noPD)* algorithms, it is observed that diffusion performs better than *MaS* in all cases. This is attributed to the overhead of *MaS* which is incurred at every load balancing step, in the absence of PD. In diffusion algorithm, the overhead is less and hence diffusion outperforms *MaS*.

   In summary, it means that when the right period is set, *MaS* outperforms diffusion but when load balancing is repeated at every period, *MaS* performs worse than diffusion. This all has to do with the architecture of the two algorithms in relation to their load balancing overheads.

4.5.3. Load Index Experiments

Measuring the Resource Load (RL)

As stated in section 3.7.1, RTLI is a measure of both the application(AL) load and the resource load(RL). While the measurement of AL was discussed in section 3.5, the measurement of RL is discussed in this section. Resource load (RL) is defined as *the total load on the system ($Load_{Total}$) minus the research Madcity application load ($Load_{Madcity}$).* In other words, it comprises of all the running processes on the system minus the load due to Madcity.

$$RL = Load_{total} - Load_{madcity} \tag{4.5}$$

CPU utilization was monitored by the Unix/Linux utility called 'Top'. Top provides an ongoing look at processor activity in real time and has additional functions that can save snapshots of Top results to a file and give the average summary of the resulting file, including information such as *CPU, memory, and swap.* In this way, analysis can be performed on the resource of interest.

RTLI is primarily designed for a heterogeneously-loaded distributed platform. To simulate such a platform, different compute-nodes were loaded with different levels of synthetic load with a Unix/Linux utility called 'Stress'. Stress is a utility that can be used to put a system under a varying amount of load by varying the command line parameters.

In preliminary experiments done on a single core CPU, monitored with *top*, it was observed that when 2 instances of Stress are run, the system allocates 50% of CPU to each instance of Stress. When 4 instances are run, the system allocates 25% to each instance. Similarly, in a multicore CPU, load is distributed to the processors in equal measure. For example, if 4 processes are sent to a quad core CPU, each CPU would have one process each. If 6 are sent, 2 CPUs would have

2 processes each and 2 CPUs would have one process each. Similarly, when an instance of Madcity research traffic simulator is run alone, the CPU is allocated 100% to it. When this is run at the same time with other instances of Stress, it is expected that the CPU is allocated fairly equally between the instances of Madcity and Stress tool. For example if 3 instances of Stress and 1 instance of Madcity are run, ideally, the CPU should be allocated in 4 parts of 25% each.

For the experiments reported in this section, which were performed on 5 compute-nodes, below are the load distribution of Stress on 4 compute-nodes. The 1st compute-node has no instance of stress running.

```
compute-node 2:
    stress --cpu 4 --io 4   --vm 8   --vm-bytes 512M &
compute-node 3:
    stress --cpu 8 --io 8   --vm 8   --vm-bytes 512M &
compute-node 4:
    stress --cpu 12 --io 12 --vm 8   --vm-bytes 512M &
compute-node 5:
    stress --cpu 16 --io 16 --vm 8   --vm-bytes 512M &

where:
         --cpu N        spawn N workers spinning on sqrt()
         --io N         spawn N workers spinning on sync()
         --vm N         spawn N workers spinning on malloc()/free()
         --vm-bytes B   malloc B bytes per vm worker (default is 256MB)
```

Note the following:

- —cpu 16 means that 16 instances of stress are running. Each compute-node is a quad-core CPU meaning that each CPU is allocated 4 instances of stress. This is to ensure that whichever CPU Madcity application runs, since Madcity cannot be restricted to a particular CPU, it would be running

	data 1 %	data 2 %	data 3 %	data 4 %	data 5 %	Average % CPU Utilization
Node 1	96.4	97.0	98.0	98.3	98.1	97.56
Node 2	29.5	30.4	30.6	29.6	30.4	30.1
Node 3	27.1	27.4	26.5	26.8	27.4	27.04
Node 4	21.7	21.9	21.7	20.7	21.6	21.52
Node 5	15.3	15.5	15.3	14.9	15.0	15.2

Table 4.3: Table showing % CPU Utilization of Madcity on a heterogeneously loaded platform

on a slowed CPU.

- —io 16 behaves similar to cpu.
- It was observed that allocating more than 8 to the VM or allocating more than 512M to vm-bytes killed the compute-node so these were the maximum values used. These restrictions has to do with the maximum memory available in the compute-node

With this distributions, experiments were conducted with serial Madcity, on 1 compute-node, to know how much percentage the CPU allocates to Madcity. The compute-node was loaded, in turn, with *Stress* utility and then readings of the Madcity percentage CPU utilization were recorded with a Linux/Unix utility called 'ps'. *ps* has the capability to read the % CPU utilization of a process when passed the process id of the process. 5 readings were taken at intervals and then the average calculated. These are tabulated in Table 4.3. The table shows that the compute-node dedicated about 98% of CPU to Madcity on compute-node 1, 30% on compute-node 2, 27% on compute-node 3, 23% on compute-node 4 and 15% on compute-node 5. The remaining percentages were taken by the resource load.

These experiments were as expected. This generally means that a heterogeneously-loaded platform can be simulated by varying instances of Stress running on the system though the actual percentage allocated to Madcity need to be verified by experimentation.

*Comparing Vehicles Load Index (VLI) and Response Time Load Index (RTLI) of Centralized (*MaS*) Algorithm*

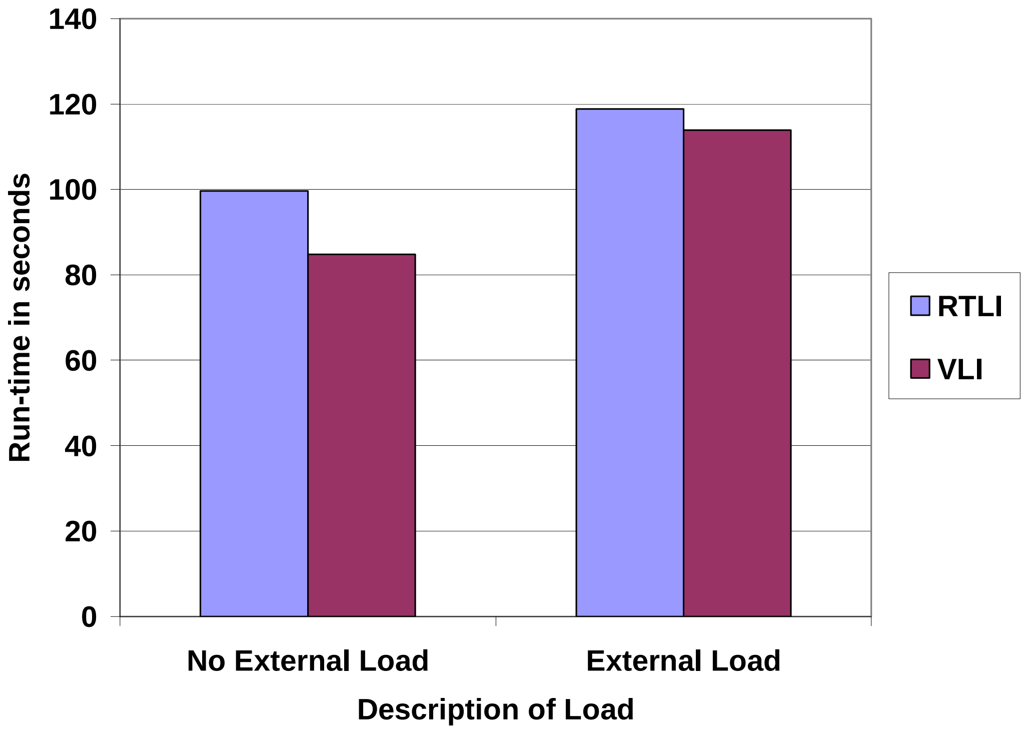


Figure 4.7: Comparing Response Time load Index (RTLI) and Vehicle Load Index (VLI) of Centralized (MaS)Algorithm.

Load index can not be run as a stand alone component but as part of a load balancing algorithm. The comparison here is between Basic Centralized Algorithm with global PD (BCA_{PD}) and Extended Centralized Algorithms(ECA). The main objective here is to compare a *MaS* algorithm with vehicles load index with another *MaS* algorithm with a response time load index. Experiments were conducted on 5 compute-nodes of the Carmen cluster. The results are shown in Figure 4.7. In one set of experiments, stress utility were run at background to simulate variable system load in a heterogeneous environment, as explained in the section above. The other set of experiments were run in a normal cluster environment. In Figure 4.7 the former is identified as *external load* while the latter is identified as *no external load*.

In the experiments with *no external load*, the vehicles load index performed better than the response time load index by about 15%. This is because in a homogeneous cluster environment, vehicles load index is sufficient as load index. Response time load index presents extra overhead of converting from the response time load index to vehicles which gives vehicles load index an advantage. In the experiments with external load, the vehicles load index still performs better than the response time load index but the difference in run-time in this case is smaller. The difference is 4% as against the 15% in the above case. This shows that as the environment moves towards a heterogenous environment, response time load index shows a qualitative evidence that it improved in performance compared to the vehicles load index, when used in centralized algorithm.

Using the same road network in the above experiments, the experiments were repeated on Mut cluster and results are in Fig. 4.8. External loads are easier to manage here since the compute-nodes has only one processor each. Here experiments were done with the following features:

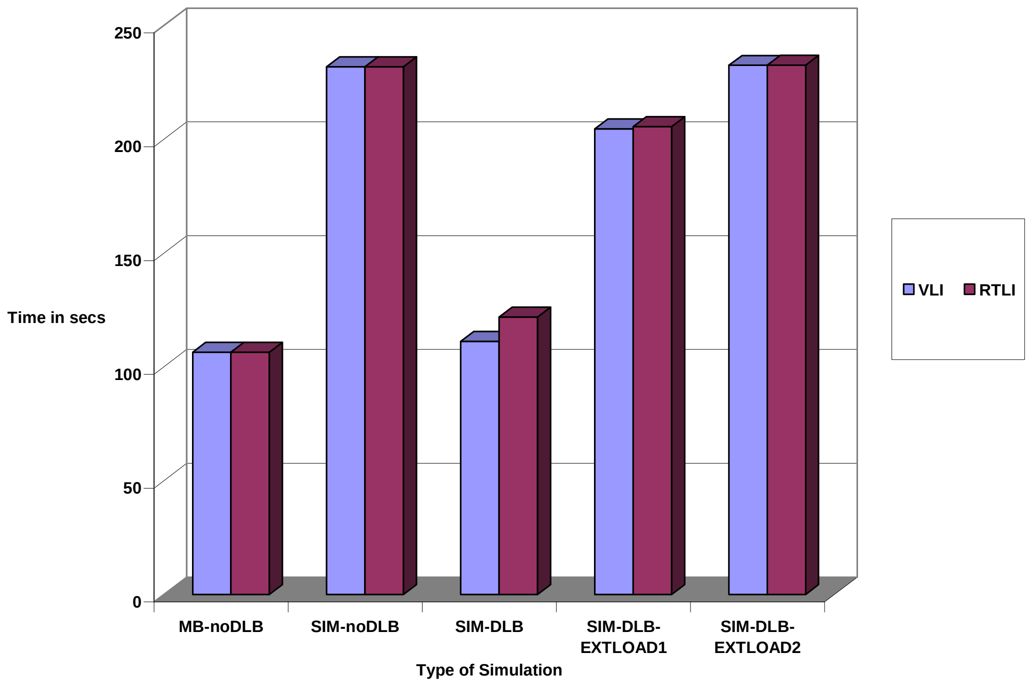


Figure 4.8: Comparing RTLI and VLI of Centralized (MaS)Algorithm.on Mut Cluster.

1. A statically balanced network with no DLB *(MB-noDLB)*

2. A manually unbalanced network and no DLB *(SIM-noDLB)*
3. A manually unbalanced network, no external load but DLB for both RTLI and VLI *(SIM-DLB)*
4. A manually unbalanced network with external load distributed in random *(SIM-DLB-EXTLOAD1)*
5. A manually balanced network with external load distributed in an organized way (i.e big AL on compute-nodes with high RL) *(SIM-DLB-EXTLOAD2)*

In the first 2 set of experiments, there is no difference in performance between RTLI and VLI. This is expected as both DLBs are turned off. This experiments only help to show that the two algorithms, BCA and ECA, are the same and the only difference between then is the load index.

When experiments were done on a homogeneous platform (i.e no external load), VLI performed better than RTLI. Again, this is expected as these are the same results obtained on Carmen cluster. It confirms the fact that VLI performs better in a homogeneous system

When external load were introduced in a random way with no regards to which compute-node were simulating high vehicles, VLI still performs better than RTLI. But, the difference in performance between RTLI and VLI is very little.

When external load were introduced in a controlled way such that compute-nodes with RL are also the compute-nodes with high AL, the two algorithms performed the same. This further supports the fact that the strength of RTLI is in a heterogeneously-loaded platform.

Conclusions

For the experiments conducted using 2 different clusters under varying ranges of heterogeneously- loaded platforms, the results indicate that VLI performs best in a homogeneous platform. This is due to the simplicity of VLI. It is quite simple to measure and no additional overhead are introduced due to load collection. For RTLI, the overhead of converting load from RTLI into an equivalent number of vehicles also add to its overhead in a homogeneous platform. In a heterogeneously-loaded platform however, vehicles load index is incapable of properly quantifying the load of the system and hence its performance starts slowing down while RTLI keeps improving. The result of a properly quantified load helps RTLI improve in performance. Both results confirm that RTLI shows a qualitative evidence of improved performance compared to the vehicles load index, when used in a heterogeneously-loaded platform.

Comparing Vehicles Load Index (VLI) and Response Time Load Index (RTLI) of Diffusion Algorithm

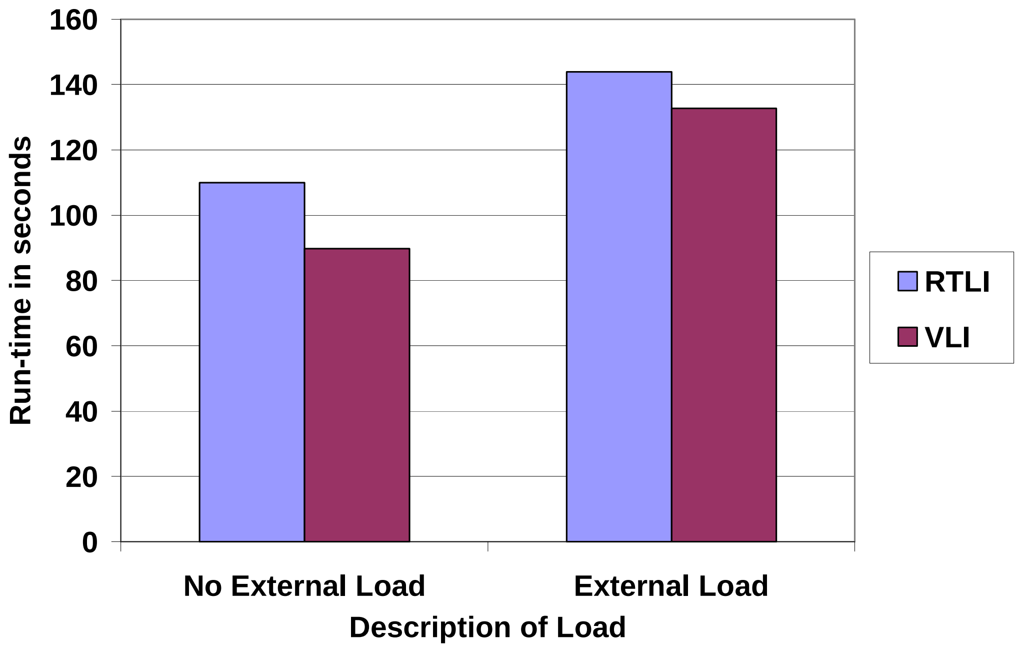


Figure 4.9: Comparing Response Time load Index (RTLI) and Vehicle Load Index (VLI) of Diffusion Algorithm.

The comparison here is between BDA with global PD (BDA_{PD}) and EDAG algorithms. BDA has profitability determination so that only the load indexes can be compared with each other. The results are shown in Figure 4.9. The results follow the same pattern as the *MaS* algorithms i.e vehicles load index still performed better than response time load index in all cases. Also, response time load index showed a qualitative evidence that it improved in performance compared to the vehicles load index, when used in distributed algorithm. However, the percentage differences in performance between the two load indexes, in both set of experiments, were wider than those in *MaS* algorithm. 18% for no external load and 8% for external load as against 15% and 4% respectively for *MaS*. In all the experiments conducted, *MaS* always performed better than diffusion. These experiments followed the same pattern. One of the reasons is that *MaS* algorithms better suite the properties of SPMD algorithms that has master slave structure.

Comparing RTLI of Extended Centralized Algorithm (ECA) and Extended Diffusion Algorithm with Global PD (EDAG)

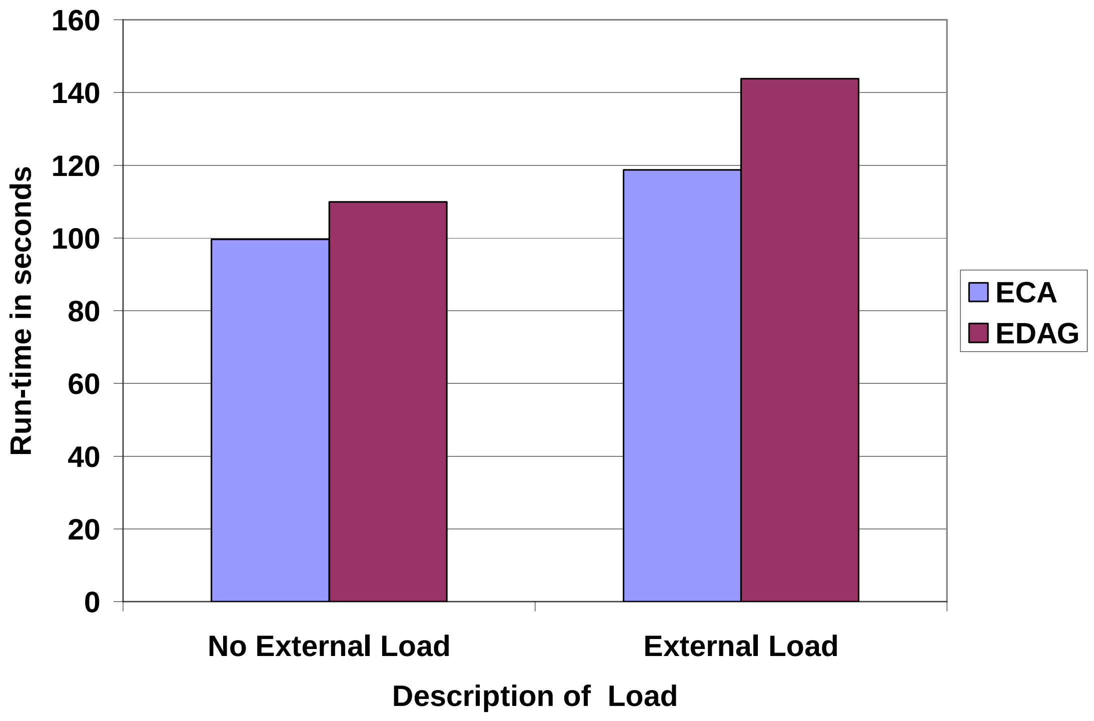


Figure 4.10: Comparing RTLI of Extended Centralized Algorithm (ECA) and Extended Diffusion Algorithm with Global PD (EDAG).

The main objective here is to compare the performances of the 2 extended algorithms of MaS and diffusion, both using RTLI load index. 2 experimental sets (with external load and with no external load) were conducted on 5 compute-nodes of the Mut cluster. The results are shown in Figure 4.11. In both experimental sets, the *MaS* algorithm performed better than the diffusion algorithm by about 9% in the case of no external load and 17% in the case of external load.

Comparing VLI of Basic Centralized Algorithm (BCA) and Basic Diffusion Agorithm (BDA)

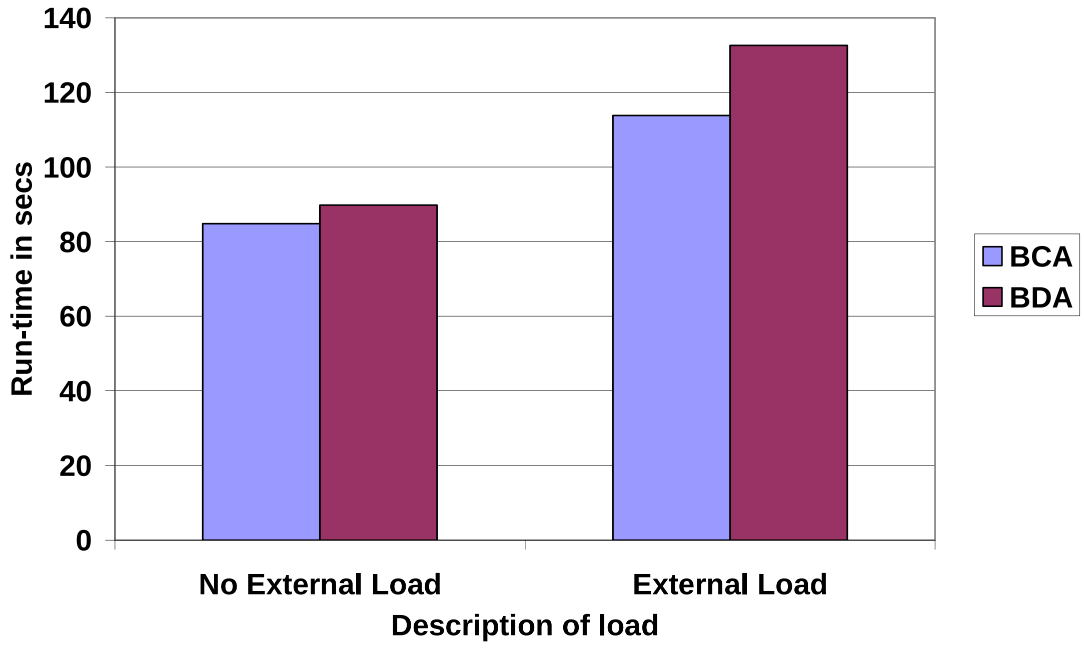


Figure 4.11: Comparing VLI of Basic Centralized Algorithm (BCA) and Basic Diffusion Agorithm (BDA).

The main objective here is to compare the performances of the 2 basic algorithms of MaS and diffusion, both using VLI load index. Experiments were conducted on 5 compute-nodes of the Mut cluster. The results are shown in Figure 4.11. In both environments, the *MaS* algorithm performed better than the diffusion algorithm by about 6% in the case of no external load and 14% in the case of external load.

The superiority of *MaS* to diffusion algorithm in both experiments is because it is observed that *MaS* fits into the structure of SPMD applications better than diffusion. Also, it is better and quicker in making load balancing decisions because it has a global view of the system. It can be concluded from these results that *MaS* better suits SPMD applications.

Performance Comparison of 2 Basic and 2 Extended Algorithms of MaS and Diffusion Algorithms

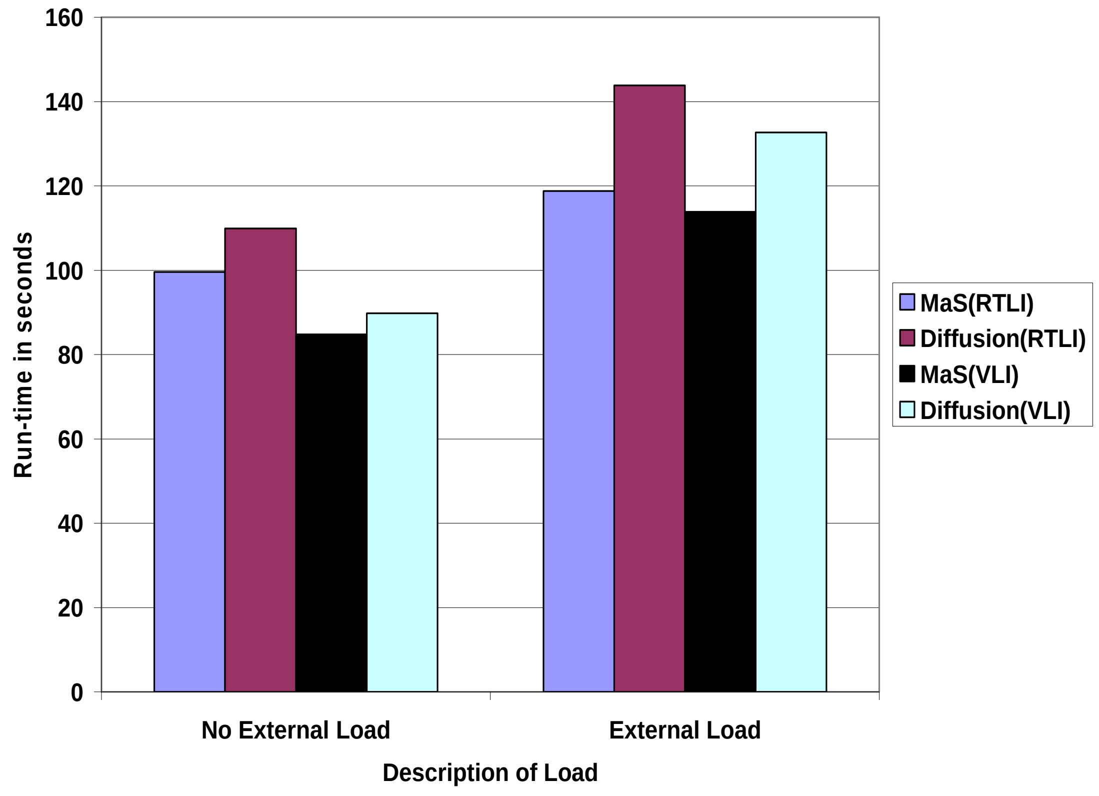


Figure 4.12: Comparing 2 Basic and 2 Extended Algorithms of MaS and Diffusion Algorithms.

These experiments compared the performance improvements of 2 basic and 2 extended load balancing algorithms of *MaS* and diffusion, on 5 compute-nodes. Performance is measured in *run-time*. The data for these experiments are the same as for the last 2 experimental discussions. There are 2 sets of experiments, one set with no external load and the other set with external load. The run-times of the different simulations were measured and plotted in Figure 4.12. In each of the 2 sets of experiments, *MaS* with VLI had the lowest run-time. In experimental set 1, this was followed by diffusion (VLI). This implies that VLI outperformed RTLI in all the experiments in experimental set 1 for both *MaS* and diffusion. In experimental set 2 however, the best two performances were for *MaS* in both VLI and RTLI. This means that *MaS* outperformed diffusion in all the experiments in experimental set 2. The fact that MaS (RTLI) outperformed diffusion (VLI), in the second set of experiments, means that as the experiments move to a heterogeneous-like environment, the advantages of RTLI over VLI becomes apparent. Also, in the second set of experiments, comparing the lowest run-time with the highest run-time, *MaS* VLI showed a percentage difference of 26% over diffusion RTLI while in the first set of experiments, the percentage difference was 30%. The 26% and 30% variations between the best performance and the worst performance in both cases was considered significant. However, the 4% reduction in the heterogeneous-like environment again showed the strength of RTLI in a heterogeneous environment. This is indicative of the fact that in a truly heterogenous environment, the performance difference between RTLI and VLI would be minimal and hence insignificant.

Performance Comparison of 2 Load Balancing Algorithms(Basic Diffusion Algorithm with global PD(BDA_{PD}) and Basic Centralized Algorithm with global PD (BCA_{PD})

This experiment aimed to measure the performance improvement achieved by the 2 regular load balancing algorithms, *MaS* and diffusion, on 5 compute-nodes. Performance is measured in 2 ways: *run-time and quality of load evenness* achieved. The experiments were performed on the following scenarios: a road network of 0.04 initial load evenness without load correction ("initially even, uncorrected"); a road network of 1.35 initial load evenness without load correction ("initially uneven, uncorrected"); and a road network of 1.35 initial load evenness

No of Vehicles	No of Junctions	No of Roads	No of partitions	Time steps
8400	225	840	5	5000

Table 4.4: Description of the Experimental Road Network for 5 Compute-nodes

employing 2 different load balancing algorithms: *MaS* and diffusion ("initially uneven, corrected"). The run-times of the different simulations were measured and the performance improvements calculated for the 2 algorithms. The full network description, which is the same for all the scenarios, is shown in Table 4.4.

Analysis of Experimental Results

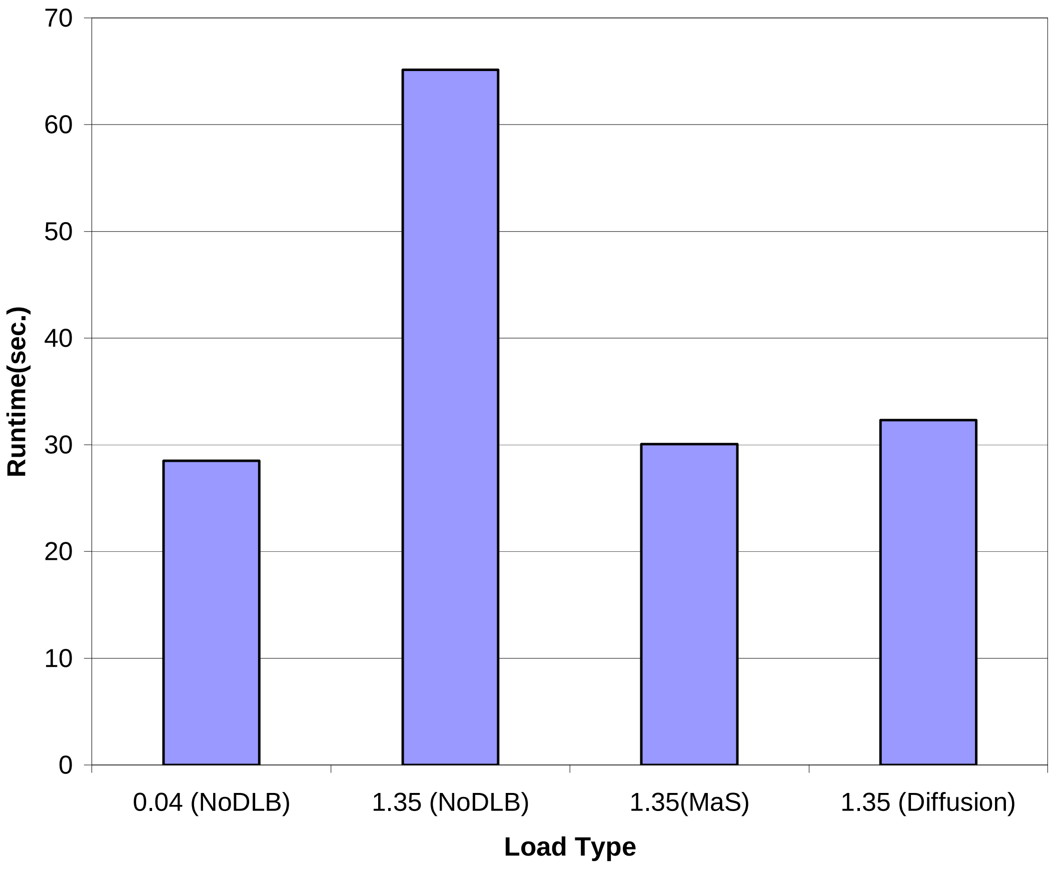


Figure 4.13: Description of Load Distribution for the Experiments

The experimental results measuring the run-time of the simulation experiments are presented in Fig. 4.13. The first observation in Fig. 4.13 is that for networks initially uneven ($\sigma = 1.35$), both DLB algorithms provided substantial improvements over the uncorrected system.

With a load evenness of 1.35, the road network was designed in such a way that one of the compute-nodes was heavily loaded, and the run-time without load balancing increased. When load balancing is turned on, a 53% reduction of simulation run-time in *MaS* and 51% reduction for diffusion are achieved. The DLB algorithms were applied at the start of the simulation, bringing the system into balance very quickly as seen in Table. 4.6. In the table, with an initial load unevenness of 1.35, *MaS* reduced it to 0.38 while diffusion reduced it to 0.50.

The initiation of load balancing at start of simulation rather than later in the simulation is to minimize the effect of prolonged load unevenness. Fig. 4.14 shows the effect of delayed load correction on the performance of the system. In the graph, there was a performance improvement of 47% when load balancing was carried out at the beginning of simulation (time step 0) rather than close to the end of the simulation (time step 4000). As can be seen in the graph, this is a linear graph showing the increase of simulation run-time as the load balancing initiation time increases. This shows that leaving the load uncorrected for a longer period leads to performance degradation.

It is interesting to note in Fig. 4.13 that the performance of both the two corrected systems and the initially even, uncorrected systems are almost the same (as predicted in section §4.1). The initially even, uncorrected system provides the best performance in this case (57% improvement), followed by *MaS* algorithm (53% improvement) and then diffusion algorithm (51% improvement) compared with the initially uneven, uncorrected case. The initially even, uncorrected network ($\sigma = 0.04$) provides the best performance because it is (almost) perfectly balanced throughout and there is no run-time overheads introduced by load balancing. However, the fact that *MaS* algorithm is very slightly, only 2% better than diffusion is interesting because it has been widely assumed that in distributed systems, centralized solutions are undesirable because they tend to

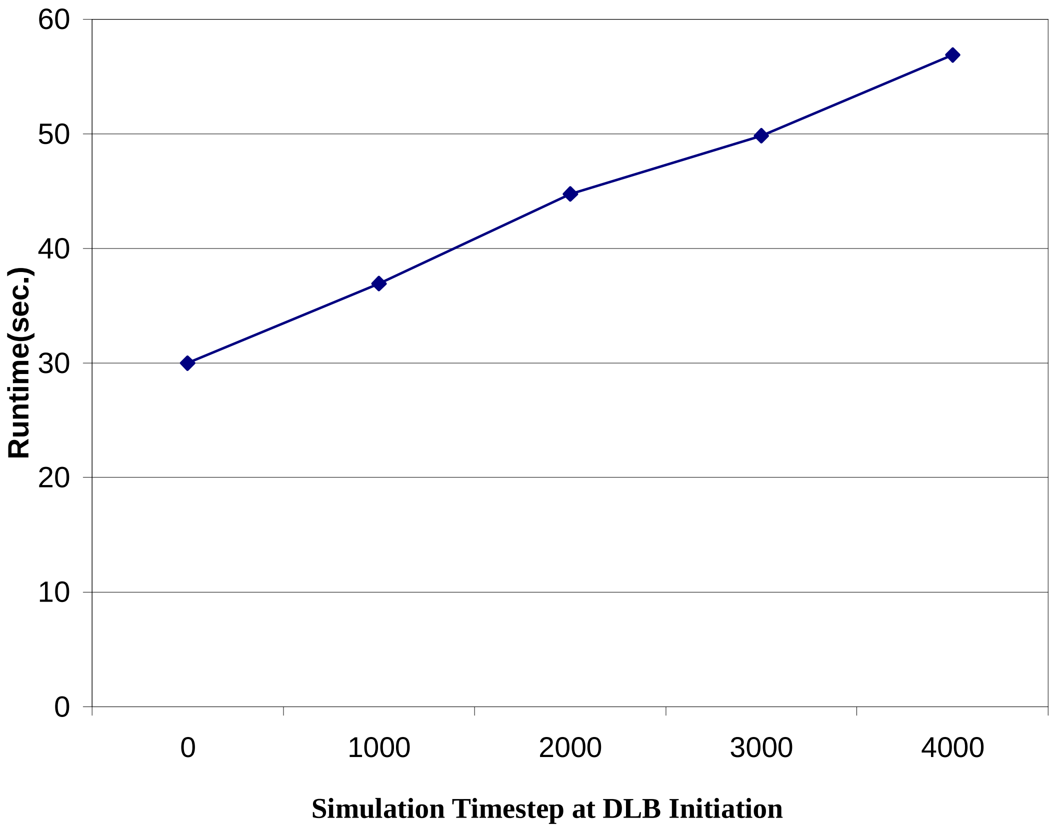


Figure 4.14: Performance of Simulation at Different Initiation of *MaS* Algorithm

create performance bottlenecks. However, if the inter-processor communication is relatively efficient (which is true in this experiments because of the Infiniband fabric [69][70]) the centralized approach to work transfer calculation is simple and efficient. The 2 percentage-point difference is not very significant but whether this difference would improve on larger networks is to be explored in later experiments.

Another reason that *MaS*'s performance was better than diffusion is that with the *MaS* algorithm, excellent work transfer calculation decisions, based on complete system information are achieved, unlike diffusion, which is only aware of the load of its neighbours. This is why *MaS* had a better quality of load balance than diffusion (see table 4.6). Table 4.5 shows the load distributions of different simulation scenarios (without and with DLB experiments) and Table 4.6 shows the quality of load balance for the corresponding simulation scenarios. In Table 4.5, the order of the load distributions correspond to the order of the

Uneven load		Diffusion			MaS			Initially balanced Uncorrected	
ILD	FLD	ILD	LDAD	FLD	ILD	LDAD	FLD	ILD	FLD
430	427	430	3330	3238	430	1590	1551	1590	1520
580	545	580	1160	1165	580	2900	2808	1740	1679
580	591	580	1160	1174	580	1160	1170	1740	1777
580	544	580	1160	1214	580	1160	1213	1740	1789
6230	6293	6230	1590	1609	6230	1590	1658	1590	1635

Table 4.5: Load (vehicles) Distribution of Simulations with and without DLB. ILD (Initial Load Distribution), FLD (Final Load Distribution), LDAD (Load Distribution After DLB).

Uneven load		Diffusion			MaS			Initially balanced Uncorrected	
ILE	FLE	ILE	LEAD	FLE	ILE	LEAD	FLE	ILE	FLE
1.35	1.37	1.35	0.50	0.47	1.35	0.38	0.35	0.04	0.04

Table 4.6: Quality of Load Balancing with and without DLB. ILE (Initial Load Evenness), FLE (Final Load Evenness), LEAD (Load Evenness After DLB).

compute-nodes in which the network partitions were executed.

The Quality of load balance (σ) of Table 4.6 are presented graphically in Fig. 4.15. With the initial load evenness of σ=1.35, diffusion reduced it to σ =0.50 while *MaS* reduced it to σ= 0.38, representing a load evenness improvement of 62% and 71% respectively, after load balancing. It is observed that none of the algorithms is able to achieve the load evenness of the manually balanced network of σ =0.04 due to performance design constraints, such as minimizing partition edges and hence communication overhead, as discussed in section 3.5. As simulation progresses and vehicles migrate from one partition to another, the load distribution changes. Hence, load distributions and the values of σ at the end of the simulation for the 3 experiments (σ = 0.47 for diffusion, σ = 0.35 for *MaS*, and σ= 0.06 for even, uncorrected networks) differ slightly from the initial conditions after the DLB operations. Similarly, the final state of the uneven, uncorrected network remains uneven (σ = 1.37). This suggests that the load has not changed much at all throughout the experiments. Since performance is directly linked to load evenness, the performance of *MaS* is thus better than diffusion, as confirmed by the run-time performance.

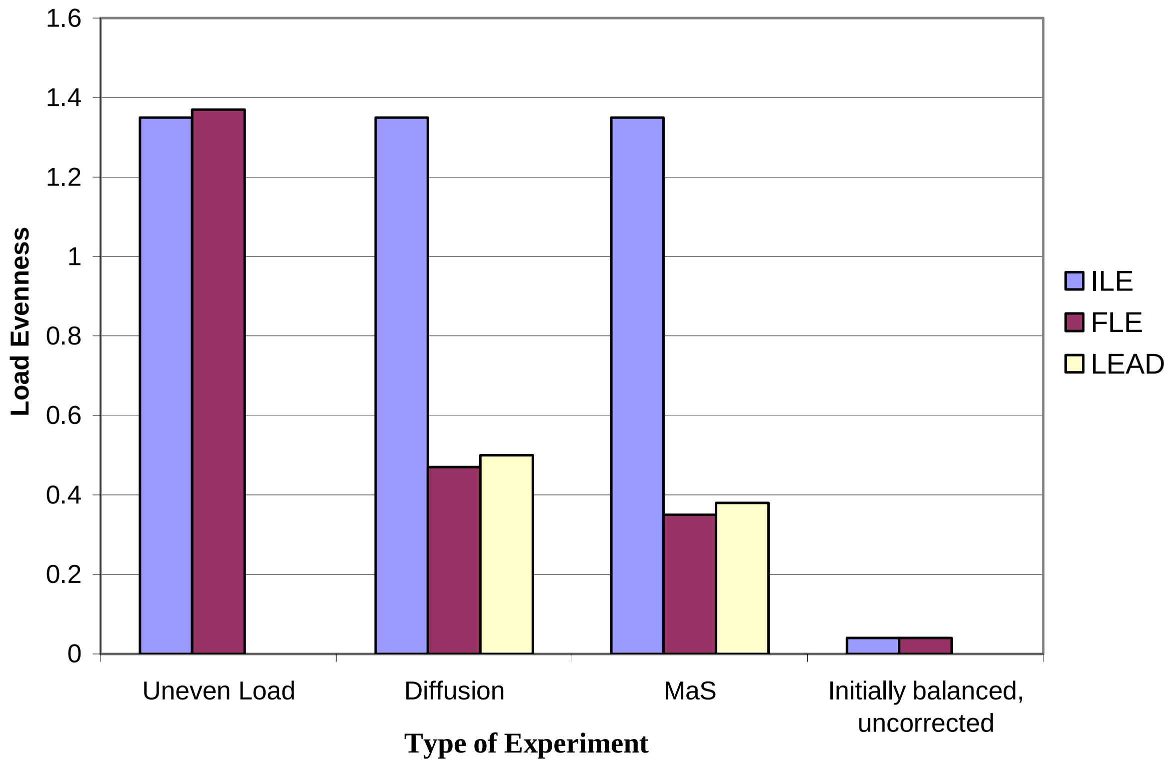


Figure 4.15: Comparison of Load Evenness(σ) of Simulation with and without DLB. ILE (Initial Load Evenness), FLE (Final Load Evenness), LEAD (Load Evenness After DLB).

Conclusion

Load balancing is capable of improving the run-time performance and load evenness of an unbalanced system on a 5 compute-node processor system, when load balancing is initiated once, at start of simulation. *MaS* reduced load evenness from 1.35 to 0.38 while diffusion reduced load evenness from 1.35 to 0.50. In all the experiments, the performance of an almost perfectly evenly balanced system (0.04 evenness) road network is best because it does not have the overheads associated with run-time of dynamic load balancing. In the experiments, *MaS* achieved a 53% improvement of simulation run-time while diffusion achieved a 51% improvement, when both algorithms were applied to a 1.35 load evenness network. The 2 percentage point difference suggests that *MaS* is slightly better than diffusion algorithm for this set of experiments. It was observed that load distribution remained fairly constant throughout the simulation, after load balancing at the start of the simulations.

4.5.4. A Study of the Scalability of the Load Balancing Algorithms (Basic Diffusion Algorithm with global PD(BDA_{PD}) and Basic Centralized Algorithm with global PD (BCA_{PD})

In the previous section, the impact of DLB on 5 compute-nodes were studied. This section studied the impact of DLB on higher number of compute-nodes ranging from 10 to 25 compute-nodes. The purpose of the experiments was to study the scalability of the load balancing algorithms. Scalability is measured in terms of *DLB overhead, run-time and the quality of load balance.* For an algorithm to be scalable, the performances should be relatively stable as the number of compute-nodes and/or problem size increases.

The efficiency of scalability is an important factor of load balancing algorithms. The ability of an algorithm to execute on numerous processors without performance degradation (i.e. performance independent of both system and load size) is an ideal characteristic of any load-balancing algorithm. A scalable algorithm must be both efficient in execution time and the quality of load balance for high degrees of parallelism and large number of compute-nodes. For n number of computational load and p number of processors on which the unit of computation must be executed, scalability efficient implies that the total time taken to execute DLB should be negligible for small values of n and grow no faster than O(p) for large values of n and p [4].

Generally, diffusive algorithms are known to be more scalable than global algorithms [51][46]. The argument for lack of scalability of global method is due to the fact that as the number of compute-nodes increases, the communication overhead between the compute-nodes and the central compute-node presents a bottleneck. This is not the case with diffusion algorithms because regardless of

No of Vehicles	No of Junctions	No of Roads	Time steps
9800	2500	9800	5000

Table 4.7: Description of the Experimental Road Network for 10-25 Compute-nodes

No of nodes	NoDLB Runtime (sec)	*MaS* Runtime (sec.)	*MaS* DLB Overhead	Diffusion Runtime (sec.)	Diffusion DLB Overhead
10	137.29	71.5	1.22	74.65	1.41
15	132.50	64.45	1.73	61.47	1.62
20	118.43	54.84	2.12	47.8	1.92
25	109.49	43.87	2.67	40.32	2.38

Table 4.8: Run-time Overhead of the Load Balancing Algorithms

the number of compute-nodes, every compute-node only communicates with its immediate neighbours.

The experiments were designed with a simplified network model (Table. 4.7) in which individual imbalances arise, one after another, with each imbalance confined to a single computer. In real applications this simplified model may not be realistic but suitable to create synthetic experimental load imbalance (in real life, imbalances may arise on several computers simultaneously, as a result of several unrelated events, or as a result of a single event that involves several computers). The size of the road network and hence the load was kept constant for all the experiments. In this way, the behaviours of the simulation on various compute-nodes can be compared with each other.

Discussion of the Results

In these experiments, the focus is on the effect of cluster scale on the performance of both algorithms (*MaS* and diffusion). Four levels of cluster scale (10, 15, 20, 25) and three sets of empirical measurements are reported and discussed. The first experiment discusses the impact of cluster scale on DLB overhead. The second set of experiments discusses the impact of scalability on the quality of load balancing. The third set of experiments discusses the effect of cluster scale on the run-time of the simulation.

Effect of Scalability on DLB Overhead

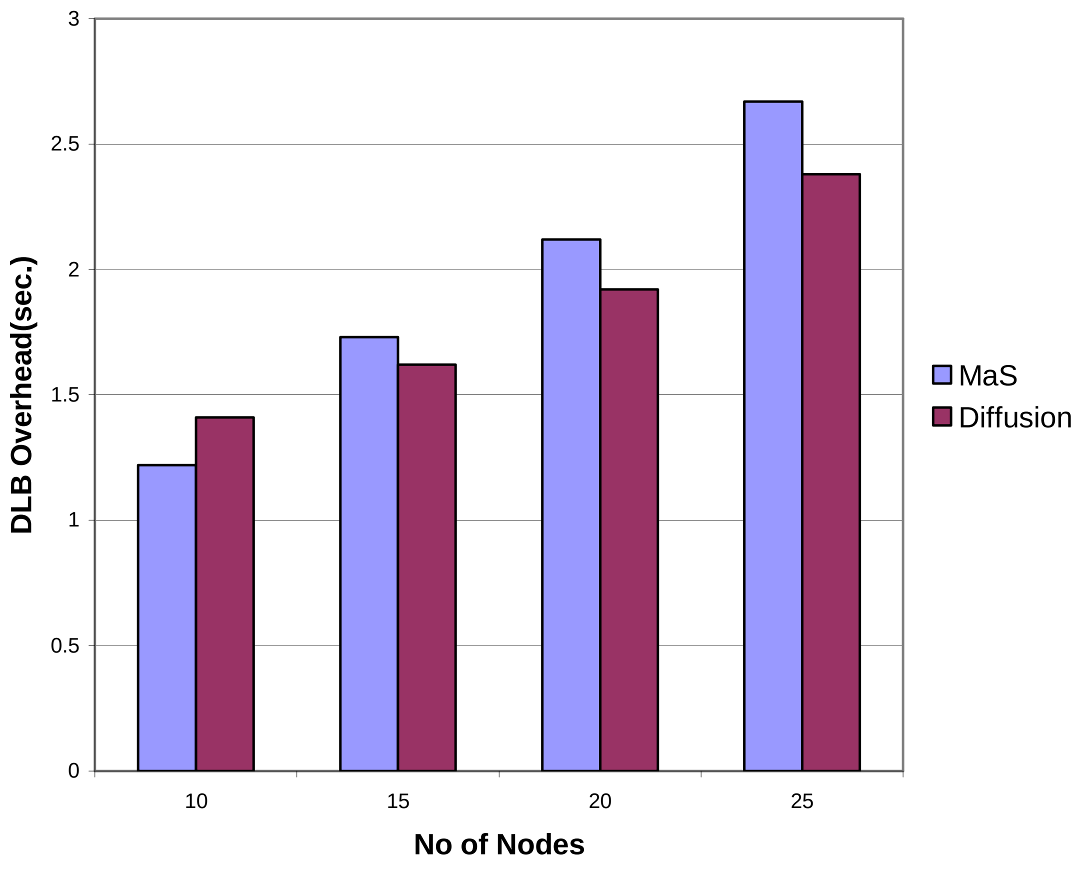


Figure 4.16: Comparisons of the Overhead of *MaS* and Diffusion Algorithms

Fig. 4.16 shows the different values of DLB overhead on a cluster scale of 10, 15, 20 and 25 compute-nodes. It was observed that for the two algorithms, there was linear increase in the DLB overhead as the number of cluster compute-nodes increased from 10 to 25 compute-nodes. These results were expected because as the number of compute-nodes increased, more communication and computation was required to achieve load balance. As the cluster scaled from 10 to 25 compute-nodes, the overhead increased by 1.45 representing a 118% increase for *MaS* and 0.97 representing a 68% increase in overheads for diffusion. The increase in overhead from one cluster scale to another was smaller for *MaS* except on 10

No of Nodes	MaS (FLE)	Diffusion (FLE)	ILE
10	0.08	0.09	2.41
15	0.29	0.37	2.63
20	0.34	0.35	2.62
25	0.15	0.15	2.45

Table 4.9: Quality of Load Balancing(σ) of *MaS* and Diffusion. ILE (Initial Load Evenness), FLE (Final Load Evenness.)

compute-nodes. For example, the increase from 10 to 15 compute-nodes, 15 to 20 compute-nodes and 20 to 25 compute-nodes were 0.51 (41%), 0.39 (22%) and 0.57 (25%) respectively for *MaS* and 0.21 (14%), 0.30 (18%), 0.46 (23%) for diffusion. The comparatively lower percentage increase for diffusion means that diffusion is more scalable than *MaS*.

It is interesting to note from Fig. 4.16 that diffusion showed higher overhead than *MaS* at 10 compute-nodes but on higher compute-nodes, its overhead was comparatively lower. Similar result pattern was observed on 5 compute-nodes in the previous section showing that MaS performed better at lower compute-nodes. In both algorithms, the major overhead of the two algorithms come from communication overhead of the load evaluation calculation. It would be observed, as discussed in section §3.6.3, that both algorithms in order to estimate the profitability of the algorithms, first of all perform a global communication to ensure that the master process receives the load of every compute-node. At this stage, the master process computes for profitability determination. If it is profitable to balance the load however, *MaS* goes straight ahead to analyse the load gradient. Diffusion on the other hand, broadcast to all the compute-nodes to inform them of the load profitability decision. At this stage, all the compute-nodes engage in an individual load calculation and then proceed to the next stage of the calculations.

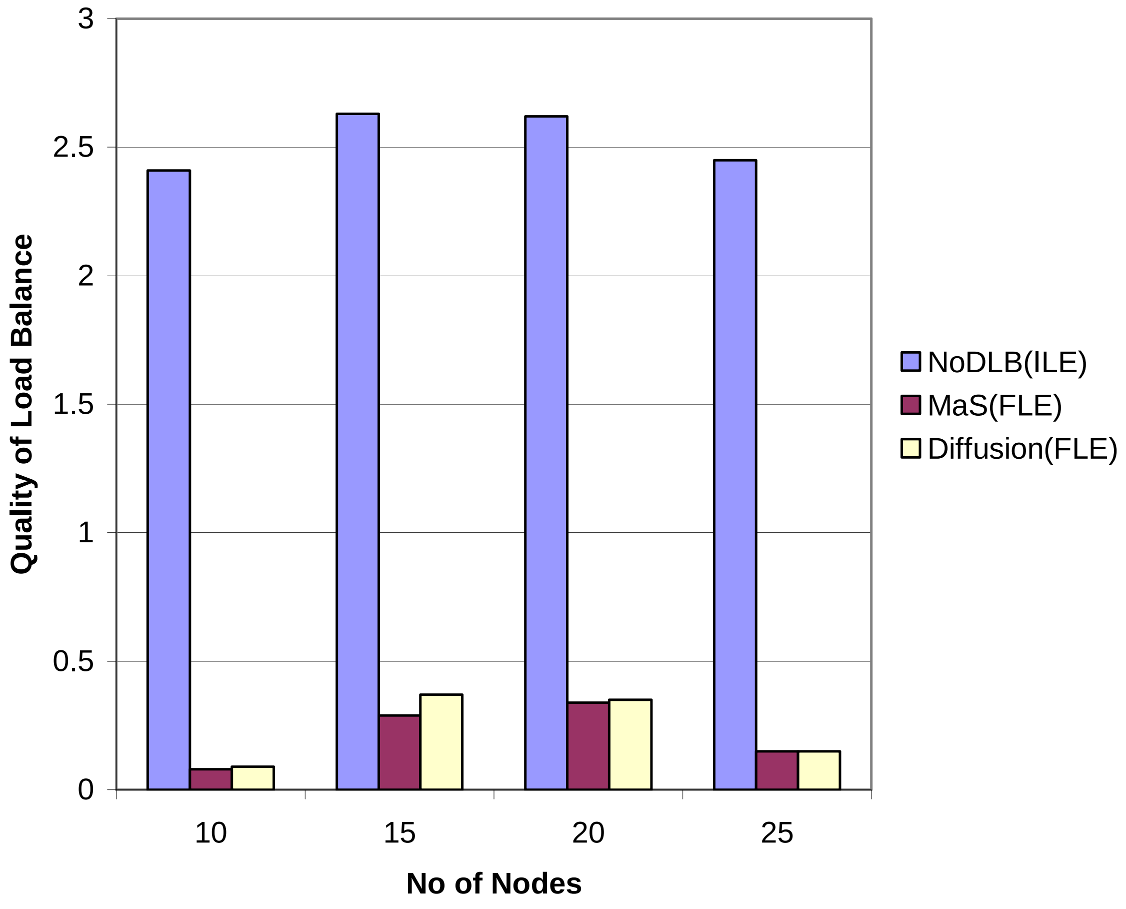


Figure 4.17: Comparisons of the Quality of Load Balancing(σ) of *MaS* and Diffusion Algorithms. ILE(Initial Load Evenness), FLE(Final Load Evenness)

Effect of Scalability on the Quality of Load Balancing (σ)

A critical factor affecting the effectiveness of load balancing algorithms is the quality of load balancing achieved. This is designated σ as discussed in section §4.2. The ideal value is zero, even if this is almost impossible to achieve in reality. However, the aim is to achieve a value as close to zero as possible. The quality of load balancing was calculated for the experiments and Table 4.9 summarizes the results. These figures are plotted in Fig. 4.17. For these set of results, the initial load evenness(ILE) was significantly reduced after the application of DLB algorithms on all the compute-nodes were considered. For example, on 10 compute-nodes, σ was improved from 2.41 to 0.08 representing a 97% improvement for *MaS*. Similarly, there was a 96% improvement on 10 compute-nodes for the diffusion. Comparing the 2 algorithms, *MaS* showed slightly better σ on most of the compute-nodes though the differences were not considered significant. In most cases, the quality of load balance reduced as the cluster scaled from 10 to 25 compute-nodes though this is not a consistent pattern. It was concluded from the significant improvement in the quality of load balance of the 2 algorithms observed on all the compute-nodes that the algorithms are scalable.

Effect of Scalability on the Run-time of the Algorithms

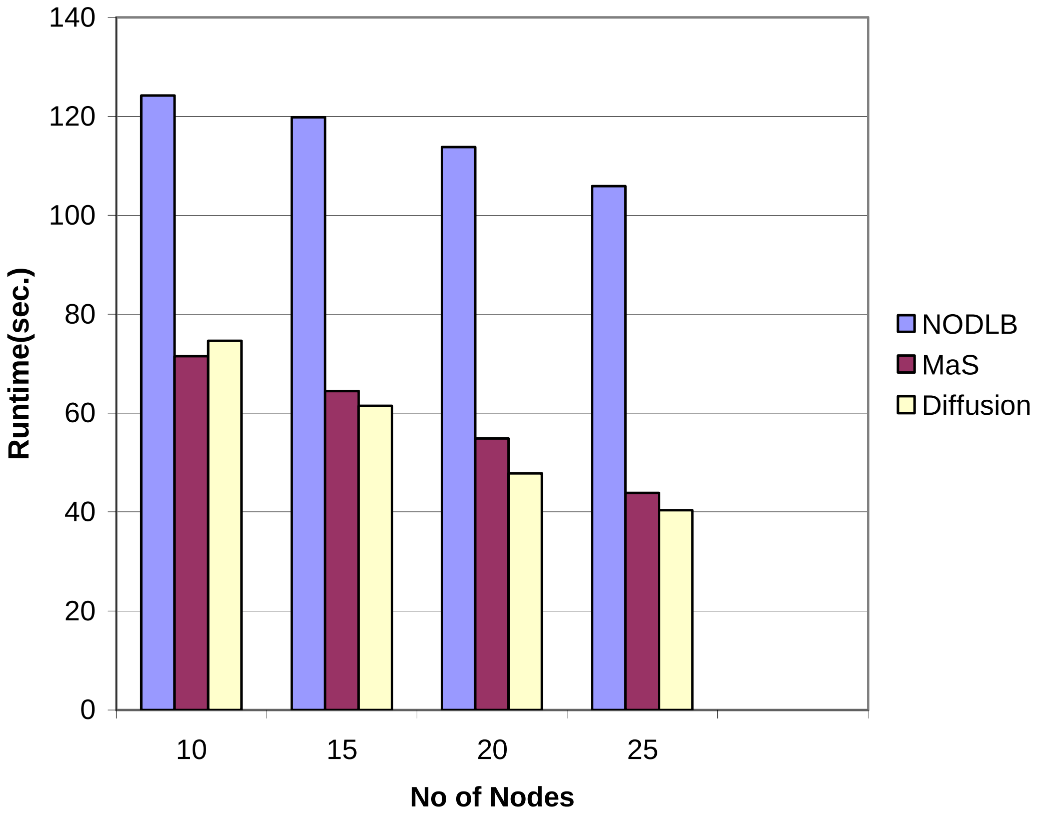


Figure 4.18: Comparisons of the Run-time of *MaS* and diffusion Algorithms and Simulation with no DLB.

For a fixed load size, the run-time of both algorithms reduced as the cluster scaled from 10 to 25 compute-nodes. The reduction in run-time is one of the advantages of parallel processing because as more compute-nodes are added to

No of nodes	*MaS* Improvement over No DLB	*MaS* equivalent % improvement	Diffusion Improvement Over No DLB	Diffusion equivalent % improvement
10	52.68	42	49.53	40
15	55.38	46	58.36	49
20	58.96	52	66.00	58
25	62.01	59	65.56	62

Table 4.10: Effect of DLB on Run-time of the Simulation

solve the same problem, each processor has less load and hence less time is required to compute the problem. However, except for experimental purposes, the load is usually increased as the cluster scales to bigger compute-nodes. In this way, more resources are deployed to compute bigger problems which is the aim of parallel processing. The effect of run-time on the scalability of the algorithms are presented in Fig. 4.18. In both cases, the run-time of the simulation without DLB was higher than the run-time of the 2 algorithms. Table 4.10 (values are calculated from Table 4.8) shows the run-time improvement and the corresponding % run-time improvement as a result of the application of DLB algorithms. On 10 compute-nodes, the run-time was reduced by 52 seconds representing a 42% improvement for *MaS*. Similarly, diffusion was reduced by 50 seconds showing a 40% improvement in run-time. In the table, a significant reduction in run-time was observed on all the compute-nodes showing that DLB has performance advantages even on higher compute-nodes. The ability of the algorithms to run on various number of high compute-nodes also shows that both algorithms are scalable in terms of the run-time.

It was observed that at 10 compute-nodes, diffusion showed higher run-time values than *MaS* (this trend was also observed in the last experiments on 5 compute-nodes), while at higher compute-nodes (15, 20 and 25), diffusions showed lower values than *MaS*. This means that diffusion is better scalable than *MaS* because as the number of compute-nodes increased, the run-time of diffusion was comparatively lower than *MaS*. Again, this confirms to the theoretical expectations discussed in the introductory section that diffusion is more scalable than *MaS*.

Conclusion

The previous section (Effect of Load Evenness on Parallel Traffic Simulation) studied the impact of DLB on 5 compute-nodes and concluded that further experiments would study the effect of DLB on higher number of compute-nodes. The scalability experiments in this section studied the effect of DLB on higher number of compute-nodes ranging from 10 to 25 compute-nodes. The study showed that DLB is capable of improving the performance of parallel simulation on higher number of compute-nodes. Also, both algorithms exhibited good scalability, at least up to the number of compute-nodes considered.

Several features of the impact of cluster scale on the performance of the algorithms were identified. Firstly, load balancing overheads were generally higher as the cluster scale increased. Secondly, for the experiments and the cluster scale considered, the quality of load balancing (σ) of the simulation significantly improved as a result of the DLB algorithms. Thirdly, the run-time of the algorithms decreased as more compute-nodes were used to execute the given problem. This shows that the algorithms are equally capable of executing on higher number of compute-nodes. In comparing the two algorithms, based on the experimental results, it can be observed that diffusion shows slightly better scalability than *MaS*.

On all the compute-nodes, the performances of the 2 algorithms are very competitive. Though *MaS* showed comparatively better performances on lower compute-nodes (5 and 10) while diffusion showed comparatively better performances on higher compute-nodes (15-25), these differences are considered insignificant. These are only indicative of the performance trend of the algorithms as they scale to even higher compute-nodes.

4.6. Chapter Summary

In this chapter, dynamic load balancing was studied using Madcity parallel traffic simulation. Performance evaluation of the parallel simulator was also studied. The parallel Madcity was optimized with four dynamic load-balancing algorithms and their performances evaluated. The work is concentrated on the cost of load evaluation strategy and the scalability of the algorithms.

Some of the factors that affect load-balancing performances were studied. It was found that load balancing has a profound impact on the system behaviour. Under load balancing, the performances of all hosts, even those originally with heavy loads improved under effective load balancing. This is very encouraging and shows when the compute-nodes balance the load with each other, the overall efficiency of the simulation is improved.

The important results from this study include:

- Load balancing reduced the run-time of the parallel application under a wide range of load imbalance.
- Generally, *MaS* showed better performance at low number of compute-nodes (5 and 10 compute-nodes) while diffusion performed better as the cluster scaled to larger number of compute-nodes (15-25 compute-nodes). Apart from scalability, from the research results presented, the two are competitive in their performances. These results are indicative of performance trend on higher cluster scale.

Chapter 5

Contributions to Knowledge and Final Remarks

5.1. Chapter Overview

In the previous chapters, dynamic load balancing has been studied in a number of different compute-nodes using experimental implementations and measurements. With the observations made from the case studies, the following remarks can be made.

In a distributed memory parallel processing platform such as the Mut cluster (the experimental platform), load balancing can improve performance significantly as reported in the experiments performed. For the algorithms studied, the loads of the hosts were made quite even by load balancing, hence, improving the performance of the system. The impact of load balancing on individual hosts is found to be quite uniform since the cluster is homogeneous.

Experiments were performed to study load balancing algorithms/methods developed in this thesis. For profitability determination (PD) algorithm, experiments were first performed to establish an optimum threshold value for profitability determination. This value is called $PD_{threhold}$. This was subsequently used in the load balancing decisions of the algorithms. Experiments were performed with and without PD routines to know if PD helps in load balancing algorithms. It was observed that in both cases of *MaS* and diffusion algorithms, simulations using PD outperformed simulations without PD. Global profitability determination has a general view of the load distribution in the system. Hence, PD makes intelligent choice when it is profitable to perform load balancing and thereby eliminate unnecessary DLB overheads.

Two load indexes were also compared with each other, the vehicles load index (VLI) and the response time load index (RTLI). The vehicles load index outperformed the response time index in most situations especially when the system is homogeneous. The experiments were repeated in a heterogeneously-loaded platform. In this platform, vehicles load index still outperformed response time load index but the difference between the two were closer than in the first case. The advantage of vehicles load index is mainly because of the simplicity of using vehicles as load index. The response time load index incurred additional overheads in converting between RTLI and vehicles. However, the fact that response time load index showed improvement in the heterogeneously-loaded platform environment is encouraging. This shows that as the environment moves towards a heterogenous platform, response time load index shows a qualitative evidence of improvement in performance compared to the vehicles load index, for both *MaS* and diffusion.

Generally, it was observed that the *MaS* algorithms performed better than diffusion in all cases. These are due to the following reasons:

- In the case of *MaS* algorithm, the decision-making compute-node have a view of the entire system such that work transfer calculations were quicker and better in quality. This subsequently lead to performance improvement. Work transfer calculation in diffusion was local such that it was harder to reach a global balanced view of the system. The quality of load balance was also less that that of the *MaS* because the compute-nodes only balanced their lodes with their neighbours.

- Both *MaS* and SPMD utilized master slave communication pattern. That is, *MaS* fitted naturally into the SPMD parallel traffic simulation. This gave *MaS* an advantage over diffusion whose structure was inherently distributed.

5.2. Contributions to Knowledge

- *Design of an adaptive load index and a Profitability determination algorithms*

  A simple adaptive load index was designed using the time per simulation time step (response time load index) per vehicle. This implicitly took into consideration the load of the application (vehicles) and the hardware utilization of the platform (e.g. CPU percentage utilization, memory, and network backbone etc) making it adaptive to the hardware platform.

  Similarly, considering the fact that dynamic load balancing incurs its own overhead, profitability determination is an essential part of any load balancing algorithm because it ensures that load balancing is performed only when it is profitable to do so. A global profitability determination was designed which was based on the most loaded compute-node in the system. The algorithm uses a sorting method to determine the most loaded compute-node and then makes decisions based on this. It is sometimes wise to allow some degree of load imbalance. Therefore, the profitability algorithm ensures that load balancing is performed only when the degree of imbalance is more then a certain degree of imbalance ($PD_{threshold}$), which is determined prior to running the simulation.

- *Design of a centralized load balancing algorithm called MaS*

  In centralized algorithms, load balance advice is estimated by the central compute-node based on the overall load distribution and then distributed to the participating compute-nodes who perform local load adjustments. In some algorithms, the most loaded compute-node can give load directly to the least loaded compute-node to balance the load in the system. In other algorithms, load can only 'diffuse' from the most loaded compute-node to the least loaded compute-node through its neighbours. This method ensures that the boundary partitions between the neighbours are well preserved which in turn reduces the communication overhead.

  The problem with this method is that at the time the participating compute-nodes receive the load balance advice, some compute-nodes don't yet necessarily have enough load to meet the requirements of the load to give to its neighbours. Thus the compute-nodes are sometimes required to give out what they don't yet have. Such compute-nodes need to wait and receive enough load from the overloaded neighbours through the neighbours before they have enough to give out. The design in this thesis solved this problem by ensuring that the compute-nodes give out their load only when they have received sufficient load from the neighbours.

  In calculating the load to transfer, it starts with the most loaded compute-node. Once load has been recalculated, it reorders the load and then goes to the most loaded processor. This process is repeated until the load is balanced, all in one time step. The design of the load evaluation is enforced in the load migration stage in which the decision-making processor supervises the load migration procedure by handling control to the most loaded compute-node in each load migration stage. This ensures that compute-nodes have enough load before they are allowed to give out load to their neighbours. A unique feature of the design is that all these operations happen in one time step of simulation. The algorithm uses a *response time* load index as well as a *profitability determinant* based on the most loaded compute-node in the system, which were all designed in this thesis.

5.3. Remarks on Future Work

Load balancing is a research topic with many dimensions, and involves a large number of research issues. The problems believed to be of fundamental importance to parallel traffic simulation, as observed in the literature review, were studied. This work is the first in which suitable load balancing algorithms for traffic simulation and their performances were studied. The work may be extended in a number of directions.

1. Load balancing was mainly treated as a performance optimization research and studied on a relatively small number of compute-nodes compared to the sizes of recent clusters. More experiments on the scalability of the algorithms would need to be done with higher number of compute-nodes.
2. The research hardware platform is considered homogeneous. Though heterogenous environments were simulated in this research, a logical extension of this work (especially for response time load index) would be to experiment on a heterogeneous platform such as the Grid. Another advantage for the grid platform is that as microscopic traffic simulations get more complex with the inclusion of other modules such as the Intelligent Transport Systems (ITS), more hardware research platforms would be needed and the grid provides such a platform.
3. More features and comparisons of the different designed algorithms need to be studied. For example, different heterogeneously-loaded platform can be simulated so that more studies can be done on the algorithms. A different set of experiments could be studied using randomly generated load.
4. A wide range of applicable algorithms revised in the literature review still

remains to be studied. Research effort into the study and comparisons of the performances of other algorithms such as *dimension exchange, gradient model*, etc should be carried out.

Publications During Research

D.Igbe, N.Kalantery, S.E Ijaha, S.C Winter (2002). Parallel Traffic Simulation in Spider Programming Environment. *In Distributed and parallel systems (Cluster and Grid computing).* Edited by Peter Kacsuk et. al. Kluwer academic publishers, page 165-172.

D.Igbe, N.Kalantery, S.E Ijaha, S.C Winter (2003). An Open Interface for the Parallelization of Traffic Simulators. *In proceedings of the 7th IEEE international symposiums on distributed simulations and real time applications*, page 158-163.

Bibliography

[1] Barceló, J., J. Ferrer, D. García, and R. Grau (1998). Microscopic traffic simulation for ATT systems analysis. A parallel computing version. Available Online. http://www.tss-bcn.com/documents.html.

[2] Barnard, S. T. and H. D. Simon (1994). A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience 6*, 101–107.

[3] Beguelin, A., J. J. Dongarra, G. A. Geist, R. Mancheck, and V. S. Sunderam (1991). PVM: A user's guide to PVM parallel virtual machine. *Oak, Ridge National Laboratory, ORNL/TM-11826*, 315 – 339.

[4] Berger, E. and J. Browne (1999). Scalable load distribution and load balancing for dynamic parallel programs. Available online. http://citeseer.ist.psu.edu/berger99scalable.html.

[5] Bhatelé, A., L. V. Kalé, and S. Kumar (2009). Dynamic topology aware load balancing algorithms for molecular dynamics applications. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, New York, NY, USA, pp. 110–116. ACM.

[6] Boillat, J. E. (1990). Load balancing and poisson equation in a graph. *Practice and Experience 2*, 289–313.

[7] Bokhari, S. H. (1981). On the mapping problem. *IEEE Trans. on Computers*, 550–557.

[8] Bokhari, S. H. (1987). *Assignment problems in Parallel and Distributed Computing.* Kluwer Academic publishers.

[9] Bokhari, S. H. (1988). Partitioning problems in parallel,pipelined and distributed computing. *IEEE transactions on computers 37*, 48–57.

[10] Bowen, N. S., C. N. Nikolau, and A. Ghafoor (1992). On the assignment problem of arbitrary process systems to heterogeneous distributed computer systems. *IEEE transactions on computers 41*, 257–273.

[11] Branco, K., M. Santana, R. Santana, and S. Bruschi (2006, July). Piv and wpiv: Performance index for heterogeneous systems evaluation. In *Industrial Electronics, 2006 IEEE International Symposium on*, Volume 1, pp. 323–328.

[12] Brugè, F. and S. L. Fornili (1990). A distributed dynamic load balancer and its implementation on multi-transputer systems for molecular dynamics simulation. *Computer Physics Communications 60*, 39–45.

[13] Bui, T. and C. Jones (1993). A heuristic for reducing fill in sparse matrix factorization. *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 445–452.

[14] Burghout, W. (2004, December). *Hybrid Microscopic-Mesoscopic Traffic Simulation*, Volume Trita-INFRA, ISBN 1651-0216. Stockholm: KTH, Infrastructure.

[15] Cameron, G. D. B. and C. I. D. Duncan (1996). Paramics-parallel microscopic simulation of road traffic. *J. Supercomputing 10*, 25–53.

[16] Campos, L. M. and I. D. Scherson (2000). Rate of change of load balancing in distributed and parallel systems. *Parallel Computing 26*, 1213–1230.

[17] Casavant, T. L. and J. G. Kuhl (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering 14*, 141–154.

[18] Catalyurek, U., E. Boman, K. Devine, D. Bozdag, R. Heaphy, and L. Riesen (2007, March). Hypergraph-based dynamic load balancing for adaptive scientific computations. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–11.

[19] Chang, G. L., T. Junchaya, and A. Santiago (1994). Real-time network traffic simulation for ATMS applications: Part 1-simulation methodologies. *IVHS journal 1*, 241–245.

[20] Chaube, R., I. Banicescu, and R. Cariño (2008). Parallel implementations of three scientific applications using lbmigrate. In *IPDPS*, pp. 1–8. IEEE.

[21] Chaudhary, V. and J. Aggarwal (1993). A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Paralel and Distributed Systems*, 328–346.

[22] Chisman, J. A. (1992). *Introduction to Simulation Modelling using GPSS/PC.* Prentice-hall, Englewood Cliffs, New Jersey.

[23] Choi, H. and B. Narahari (1991). Algorithms for mapping and partitioning chain structured parallel computations. *Proceedings of International Conference on Parallel Procesising*, 625–628.

[24] Corradi, A., L. Leonardi, and F. Zambonelli (1999). Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency 7*(1), 22–31.

[25] Corradi, A., L. Leonardi, and F. Zambonelli (2001). Parallel object allocation via user-specified directives: A case study in traffic simulation. *Parallel Computing 27*(3), 223–241.

[26] Cybenko, G. (1989). Dynamic load balancing for distributed memory multiprocessors. *J. of Parallel Distributed Computing 7*, 279–301.

[27] Cynthia Bailey Lee and Scott Baden (2002). A parallel simulation of traffic. Available online. `http://www.cs.ucsd.edu/users/clbailey/ParallelTrafficSimulator.pdf`.

[28] Deliverable (2000, July). Open framework for simulation of transport strategies and assessment (OSSA)DG TREN, GRD1-10917. D1.1 User and policy requirements.

[29] Deliverable (2002, July). Open framework for simulation of transport strategies and assessment (OSSA)DG TREN, GRD1-10917. D1.1 User and policy requirements.

[30] Devine, K. and J. Flaherty (1996). Parallel adaptive hp-refinement techniques for conservation laws. *Numer. Math.*, 367Ű386.

[31] Diekmen, R. and B. Monien (1999). Load balancing strategies for distributed memory machines. *Parallel and Distributed Processing for Computational Mechanics: Systems and Tools, B.H.V Topping (ed), Saxecoburg*, 124–157.

[32] Diekmenn, R., A. Frommer, and B. Monien (1999). Efficient schemes for nearest neighbour load balancing. *Parallel Computing 32*, 789–812.

[33] Dongarra, J. J. and et. al. (1994). The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing 20*, 531 – 547.

[34] Driessche, R. V. and D. Roose (1995). Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem. In B. Hertzberger and G. Serazzi (Eds.), *High Performance Computing and Networking*, Number 919, pp. 392–397. Springer.

[35] Duan, Z. and Z. Gu (2008). Dynamic load balancing in web cache cluster. *Grid and Cooperative Computing, International Conference*, 147–150.

[36] Eager, D. L., E. Lazowska, and J. Zahorjan (1986). Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on soft. engineering 12*, 662–675.

[37] Ferarri, D. and S. Zhou (1987). An empirical investigation of load indices for load balancing applications. *proceddings of performance'87 international symposium on computer Performance Modeling,Measurement and Evaluation*, 515–528.

[38] Ferscha, A. and S. K. Tripathi (1994). Parallel and distributed simulation of discrete event systems. Technical Report CS-TR-3336.

[39] Fishwick, P. A. (1995). Computer simulation: The art and science of digital world construction. Available Online. http://www.cis.ufl.edu/ fishwick/introsim/paper.html.

[40] Franklin, M. A. and V. Govindan (1996). A general matrix iterative model for dynamic load balancing. *Parallel Computing 22*(7), 969–989.

[41] Furuichi, M., T. Taki, and N. Ichiyoshi (1990). A multi-level load balancing scheme for or-parallel exhaustive search programs. *SIGPLAN notices 25*, 50–59.

[42] Hac, A. and T. J. Johnson (1986). A study of dynamic load balancing in a distributed system. *Proc. ACM SIGCOMM Symposium on Communication, Architecture, and Protocols 16*(3), 348 – 356.

[43] Han, X., D. Chen, and J. Chen (2009). One centralized scheduling pattern for dynamic load balance in grid. *Information Technology and Applications, International Forum on 2*, 402–405.

[44] Harget, A. J. and I. Johnson (1990). *Load balancing algorithms in loosely-coupled distributed systems: A survey.* Distributed computer systems: Theory and Practice.

[45] Heirich, A. (1997). A scalable diffusion algorithm for dynamic mapping and load balancing on networks of arbitrary topology. *The International Journal of Foundations of Computer Science 8*, 329–346.

[46] Heirich, A. (1998). *Analysis of Scalable Algorithms for Dynamic Load Balancing and Mapping with Application to Photo-realistic Rendering.* Ph. D. thesis, California Institute of Technology, Pasadena.

[47] Hendrickson, B. and K. Devine (2000). Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engeneering 184*, 485–500.

[48] Hill, J. M. and D. Skillicorn (1996). Practical barrier syncronisation. Article PRG-TR-16-96, Oxford University Computing Laboratory, Wofson Building, Parks Road, Oxford, OX1 3QD. Programming Research Group.

[49] Horton, G. (1993). A multilevel diffusion method for dynamic load balancing. *Parallel Computing 19*, 209–218.

[50] Hu, Y. F. and R. J. Blake (1995). An optimal dynamic load balancing algorithm. Technical Report DL-P-95-011.

[51] Hu, Y. F. and R. J. Blake (1999). An improved diffusive algorithm for dynamic load balancing. *Parallel Computing 25*, 417 – 444.

[52] IBM (2005). Ibm informix dynamic server performance guide. Available Online. http://publib.boulder.ibm.com/infocenter/idshelp/v10/index.jsp?topic=/com.ibm.perf.doc/perf48.htm.

[53] Igbe, D., N. Kalantery, S. Ijaha, and S. C. Winter (2002). Parallel traffic simulation in Spider programming environment. *Distributed and Parallel Systems (Cluster and Grid Computing)*, 165–172.

[54] Igbe, D., N. Kalantery, S. Ijaha, and S. C. Winter (2003). An open interface for the parallization of traffic simulators. *Proceedings of the 7th IEEE International Symposiums on Distributed Simulations and Real Time Applications)*, 158–163.

[55] Ijaha, S. E., S. C. Winter, N. Kalantery, and B. K. Daniels (1998). Hipertrans: A road traffic simulation as an operational tool. *IEE International conferences on simulation*, 85–92.

[56] Ijaha, S. E., S. C. Winter, N. Kalantery, and B. K. Daniels (2000). Hipertrans: High performance transport network modelling and simulation. *6th International Euro-par conference*, 269–274.

[57] Kaddoura, M. and S. Ranka (1997). Runtime support for parallelization of data-parallel applications on adaptive and nonuniform computational environments. *Journal of Parallel Computing and distributed computing 43*, 163–168.

[58] Karypis, G. and V. Kumar (1995). Parallel multilevel graph partitioning. Technical Report TR 95-036.

[59] Karypis, G., V. Kumar, A. Grama, and A. Gupta (1994). *Introduction to parallel computing: Design and analysis of paralel algorithms.* Benjamin/Cummings.

[60] Kernighan, B. and S. Lin (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal 49*, 291–307.

[61] Kernighan, B. and S. Lin (1973). An effective heuristic algorithm for the traveling salesman problem. *Operations Research 21*(2), 498–516.

[62] Kirkpatrick, S. and et al. (1984). Optimization by simulated annealing. *Science 220*, 671–680.

[63] Kumar Chandra, P. and B. Sahoo (2008, Nov.). Dynamic load distribution algorithm performance in heterogeneous distributed system for i/o- intensive task. In *TENCON 2008 - 2008, TENCON 2008. IEEE Region 10 Conference*, pp. 1–5.

[64] Kunz, T. (1991). The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering 17*(17), 725–730.

[65] Lan, Z., V. E. Taylor, and G. Bryan (2001). Dynamic load balancing of samr applications on distributed systems. In *In Proceedings of the ACM/IEEE Symposium on Supercomputing (SCŠ01). IEEE Computer*. Society Press.

[66] Law, A. M. and W. D. Kelton (1991). *Simulation Modelling and Analysis*. McGraw-Hill, Second edition.

[67] Lieberman, E. and A. K. Rathi (1992). Traffic flow theory: A state of the art report. Available Online. http://www.tfhrc.gov/its/tft/tft.htm.

[68] Lin, F. C. H. and R. M. Keller (1987, Jan). The gradient model load balancing method. *IEEE Transactions on Software Engineering 13*(1), 32–38.

[69] Liu, J., B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kinis, P. Wyckoff, and D. Pand (2003). Micro-benchmark level performance comparison of high-speed cluster interconnects. Available Online. http://citeseer.ist.psu.edu/liu03microbenchmark.html.

[70] Liu, J., B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. Kinis, P. Wyckoff, and D. Pand (2004). Micro-benchmark performance comparison of high-speed cluster interconnects. *Journal of Parallel and Distibuted Computing 24*, 42– 51.

[71] Lo, V. A. (1988). Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on computers*, 1384–1397.

[72] Lucas, K. R. and E. D. Morenno (2006). Load indices-past,present and future. *International conference on Hybrid Information Technology 02*(06), 206–214.

[73] Lüling, R., B. Monien, and F. Ramme (1991). Load balancing in large networks: A comparative case study. In *3th IEEE Symposium on Parallel and Distributed Processing*.

[74] Luque, E., A. Ripoll, A. Cortes, and T. Margalef (1995). A distributed diffusion method for dynamic load balancing on parallel processors. *Proceedings of the Euromicro on Parallel and Distributed Processing*, 43–50.

[75] Martin, O. C. and S. W. Otto (1994). Partitioning of unstructured meshes for load balancing. *Concurrency and Computation: Practice and Experience 7*, 303 – 314.

[76] Mehra, P. and B. W. Wah (1993). Automated learning of workload measures for load balancing on a distributed system. In *Distributed System, in Intl. Conference on Parallel Processing*, pp. 263–270. CRC Press.

[77] Muniz, F. J. and E. J. Zaluska (1995). Parallel load balancing: an extension to the gradient model. *Parallel Computing 21*, 287–301.

[78] Nagel, K. and M. Rickert (2000). Parallel implementattion of the transims micro-simulation. *Parallel Computing 27*, 1611–1639.

[79] Nian, S. and L. Guangmin (2009). Dynamic load balancing algorithm for mpi parallel computing. *New Trends in Information and Service Science, International Conference 0*, 95–99.

[80] Nicol, D. M. and D. R. O'Hallaron (1991). Improved algorithms for mapping pipelined and parallel computations. *IEEE transactions on computers 40*, 295–306.

[81] Nicol, D. M., R. Simha, and D. Towsley (1996). Static assignment of stochastic tasks using majorization. *IEEE transactions on computers 45*, 730–740.

[82] Oliker, L. and R. Biswas (1998). Plum: parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput. 52*(2), 150–177.

[83] Overeinder, B. J., L. O. Hertzberger, and P. M. A. Sloot (1991, May 15). Parallel discrete event simulation. In W. J. Withagen (Ed.), *The Third Workshop Computersystems*, Eindhoven, The Netherlands, pp. 19–30.

[84] Plastino, A., C. C. Ribeiro, and S. Lifschitz (1997). Exploring load balancing in parallel processing of recursive queries. *In Proceedings of the 3rd intl. Euro-Par Parallel processing conference,Passau*, 1125–1129.

[85] Plastino, A., C. C. Ribeiro, and N. Rodriguez (1999). Load balancing algorithms for spmd applications. Available online. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.50.4157.

[86] Procassini, R. J. and et al. (September 2005). Load balancing of parallel monte carlo transport calculations. International Topical meeting on Mathematics and Computations. UCRL-PROC-212700.

[87] Rickert, M. (1998). *Traffic Simulation on Distributed Memory Processors.* Ph. D. thesis, University of Cologne. Available online from http://www.the-rickerts.de/mr/dissertation.en.html.

[88] Rickert, M., P. Wagner, and C. Gawron (1996). Real-time traffic simulation of the german autobahn network.

[89] Ross, K. W. and D. D. Yao (1991). Optimal load balancing and scheduling in a distributed computer system. *Journal of ACM 38*, 676–690.

[90] Schloegel, A., G. Karypis, V. Kumar, R. Biswas, and L. Oliker (1998). A performance study of diffusive vs. remapped load-balancing schemes. *Parallel and Distributed Computing Systems 11*, 1–12.

[91] Schloegel, K., G. Karypis, and V. Kumar (1997). Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Technical Report TR 97-014.

[92] Schloegel, K., G. Karypis, and V. Kumar (2001). Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems 12*(5), 451–466.

[93] Schulze, T. and T. Fliess (1997). Urban traffic simulation with psycho-physical vehicle-following models. In *Winter Simulation Conference*, pp. 1222–1229.

[94] Shirazi, B. A., A. R. Hurson, and K. M. Kavi (1995, April). *Scheduling and Load Balancing in Parallel and Distributed Systems.* Wiley-IEEE Computer Society Press.

[95] Song, J. (1994). A partially asyncronous and iterative algorithm for distributed load balancing. *Parallel Computing 20*, 853–868.

[96] Sunderam, V. S. (1990). PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 315 – 339.

[97] Tian, J.-F., Y.-L. Liu, and R.-Z. Du (2005, Aug.). Design and analysis of load balancing mode mobile agent-based in DDSS. In *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, Volume 2, pp. 788–796 Vol. 2.

[98] Tibaut, A. (1999). Parallel traffic simulation with an algorithm for adaptive load-balancing. Available online. http://euklid.bauing.uni-weimar.de/ikm1997/PROC97/DOCS/061/061IKM97.PDF.

[99] Tropper, C. (2002). Parallel discrete-event simulation applications. *Journal of Parallel and Distributed Computing 62*(3), 327 – 335.

[100] Vee, V. and W. Hsu (1999). Parallel discrete event simulation: a survey. Available Online. http://citeseer.ist.psu.edu/vee99parallel.html.

[101] Walsaw, C., M. Cross, and M. Everett (1995). A localized algorithm for optimizing unstructured mesh partitions. In *International Journal of High Performance Computing Applications*, pp. 280–295.

[102] Walsaw, C., M. Cross, and M. Everett (1997). Parallel dynamic graph-partitioning for unstructured meshes. Volume 2, pp. 102–108.

[103] Walshaw, C. and M. Berzins (1995a, July). Dynamic load-balancing for pde solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience 7*(1), 17–28.

[104] Walshaw, C., M. Cross, and M. Everett (1995, Jan). A parallelisable algorithm for optimising unstructured mesh partitions. Technical report, Wellington St, Woolwich, London SE18 6PF, UK.

[105] Walshaw, C. H. and M. Berzins (1995b, Feb). Dynamic load-balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice & Experience 7*(1), 17–28.

[106] Waraich, S. S. (2008). Classification of dynamic load balancing strategies in a network of workstations. In *ITNG '08: Proceedings of the Fifth International Conference on Information Technology: New Generations*, Washington, DC, USA, pp. 1263–1265. IEEE Computer Society.

[107] Watts, J., M. Rieffel, and S. Taylor (1996). Practical dynamic load balancing for irregular problems. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pp. 299–306.

[108] Watts, J. and S. Taylor (1998). A practical approach to dynamic load balancing. *IEEE Transactions on Parallel and Distributed System 9*, 235–248.

[109] Wheat, S. (1992). *A fine-grained data migration approach to application load balancing on MP MIMD machines.* Ph. D. thesis, The University of New Mexico,Department of Computer Science, Albuquerque, NM.

[110] Wheat, S., K. Devine, and A. Maccabe (1994). Experience with automatic, dynamic load balancing and adaptive finite element computation. Available Online. http://citeseer.ist.psu.edu/wheat94experience.html.

[111] Willebeek-LeMair, M. and A. Reeves (1993). Strategies for dynamic load balancing on parallel processors. *IEEE transactions on Parallel and Distributed System 4*, 979 – 993.

[112] Wolffe, G. S. and et al. (1997). An empirical study of workload indices for non-dedicated, heterogeneous systems. *Proceedings of PDPTA'97 v1*, 470–478.

[113] Xu, C., F. C. M. Lau, and R. Diekmann (1997). Decentralized remapping of data parallel applications in distributed memory multiprocessors. *Concurrency: Practice and Experience 9*(12), 1351–1376.

[114] Xu, C. Z. and F. Lau (1992). Analysis of the generalized dimension exchange method for dynamic load balancing. *Parallel and Distributed Computing 16*, 385–393.

[115] Xu, C. Z. and F. Lau (1997). *Load Balancing in Parallel Computers: Theory and Practice.* Kluwer Academic Publishers.

[116] Xu, C. Z., B. Monien, R. Luling, and F. Lau (1995). An analytical comparison of nearest neighbour algorithms for load balancing. *IEEE Processor Society Press 112*, 472–479.

[117] Xu, J. and K. Hwang (1993). Heuristic methods for dynamic load balancing in a message-passing processor. *Parallel and Distributed Computing 18*, 1–13.

[118] Zaki, M. J., W. Li, and S. Parthasarathy (1997a). Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing 43*(2), 156–162.

[119] Zaki, M. J., W. Li, and S. Parthasarathy (1997b). A multi-level load balancing scheme for OR-parallel exhaustive search programs on the multi-PSI. *Journal of Parallel and Distributed Computing 43*(2), 156–162.

[120] Zhang, D., C. Jiang, and S. Li (2007). Research on dynamic load balancing algorithms for parallel transportation simulations. In *APPT*, Volume 4847 of *Lecture Notes in Computer Science*, pp. 560–568. Springer.

[121] Zhou, S. (1987). *Performance Studies of Dynamic Load Balancing in Distributed Systems*. Ph. D. thesis, EECS Department, University of California, Berkeley.