Universitat
Autònoma
de Barcelona

A new distributed diffusion algorithm for dynamic load-balancing in parallel systems

Departament d'Informàtica
Unitat d'Arquitectura d'Ordinadors
i Sistemes Operatius

A thesis submitted by Ana Cortés Fité in fulfilment of the requirements for the degree of *Doctor per la Universitat Autònoma de Barcelona*.

Barcelona (Spain), September 2000

A new distributed diffusion algorithm for dynamic load-balancing in parallel systems

Thesis submitted by Ana Cortés Fité in fulfilment of the requirements for the degree of *Doctor per la Universitat Autònoma de Barcelona*. This work has been developed in the Computer Science Department of the *Universidad Autònoma de Barcelona* and was advised by Dra. Ana Ripoll Aracil,

Bellaterra September, 2000

Thesis Advisor

Ana Ripoll

Ana Ripoll Aracil

A la meva filla Júlia:

"... mami explica'm aquest conte."

ACKNOWLEDGEMENTS

A lot of people have made this work possible. I wish to express my sincere gratitude to them all for being there, for working with me and for helping me.

First of all I want to thank Ana Ripoll Aracil for being my advisor throughout this work, for her constant advice and never-ending encouragement. I have learned from her how to organise basic concepts and new ideas, and how to describe and transmit them with clarity.

I would like to express my gratitude to Emilio Luque for his suggestions at the outset of this work, as well as for his inestimable contribution to its development at certain critical moments.

My deepest thanks go to Miquel Angel Senar for his technical support and the clarity of his discussion of this work, and, particularly, for not being discouraged at my own discouragement, for bringing calm and good sense in moments of desperation and madness and, above all, for being there at all times. Without his support and affection this work would never have reached its conclusion.

Thanks to Ferran Cedó, member of the Department of Mathematics at the *Universitat Autònoma de Barcelona*, for his mathematical support in the formal detailed descriptions and analysis of this work.

To Tomàs Margalef for thinking of me at the very beginning of this work, and for getting me under starters' orders.

To Dani Franco and Indira Garcés for lending me the beta version of their network simulator in order to obtain "deadlock" free reliable communication times.

I would also like to thank María Serrano, Lola Reixachs, Eduardo César and Elisa Heymann for putting up with long talks with me without ever losing their patience.

Thanks to Josep Pons and Jaume Jo who have collaborated in the implementation of some of the programs related to this work.

Special thanks to Porfidio Hernández for his academic help during the period in which I was intensively preparing my lectures.

I am also grateful to the rest of my colleagues from the Computer Architecture and Operating Systems Group as well as to those who have passed through the Group during the gestation and preparation periods of this work, for their constant support and encouragement.

Last, but decidedly not least, I would like to thank my family for their support throughout this time, and especially my daughter Júlia who, without being aware of it, has given me the strength to cope with the work's tough final stage, and who has had to suffer in ignorance all of my many and various moods and frames of mind.

Contents

CHAPTER 2

CHAPTER 3

CHAPTER 4

COMPARATIVE STUDY OF NEAREST-NEIGHBOUR LOAD-BALANCING ALGORITHMS

CHAPTER 5

CHAPTER 6

CHAPTER 7

PREFACE

Advances in hardware and software technologies have led to increased interest in the use of large-scale parallel and distributed systems for database, real-time, and large-scale scientific and commercial applications. The operating systems and management of the concurrent processes constitute integral parts of the parallel and distributed environments. One of the biggest issues in such a system is the development of effective techniques for the distribution of processes among processing elements to achieve some performance goal(s), such as minimising execution time, minimising communication delays, and/or maximising resource utilisation. Load-balancing is one of the most important problems which have to be solved in order to enable the efficient use of multiprocessor systems. Load-balancing aims at improving the performance of multiprocessor systems by equalising the computational load over all processors in the system since it is commonly agreed that equally balancing loads between all processors in the system directly leads to a minimisation of total execution time.

There are applications that can be partitioned into tasks with regular computation and communication patterns and, therefore, load-balancing algorithms can be used to assign computational tasks to processors before beginning the execution. This is called static load-balancing. However, there is an important and increasingly common class of scientific applications (such as particle/plasma simulations, parallel solvers for partial differential equations, numerical integration, N-body problem to name just a few) where the computational load associated with a particular processor may change over the course of a computation and cannot be estimated beforehand. For this class of non-uniform problems with unpredictable *a priori* computation and communication requirements, dynamic load-balancing algorithms are needed to efficiently distribute the computational load at run time on the multiprocessor system. This work is about dynamic load-balancing in message-passing parallel computers where, in general, a direct, point-to-point interconnection network is used for communication. Load-balancing is performed by transferring load from heavily to lightly loaded processors. For that purpose, a load-balancing algorithm has to resolve the issues of *when* to invoke a balancing operation, *who* makes load-balancing decisions according to what information and *how* to manage load migrations between processors. We can find several answers to these questions which results in a wide set of load-balancing techniques.

A highly popular class of load-balancing strategies are nearest-neighbour approaches which are edge-local, that is, methods that can be implemented in a local manner by each processor consulting only its neighbours, thereby avoiding expensive global communication in distributed applications. The load moved along each edge is related to the gradient in the loads across it. These kinds of distributed load-balancing algorithms are appealingly simple and they degrade gracefully in the presence of asynchrony and faults. Most of these algorithms are implemented in an iterative way to achieve a global load balanced state and, therefore, they are referred to as iterative load-balancing algorithms. The load balance stems from successive approximations to a global optimal load distribution by only being concerned with local load movements. In the ideal case, a perfect state is achieved when all processors have the same load. These kinds of load-balancing algorithms are suited to appreciably decreasing large imbalances.

Most iterative load-balancing algorithms proposed in the literature consider an idealised version of the load-balancing problem in which the loads are treated as real numbers; therefore, loads can be split to arbitrary. However, in a more realistic setting of the problem, which covers medium and large grain parallelism, the loads (processes, data, threads) are not infinitely divisible and, as a consequence, they are treated as natural numbers. There are two categories of load-balancing algorithms that consider discrete load model. On the one hand, the load-balancing algorithms that were originally designed under the discrete load model assumption, and, on the other hand, the discrete adaptation of the idealised load-balancing algorithms by performing rounding operations. Iterative load-balancing algorithms using discrete load model produce situations in which a global load balance cannot be guaranteed when the load-balancing process terminates. Furthermore, the convergence analysis of these iterative load-balancing algorithms using discrete load model has not been explored in the literature.

We raised the issue of the development of a realistic iterative load-balancing algorithm which was able to solve the balancing problems of existent discrete load-balancing algorithms in an asynchronous fashion. One important goal in this work was to derive a rigorous mathematical description of the proposed algorithm, which allows us to analyse its convergence, as well as other formal aspects of the algorithm such as its complexity and convergence rate. The proposed algorithm called DASUD (Diffusion Algorithm Searching Unbalanced Domains) is noteworthy for its ability to detect locally unbalanced situations that are not detected by other algorithms and for *always achieving the optimal local balance distribution*

Once DASUD was fully described, we were interested in comparing our proposal to the most relevant load-balancing strategies within the same family, in order to evaluate the goodness of DASUD. For that purpose, we raised the need to develop a simulation environment in which to be able to simulate the whole load-balancing process for different iterative load-balancing algorithms under the same conditions. Parameters such as topology, load distributions, system size and problem size should be easily variable in order to analyse the influence of each of them on the behaviour of the simulated load-balancing algorithms.

By simulation we have compared our algorithm with three well-known nearest-neighbours load-balancing algorithms within the literature attending to two quality measurements: stability and efficiency. Stability measures the ability of the algorithm to coerce any initial load distribution into a global stable state as close to even as possible. Efficiency measures the time delay for arriving at the globally stable state. From the results we are able to conclude that DASUD exhibits the best trade-off between the degree of balance achieved and the time incurred to reach it.

This work is organised as follows:

- The first chapter gives an overview of the dynamic load-balancing problem in parallel computing where the key issues that must be considered in this problem are described. Following this, a load-balancing algorithm taxonomy that illustrates how load distribution can be carried out is developed, and a simple description of the most relevant dynamic load-balancing strategies is included.

- The second chapter is focused on the nearest-neighbour load-balancing methods. Since this kind of load-balancing algorithms works in an iterative way, two important issues such as the algorithm convergence and its termination detection are discussed. Three relevant load-balancing algorithms from this category are described and analysed in detail.

- In chapter three the proposed dynamic load-balancing algorithm DASUD is described and analysed. DASUD's complexity is provided, as well as its convergence proof and upper bounds for the final balance degree and the number of balance iterations needed to achieve it.

- In chapter four the proposed load-balancing algorithm is compared by simulation to three of the most relevant load-balancing algorithms within the nearest-neighbour category that were described in chapter 2. The simulation framework has been designed including different interconnection networks as well as a wide range of system sizes. Moreover, the load distribution patterns simulated vary from situations which exhibit lightly unbalanced degree to highly unbalanced situations. The comparison has been carried out in terms of the unbalance degree reached by the algorithms and the time needed to achieve this final state.

- In chapter five the scalability of the proposed strategy is analysed in order to show the capacity of reacting similarly under different problem and system sizes.

- A question that ocurred to us during the evaluation of DASUD was: *how would DASUD work if it was able to collect more load information than only that of its immediate neighbours*?. For that purpose, in chapter six, an extended system model is provided. The influence of enlarging the domain into the time incurred in transferring messages beyond one link, and the extra computational cost incurred by the extended version of DASUD (d_s-DASUD) has been also considered to evaluate which enlargement provides the best trade-off between balance improvement and the load-balancing time spent.

- Chapter seven summarises the main conclusions derived from this thesis, outlining, in addition, current and future work in this field.

- Finally, the complete bibliography is provided, and complementary tables and figures are included in four appendixes.

Chapter 1

The dynamic load-balancing problem

Abstract

This chapter gives an overview of the dynamic load-balancing problem in parallel computing where, firstly, the key issues that must be considered in this problem are described. Following this, a load-balancing algorithm taxonomy that illustrates how load distribution can be carried out is presented, as well as a simple description of the most relevant dynamic load-balancing strategies. Finally, environments and existing tools for supporting load-balancing are described.

1.1 Introduction

When a parallel application is divided into a fixed number of processes (tasks)* that are to be executed in parallel, each processor performs a certain amount of work. However, it may be that some processors will complete their tasks before others and become idle because the work is unevenly divided, or some processors operate faster than others, or both situations. Ideally, all the processors should be operating continuously on tasks that would lead to the minimum execution time. Achieving this goal by spreading the tasks evenly across the processors is called load-balancing. Load-balancing can be attempted statically before the execution of any process, or dynamically during the execution of the processes. Static load-balancing is usually referred to as the mapping problem or scheduling problem. Dynamic load-balancing techniques assume little or no compile-time knowledge of the runtime parameters of the problem, such as task execution times or communication delays. These techniques are particularly useful in efficiently resolving applications that have unpredictable computational requirements or irregular communication patterns. Adaptive calculations, circuit simulations and VLSI design, N-body problems, parallel discrete event simulation, and data mining are just a few of those applications.

Dynamic load-balancing (DLB) is based on the redistribution of load among the processors during execution time, so that each processor would have the same or nearly the same amount of work to do. This redistribution is performed by transferring load units (*data*, *threads*, *processes*) from the heavily loaded processors to the lightly loaded processors with the aim of obtaining the highest possible execution speed. DLB and load sharing are used as interchangeable terms in the literature. While DLB views redistribution as the assigning of the processes among the processors, load sharing defines redistribution as the sharing of the system's processing power among the processes. The results of applying an ideal DLB algorithm to a 3x3 torus is shown in figure 1.1. The numbers inside the circles denote

* In this context, both terms (process and tasks) are used indistinctly.

the load value of each processor. Initially, at time (t_0) the load is unevenly distributed among the processors. The load becomes the same in all processors after executing an ideal DLB strategy (time t_f).

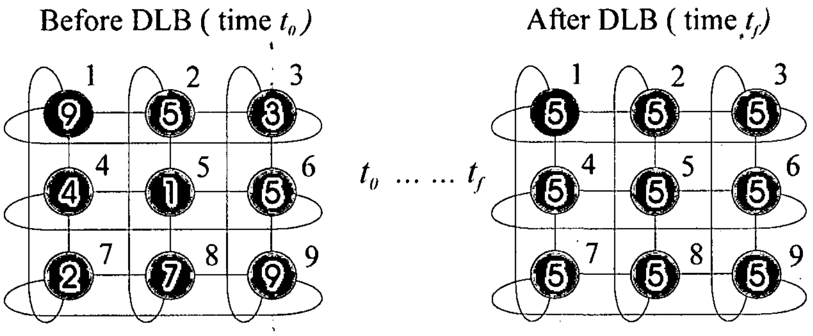


Figure 1.1. Dynamic load-balancing process.

Every DLB strategy has to resolve the issues of *when* to invoke a balancing operation, *who* makes load-balancing decision according to what information and *how* to manage load migrations between processors. There has been much research on DLB strategies for distributed computing systems. However, on parallel computing systems, the DLB problem takes on different characteristics. First, parallel computers typically use a regular point-to-point interconnection network, instead of random network configuration. Second, the load imbalance in a distributed computer is due primarily to external task arrivals, whereas the load imbalance in a parallel computer is due to the uneven and unpredictable nature of tasks.

The advantage of dynamic load-balancing over static load-balancing is that the system need not be aware of the run-time behaviour of the applications before execution. Nevertheless, the major disadvantage of DLB schemes is the run-time overhead due to the load information transfer among processors, the execution of the load-balancing strategy, and the communication delays due to load relocation itself.

1.2 Key issues in the Load-Balancing process

The design of a dynamic load-balancing algorithm requires resolving issues such as: who specifies the amount of load information made available to the decision-maker; who determines the condition under which a unit of load should be transferred; who identifies the destination processor of load to transfer; and how to manage load migrations between processors, amongst other issues. Combining different answers to the above questions results in a large area of possible designs of load-balancing algorithms with widely varying characteristics. On the one hand, there are decisions which must be taken at processor level, and others that require a greater or lesser degree of co-ordination between different processors, so the latter become system-level decisions.

In order to be systematic in the description of all necessary decisions related to the load balancing process, we distinguish two different design points of view: the *processor* level point of view and the *system* level point of view (see figure 1.2). We refer to *processor level* when the load-balancing operations respond to decisions taken by a processor. Otherwise, we talk about *system level* when the decisions affect a group of processors. In the following sections, we outline the description of each one of these levels.

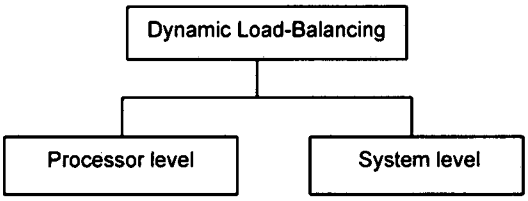


Figure 1.2. Load-balancing: two design points of view

1.3 Processor level.

A processor which intervenes in the load-balancing process will execute computational operations both for applications tasks as well as for load-balancing operations. This section describes the load-balancing operations carried out at processor level and their design alternatives. In order to perform the load-balancing operations, a processor must allocate three functional blocks to effectively implement the load-balancing process: the *Load Manager (LM)*, the *Load-Balancing Algorithm (LBA)* and the *Migration Manager (MM)*, as shown in figure 1.3. The *Load Manager block* is the one related to all load-keeping issues. The *Load-Balancing Algorithm block* is related to the concrete specification of the load-balancing strategy. Finally, a *Migration Manager block* is needed in order to actually perform the load movements.

The processors, which incorporate each one of these three blocks, will be referred to as *running processors*.

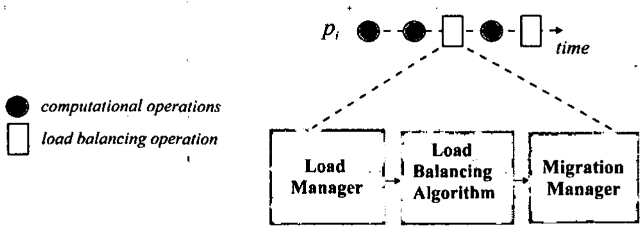


Figure 1.3 Functional blocks that integrate the load-balancing process within a processor.

The next sections will discuss the implementation issues for each one of these blocks, as well as their co-operation.

1.3.1 Load Manager block

One of the most important issues in the load-balancing process is to quantify the amount of load (*data*, *threads* or *processes*) of a given processor (*load index*). It is impossible to quantify exactly the execution time of the resident processes of a processor. Therefore, some measurable parameters should be used to determine the load index in a system such as the process sizes, the number of ready processes, the amount of data to be processed and so on. However, previous studies have shown that simple definitions such as the number of ready processes are particularly effective in quantifying the load index of a processor [Kun91].

The Load Manager block has the responsibility of updating the load information of the running processor, as well as gathering load information from a set of processors of the system (underlying domain). The time at which the load index of each processor is to be updated is known as *the load evaluation instant*. A non-negative variable (integer or real number), taking on a zero value if the processor is idle and taking on increasing positive values as the load increases, will be measured at this time according to the load unit definition [Fer87]. There must be a trade-off between the load gathering frequency and the ageing of the load information kept by the LM block, in order to avoid the use of obsolete values by the Load Balancing Algorithm block. This trade-off is captured in the following three load collection rules:

- On-demand: Processors collect the load information from each other whenever a load-balancing operation is about to begin or be initiated [Sta84][Zna91].

- Periodical: Processors periodically report their load information to others, regardless of whether the information is useful to others or not [Yua90][Kal88].

- On-state-change: Processors disseminate their load information whenever their state changes by a certain degree [Xu95] [Sal90].

The on-demand load-gathering method minimises the number of communication messages, but postpones the collection of system-wide load information until the time when a load-balancing operation is to be initiated. Its main disadvantage is that it results in an extra delay for load-balancing operations. Typically, this category includes bidding methods, where underloaded processors ask for load information from other processors to choose the best partner in performing load-balancing [Sta84][Zna91]. Conversely, the periodic method allows processors in need of a balancing operation to initiate the operation based on maintained load information without delay. The problem with the periodical scheme is how to set the interval for information gathering. A short interval would incur heavy communication overheads, while a long interval would sacrifice the accuracy of the load information used in the load-balancing algorithm. A protocol to exchange load information periodically called LIEP (Load Information Exchange Protocol) was presented in [Yua90]. In that work processors were arranged into a logical hypercube with dimension *d* (the topology diameter). During each period of load information exchange, a processor invoked *d* rounds of load exchanges in such a way that a processor exchanged its load value with the directly connected processor in the inspected dimension. A way to optimise the load collection process is reported in [Kal88] where the methodology proposed consists of periodically piggy-backing the load information with regular messages. The on-state-changing rule is a compromise of the on-demand and periodic schemes. In [Sal90] an on-state-changing method is reported in order to include the advantages of both approaches. In this case, a processor sends a status message to its neighbours only if its own load has changed by a certain value and an updated interval has elapsed since the last update. This reduces unnecessary frequent updates.

Nevertheless, how the LM block proceeds to collect and hold load information is not relevant to the Load Balancing Algorithm block. The information required by this block is limited to a set of non-negative numbers that represent the load index of each one of the processors belonging to the underlying domain. These values will be used subsequently to evaluate whether it is necessary to perform load movements or not and how these movements must be performed.

1.3.2 Load Balancing Algorithm block

The Load Balancing Algorithm block uses the load information provided by the previous LM block in deciding if it is necessary or not to balance the load, source and destination processors of load movements, as well as the amount of load to be transferred. An LBA algorithm, can be divided into two phases: *Load Balancing Activation* and *Work Transfer Calculation.*

Load Balancing Activation

This phase uses the load information kept by the LM block to determine the presence of a load imbalance. The criterion used to evaluate whether a processor is balanced or not is known as *trigger condition* and is normally associated to a threshold value that can be defined as:

- Fixed threshold: one or several fixed values are used as criteria to determine whether a processor is an overloaded processor or not [Zho88] [Mun95].
- Adaptive threshold: the threshold values are evaluated during the execution of the load-balancing algorithm and their values are usually state dependent [Wil93][Xu93][Cor99c].

Usually, the election of fixed thresholds as trigger condition produces simple strategies where each processor compares its load index with a fixed threshold to determine whether a processor has load excess (overloaded) or not (underloaded). In applications where the total load is expected to remain fairly constant, the load

balancing would be undertaken only in those cases where the load index of some processor falls outside specified upper and lower thresholds [Mun95].
Another method that has been suggested for situations in which the total load is changing is to balance load if the difference between a processor's load and the local load average (i.e. the average load of a processor and its neighbors) exceeds some threshold [Cor99c][Wil93]. Another similar approach consists of setting the threshold value using the global load average instead of the local load average to determine the trigger condition at each processor [Xu93].

All running processors in the system will evaluate the trigger condition at the start of executing the load-balancing algorithm. However, not all the running processors will overcome their trigger condition. The processors whose trigger condition evaluation does not fail will be called *active processors*. We refer to sender-initiated (SI) approaches when active processors are the ones with load excess and we refer to received-initiated (RI) schemes, [Eag85][Wil93], when the underloaded processors will become the active processors by requesting load from their overloaded counterpart.

Work Transfer Calculation

This phase is concerned with devising an appropriate transfer strategy to correct the imbalance previously detected and measured. After determining that load-balancing is required, *source* and *destination* processor pairs are determined, as well as *how much work* should be transferred from one processor to another. The function used to determine the destination of load can be implemented using one of the following choices:

- Randomly: no information about the domain state of the underlying processor is needed and destination processors are chosen in a random fashion [Zho88].
- Fixed: decisions produced by the active processors are not state dependent. The quantity of load to be transferred from one processor to another is set *a priori* as a fixed value. [Xu97][Cyb89].
- Evaluated: the amount of load to be moved between processors is evaluated at run time following some predetermined function [Wil93][Cor99c]

1.3.3 Migration Manager block

Finally, the last block is the Migration Manager (MM) block. This block receives as input the information generated by the Load Balancing Algorithm block, i.e., the destination processors and the amount of load that should be transferred to them. This block can be divided into two phases: *load unit selection* and *load transfer,* in order to differentiate between the way of choosing the individual load units to be transferred, and the physical transfer of those elements. Both phases are described below.

Load unit selection

Source processors *select* the most suitable *load units* (process, threads, data,...) which properly match with the load value to be moved. The quality of load-unit selection directly affects the ultimate quality of load-balancing. Sometimes, it may prove to be impossible to choose a group of load units whose associated load index corresponds exactly to the value that needs to be moved. The problem of selecting which load units to move is weakly NP-complete, since it is simply the subset sum problem. Fortunately, approximation algorithms exist which allow the subset sum problem to be solved to a specified non-zero accuracy in polynomial time [Pap94]. Before considering such an algorithm, it is important to note that other concerns may constrain load transfer options. In particular, we would like to avoid costly transfers of either large numbers of processes or large quantities of data unless absolutely necessary. We would also like to guide load selection to preserve, as best as possible, existing communication locality in the application. In general, we would like to associate a cost with the transfer of a given set of load units and then find the lowest cost set for a particular desired transfer.

Load transfer

This module should provide the appropriate mechanisms to correctly migrate several selected load units (which can be either processes, data or threads) to any destination processor. Data migration load-balancing systems support dynamic balancing through transparent data redistribution. Data migration mechanisms usually exhibit the lowest complexity amongst the three mechanisms' as they only have to move data-systems based on thread migration support dynamic load balancing through thread redistribution in multithreading environments. In such systems, a user application consists of a number of processes assigned to different processors and each process encapsulates a certain number of threads that can be created/destroyed dynamically. Transparent migration of threads implies the movement of the data and the computation state of a particular thread for one process located in a processor to another process located in a different processor. Process migration load-balancing systems support dynamic load balancing through transparent process redistribution in parallel and/or distributed computing environments. As in thread migration load-balancing systems, process migration implies the movement of the data and the computation state. However, process migration mechanisms exhibit the highest complexity as they must be aware of a huge amount of information. In the case of a process, the computation state is considerably more complex compared to the thread case and, moreover, application binaries must also be moved. In section 1.8, a more detailed description of the migration mechanisms provided by some load-balancing software packages is reported.

After having described the behaviour of each one of the blocks corresponding to one load-balancing operation, it is important to indicate that this decomposition in the load-balancing process in different modules allows us to experience in a plug-and-play fashion with different implementations at each one of the above blocks, allowing the space of techniques to be more fully and readily explored. It is also possible to customise a load-balancing algorithm for a particular application by replacing some general methods with those specifically designed for a certain class of computations.

1.4 System level

This level analyses which processors are involved in the load-balancing process and how their co-operation is carried out. Hence, the first decision that must be considered is the election of the set of running processors that participates in the load-balancing process. Depending on the number of processors belonging to this set we can distinguish between: *centralised*, *totally distributed* and *semi-distributed* approaches. In totally distributed and semi-distributed schemes the load-balancing goal is obtained because load-balancing operations are concurrently executed in more than one processor as time goes by. In particular, when the load-balancing blocks are executed simultaneously in all running processors of the system, we are considering a *synchronous* implementation of the load-balancing process. Otherwise, the system works in an *asynchronous* way. The influence of each one of above characteristics in the load-balancing process will be discussed in next subsections.

1.4.1 Centralised

Centralised load balancing strategies are characterised by the use of a dedicated processor for maintaining a global view of the system state and decision making. This processor is called central scheduler (or *central job dispatcher*). A central strategy can improve resource utilisation by having all the information of processors and it can achieve optimal performance by using sophisticated algorithms. It can also impose less overhead on the communication network by avoiding transfers of duplicate or inaccurate host state information. Global scheduling can also avoid task thrashing caused by contradictory load balancing decisions. However, centralised models have low reliability. If the central processor fails, the operation of the whole system can be corrupted. In addition, in large systems with high load fluctuation, the messages with load information can overload the interconnection structure around the central processor.

1.4.2 Totally distributed

An alternative to centralised approaches is a *distributed* scheme, in which the load-balancing decisions are carried out by all the processors of the system. Load information exchanges are restricted to a local sphere of processors and load-balancing operations are also performed within this sphere or domain. Depending on the existing relationship between different domains, we can distinguish between *overlapped* domains or *non-overlapped* domains. In figure 1.4, the processors in red are chosen as the running processor. In this example, we consider the domain of a given processor as the processors directly connected to it. Therefore, with the blue and yellow colours we indicate the domains of each one of the running processor. In figure 1.4.a we can observe that there are some common processors between the blue and the yellow domains. Hence, we refer to them as overlapped domains. Otherwise, we refer to non-overlapped domains (figure 1.4.b).

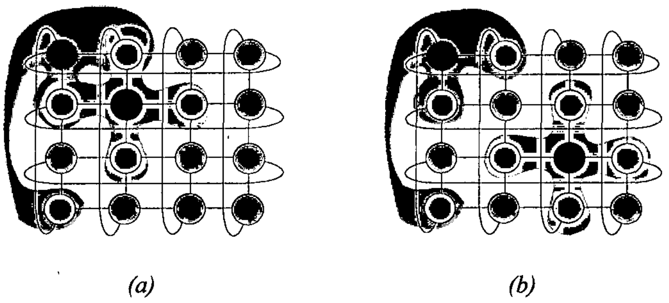


Figure 1.4. Overlapped domains (a) and non-overlapped domains (b).

When the domain includes a given processor and its immediate neighbours we refer to it as a *nearest–neighbour* approach. Nearest-neighbour load-balancing methods operate on the principle of reducing the load imbalance between each processor and its immediate neighbours with the aim of diffusing load through the system converging toward a system-wide balance. Otherwise, load-balancing strategies are categorised as *non-nearest-neighbour* approaches. Non-nearest-neighbour load- balancing alternatives work in a decentralised form by using local information, which is not restricted to immediate neighbours. Under this assumption the scope of the domain is extended to a large *radius* that may also include the neighbours' neighbours and so on.

Totally distributed approaches, in using local information, do not make such effective balance decisions as Centralised approaches, but, in contrast, they incur smaller synchronisation overheads.

1.4.3 Partially distributed.

For large systems (more than 100 processors), neither centralised nor distributed strategies proved to be appropriate. Although centralised strategies have the potential of yielding optimal performance, they also have disadvantages that make them suitable only for small or moderate systems [Bau88]. On the other hand, distributed strategies have good scalability, but for large systems it is difficult to achieve a global optimum because the processors have a limited view of the global system state. Partially distributed strategies (also called semi-distributed) were proposed as a trade-off between centralised and fully distributed mechanisms. The main idea is to divide the system into regions and thus split the load-balancing problem into subtasks. These strategies can be viewed at two levels: (i) load-balancing within a region and (ii) load-balancing among all the regions.

Different solutions can be devised for each level of the strategy. Each region is usually managed by a single master-processor using a centralised strategy and, at the level of the region, master-processors may (or may not) exchange aggregated information about their corresponding regions.

1.4.4 Synchronous versus asynchronous

Taking into account the instant at which load-balancing operations are invoked, both totally and partially distributed strategies can be further subdivided into synchronous and asynchronous strategies. From this point of view, we talk about *synchronous* algorithms when all processors involved in load-balancing (running processors) carry out balancing operations at the same instant of time so that each processor cannot proceed with normal computation until the load migrations demanded by the current operations have been completed. Otherwise, if each running processor performs load-balancing operations regardless of what the other processors do, we refer to *asynchronous* approaches. Figure 1.5 shows these behaviours for a four processor system. Notice that the distinction between synchronous and asynchronous does not apply for centralised schemes due to the existence of only one running processor in the entire system.

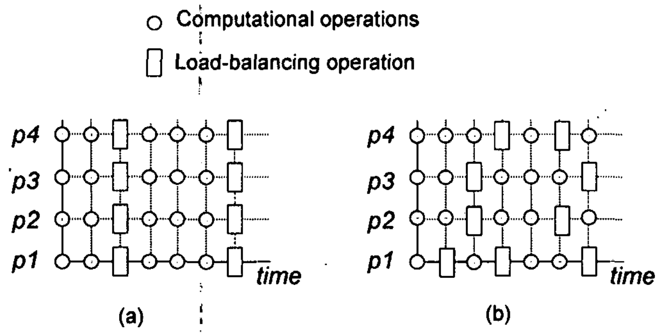


Figure 1.5. Synchronous (a) and asynchronous (b) load-balancing operations.

1.5 Load-balancing algorithm taxonomy

Most of the load-balancing strategies proposed in the literature are focused basically on the development of approaches for solving the Load Balancing Algorithm block mentioned in section 1.3.2. In terms of the algorithmic method used by these strategies we can derive the taxonomy shown in figure 1.6. The main criteria in classifying these algorithms concerns the way in which load distribution is carried out.

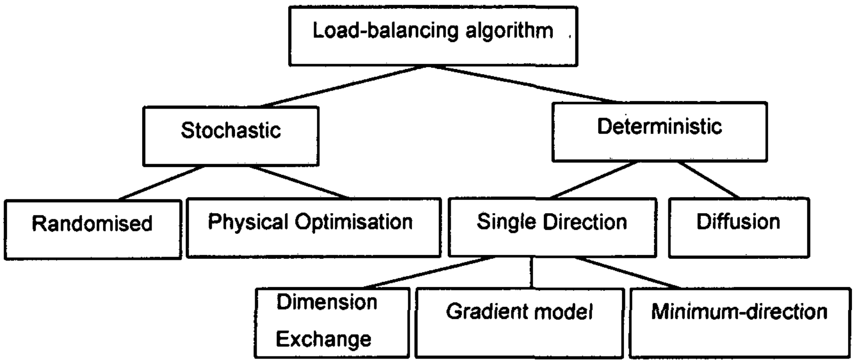


Figure 1.6. Load-Balancing taxonomy in terms of algorithmic aspects from the processor level point of view.

In **Stochastic** methods, the load is redistributed in some randomised fashion, subject to the objective of load balancing. Stochastic load balancing methods attempt to drive the system into an equilibrium state with high probability. Two different approaches can be found: *Randomised* allocation and *Physical optimisation.*

Randomised allocation methods are very simple methods that do not use information about potential destination processors. A neighbour processor is selected at random and the process is transferred to that processor. No exchange of state information among the processors is required in deciding where to transfer a load unit. On the other hand, Stochastic algorithms, where physical optimisation is applied, are based on analogies with physical systems. They map the load-balancing problem onto some physical systems, and then solve the problem using simulation or techniques from theoretical or experimental physics. Physical optimisation algorithms offer a slightly more variety in the control of the randomness in the redistribution of load units. This control mechanism makes the process of load balancing less susceptible to being trapped in local optima and therefore these stochastic algorithms are superior to other randomised approaches which could produce locally but not globally optimal results.

Deterministic methods proceed according to certain predefined strategies. These solutions are usually performed in an iterative form when the execution of the load-balancing algorithm is repeated more than once in a given processor, before restarting the execution of the user application [Xu94]. Deterministic methods can be classified into two categories according to the load distribution within the domain: *Diffusion* and *Single-direction*.

Firstly, in diffusion methods the load excess of an overloaded processor is simultaneously distributed amongst all processors of the underlying domain, following an iteration of the load-balancing algorithm. In contrast, in single-direction methods only one processor of the underlying domain can be chosen as destination processor after .executing one iteration of the load-balancing algorithm. Single Direction methods are further classified according to how the destination processor is selected. When the direction of the closer lightly loaded processor is used as a selection criterion we refer to *Gradient Model*, and when the chosen processor is the least loaded processor of the underlying domain we talk about *Minimum-Direction* schemes. Techniques where all processors are considered one by one at each load-balancing iteration are called *Dimension Exchange* strategies.

We will now describe some of the most relevant strategies that appear in the literature.

1.6 Dynamic load-balancing strategies

Following the taxonomy described in the previous paragraph, and bearing in mind the design characteristics outlined in section 1.4, we have constructed table 1.1, which draws together all published strategies, as far as we are aware. Particularly, in the case of processor level, the algorithmic aspects seen in section 1.5 are used. At each box of the table, the mnemonic for the strategy and its reference are given. Strategies indicated with a continuos line are not feasible or have not been proposed as far as the author knows. We now describe some of the strategies indicated in the table, starting with those classified in the randomised category.

<table>
<tr><th colspan="4" rowspan="3">Dynamic Load-Balancing</th><th colspan="4">System Level View</th></tr>
<tr><th rowspan="2">Centralised</th><th colspan="2">Totally Distributed</th><th rowspan="2">Semi-Distributed</th></tr>
<tr><th>Non-Nearest Neighbours</th><th>Nearest-Neighbours</th></tr>
<tr><td rowspan="6">Algorithmic Aspects from Processor Level View</td><td rowspan="2">Stochastic</td><td colspan="2">Randomised</td><td>—</td><td>Reservation [Eag85]
RANDOM[Zho88]
THRDL[Zho88]
LOWEST[Zho88]
MYPE[Yua90]</td><td>—</td><td>—</td></tr>
<tr><td colspan="2">Physical Optimisation</td><td>GTCA [Bau95]
CBLB[Bau95]</td><td>—</td><td>S.-A.[Fox89]</td><td>CSAM[Cha95]
MFAM [Cha95]</td></tr>
<tr><td rowspan="4">Deterministic</td><td colspan="2">Diffusion</td><td>Multi-level-Diffusion [Hor93]</td><td>NA [Sal90]
DDE [Wu96]
AN-n [Cor99c]</td><td>Diffusion [Cyb89][Boi90]
SID,RID [Wil93]
ATD [Wat98]
AN [Cor99c]
[Son94][Die99] [Hu99]</td><td>Membership-exc. [Eva94]
Joint-membership [Eva94]</td></tr>
<tr><td rowspan="3">Single-direction</td><td>Dimension Exchange</td><td>—</td><td>EDN [Cor99c]</td><td>DE [Cyb89]
GDE [Xu97]
DN [Cor99c]
Graph-Colouring[Hos90]</td><td>—</td></tr>
<tr><td>Gradient Model</td><td>—</td><td>—</td><td>GM [Lin87]
B[Bar90]
X-GM [Lul91]
EG [Mun95]</td><td>—</td></tr>
<tr><td>Minimum-direction</td><td>Central [Zho88]
LBC[Lin92]</td><td>GLRM[Xu93]
GMLM [Xu93]</td><td>CWN [Kal88]
ACWN [Shu89]
LLRM[Xu93]
LMLM [Xu93]</td><td>Sphere-like [Ahm91]
Hierch. Sched. [Dan97]</td></tr>
</table>

Table 1.1 Load-Balancing techniques classified with respect to system level view and processor level view.

1.6.1 Randomised

As we have seen, in randomised load-balancing algorithms, the destination processors for load transfer are chosen in a random fashion. Therefore, these kinds of algorithms use less system state information than deterministic algorithms. These algorithms use only local load information to make movement decisions. In such cases a threshold value is preset as a criterion in determining whether a processor must send out part of its load or not. Several randomised algorithms based on a threshold value (*T_l*) as a trigger condition (RANDOM, THRHLD,LOWEST) are reported in [Zho88]. In the RANDOM algorithm when a processor detects that its local load is bigger that T_l, a processor is randomly selected as a destination of load movements. Since all processors are able to make load movement decisions, this algorithm is classified as a totally distributed and non-nearest-neighbours approach. THRHLD and LOWEST algorithms are similar to the RANDOM algorithm in the sense that they also select the destination processor in a random way. However, a number of randomly selected processors, up to a limit L_p, are inspected instead of selecting only one candidate. In the THRHLD algorithm, extra load is transferred to the first processor whose load is below the threshold value. In contrast, in the LOWEST algorithm a fixed number of processors (L_p) are polled and the most lightly loaded processor is selected as destination processor. A similar scheme is used in the MYPE algorithm [Yua90]. Instead of using only one threshold value, the MYPE algorithm uses two threshold values to determine the state of the underlying processor, N_u and N_l. A processor is overloaded when its load is higher than N_u. Underloaded processors are the ones whose load is lower than N_l. Otherwise, they refer to neuter processors. An overloaded processor randomly selects a number of processors (up to the a preset limit) whose load indexes are lower than N_l, as potential receivers. Then a polling scheme is used to determine the final destination of the load. The load excess will be sent to the first processor whose current load is lower than N_l.

1.6.2 Physical Optimisation

The most common physical optimization algorithm for the load-balancing problem is simulated annealing. Simulated annealing is a general and powerful technique for combinatorial optimization problems borrowed from crystal annealing, in statistical physics. Since simulated annealing is very expensive and one of the requirements for dynamic load balancing is yielding the result in limited time, two hybrid methods, combining statistical and deterministic approaches, are proposed in [Cha95]: the Clustering Simulated Annealing Model (CSAM) and the Mean Field Annealing Model (MFAM). They were proposed to allocate or reallocate tasks at run time, so that every processor in the system had a nearly equal execution load and load interprocessor communication was minimised. In these methods, the load balancing was activated on a specific processor called the local balancer. The local balancer repeatedly activates the task allocation algorithm among a subset of processors. Each local balancer makes task allocation decisions for a group of four to nine processors. Groups are overlapped with each other, allowing tasks to be transferred through the whole system.

The CSAM combines a heuristic clustering algorithm (HCA) and the simulated annealing technique. The HCA generates clusters, where each cluster contains tasks which involve high intertask communication. Various task assignments (called system configurations) are generated from the HCA to provide clusters in various sizes that are suitable for the annealing process. During the annealing process, system configurations are updated by reassigning a cluster of tasks from one processor to another. The procedure of simulated annealing is used to either accept or reject a new configuration. The MFAM (Mean Field Annealing Model) was derived from modelling the distributed system as a large physical system in which load imbalance and communication costs causes the system to be in a state of non-equilibrium. Imbalance is balanced through a dynamic equation whose solution reduces the system imbalance. The dynamics of the MFAM are derived from Gibbs distribution. Initially all tasks have the same probability of being allocated in each processor. Several iterations of an annealing algorithm are carried out so that the system is brought to a situation where each process is assigned to only one processor. A major advantage of the MFAM is that computation of the annealing algorithm can be implemented in parallel. A similar load-balancing algorithm that uses simulated annealing technique is reported in [Fox89]

In addition to the simulating annealing technique, genetic algorithms constitute another optimisation method that has borrowed ideas from natural science and has also been adapted to dynamic load-balancing. Examples of genetic load-balancing algorithms can be found in [Bau95]. The first algorithm presented in the paper is Genetic Central Task Assigner (GCTA). It uses a genetic algorithm to perform entire load-balancing action. The second, Classifier-Based Load Balancer (CBLB), augments an existing load-balancing algorithm using a simple classifier system to tune the parameters of the algorithm.

1.6.3 Diffusion

One simple method for dynamic load-balancing is for each overloaded processor to transfer a portion of its load to its underloaded neighbours with the aim of achieving a local load balance. Such methods correspond closely to simple iterative methods for the solution of diffusion problems; indeed, the surplus load can be interpreted as diffusing through the processors towards a steady balanced state. Diffusion algorithms assume that a processor is able to send and receive messages to/from all its neighbours simultaneously.

Corradi *et alter* propose a more precise definition of diffusive load-balancing strategies in [Cor99c]. In particular, they define an LB strategy as diffusive when:

- It is based on replicated load-balancing operations, each with the same behaviour and capable of autonomous activity;
- The LB goal is locally pursued: the scope of the action for each running processor is bound to a local area of the system (domain). Each running processor tries to balance the load of its domain as if it were the whole system, based only on the load information in its domain; and
- Each running processor's domain overlaps with the domain controlled by at least one other running processor and the unification of these domains achieves full coverage of the whole system.

Cybenko describes in [Cyb89] a simple diffusion algorithm where a processor *i* compares its load with all its immediate neighbours in order to determine which neighbouring processors have a load value smaller than the underlying processor's load. Such processors will be considered underloaded neighbour processors. Once underloaded neighbours are determined, the underlying processor will evaluate the load difference between itself and each one of its neighbours. Then, a fixed portion of the corresponding load difference is sent to each one of the underloaded neighbours. This strategy, as well as other strategies from the literature based on this, [Ber89][Die99][Hu99] were originally conceived under the assumption that load can be divided into arbitrary fractions, i.e., the load was treated as a non-negative real quantity. However, to cover medium and large grain parallelisms which are more realistic and more common in practical parallel computing environments, we must treat the loads of the processors as non-negative integers, as was carried out in [Cor99c][Son94][Wat98][Wil93]. A relevant strategy in this area is the SID (*Sender Initiated Diffusion*) algorithm [Wil93]. In this algorithm, each processor *i* has a load value equal to w_i and it evaluates its local load average ($\overline{w}_i$) to be used as a trigger condition. If the load of processor *i* is bigger than the load average of its domain, then it is an overloaded processor. Otherwise, the processor was referred to as underloaded. An overloaded processor distributes its excess load among its underloaded neighbours. A neighbour processor *j* of the underlying processor *i*, is a neighbour with deficit if its load is smaller than the load average of the underlying domain ($\overline{w}_i > w_j$). Then, the surplus of a given processor *i* is distributed among its deficient neighbours in a proportional way. This strategy is classified as a sender-initiated scheme because the overloaded processors are the active processors. The same authors described a similar strategy called RID (*Receiver Initiated Diffusion*) which is based on the same idea as the SID algorithm, but using a receiver-initiated scheme to determine the processors which take load-movements decisions. An example of the behaviour of this algorithm is shown in figure 1.7. The number inside the nodes represents the load of the corresponding processor. Processor 6 has a load value equal to 40 ($w_6 = 40$) and the load average within its domain is 12 ($\overline{w}_6 = 12$). Therefore, the load excess of processor 6 is equal to 28 units of load. After applying the SID algorithm, processor 6 decides to move 1, 11, 5 and 7 load unit to processors 2, 5, 7 and 10 respectively. These load movements are denoted by the numbers behind the arrows.

$$w_6 = 40$$

$$\overline{w}_6 = 12$$

$$\overline{w}_6 - w_6 = 28$$

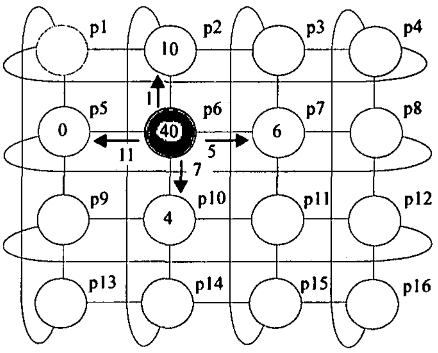


Figure 1.7 An example of the execution of one iteration of the SID strategy in processor 6.

The reader can find more examples of deterministic diffusion load-balancing strategies in [Cor99c][Eva94][Hor93][Sal90][Wat98] and [Wu96].

1.6.4 Dimension Exchange

This load-balancing method was initially studied for hypercube topologies where processor neighbours are inspected by following each dimension of the hypercube. Thus, this is the origin of the dimension exchange (DE) name. Originally, in DE methods, the processors of a *k*-dimensional hypercube pair up with their neighbours in each dimension and exchange half the difference in their respective loads [Cyb89]. The load value of the underlying processor is updated at each neighbour inspection and the new value is considered for the next revision. Going through all the neighbours once consists of carrying out a "sweep" of the load-balancing algorithm. Such behaviour is shown in figure 1.8.

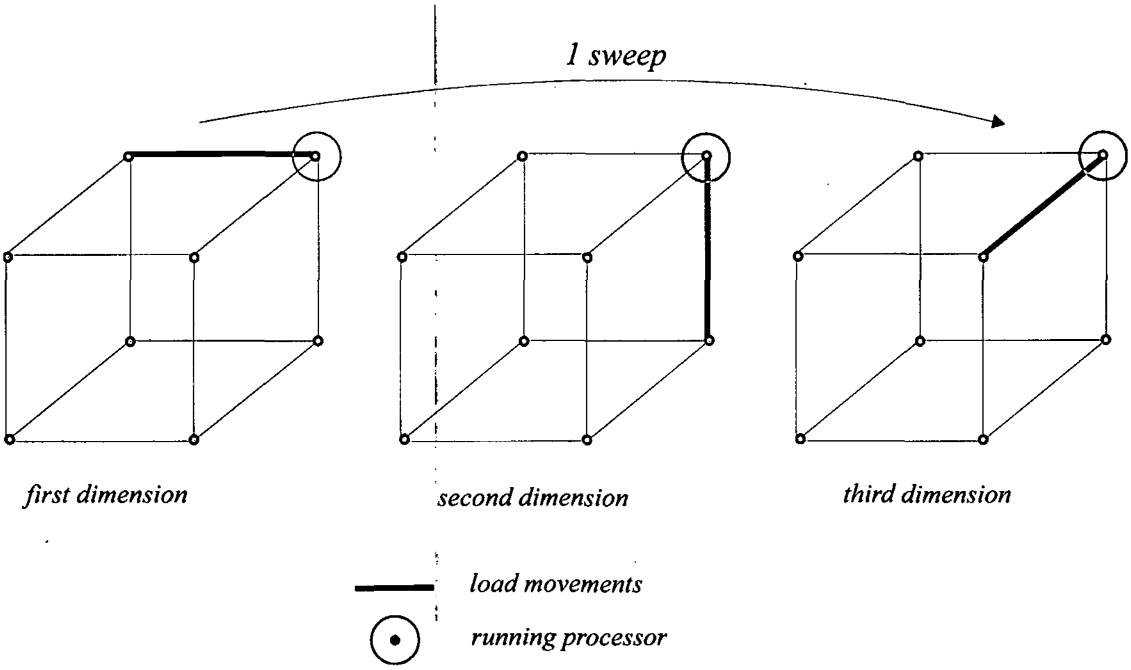


Figure 1.8 Load movements for DE methods in a 3-dimensional hypercube through three iterations of the load-balancing algorithm (1 sweep) in a running processor.

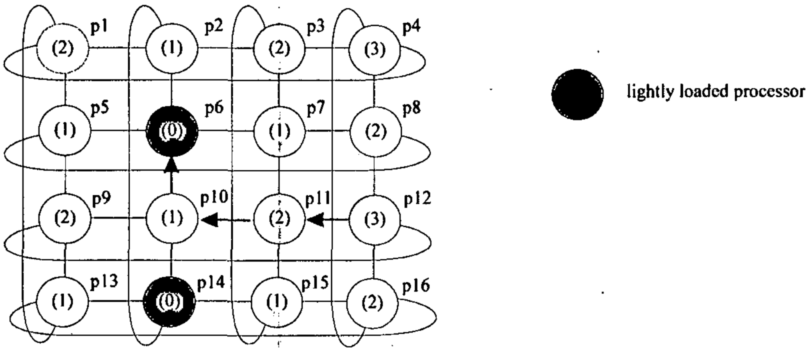
Xu *et alter* present in [Xu97] a generalisation of this technique for arbitrary topologies, which they call the GDE (Generalised Dimension Exchange) strategy. For arbitrary topologies the technique of edge colouring of undirected graphs (where each node of the graph identifies one processor of the system and the edges are the links) is used to determine the number of dimensions and the dimension associated at each link. The links between neighbouring processors are minimally coloured so that no processor has two links of the same colour [Hos90]. Subsequently, a "dimension" is then defined as being the collection of all edges of the same colour. At each iteration, one particular colour/dimension is considered, and only processors on edges with this colour execute the dimension exchange procedure. The portion of load exchanged is a fixed value and is called the exchange parameter. This process is repeated until a balanced state is reached. The DE algorithm uses the same value of the exchange parameter for all topologies, while the GDE algorithm uses different values depending on the underlying topology.

The DN (Direct Neighbour) algorithm is a strategy based on the dimension exchange philosophy, which uses a discrete load model [Cor99c]. This strategy allows load exchange between two processors only directly connected by a physical link. A balancing action within a domain strives to balance the load of the two processors involved. In order to assure the convergence of this method, the running processors must synchronise amongst themselves in such a way that the running processors active in any given moment have non-overlapping domains. The same authors describe an extension to this algorithm, the EDN (Extended Direct Neighbour) algorithm, which works as a non-nearest neighbour strategy. This strategy allows a dynamic domain definition by moving load between direct neighbours, overcoming the neighbourhood limit through underloaded processors. Load reallocation stops when there are no more useful movements, i.e., a processor is reached whose load is minimal in its neighbourhood.

1.6.5 Gradient Model

With gradient-based methods, load is restricted to being transferred along the direction of the most lightly loaded processors. That is, an overloaded processor will send its excess load only to one neighbor processor at the end of one iteration of the load-balancing algorithm. Therefore, the main difference between the gradient model and the dimension exchanged scheme is that at each iteration the load information of the entire underlying domain is considered in deciding the destination processor, whilst in DE methods only one processor is considered at each iteration.

In the Gradient Model algorithm described in [Lin87] two-tiered load-balancing steps are employed. The first step is to let each individual processor determine its own loading condition: light, moderate or heavy. The second step consists of establishing a system-wide gradient surface to facilitate load migration. The *gradient surface* is represented by the aggregate of all *proximities*, where a proximity of a processor *i* is the minimum distance between the processor and a lightly loaded processor in the system. The gradient surface is approximated by a distributed measurement called the *pressure surface*, then the excessive load from heavily loaded processors is routed to the neighbour of the least pressure (proximity). The resulting effect is a form of relaxation where load migrating through the system is guided by the proximity gradient and gravitates towards underloaded processors. Figure 1.9 shows an example of a gradient surface in a 4x4 torus network where there are two lightly loaded processors (processors 2 and 10). The value between brackets (x) represents the pressure surface of each processor. Let us suppose that processor 12 is an overloaded processor (yellow colour). By following the proximities depicted in the figure, the load excess of processor 12 will be guided to be moved through the red links in order to achieve one of the two lightly loaded processors within the system (in this case processor 6).



1.9 The GM scheme on a 4x4 torus..

This basic gradient model has serious drawbacks. First, when a large portion of moderately loaded processors suddenly turns lightly loaded, the result is considerable commotion. Lüling *et alter* proposed an improved version of the GM algorithm to remedy this problem, the Extended Gradient Model (X-GM) [Lül91]. This method adds a suction surface which is based on the (estimated) proximities of non-heavily-loaded processors to heavily-loaded processors. This information would cause load migration from heavily-loaded processors to nearby local minima which may be moderately-loaded processors. Since the system load changes dynamically, the proximity information kept by a processor may be considerably out-of-date. And finally, if there are only a few lightly-loaded processors in the system, more than one overloaded processor may emit some load toward the same underloaded processor. This "overflow" effect has the potential to transform underloaded processors into overloaded ones. The authors of [Mun95] propose another extension to the GM scheme, **EG** (Extended Gradient) mechanism, to overcome the problems mentioned. The EG mechanism is a two-phase strategy, where an overloaded processor confirms that a processor is still underloaded before transferring load to it, and then the underloaded processor is reserved in transferring the load.

1.6.6 Minimum-direction

The minimum-direction scheme is an alternative to dimension exchange methods and gradient model within the single-direction category of deterministic load-balancing algorithms. In the strategies based on this scheme, the running processor chooses the least loaded processor within its domain as the only destination of a load movement after executing the load-balancing algorithm once. Notice that, depending on the scope of the domain, the least loaded processor within the underlying domain may coincide with the least loaded processor in the whole system. Such a match is typically produced in centralised load-balancing systems where the running processors have access to the load of the entire system.

The LBC (Load-Balancing with a Central job dispatcher) strategy reported in [Lin92] makes load-balancing decisions based on global state information which is maintained by a central job dispatcher. Each processor sends a message to the central site whenever its state changes. Upon receiving a state-change message, the central dispatcher updates the load value kept in its memory accordingly. When a processor becomes underloaded, the state-change message is also used as a load request message. In response to this load request, the dispatcher consults the table where load values are kept, and the most loaded processor is chosen as load source.

Then this processor is notified to transfer some load to the requesting processor. The LBC strategy is a centralised algorithm because the central site guides all load movements. This strategy is also classified as a receiver-initiated method in the sense that the underloaded processors are the ones which start the load-balancing operations.

The CENTRAL algorithm described in [Zho88] is a centralised sender-initiated algorithm that works in a complementary form to the LBC strategy. When a processor detects that it is an overloaded processor, it notifies the *load information center* (LIC) about this fact by sending a message with its current load value. The LIC selects a processor with the lowest load value and informs the originating processor to send the extra load to the selected processor.

GLRM (Global Least Recently Migrated) and GMLM (Global Minimum Load Maintained) are two totally distributed non-nearest-neighbour strategies where the domain of each processor includes all processors in the system [Xu93]. Both GLRM and GMLM strategies use the global load average in the system as a threshold to determine whether a processor is overloaded or not. This threshold is computed at each processor using the load values received from the information collector (IC) processor. The IC processor has the responsibility of collecting the load of the entire system and broadcasting it to all processors. These actions will be performed on a time window basis. Once a processor is considered to be overloaded, a destination processor must be chosen. GLRM selects the destination processor by applying the last recently migrated discipline in a window time and the GMLM strategy determines the destination processor as the processor with minimum load value in the current time window. If the domain of each processor is restricted to the immediate neighbours, two nearest-neighbour strategies are easily derived from the two previous ones: LLRM (Local Least Recently Migrated) and LMLM (Local Minimum Load Maintained).

Another algorithm based on the minimum-direction scheme is the CWN (Contracting Within Neighbourhood) strategy [Kal88]. CWN is a totally distributed strategy where each processor only uses load information about its immediate neighbours. A processor would migrate its excess load to the neighbour with the least load. A processor that receives some load keeps it for execution if it is most lightly loaded when compared with all its neighbours; otherwise, it forwards the load to its least loaded neighbour. This scheme has two parameters: the *radius*, i.e. the maximum distance a load unit is allowed to travel, and the *horizon*, i.e. the minimum distance a load unit is required to travel. If we allow these parameters to be tuneable at run-time, the algorithms become ACWN (Adaptive Contracting Within a Neighbourhood) [Shu89].

In the semi-distributed strategy proposed by Ahmad [Ahm91], called Sphere-like, the system is divided into symmetric regions called 'spheres'. Considering the load-balancing method applied among these spheres, this strategy falls into the minimum-direction category. This strategy has a two-level load-balancing scheme. At the first level the load is balanced among different spheres using global system information. At the second level, load balancing is carried out within individual spheres. Each sphere has a processor that acts as a centralised controller for its own sphere. Since this strategy is primarily designed for massively parallel systems, it also addresses the problem of creating the spheres. State information, maintained by each centralised controller, is the accumulative load of its sphere. In addition, a linked list is maintained in non-decreasing order that sorts the processors of the sphere according to their loads. The scheduling algorithm first considers the load of the least loaded processor in local sphere and if it is less than or equal to chosen *threshold1*, the task is scheduled on that processor. Otherwise the scheduler checks the cumulative load of other spheres. If the load of the least loaded sphere is less than *threshold2*, the task is sent to that sphere where it is executed without further migration to any other sphere. If there is more than one such sphere, one is selected randomly. In the case that there is no such sphere, the task is scheduled in the least loaded processor of the local sphere. The parameters *threshold1* and *threshold2* are adjustable depending upon system load and network characteristics.

More dynamic load-balancing algorithms included within the minimum-direction category are reported in [Dan97].

1.7 Software facilities for supporting dynamic load-balancing

Software systems that support some kind of adaptive parallel application execution are basically classified into two main classes: *system-level* class and *user-level* class.

In the system-level class, load-balancing support is implemented at the operating system level [Sin97]. In contrast, the user-level class includes all the systems where the load-balancing support is not integrated into the operating system level. They are built on top of existing operating systems and communication environments. In that sense, load-balancing systems supported at the system level provide more transparency and less interference (migration can be carried out more efficiently, for instance) compared to load-balancing systems supported at the user level. However, they are not as portable as the user-level implementations. In the remainder of the section we focus mainly on the second class of systems. Readers interested in load-balancing systems based on system-level support could refer to the description of systems such as Sprite [Dou91], V System [The85] and Mach [Mil93].

Load-Balancing Systems (LBS) implemented at the user level can be further subdivided into data-based, thread-based or process-based systems, according to the load unit that is migrated, as was mentioned previously in sections 1.3.1 and 1.3.3. Therefore, we will refer to load unit migration as a general term that does not differentiate whether migration involves data, threads or processes.

Moreover, data-based and thread-based LBS are usually based on a distributed shared memory paradigm. As a consequence, some problems addressed in process-based LBS (process communication, for instance) do not always appear in data-based and thread-based LBS. We will focus below mainly on the design issues related to process-based LBS, although the reader should bear in mind that many issues are also applicable to the other two classes of LBS.

1.7.1 Architecture of Process-based LBS

Despite the particular characteristics of different LBS, a similar system architecture is shared by most of them. This architecture is based on a set of layers where upper layer components interact with lower layer components through library functions (see figure 1.10)

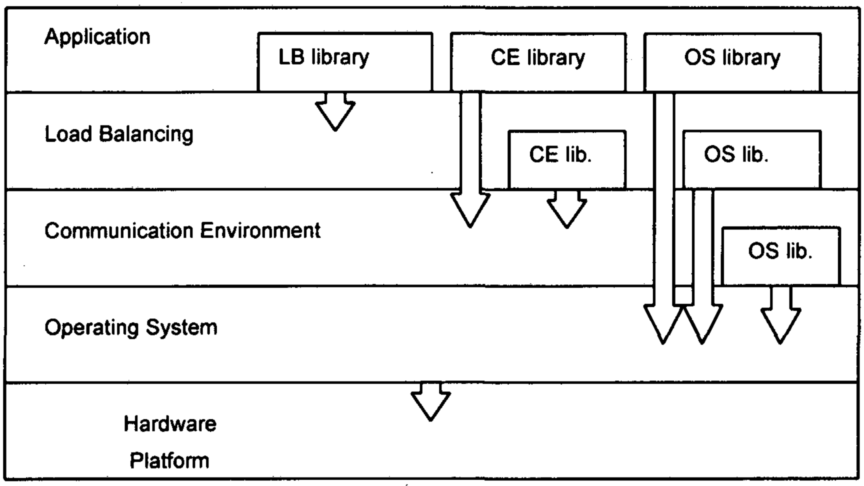


Figure 1.10 Layered structure of a Load Balancing System

A parallel application is made of a set of processes that execute in a distributed environment and co-operate/synchronise by means of message passing. For that purpose the Communication Environment (CE) offers a set of services that serve to communicate information between tasks in a transparent way, and conceal the particular OS network characteristics. Similarly, the Load Balancing (LB) layer will take advantage of the services offered by the Communication layer. PVM and MPI constitute common examples of such a communication layer that have been used in many existing LBS.

The LB layer is responsible for carrying out all the actions involved in process migration. In that sense, this layer includes all the mechanisms that implement the object migration. Moreover, this layer should also include the policies mentioned in section 1.6 that manage the resources and are responsible for maintaining load balancing. Interaction between the user application and the LB layer could be carried out by invoking certain functions of the LB layer directly. Alternatively, the programming language of the application may be augmented with new constructs and a code pre-processor will transform those constructs to LB functions. In both cases, the LB functions will be linked to the user application at a later stage. In contrast to LBS where the interaction between the application and the LB layer is accomplished through a linked library, there are LBS where such interaction is carried out by means of messages using the services of the Communication layer. In this case, the user application treats the LB system as another application task within the communication domain of the message-passing environment.

1.7.2 Design issues of a process migration mechanism

A major requirement for LBS is that the migration should not affect the correctness of the application. Execution of the application should proceed as if the migration had never taken place, the migration being "transparent". Such transparency can be ensured if the state of a process on the source processor is reconstructed on the target processor. The process migration mechanism can be roughly divided into four main stages: migration initiation, state capture, state transfer and process restart. Below, we present the most important issues that a process migration environment must address in practice.

Migration Initiation

This stage triggers the decision of starting the migration of a given process, i.e. it decides *when* migration occurs. Moreover it should indicate which process should migrate and where to. In principle, this information depends on the decisions adopted by the load balancing strategy running in the system. The scope of the migration event causes the migration mechanism to be synchronous or asynchronous. *Asynchronous migration* allows a process to migrate independently of what the other processes in the application are doing. *Synchronous migration* implies that all the processes are first executing and agree to enter into a migration phase where the selected processes will be finally relocated.

State Capture

This stage implies capturing the process' state in the source processor. In this context, the process' state includes: (i) the processor state (contents of the machine registers, program counter, program status word, etc.), (ii) the state held by the process itself (its text, static and dynamic data, and stack segments), (iii) the state held by the OS for the process (blocked and pending signals, open files, socket connections, page table entries, controlling terminals, process relationship information, etc.), and (iv) the OS state held by the process (file descriptors, process identifiers, host name and time).

The previous information, known to the process, is only valid in the context of the local execution environment (local operating system and host). Furthermore, a process has an state as viewed from the perspective of the communication layer. In this regard, a process' state includes its communication identifiers and the messages sent to/from that process.

Capturing the process state can be non-restricted or restricted.

- *Non-restricted capture* of the process' state means that the state can be saved at any moment. The LBS must block the process (for instance, using the Unix signal mechanism) and capture its state.
- *Restricted capture* means that the state will be saved only when the process executes a special code that has been inserted into the application. This implies that all points where a process may be suspended for migration must be known at the time of compilation. The special code usually consists of a call to an LB service that passes control to the LB layer. Then the LB layer decides whether the process should be suspended for migration or if the call is serviced and the control returned to the process. In the latter case, the service invocation also serves to capture and preserve the process state information.

The process state can be saved on disk, creating a checkpoint of the process. Readers interested in this area should refer to [Tan95] for a detailed description of the checkpointing mechanism applied to a single user process in the Condor system. The checkpoint mechanism has the advantage of being minimally obtrusive and providing fault-tolerance. However, it requires significant disk space consumption. Other migration mechanisms do not store the process state on disk. They create a skeleton process on the target processor to receive the migrating process, and then the process state is sent by the migrating process directly to the skeleton process.

State Transfer

For LBS implemented at user-level, the entire virtual address of a process is usually transferred at this stage [Cas95]. There are different mechanisms to transfer this information and they depend on the method that was used to capture the process state.

When the checkpoint of the process has been stored on disk (indirect checkpointing), the state transfer is carried out by accessing the checkpoint files from the target processor. The use of a certain global file system (NFS, for instance) is the simplest solution in this case. Otherwise, checkpoint files must be transferred from one host to another through the network.

When a skeleton process mechanism is used (direct chekpointing), this stage of the migration protocol implies that the skeleton process was successfully started at the target processor, (using the same executable file automatically "migrates" the text of the process). Then the process must detach from the local processor and its state, which was previously preserved (data, stack and processor context), must be transferred to the target processor through a socket [Cas95][Ove96].

Process Restart

The process restart implies that a new process is created in the target host and its data and stack information is assimilated according to the information obtained from the state of the process in the source host. The new process in the target host reads data and stack either from disk or a socket, depending on the mechanism used to capture the process state. Once the new process has assimilated all the information needed as its own process state, the process in the source host is removed.

Before the new process can re-participate as part of the application, it first has to re-enrol itself with the local server of the Communication layer. This implies that some actions are carried out to ensure the correct delivery of messages. It must be ensured that all processes send all their future messages destined for the migrated process to the new destination, and that no in-transit messages are dropped during migration. These actions must also solve problems related to the process identifiers within the communication environment and message sequencing. Different mechanisms have been proposed to ensure the correct delivery and sequencing of in-transit messages. They can be roughly classified into three main categories:

- *Message forwarding*: a shadow process can be created in the processor where the process was originally created. This shadow process will be responsible for forwarding all the messages directed to the process in its new location. When a message arrives at a processor and finds that the destination process is not there, the message is forwarded to the new location [Cas95]).
- *Message restriction*: this technique ensures that a process should not be communicating with another process at the moment of migration. That imposes the notion of critical sections where all interprocess communication is embedded in such sections. Migration can only take place outside a critical section [Ove96].
- *Message flushing*: in this technique, a protocol is used to ensure that all pending messages have been received. Therefore, the network will be drained when all the pending messages are received [Pru95].

Prior to restarting a migrated process, it must be connected again to the Communication Environment in order to establish a new communication identifier. The new identifier must be broadcast to all the hosts so that a mapping of old identifiers to new identifiers for each process is maintained in a local table. All future communications will go through this mapping table before they are passed into and out of the Communication Environment.

1.7.3 Limitations of process migration mechanisms

In practice, there are additional problems that must also be addressed in the implementation of the LBS. These problems are related to the management of file I/O (which includes application binaries, application data files and checkpoint files), *management of terminal I/O* and GUIs, *and management of cross-application* communication and inter-application communication.

Access to the same set of files can be carried out via a networked file system (NFS, for example). When there is no common file system, the remote access is accomplished by maintaining a "shadow" process on the machine where the task was initially running. The "shadow" process acts as an agent for file access by the migrated process. Similar solutions can be devised for accessing terminal I/O.

However, some limitations are imposed by existing LBS. For instance, processes which execute *fork ()* or *exec ()*, or which communicate with other processes via signals, sockets or pipes are not suitable for migration because existing LBS cannot save and restore sufficient information on the state of these processes. This limitation is reasonable according to the layering architecture of figure 1.10. User applications are restricted to using the facilities provided by the Communication layer or the LB layer to establishing communication between processes or to creating/destroying processes.

Additionally, process migration is normally restricted between machines with homogeneous architectures, i.e., with the same instruction sets and data formats. However, there are some systems that allow migration of sequential processes between heterogeneous machines. For instance, Tui [Smi97] is a migration system that is able to translate the memory image of a program (written in ANSI-C) between four common architectures (MC68000, SPARC, i486 and PowerPC). Another example is the Porch compiler [Str98] that enables machine-independent checkpoints by automatic generation of checkpointing and recovery code.

1.7.4 Examples of existing packages for supporting dynamic load balancing

In this subsection, we briefly review some of the most significant software packages that have been developed or are in an early stage of development in the framework of dynamic load balancing and load unit migration. These tools usually fall into two main classes [Bak96]:

- *Job Management Software*: these software tools are designed to manage application jobs submitted to parallel systems or workstation clusters. Most of them might be regarded as direct descendants from traditional batch and queuing systems [Bak96][Jon97]. Process-based LBS usually belong to this group.
- *Distributed Computing Environments*: these software tools are used as an application environment, similar in many ways to a distributed shared memory system. The application programmer is usually provided with a set of development libraries added to a standard language that allows the development of a distributed application to be run on the hardware platform (usually, a distributed cluster of workstations). The environment also contains a runtime system that extends or partially replaces the underlying operating system in order to provide support for load unit migration. Data-based and thread-based LBS mainly belong to this group of tools.
-

For each class of LBS, we briefly describe the main characteristics of one of the most relevant tools, which serves as a representative example of tools of that class. This description is completed with a list of references for other similar tools.

a) *Data-based LBS*. Most of the dynamic migration environments that distribute data are based on the SPMD model of computation, where the user program is replicated in several processors and each copy of the program, executing in parallel, performs its computations on a subset of the data. Dome [Ara96] is a computing environment that supports heterogeneous checkpointing through the use of C++ class abstractions. When an object of one class is instantiated, it is automatically partitioned and adapted within the distributed environment. Load-balancing is performed by remapping data based on the time taken by each process during the last computational phase. Due to the SPMD computational nature of the applications, the synchronisation between computational phases and load-balancing phases is straightforward. Other systems similar to Dome that also provide data migration are described in [Sil94] and [Bru99]. An architecture-independent package is presented in [Sil94], where the user is responsible for inserting calls to specify the data to be saved and perform the checkpoints. A framework implemented in the context of the Charm++ system [Kal96] is presented in [Bru99]. This framework automatically creates load balanced Charm++ applications by means of object migration. Load balancing decisions are guided by the information provided by the run-time system, which can measure the work incurred by particular objects and can also record object-to-object communication patterns. In this framework migration can only occur between method invocations, so that migration is limited to data members of the object.

b) *Thread-based LBS*. These systems are usually object-based systems that provide a programming environment that exports a thread-based object oriented programming model to the user. The objects share a single address space per application that is distributed across the nodes in the network, and the objects are free (under certain constraints) to migrate from one node to another. Arachne [Dim98] is a thread system that supports thread migration between heterogeneous platforms. It is based on C and C++ languages, which have been augmented in order to facilitate thread migration. Conventional C++ is generated by a pre-processor that inserts special code to enable the saving and subsequent restoration of a thread state. Migrating threads must be previously suspended, and suspension takes place when a thread invokes an Arachne primitive. Therefore, threads may be suspended (and potentially migrated) only at particular points that must be known at the time of compilation. The Arachne environment includes also a runtime system that manages the threads during program execution. Generating executables beforehand for each machine supports the heterogeneity of the environment. Other examples of object-based environments that support migrating threads are Ariadne [Mas96], Emerald [Ste95] and Ythreads [San94]. In contrast, UPVM [Kon97] is a process-based environment that provides thread migration for PVM programs written in single program multiple data (SPMD) style.

c) *Process-based LBS*. Condor/CARMI [Pru95] constitutes one of the most notable examples of process migration environments implemented at the user-level. It is based on Condor, a distributed batch processing system for Unix that was extended with additional services to support parallel PVM applications. Condor uses a checkpoint/roll-back mechanism to support migration of sequential processes. The Condor system takes a snapshot of the state of the programs it is running. This is done by taking a core dump of the process and merging it with the executable file. At migration time, the currently running process is immediately terminated and it is resumed on another host, based on the last checkpoint file. Condor was extended with CARMI (Condor Application Resource Management Interface) which provides an asynchronous Application Programming Interface (API) for PVM applications. CARMI provides services to allocate resources to an application and allows applications to make use of and manage those resources by creating processes to run there. CoCheck (Consistent Checkpointing) is the third component of the system. It is built on top of Condor and implements a network consistency protocol to ensure that the entire state of the PVM network is saved during a checkpoint and that communication can be resumed following a checkpoint. Process migration and checkpointing (with certain limitations in most cases) have also been developed or are under development in some research packages such as MIST [Cas95], DynamicPVM [Ove96], Pbeam [Pet98] and Hector [Rus99], and in some commercial packages that were also based on Condor such us [Cod97] and LoadLeveler [IBM93].

Chapter 2

Nearest-neighbour load-balancing methods

Abstract

As reported in the previous chapter, totally distributed dynamic load-balancing algorithms seem to be particularly adequate in the case of parallel systems. In this chapter, we focus on nearest-neighbour load-balancing algorithms where each processor only considers its immediate neighbour processors to perform load-balancing actions. First of all, we introduce a generic notation for nearest-neighbour load-balancing algorithms. Due to the iterative nature of most of the nearest-neighbour load-balancing methods, two important issues such as the algorithm convergence and its termination detection are subsequently discussed. Finally, three relevant LB algorithms from this category (SID, GDE and AN) are described and analysed in detail.

2.1 Introduction

As was commented in chapter 1, nearest-neighbour load-balancing algorithms have emerged as one of the most important techniques for parallel computers based on direct networks. This fact is corroborated by the fact that, amongst all categories of load-balancing algorithms reported in table 1.1, greater concentration of strategies are found in this (nearest-neighbours column). Amongst the two families into which we can divide this category when we focus upon the algorithmical aspects of the strategy, the load-balancing algorithms described below belong to the deterministic category. As was commented in section 1.5, deterministic algorithms are, in turn, divided into the subfamilies of diffusion and single direction. The load-balancing algorithm proposed in this thesis (chapter 3) belongs to the diffusion schemes and it has been compared, in chapter 4, with three well known load-balancing strategies from the literature: the SID (Sender Initiated Diffusion) algorithm, the AN (Average Neighbourhood) algorithm and the GDE (Generalised Dimension Exchange) algorithm. SID and AN, are diffusive strategies and they have been chosen for their popularity and similitude to the one proposed in this thesis. GDE belongs to the single-direction family and it has been selected as a representative algorithm within this family and for being one of the most popular and traditional load-balancing algorithms of the same.

In order to be rigorous in the description of the above mentioned load-balancing algorithms, in section 2.2 some generic notation and assumptions are introduced. All these algorithms perform the load-balancing process in an iterative way, therefore the convergence of the algorithm and its termination detection are important issues to be considered. Section 2.3 deals with these two problems. Finally, section 4 of this chapter includes a detailed description of SID, AN and GDE algorithms and the balance problems that arise from each of them are also analysed.

2.2 Basic notation and assumptions

The load-balancing algorithms described in the following section as well as the proposed in chapter 3 are suitable for parallel system that may be represented by a simple undirected graph G=(P,E), i.e. without loops and with one or zero edges between two different vertices. The set of vertices P={1,2,....,n} represents the set of processors. One edge {i,j} $\in$ E if there is a link between processor *i* and processor *j*. Let r_i denote the degree of a vertex *i* in *G* (number of direct neighbours of processor *i*) and *r* denote the maximum degree of *G*'s nodes. Note that in symmetrical topologies the number of immediate neighbours for all the processors in the system is the same, i.e., r_i and *r* will have the same value for any given processor *i*. The neighbourhood or domain of a given processor *i* is defined as the following set of processors, $N_i = \{j \in P \mid \{i,j\} \in E\} \cup \{i\}$. We assume that the basic communication model is the *all-port* one. This model allows a processor to exchange message with all its direct neighbours simultaneously in one communication step, i.e., communication hardware supports parallel communications over the set of links of a processor.

Furthermore, we state that at a given instant *t*, each processor *i* has a certain load $w_i(t)$ and the load distribution among the whole system is denoted by the load vector $W(t) = (w_1(t) \;\; \ldots \;\; w_n(t))$. The components of the load vector, $W(t)$, can be real or integer values depending on the granularity of the application. When the computational load of a processor is assumed to be infinitely divisible, load can be represented by a real number and, therefore, the perfect balance is achieved when all processors in the system have the same load value. This assumption is valid in parallel programs that exploit very fine grain parallelism. To cover medium and large grain parallelism, the algorithm must be able to handle indivisible tasks. Hence, the load would be represented by a non-negative integer number and, therefore, the global balanced load distribution is the one in which the maximum load difference throughout the entire system is 0 or 1 load unit.

Let $L = \sum_{i=1}^{n} w_i(t)$ be the total load of the system at a given time *t* and $\overline{w(t)} = L/n$ denote the *global load average* at the same time. Thus, the objective of a load-balancing algorithm is to distribute loads such that at some instant *t* each $w_i(t)$ is "close" to $\overline{w(t)}$.

2.3 Iterative nearest-neighbour load balancing algorithms

Most of the nearest-neighbour load-balancing algorithms are implemented in an iterative form where a processor balances its load by exchanging load with the neighbouring processors in successive executions of the three functional blocks introduced in section 1.3: the LM block, the LBA block and the MM block. By neglecting the computational operations related with the underlying application, and only concerning with the load-balancing process, a given processor *i* should be allocated the loop shown in figure 2.1.

```
While (not_converge)
{
  LB operations

    LM block

    LBA block

    MM block

}
```

Figure 2.1 Iterative LB process in a processor i.

By its distributed nature, nearest-neighbour load-balancing algorithms lack central coordination points that control the execution of load-balancing operations in each processor. As load decision movements are taken locally by each processor, an important issue that must be considered when designing iterative distributed load-balancing algorithms is *the convergence* of the algorithm. Convergence property guarantees that there exists a finite number of balancing iterations beyond which all the processors in the system do not receive or send any amount of load, and therefore a stable state is reached. With regard to the practical side of these load-balancing methods, another important issue to be addressed is the *termination detection* problem. In order to assist the processors inferring global termination of the load-balancing process from local load information, it is necessary to superimpose a distributed termination detection mechanism on the load-balancing procedure. These two issues are discussed in the following sections.

2.3.1 Algorithm convergence

Several analytical methods for studying distributed load-balancing algorithms have been developed for the idealised scenario where loads are treated as infinitely divisible and represented by real numbers; therefore, loads can be split to arbitrary precision [Ber89][Boi90][Cyb89][Hos90][Mut98][Wil93][Xu97]. The convergence of these methods has been studied using different mathematical approaches (ranging from linear algebra to other mathematical models). Cybenko [Cyb89], and independently by Boillat in [Boi90], made an important contribution by showing that ideas from linear system theory can be employed in characterising the behaviour of load balancing. We summarise Cybenko's approach . The load assigned to processor *i* at time *t+1* is given by

$$w_i(t+1)=w_i(t)+\sum_{j=1}^{n}\alpha_{ij}\left(w_j(t)-w_i(t)\right)$$

where the α_{ij}, for $1 \leq i,j \leq n$, satisfy the following conditions:

- $\alpha_{ij} = \alpha_{ji}$ *for all i, j.*
- *if* $\{i, j\} \notin E$, *then* $\alpha_{ij} = \alpha_{ji} = 0$.
- $\sum_{j=1}^{n}\alpha_{ij} \leq 1$ *for* $1 \leq i \leq n$.

This type of load assignment can be regarded as "diffusing" loads among the processors. Cybenko has shown that with certain choices for the diffusion coefficients, α_{ij}, this process tends to balance the total load among the vertices. It is possible to express the load vector at time *t+1* in terms of a matrix and the load vector at time *t*. This matrix is called the *diffusion matrix* and it is denoted as *M* where $M_{ij}=\alpha_{ij}$ for $i\neq j$ and $M_{ii}=1-\sum_{j\neq i}\alpha_{ij}$. It immediately follows that *w(t+1)=Mw(t).* It is straightforward to check that if each row of *M* sums up 1, then *L* is conserved. Different *M*'s result in different idealised load-balancing algorithms. Works from the literature that matches with this matrix behaviour concentrate their effort in bounding the time the algorithm takes to appreciably decrease imbalance and how this process can be accelerated. Classical works in this area [Boi90][Cyb89][Hos90][Xu97] were focused on the analysis of the influence of the value of α_{ij} in the convergence rate and different distributed load-balancing algorithms were proposed by simply changing the value of the diffusive coefficient, considering the eigenvalues of the Laplacian of the underlying topological graph. More recent works try to accelerate the convergence rate of previous diffusive approaches by keeping a memory at each load-balancing iteration of what happens in past iterations [Die99][Hu99][Mut98]. In these alternatives the properties of the Chebyshev polynomial should be applied to study the convergence rate.

All the above mentioned idealised distributed load-balancing algorithms are supposed to work in synchronous environments where all processors of the system perform load-balancing operations simultaneously. Another framework where the analysis of idealised distributed load-balancing algorithms was focused on is the asynchronous framework where each processor proceeds regardless of what the other processors do. Under this assumption, no global co-ordination is needed between processors. Asynchronous idealised iterative load-balancing algorithms differ from their synchronous counterparts in the manner in which the portion of excess load to be moved is evaluated. While synchronous algorithms use a fixed value (α_{ij}) to apportion the excess load of an overloaded processor, asynchronous algorithms use a variable parameter, named P_{ij}, which tend to depend on the current local load distribution [Cor99c][Wil93]. Therefore, the matrix model cannot be directly applied to this kind of algorithms. Bertsekas and Tsitsiklis proposed a load-balancing model for asynchronous idealised load-balancing algorithms [Ber89]. They also proved their convergence under the assumption that the balancing decisions do not reverse the roles of the involved processors and including some bounds for the message delays in arbitrary message passing networks (partial asynchronism).

In the more realistic setting of the problem which covers medium and large grain parallelism, the loads are not infinitely divisible and, as a consequence, they are treated as natural numbers. An interesting question that arises is: *what do the results* *mean in the realistic setting where the load is represented by natural numbers*? Some preliminary analysis about the problems related to consider loads as indivisible can be found in [Luq95][Mut97][Sub94]. A common solution for this question is found in all the above mentioned idealised distributed load-balancing algorithms. This solution simply consists of rounding the load quantities to be moved to make them integral. Then, the key is to analyse the convergence of these algorithms with respect to the analysis performed in the idealised situation. Intuitively it is clear that a scheme with the small perturbation (of rounding down) will behave similarly to the original one. However, as it was stated in [Mut98], applying standard linear algebraic tools to handle perturbation such as Gerschgorin's theorem in order to analyse the convergence rate of the algorithms yields only weak results. In some cases the convergence of the algorithm has only been substantiated by simulation results as it happens for the realistic versions outlined in [Son94][Wil93][Xu97]. In other cases, some theoretical works related with the convergence rate of the realistic counterparts are provided but only attending to obtain thigh bounds on the global unbalance [Cor99c][Cyb89][Hos90][Mut98], whereas the convergence proof was left open.

Therefore, there is no proof in the literature of the convergence of iterative load-balancing algorithms that work with realistic load model, as far as we are aware.

2.3.2 Termination detection problem

With regard to the practical side of iterative load-balancing methods, another important issue to be addressed is the termination detection problem. With iterative LB algorithms, the load-balancing procedure is considered terminated when the system reaches a global stable load distribution. From the practical point of view, the detection of the global termination is by no means a trivial problem because there is a lack of consistent knowledge in every processor about the whole load distribution as the load-balancing progresses. In order to assist the processors inferring global termination of the load-balancing process from local load information, it is necessary to superimpose a distributed termination detection mechanism on the load-balancing procedure. There is extensive literature on the termination detection of synchronous algorithms [Eri88][Haz87][Mat87][Ran83][Rön90][Szy85] as well as asynchronous forms [Cha85][Dij83][Fra82][Kum92][Sav96][Top84]. Most of the iterative LB algorithms proposed in the literature overlooked this problem by also attending to the analysis of its convergence. Ensuring that all these algorithms lead to termination consists of including one of the existent determination detection algorithms at the expense of increasing the overhead of the load-balancing process by the termination delay.

In continuation, and as an illustration of what has been mentioned, we shall now describe a synchronic termination algorithm called SSP, whose name derives from the initials of its authors [Szy85]. At the point of termination evaluation, two states should be distinguished in a processor: the *busy* and the *idle* state. A processor is in an idle state when the load-balancing process has locally finished, i.e., when the load remains unchanged after an iteration of the load-balancing process. Otherwise, a processor is considered to be in a busy state. But, subsequently, a busy processor can become idle and an idle processor may return to the busy state. Global termination occurs when all processors become idle. In order to detect this global termination, control messages are used to pass information about termination around, which can emanate from both busy and idle processors. All system processors keep a counter *S* to determine how far the farthest busy processor might be. The counter's value changes as the termination detection algorithm proceeds. *S* is equal to 0 if and only if the processor is in a busy state. At the end of a load-balancing iteration, each processor exchanges its counter value with all of its nearest neighbours. Then each idle processor updates its counter to be 1+min{*S, InputS*}, where *InputS* is the set of all received counter values. Evidently, the counter in a processor actually corresponds to the distance between this processor and the nearest busy processor. Since the control information of a busy processor can propagate at most over one edge in one iteration, when the value of S is *d*+1, where *d* is the diameter of the underlying topology, the global termination condition has been accomplished. This pseudo-code of this termination algorithm is shown in figure 2.2.

```
Algorithm SSP for Processor i
S = 0;
while ( S < d+1 ) {
        collect the S value from all neighbour processor and stored in InputS;
        S = min {S, InputS};
        if (LocalTerminated)
                S = S+1;
        else
                S = 0;
}
```

Figure 2.2 SSP algorithm for global termination detection.

2.4 Analysis of relevant nearest-neighbours load balancing algorithms.

As has been mentioned in the previous section, the LM, LBA and MM blocks are the three blocks that constitute the load-balancing process. In this section, we focus on the description of the LBA block for three well-known load-balancing algorithms by overlooking the LM and MM blocks: the SID (Sender Initiated Diffusion) algorithm, the GDE (Generalised Dimension Exchange) algorithm and the AN (Average Neighbourhood algorithm.

The LBA block implements the load-balancing rules that allow to a given processor *i* to obtain, based on its load and on that of the processors relevant to its domain evaluated in time *t*, its new load value at time *t*+1. Therefore, the load value of processor *i* at time *t*+1 can be defined as a function of all load values within the underlying domain at time *t* ($w_j(t) \quad \forall\, j \in N_i$) and it can be expressed as follows:

$$w_i(t+1) = LBA\big(w_j(t)\big) \;\; \forall\, j \in N_i$$

Below, the formal descriptions and analysis of the LBA block of the SID, the GDE and the AN algorithms are reported.

2.4.1 The SID (Sender Initiated Diffusion) algorithm

The SID strategy is a nearest-neighbour diffusion approach which employs overlapping balancing domains (defined in section 1.4.2) to achieve global balancing [Wil93]. The scheme is purely distributed and asynchronous. Each processor acts independently, apportioning excess load to deficient neighbours. Balancing is performed by each processor whenever it receives a load update message from a neighbour *j*, indicating that the neighbour load is lower than a preset threshold L_{LOW} ,i.e., ($w_j(t) < L_{LOW}$). Each processor is limited to load information from within its own domain, which consists of itself and its immediate neighbours. All processors inform their nearest neighbours of their load levels and update this information throughout program execution. Load-balancing activation is determined by first computing the average load in the domain, $\overline{w_i(t)}$, as follows.

$$\overline{w_i(t)} = \frac{1}{r+1}\left(\sum_{\forall j \in N_i} w_j(t)\right)$$

Next, if a processor's load exceeds the average load by a prespecified amount, $L_{threshold}$,that is $w_i(t) - \overline{w_i(t)} > L_{threshold}$, then it proceeds to perform the load movements decisions. Load distribution is performed by apportioning excess load to deficient neighbours. Each neighbour *j* is assigned a weight $d_{ij}^{+}(t)$ according to the following formula.

$$d_{ij}^{+}(t) = \begin{cases} \overline{w_i(t)} - w_j(t), & if\, w_j(t) < \overline{w_i(t)} \\ 0 & otherwise \end{cases}$$

These weights are summed to determine the total deficiency which is obtained as following

$$D_i(t) = \sum_{\forall j \in N_i} d_{ij}^{+}(t)$$

Finally, the portion of processor *i*'s excess load that is assigned to neighbour *j*, $P_{ij}(t)$, is defined as

$$P_{ij}(t) = \frac{d_{ij}^{+}(t)}{D_i(t)}.$$

Then, a non-negative amount of load, denoted by $s_{ij}(t)$, is transferred from processor *i* to processor *j* at time *t* and is computed as,

$$s_{ij}(t) = P_{ij}(t)\left(w_i(t) - \overline{w_i(t)}\right)$$

Balancing continues throughout application execution whenever a processor's load exceeds the local average by more than a certain amount. Figure 2.3 summarises the LBA block for the SID algorithm. In this figure $r_{ji}(t)$ denotes the amount of load received by processor *i* from the neighbour processor *j* at time *t*. A typical value for the $L_{threshold}$ parameter is zero to force the load-balancing algorithm reaching an evenly local load distribution.

```
LBA block
  evaluate $\overline{w_i(t)}$
  if ( $w_i(t) - \overline{w_i(t)} > L_{threshold}$ )
  {
      $w_i(t+1) = w_i(t) + \sum_{\forall j \in N_i \setminus i} r_{ji}(t) - \sum_{\forall j \in N_i \setminus i} s_{ij}(t)$
  }
```

Figure 2.3 LBA block for the SID algorithm in processor i.

The SID algorithm was originally devised to be applied under the assumption that load can be treated as infinitely divisible. Under this assumption, the algorithm has been experimentally proved to converge [Wil93]. However, if the algorithm is modified to support integer loads, some problems arise. There are two rounded operations that allow transforming $s_{ij}(t)$ into an integer value: *floor* or *ceiling*. Under the assumption of using discrete load values, the load-balancing process should coerce into a perfect balanced load distribution if all processors of the system end up with a load value equal to $\lceil L/n \rceil$ or $\lfloor L/n \rfloor$. We recall from section 2.1 that *L* is the total amount of load across the system, and *n* is the number of processors.

In figure 2.4 it is possible to observe how the convergence of the integer version of the SID algorithm greatly depends on how $s_{ij}(t)$ is rounded as it has been reported in [Cor97]. The nodes in yellow colour are the processors with load excess. The execution of the discrete SID algorithm in these processors will produce the load movements indicated with the black arrows. If we apply ceiling rounding operations, the processor loads could oscillate between unbalanced states. In the example shown in figure 2.4(a) the central node starts the LB process with load excess equal to 3.25. However, after executing one iteration of SID, it becomes an idle processor, while its neighbouring processors suddenly become overloaded processors. As a consequence, the external processors will try to distribute their new excess load by returning some units of load to the central processor, starting a new cycle of ping-pong load movements. A clear consequence of such behaviour is that the LB algorithm will never converge, and overloaded processors can suddenly become idle processors as happens to the central one. On the other hand, with the floor approach, the load distribution has converged to a situation that does not exhibit a perfect global balance (see figure 2.4(b)). Therefore, when a discrete version of the SID algorithm is needed to be implemented, the floor approach is chosen because it stops at a stable state although it may not be the even one. Thus, the LBA block of the discrete version of SID in a given processor *i* is summarised in figure 2.5.

The conditional sentence in figure 2.5 allows each processor to detect whether it is balanced or not according to its domain-load average. By taking an $L_{threshold}$ value equal to 0, if the comparison $w_i(t)-\overline{w_i(t)} > L_{threshold}$ is not accomplished, then the load of processor *i* coincides with its domain load average, and it is considered to be balanced. In this case, no load movements are guided by the LBA block. Otherwise, if the load of processor *i* is different from its own load average, then it is considered as unbalanced and the load-balancing rules are evaluated. However, the LBA block may provide no load movements as a consequence of the rounding down operation. Therefore, although the discrete version of the SID algorithm is able to detect that the underlying processor is not balanced, it is not always able to correct this situation. Below, we analyse in more detail, the reasons for such a situation.

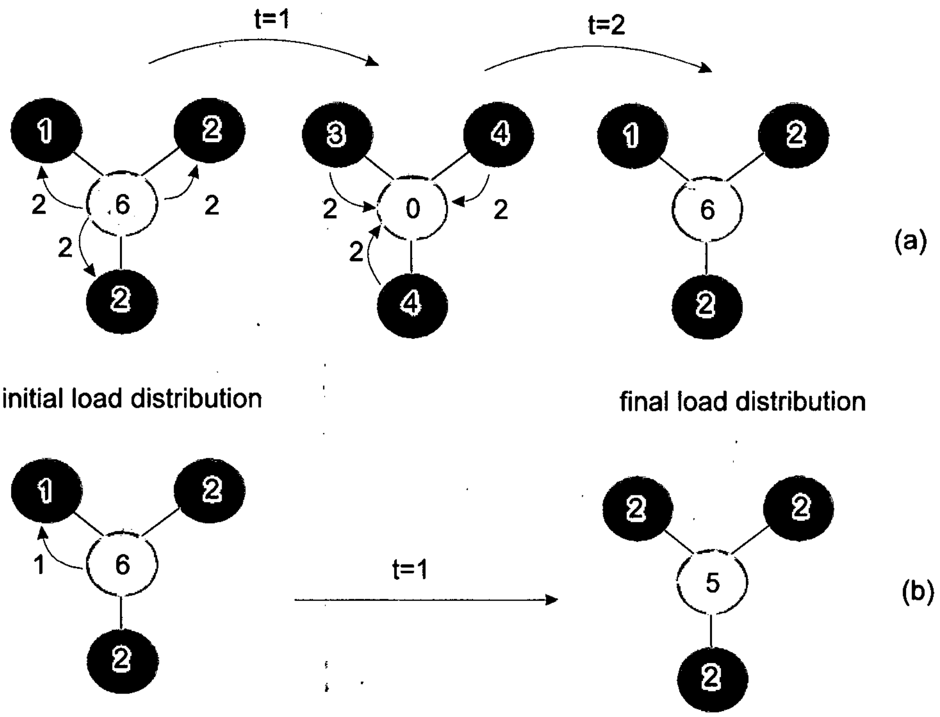


Figure 2.4 Integer versions of the SID algorithm by applying ceiling (a) and floor(b) rounding operation

```
LBA block
evaluate $\overline{w_i(t)}$
if ( $w_i(t) - \overline{w_i(t)} > L_{threshold}$ )
{
    $w_i(t+1) = w_i(t) + \sum_{\forall j \in N_i \setminus i} \lfloor r_{ji}(t) \rfloor - \sum_{\forall j \in N_i \setminus i} \lfloor s_{ij}(t) \rfloor$
}
```

Figure 2.5 LBA block for the discrete SID algorithm in processor i.

More precisely, there are three unevenly balanced local load distributions that the discrete SID algorithm is not able to arrange, in spite of being detected as unbalanced. These three situations are outlined below.

a) Processor *i* is an overloaded processor which has the highest load value within its own domain. In particular, the load value of processor *i* is equal to *m+s* ($w_i(t) = m + s$) and all its *r* neighbours have the same load value *m* ($w_j(t) = m \ \forall j \mid \{i,j\} \in E$), (see figure 2.6(a)). If the value of *s* is less than the number of neighbours ($0 \leq s < r$), then processor *i* guides no load movements. Moreover, if s>1, the underlying domain is unbalanced. Otherwise, if *s* is equal to 1, the maximum load difference among the underlying domains is 1, and then this situation it is considered as balanced. Figure 2.6(b) shows an example of this kind of load distribution, where the central processor has a load value equal to 8 ($w_i(t) = 8$), and neighbouring processors have a load equal to 4. Thus, *m* is 4 and, since *s* is also equal to 4, the domain is unbalanced and the discrete version of SID is not able to arrange such a situation, because after executing the LBA block the resultant load movements are all 0.

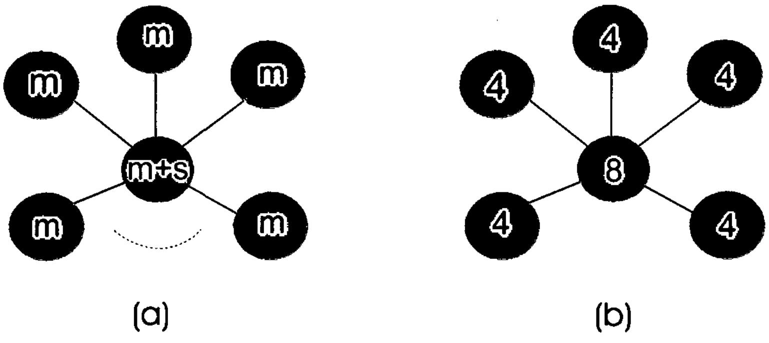


Figure 2.6 Unbalanced domain where the central processor has the highest load value and all its neighbour processors have the same load value

b) Processor *i* is the most loaded processor within its domain with a load value equal to *m* ($w_i(t) = m$) but not all its neighbours have the same load value. Suppose that the load values of the neighbouring processors are $n_1, n_2, \ldots, n_r$ and they accomplish the following two conditions:

- $n_1 \le n_2 \le \ldots \le n_r < m$,
- and $\left\lfloor \dfrac{m + \sum_{j=1}^{r} n_j}{r+1} \right\rfloor = n_1$

Then, no load movements are guided by processor *i*. Furthermore, if $m - n_1 > 1$ there exists an uneven load distribution. This generic situation is depicted in figure 2.7(a).

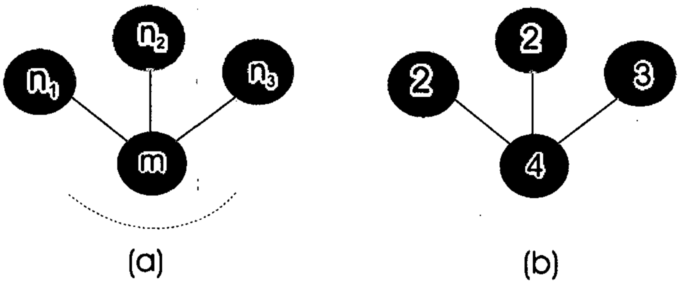


Figure 2.7 Unbalanced domain where the central processor has the highest load value within its domain, and its neighbours have different load values.

Figure 2.7(b) shows an example of this kind of load distribution. The red processor is the central processor *i* which has a load value equal to 4. The neighbouring processors have the following load values: 2, 2 and 3 which are identified with n_1, n_2, n_3 respectively. The two conditions exposed above are accomplished:

- $n_1 = 2 \le n_2 = 2 \le n_3 = 3 < 4$
- and $\left\lfloor \dfrac{4+(2+2+3)}{3+1} \right\rfloor = 2 = n_1$

Furthermore, with 4-2>1, then the underlying domain is unbalanced and SID is not able to balance it.

c) The last local unbalanced situation that SID is not able to arrange is the following. Suppose that processor *i*'s load is equal to o larger than the average load within its domain, but its load is not the biggest. Then, supposing that the load of its neighbours is denoted by $n_1, n_2, ..., n_r$ and *m* is the processor *i*'s load, then if the load distribution within the domain of processor *i* accomplishes:

- $n_1 \le ... \le n_i \le m \le n_{i+1} \le ... \le n_r$
- and $\left\lfloor \dfrac{m + \sum_{i=1}^{r} n_i}{r+1} \right\rfloor = n_1$,

processor *i* does not perform load movements. In addition, if $n_r - n_1 > 1$, the current local load distribution is unbalanced. Figure 2.8(a) shows a generic illustration of this situation.

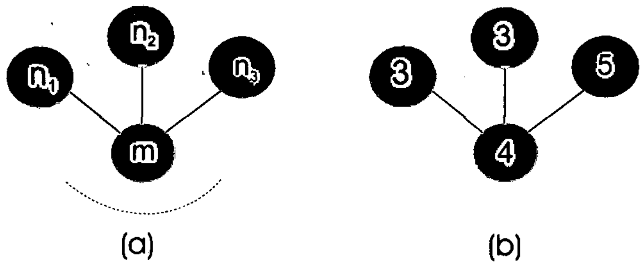


Figure 2.8 Unbalanced domain where the central processor is overloaded but is not the most loaded processor within the domain.

Figure 2.8(b) depicts an example of this situation. Processor *i* has a load value equal to 4, and its neighbours have the following load distribution: 3, 3 and 5. Under this situation the two condition, reported above are accomplished:

- $n_1 = 3 \le n_2 = 3 \le m = 4 \le n_3 = 5$ and
- $\left\lfloor \dfrac{4 + (3+3+5)}{3+1} \right\rfloor = 3 = n_1$

Furthermore, as 5 – 3 > 1 the underlying domain is unbalanced, but the underlying processor will not perform any load movement.

Finally, we can conclude that, although the original SID algorithm is able to detect unbalanced situations and arrange them, the discrete version of SID, which is a more realistic approach, is not able to achieve even load distributions. Furthermore, although it has been experimentally proved that the discrete SID stops, no theoretical proof about its convergence has been provided.

2.4.2 The GDE (Generalised Dimension Exchange) algorithm

The GDE algorithm belongs to the single-direction family within the nearest-neighbour strategies as stated in table 1.1. This strategy is based on the dimension-exchange method which was initially intensively studied in hypercube-structured multicomputers [Cyb89][Ran88][Shi89b]. In these kinds of architectures, the dimension exchange works in the way that each processor compares its load with those of its nearest neighbours one after another. At each one of these comparisons, the processor would try to equalise its load with its neighbour's. To do this systematically, all the processors could follow the order as implied by the dimension indices of the hypercube: equalising load with the neighbour along dimension 1, and then along dimension 2, and so on. Thus, the load balancing algorithm block of the Dimension Exchange algorithm is defined as follows:

```
LBA block
For each dimension
{
  if there exists an edge (i,j) along the current dimension
          $w_i(t+1) = 0.5w_i(t) + 0.5w_j(t)$
}
```

Figure 2.9 LBA block for the original DE algorithm in processor i.

The Dimension Exchange method is characterised by "equal splitting" of load between a pair of neighbours at every communication step. Due to this fact, this method does not take fullest advantage of the all-port communication model described in section 2.2, having to realise as many communication steps as dimensions exists in the underlying hypercube to execute the entire for-loop in the LBA block. This algorithms way of working adapts best to the one-port communication model where a processor is restricted to exchange messages with at most one direct neighbour at one time.

It has been shown that this simple load-balancing method yields a uniform distribution from any given initial load distribution in hypercubes topologies in a single sweep (i.e. one iteration of the for-loop) [Cyb89]. For arbitrary networks it may not be the case, and the dimension can be defined by edge-colouring techniques. With edge-colouring techniques [Hos90], the edges of a given system graph *G* are coloured with some minimum number of colours (*k*) such that no two adjoining edges are of the same colour. The colours are indexed with integer numbers from 1 to *k*, and represent the *k*-colour graph as G_k. A "dimension" is defined as being the collection of all edges of the same colour. An edge between vertices *i* and *j* with a chromatic index *c* in G_k is represented by a 3-tuple *(i,j,c)*. It is known that the minimum number of colours *k* is strictly bounded by r , and $r \leq k \leq r+1$ [Fio78]. Figure 2.10 shows an example of a 4x4 mesh coloured with four colours where the numbers beside the edges correspond to the four colours. During each sweep, all colours/dimensions are considered in turn. Since no two adjoining edges have the same colour, each node needs to deal with at most one neighbour at each iteration step (each step corresponds to one colour; a sweep corresponds to going through all the colours once –figure 2.11 (a)-).

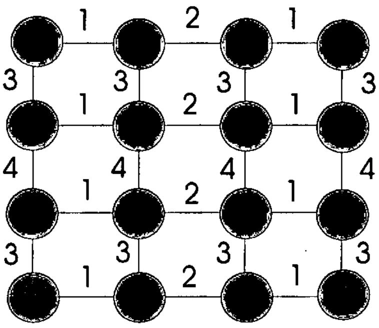
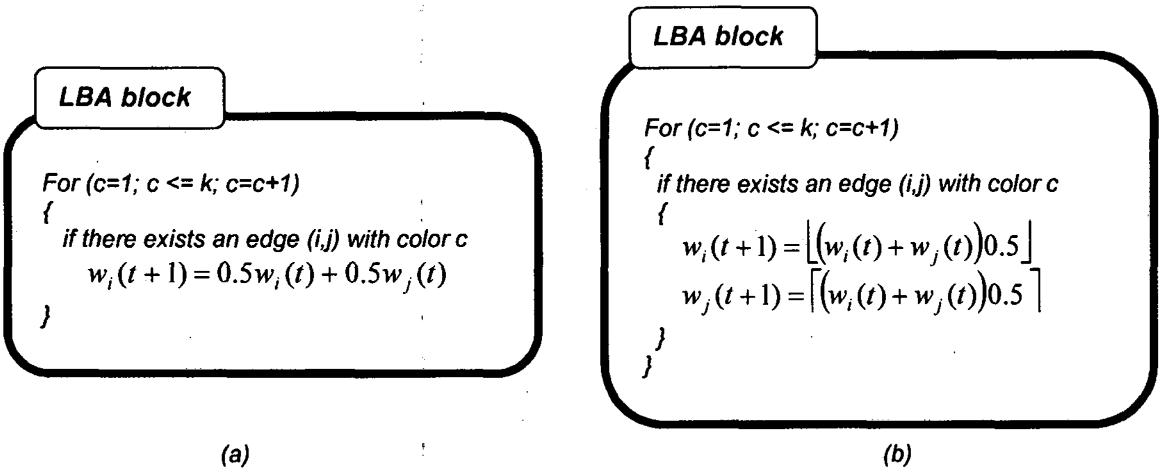


Figure 2.10 Edge coloring for a 4x4 mesh

Hosseini *et alter* also studied the convergence property of this method by treating load as real numbers and using linear system theory. Moreover, the authors also derived an integer version of this load-balancing algorithm as a perturbation of the linear version. The load-balancing algorithm block applied to this alternative is shown in figure 2.11(b).



LBA block

```
For (c=1; c <= k; c=c+1)
{
  if there exists an edge (i,j) with color c
     $w_i(t+1) = 0.5w_i(t) + 0.5w_j(t)$
}
```

(a)

LBA block

```
For (c=1; c <= k; c=c+1)
{
  if there exists an edge (i,j) with color c
  {
     $w_i(t+1) = \lfloor (w_i(t) + w_j(t))0.5 \rfloor$
     $w_j(t+1) = \lceil (w_i(t) + w_j(t))0.5 \rceil$
  }
}
```

(b)

Figure 2.11 LBA block for the DE algorithm applied to arbitrary networks in processor i using real (a) and discrete (b) load values

A question that occurred to the authors when this integer version was proposed is how does one decide which node receives the floor and which the ceiling for the average of the paired nodes?. Rather than dealing directly with this question the authors introduced a notation that records the choices made in each balance operation allowing the load-balancing algorithm to reach a load distribution, where deviation from the global load average keeps bounded. This imprecision denotes the difficulty of deriving a load-balancing algorithm that treats load as integer numbers, which is a more realistic approach using the dimension exchange idea. Xu and Lau introduced a more precise integer version of the dimension exchange idea for arbitrary networks which ensures its convergence. The revised algorithm proposed in [Xu97] was shown in figure 2.12.

```
LBA block
For (c=1; c <= k; c=c+1)
{
  if there exists an edge (i,j) with color c
    $w_i(t+1)=\begin{cases} \lceil 0.5w_i(t)+0.5w_j(t) \rceil & \mathit{if} w_i(t) \geq w_j(t) \\ \lfloor 0.5w_i(t)+0.5w_j(t) \rfloor & \mathit{otherwise} \end{cases}$
}
```

Figure 2.12 LBA block for the discrete versio of the DE algorithm for arbitrary networks in processor i.

Figure 2.13 shows the algorithm's behaviour for a 4x4 mesh where the algorithm 2.12 is applied. The edge colouring used corresponds to the one shown in figure 2.10. Suppose that the load distribution at some time instant is as in figure 2.13(a), in which the number inside a processor represents the load of the processor. Then, after a sweep of the dimension exchange procedure, the load distribution changes to that in figure 2.13(b). The load distributions obtained when executing the load balancing function at each dimension/colour is also shown. The red edges denote the pair of processor, involved in the load exchange within the inspected dimension/colour, and the arrows show the amount of load moved between them. In order to achieve a stable global load distribution, the load-balancing algorithm block (one sweep) must be executed three times more, as is shown in figure 2.14. The obtained final load distribution is not a global balance situation because the maximum load difference throughout the entire system is two load units instead of one. Thus, one can derive that an equal splitting of load between pairs of processors does not always coerce into an even global load distribution, where the maximum load difference between any two processors within the system is one load unit.

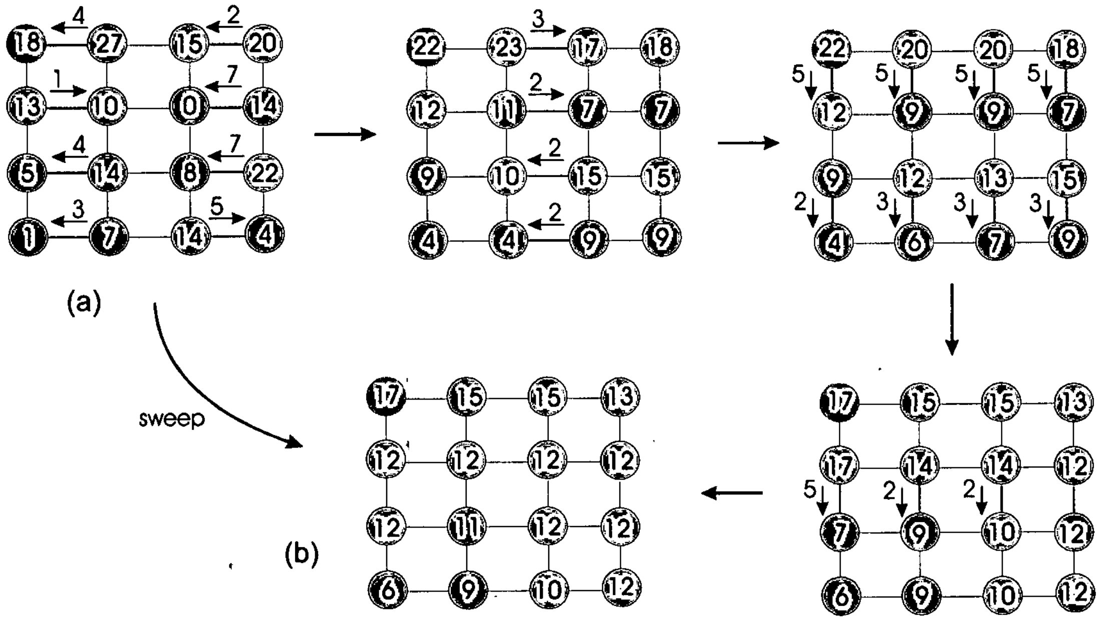


Figure 2.13 Load distribution during a sweep of DE for arbitrary networks

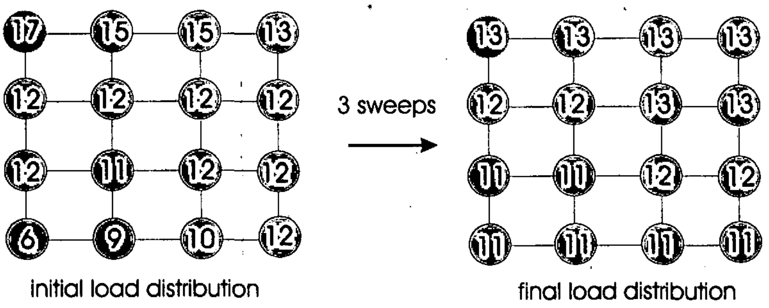


Figure 2.14 DE for arbitrary networks

In the light of the intuition that non-equal splitting of load might lead to fewer sweeps necessary for obtaining a uniform distribution in certain networks, Xu and Lau generalised the dimension exchange method by adding an *exchange parameter* to control the splitting of load between a pair of directly connected processors. They called this parameterised method the Generalised Dimension Exchange (GDE) method [Xu92][Xu94][Xu97]. The GDE method is based on edge-colouring of the given system *G* using the idea introduced by Hosseini *et alter* in [Hos90] and reported above. For a given G_k, let $w_i(t)$ denote the current local load of a processor *i* and λ denote the exchange parameter chosen. Then, the change of $w_i(t)$ in the processor using the GDE method is governed by the following LBA block:

```
LBA block
For (c=1; c <= k; c=c+1)
{
  if there exists an edge (i,j) with color c
              $w_i(t+1) = (1-\lambda)w_i(t) + \lambda w_j(t)$
}
```

Figure 2.15 LBA block for the GDE algorithm in processor i.

In order to guarantee $w_i(t) \geq 0$, the domain of the exchange parameter λ is restricted to [0,1]. There are two choices of the exchange parameter which have been suggested as rational choices in the literature:

- λ=1/2: equally splitting the total load of a pair of processors. Note that this choice reduces the GDE algorithm to the Dimension Exchange method. This special version of the GDE algorithm is referred to as the ADE (Averaging Dimension Excahge) method. As mentioned above, this value of the exchange parameter favours hypercube topologies.

- $\lambda = 1/(1+sin(\pi/k))$ in the mesh and $\lambda = 1/(1+sin(2\pi/k))$ in the torus where k is the largest dimension of the corresponding topology. This variant of the GDE method is known as the ODE (Optimally tuned Dimension Exchange) method. It has been proved in [Xu95] that under the assumption that no load is created or consumed during the load balancing process, the two previous values of the exchange parameter are optimal for meshes and torus respectively.

Xu and Lau have shown that in easy numeric terms, the optimal exchange parameter for *k*-ary *n*-cubes topologies ranges between [0.7, 0.8].

These theoretical results were obtained by Xu and Lau treating loads as real numbers. As we have previously mentioned, to cover medium and large grain parallelism which are more realistic and more common in practical parallel computing environments, load may be treated more conveniently as non-negative integers. Then, a modified version of the GDE algorithm is needed. Such a variant is included in the code of figure 2.16 where the revised formula of the LBA block is shown.

```
LBA block
For (c=1; c <= k; c=c+1)
{
  if there exists an edge (i,j) with color c
    $w_i(t+1) = \begin{cases} \lceil (1-\lambda)w_i(t)+\lambda w_j(t) \rceil & \text{if } w_i \geq w_j \\ \lfloor (1-\lambda)w_i(t)+\lambda w_j(t) \rfloor & \text{otherwise} \end{cases}$
}
```

Figure 2.16 LBA block for the discrete version of the GDE algorithm in processor i.

Because of the use of integer loads, the load balancing process will end with a variance of some threshold value (in load units) between neighbouring processors. This threshold value can be tuned to satisfactory performance. By setting this value to one load unit the closest total load-balancing is enforced. Then, it is clear that $0.5 \leq \lambda < 1$ because a pair of neighbouring processors with a variance of more than one load unit would no longer balance their loads when $\lambda < 0.5$. Xu and Lau demonstrate experimentally that the optimal exchange parameter λ when the integer load model is applied, is not always 0.5, but somewhere between 0.7 and 0.8, which is an agreement with their theoretical results.

As happens to other load-balancing algorithms that were originally devised to be used by treating loads as infinitely divisible, the discretization approach of the original GDE algorithm also arises some problems which have been analysed in [Mur97][Sub94]. Figure 2.17 shows the load changes performed in the pictured domain after executing one sweep (three iterations) of the discrete version of the

GDE algorithm, where exchange parameter λ has been chosen as 0.75. As can be observed in the example, the discrete version of GDE may converge to a situation that does not exhibit a perfect global balance.

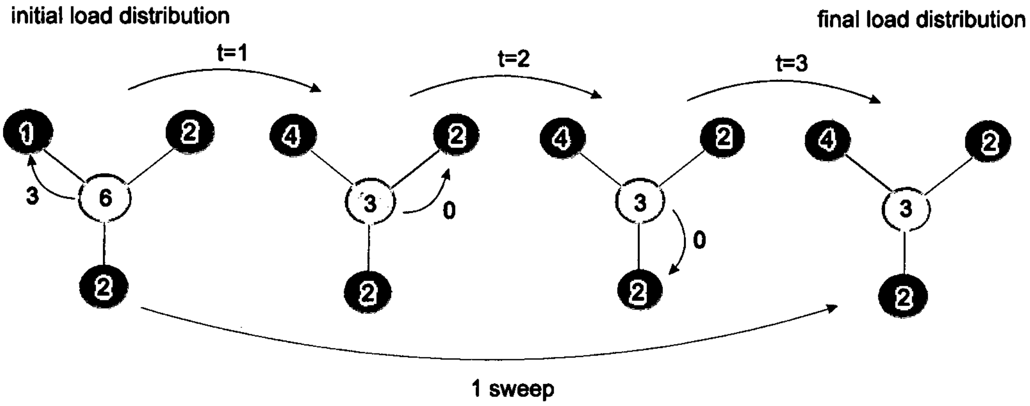


Figure 2.17 Discrete versions of the GDE algorithm with exchange parameter λ equal to 0.75

2.4.3 The AN (Average Neighbourhood) algorithm

The Average Neighbourhood algorithm is a load-balancing algorithm which belongs to the diffusion family as well as the SID algorithm (see table 1.1). These two algorithms are similar in the sense that both evaluate the load average within the domain of each processor to determine whether it has load excess or not. In other words, if the load value of a given processor *i* is bigger than the load average within its domain, the LBA block may produce some load movement decisions to distribute this surplus load among the neighbouring processors. From the analysis of SID's behaviour reported in section 2.4.1, we have concluded that although there is a time when the execution of SID in a given processor *i* provides no load movements, the load distribution obtained at that time may not be balanced. The AN algorithm tries to arrange this problem by including the capability of moving loads between non-directly connected processors. These load movements are restricted to being performed between processors that have a common neighbouring processor. Therefore, the goal of the AN's LBA block on processor *i* is to balance the load of all processors belonging to its domain [Cor99c]. This goal is wider than simply trying to balance the load of processor *i* only. The LBA block computes the average domain load using the load information from all processors belonging to its domain. Bearing in mind that the domain of a given processor *i* is denoted by N_i ,and that *r* is the number of the direct neighbours in a symmetric topology, this average is evaluated as follows:

$$\overline{w_i(t)} = \frac{1}{r+1}\left(\sum_{\forall j \in N_i} w_j(t)\right)$$

Two thresholds values named T_{sender} and $T_{receiver}$ which are centred around the domain load average, are also evaluated. A processor *i* is classified as:

- **sender** in N_i if $T_{\text{sen}\,der}$ is smaller than the current load value of processor *i* ($w_i(t) > T_{\text{sen}\,der}$);
- **receiver** if $T_{receiver}$ is bigger than the current load value of processor *i* ($w_i(t) < T_{receiver}$);
- **neuter** in any other case, i.e., $T_{receiver} \leq w_i(t) \leq T_{\text{sen}\,der}$.

Any time a processor *i* receives load information from one of its neighbours, the new load average within its domain is calculated and T_{sender} and $T_{receiver}$ are updated to evaluate the needs of load-balancing in the domain.

In a single balancing iteration, depending on the domain load situation, one action can be decided:

1. if processor *i* is sender and at least one neighbouring processor is a receiver, processor *i* sends load towards them proportional to its deficit load;
2. if processor *i* is received and there is at least one sender in the domain, processor *i* requests load from one of them, in particular the most loaded one, and the amount of load requested will be proportional to its excess;
3. if processor *i* is neuter but there are senders in the domain, it requests to provide load from the most loaded sender and to give it to the most underloaded processor of the domain.

In other words, a given processor *i* tries to push the load of all processors within its domain as close as possible to the domain load average. However, the algorithm gives priority to balancing processor *i* (point 1 and 2). Only if it is balanced does the algorithm try to balance neighbouring processors (point 3). The described implementation achieves the goal of moving the load of every domain processor within the range defined by T_{sender} and $T_{receiver}$. In order to achieve a perfect balanced domain where the maximum local load difference is no bigger than one load unit, both thresholds must coincide with the local load average, i.e., $T_{receiver} = T_{\mathrm{sen}\,der} = \overline{w_i(t)}$. It is important to remark that a load movement must not reverse the role of processors in the domain to grant stability, in other words, the load reallocation must be limited to avoid a sender processor becoming a receiver in the domain, and *viceversa*. Figure 2.18 summarises the LBA block for the AN algorithm.

An important difference in the AN algorithm with respect to the two load-balancing algorithms described in the previous sections (SID and GDE) lies in the limitation of the number of processors in the system that can be executing the balancing process simultaneously. This means that some degree of co-ordination between the LBA blocks is needed to avoid situations in which balancing operations take place concurrently in overlapping domains. For this reason, all system processors are organised in processors subgroups, in such a way that the processor domains relevant to a particular subgroup are totally disjoint. These subgroups are referred to as *serialisation sets,* and there should be the minimum possible number in existence. Since the AN algorithm is a totally distributed load-balancing algorithm, each processor of the system belongs to a serialisation set. Bearing in mind this serialisation restriction, it would imply covering all the serialisation sets at once in order to obtain that all processors execute the balancing process one single time. Figure 2.19 depicts all the serialisation sets in a 4x4 torus topology where the red processors are the processors which are allowed to execute the load-balancing process. As we can observe there are 8 serialisation sets with 2 non-overlapped domain in each one of them.

LBA block

evaluate $\overline{w_i(t)}$, T_{sender} *and* $T_{receiver}$

$$w_i(t+1)=\begin{cases} w_i(t)-\sum\limits_{\forall j\in N_i\backslash i} s_{ij}(t) & \text{if } w_i(t)>T_{sen\,der} \\ w_i(t)+\sum\limits_{\forall j\in N_i\backslash i} r_{ji}(t) & \text{if } w_i(t)<T_{receiver} \\ w_i(t) & \begin{array}{l}\textit{otherwise: in this case load movements} \\ \textit{can be commanded between processors} \\ \textit{belonging to the underlying domain}\end{array} \end{cases}$$

Figure 2.18 LBA block for the AN algorithm in a master processor i.

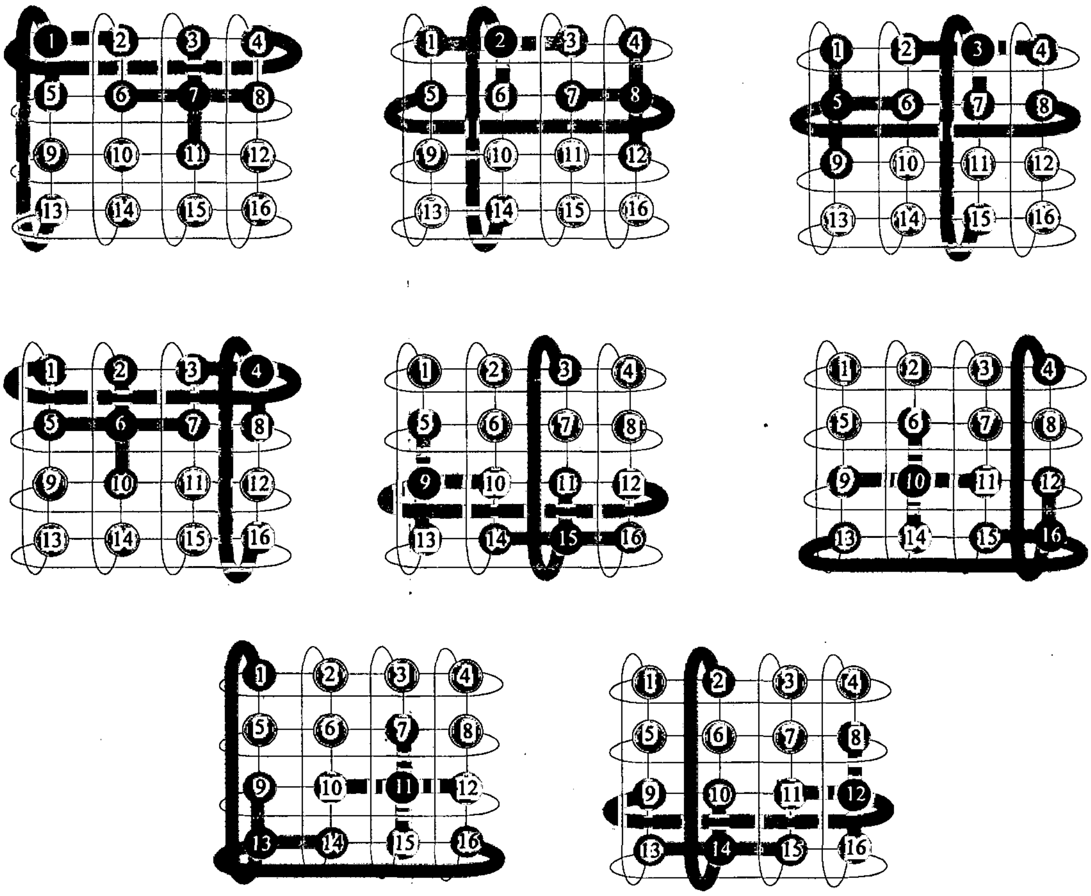


Figure 2.19 Serialisation sets for a 4x4 topology

Whenever the activity of the algorithm is stopped because competing load-balancing operations are taking place in overlapping domains, the algorithm cannot simply wait for their completion. The situation of the domain processors is likely to have been changed, and the information upon which a decision was based could have become obsolete. The load-balancing process must abort whenever it is delayed, because of competing actions. Therefore, the need for serialising the load-balancing process can be viewed as an important constraint, since it generates high synchronisation message traffic and continuous interruptions of the load-balancing operations in some processors.

The convergence of the AN algorithm has been proved with respect to keeping the global load variance bounded beyond a given time [Cor96]. However, final load distribution does not always exhibit a global balance situation as happens in the example depicted in figure 2.20 where the maximum load difference between any two processors at the end of the load-balancing process is 2.

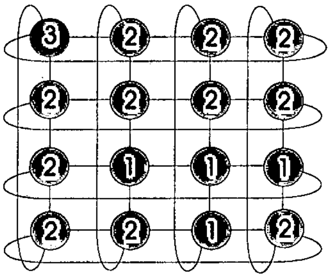


Figure 2.20 Final unbalanced situation applying the AN algorithm

2.5 Summary of this chapter

This section is oriented to summarise the main characteristics of each one of the three algorithms described above, in order to have a global view of their similarities and differences with respect the main implementation key issues and their derived capabilities. Recalling from chapter 1 that the load-balancing process viewed from the processor level point of view, is decomposed into three functional blocks: LM (Load Manager) block, LBA (Load Balancing Algorithm) block and MM (Migration Manager) block. In particular, in this chapter, we concentrate on an exhaustive description of the corresponding LBA (Load-Balancing Algorithm) bock which, recalling from chapter 1, is the block of the global load-balancing process that decides source and destination processors for load movements, as well as the amount of load to be moved between them.

MAIN CHARACTERITSTICS				SID	GDE	AN
Implementation Key Issues	Processor Level	LM block	Original Load Model	Infinitely divisible load units	Infinitely divisible load units	Discrete load units
			Domain	Nearest Neighbours	Nearest Neighbours	Nearest Neighbours
		LBA block	Load Balancing Activation (trigger condition)	One adaptive threshold (local load average)	One adaptive threshold (load difference between two neighbours	Two adaptive threshold (evaluated around the local load average)
			Work Transfer Calculation	Evaluated (depending on load distribution within the domain)	Fixed (exchange parameter)	Evaluated (depending on load distribution within the domain)
	System Level		Set of running processors	All system processors	All system processors	All system processors
			Degree of co-operation	Asynchronous	Synchronous	Synchronous
			Simultaneous active processors	All system processors	Processors involved in the same dimension	Processors belonging to the same serialisation set
Behaviour Features	Convergence			Not proved	Not proved	Not proved
	Detection of unbalanced domains			Not allowed	Not allowed	Not allowed
	Local Balance			Not always achieved	Not always achieved	Not always achieved
	Global Balance			Not always achieved	Not always achieved	Not always achieved
	Movements between non-directly connected processors			Not allowed	Not allowed	Allowed

Table 2.1 Summary of the main characteristics of SID,GDE and AN.

As was described in section 1.3.2 of this work, the LBA block is divided into two phases: *Load Balancing Activation* and *Work Transfer Calculation*. The first phase is responsible of testing the *trigger condition* and, therefore, determining which processors are the *active processors* at a certain time, i.e., which processors overcome the *trigger condition* and will consequently execute the second phase of the LBA block. In table 2.1, we have included the particular characterisation of each one of these phases for all analysed algorithms. However, not only these characteristics are reported in that table. Other relevant issues from the processor level point of view, as well as from the system level point of view are also considered in its summary. Furthermore, table 2.1 incorporates the behaviour features derived from the execution of each algorithm.

Chapter 3

DASUD load-balancing algorithm

Abstract

In this chapter a new dynamic load-balancing algorithm called DASUD (Diffusion Algorithm Searching Unbalanced Domains), which uses a local iterative scheme to achieve a global balance, is proposed. DASUD was developed for applications with a coarse and large granularity where load must be treated as non-negative integer values. Firstly, in this chapter, we describe the proposed algorithm, its complexity is analysed and its functioning demonstrated for an example in which the underlying topology is a 3-dimensional hypercube. Subsequently, the proof of DASUD's convergence is provided, and bounds for the degree of overall balance achieved are provided, as well as for the number of iterations required for such balance.

*3.1*DASUD (Diffusion Algorithm Searching Unbalanced Domains)'s motivation

The load-balancing strategies within the nearest-neighbour category exhibited some problems when load is packaged into discrete units. In particular, in chapter 2, the problems related to the three algorithms described have been analysed. Summarising, the main problem that all these strategies exhibited is that they may produce solutions which, although they are locally balanced, prove to be globally unbalanced. Figure 3.1 shows the load of 6 processors connected in a linear array, which are obtained as balanced solution by most of the nearest-neighbour load-balancing algorithms. However, this load distribution is not an even distribution because the maximum load difference through the whole system is 5 load units. We recall from chapter 2 that the system is balanced when the global maximum load difference is 0 or 1 load unit.

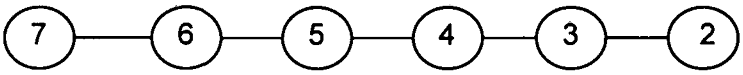


Figure 3.1 Final stable load distribution but globally unbalanced.

The proposed algorithm DASUD, is based on the SID load-balancing algorithm and incorporates the evaluation of some new parameters to detect stable unbalanced local load distributions achieved by SID and generates some load movements to slightly arrange them. DASUD detects as unbalanced situations such as the one shown in figure 3.1 and is able to drive the system to a final balanced load distribution such as the one depicted in figure 3.2.

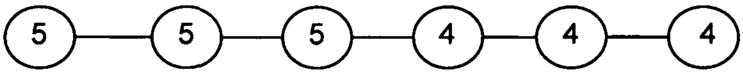


Figure 3.2 Final balanced load distribution.

In table 3.1, we enumerate the characteristics of DASUD by using the same scheme that the one used in section 2.5 to summarise the main characteristics of the three load-balancing algorithms described in chapter 2 (SID, GDE and AN). In the following section, an exhaustive description of DASUD's behaviour is provided.

MAIN CHARACTERITSTICS				DASUD
Implementation Key Issues	Processor Level	LM block	Original Load Model	Discrete load units
			Domain	Nearest Neighbours
		LBA block	Load Balancing Activation (trigger condition)	One adaptive threshold (local load average)
			Work Transfer Calculation	Evaluated (depending on load distribution within the domain)
	System Level		Set of running processors	All system processors
			Degree of co-operation	Asynchronous
			Simultaneous active processors	All system processors
Behaviour Features			Convergence	Finitude proved
			Detection of unbalanced domains	Allowed
			Local Balance	Always achieved
			Global Balance	Upper Bounded
			Movements between non-directly connected processors	Allowed

Table 3.1 Main characteristics of the DASUD load-balancing algorithm

3.2 Description of the DASUD algorithm

We now discuss the behaviour of one iteration of DASUD or, in effect the same thing, the LBA block of the DASUD algorithm. Essentially, one iteration of DASUD consists of two load-balancing stages as is shown in figure 3.3. The first stage performs a coarse load distribution of the load excess of the underlying processor, whereas the second stage produces a more accurate excess distribution in order to achieve the perfect load balance within the underlying domain. More precisely, in the first stage, if the underlying processor is an overloaded processor it will proportionally distribute its excess load among its underloaded neighbour processors in such a way that the most underloaded neighbours will receive more load than the less underloaded ones. In the second stage, firstly, each processor checks its own domain to determine whether it is unbalanced or not. In order to balance the underlying domain, each processor can proceed by completing its excess load distribution in a more refined way, sending messages to an overloaded neighbour instructing it to send load to an underloaded neighbour, or performing load movements between non-directed connected processors.

We now formally describe each one of the stages of DASUD. In order to do so, we use the same nomenclature described in chapter 2, and we introduce certain new notation that is described below.

In the DASUD algorithm, each processor sends a message, at certain given times, to all its neighbours containing its local load. This information is not updated immediately in all neighbour processors due to delays introduced by the network. Therefore, each processor *i* keeps in its local memory an estimation of processor's *j* load (we denote $w_{ij}(t)$ as the load estimation of processor *j* kept in memory by processor *i* at time *t*). Then, if *i* and *j* are neighbour processors (i.e. *{i,j}* $\in$ *E*), $w_{ij}(t) = w_j(\tau_{ij}(t))$, where $\tau_{ij}(t)$ is a certain time instant satisfying $0 \leq \tau_{ij}(t) \leq t$.

For convenience, if *i* ≠*j* and *{i,j}* $\notin$ *E* then $w_{ij}(t) = 0$. Each processor is able to assess its current load at each instant. This means that $w_{ii}(t) = w_i(t)$. For our algorithm, it is also convenient that each processor has a list of its neighbours sorted according to the assigned processor indexes. This sorted list will be used as criteria for selection in those cases in which a processor must be selected from amongst a subset of neighbouring processors.

DASUD works asynchronously. Therefore, each processor *i* can be associated with a set of instants T_i that will be denoted as a set of *load balancing times* for processor *i*. At each one of these instants, processor *i* begins the first stage of the DASUD algorithm. In this stage each processor compares its load with the load estimation of all its neighbours that are stored in its local memory. As each T_i is a time-discrete set, in order to study DASUD's behaviour, we can discriminate the variable *t*. Therefore *t* assumes the value 0,1,2,...

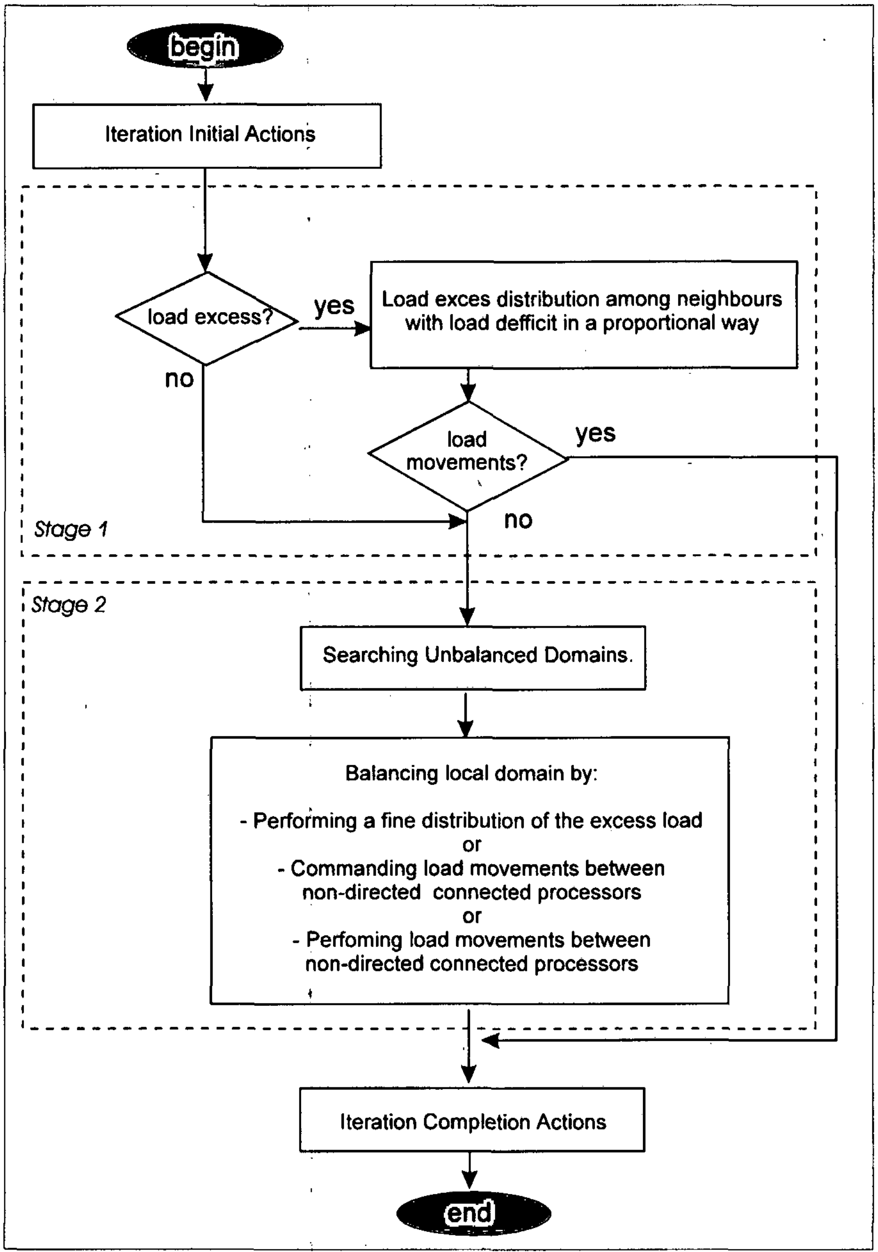

begin
Iteration Initial Actions
load excess?
yes
no
Load exces distribution among neighbours with load defficit in a proportional way
load movements?
yes
no
Stage 1
Stage 2
Searching Unbalanced Domains.
Balancing local domain by:
- Performing a fine distribution of the excess load
or
- Commanding load movements between non-directed connected processors
or
- Perfoming load movements between non-directed connected processors
Iteration Completion Actions
end

Figure 3.3 One iteration of DASUD algorithm

Firstly, processor *i* performs some initial actions starting by checking the load estimations kept in its memory of its neighbours ($w_{ij}(t)$) and it computes the load average of its domain as follows:

$$\overline{w}_i(t) = \frac{\sum_{j \in N_i} w_{ij}(t)}{\# N_i}$$

Once processor *i* has computed its local load average, $\overline{w}_i(t)$, it also evaluates a local load weight, denoted by $d_{ii}(t)$, in order to detect whether it is an overloaded processor or not, $d_{ii}(t) = \overline{w}_i(t) - w_i(t)$. If processor *i* is an overloaded processor, $d_{ii}(t)$ will be a negative value ($d_{ii}(t) < 0$). Otherwise, if processor *i* is an underloaded processor, $d_{ii}(t)$ will be a non-negative value ($d_{ii}(t) \geq 0$).

Then, depending on the value of $d_{ii}(t)$ one of the two stages of DASUD will be performed. If $d_{ii}(t) < 0$ then the first stage will be executed. Otherwise, if $d_{ii}(t) \geq 0$ the computation will go on with the second stage.

3.2.1 Description of the first stage of DASUD

In this stage, the load excess of the processor *i* is distributed among its neighbouring processors with load deficit. The load excess distribution is performed proportionally to the corresponding load deficit. For this purpose, a load weight $d_{ij}(t)$ is evaluated for each processor *j* belonging to the domain of processor *i* according to the following formula: $d_{ij}(t) = \overline{w}_i(t) - w_{ij}(t)$. An overloaded processor *i* ($d_{ii}(t) < 0$) performs load balancing by apportioning its excess load only to deficient neighbours, *j*, whose load weight is a positive value ($d_{ij}(t) > 0$). The amount of excess load to be moved from an overloaded processor *i* to one of its deficient neighbours *j* will be denoted by $s_{ij}(t)$. In order to evaluate $s_{ij}(t)$, a new weight called $d_{ij}^{+}(t)$ is computed for all processors *j*:

$$d_{ij}^{+}(t) = \begin{cases} d_{ij}(t) & if\ d_{ij}(t) > 0 \\ 0 & otherwise \end{cases}$$

The total amount of load deficits is computed on $D_i(t)$ to determine the total deficiency, $D_i(t) = \sum_{j=1}^{n} d_{ij}^{+}(t)$

Subsequently, the portion of excess load of processor *i* that is assigned to neighbour *j*, $P_{ij}(t)$, is computed as follows,

$$P_{ij}(t) = \begin{cases} \frac{d_{ij}^{+}(t)}{D_i(t)} & if\ d_{ii}(t) < 0 \\ 0 & otherwise \end{cases}$$

Then, a non-negative amount of load, denoted by $s_{ij}(t)$, is transferred from processor *i* to processor *j* at time *t* and is computed as, $s_{ij}(t) = floor(-P_{ij}(t) * d_{ii}(t))$

If $s_{ij}(t) = 0$ for all *j*, i.e., no load movements are performed by the first stage of DASUD, then the second stage will be executed. Otherwise, the second stage of DASUD is skipped and no more load movements will be performed during this iteration.

3.2.2 Description of the second stage of DASUD

In this stage, DASUD evaluates the balance degree of processor domain *i* by searching unbalanced domains. This stage is composed of four blocks which work together with the aim of completely balancing the underlying domain. These blocks are: the SUD (Searching Unbalanced Domains), FLD (Fine Load Distribution), SIM (Sending Instruction Message) and PIM (Processing Instruction Messages) blocks. Each one of these blocks are described below.

Searching Unbalanced Domains (SUD) block

In this block four parameters are evaluated:

a) maximum load value of the whole domain (included *i*): $w_i^{max}(t)$,

b) minimum load value of the whole domain (included *i*): $w_i^{min}(t)$,

c) maximum load value of neighbouring processors of processor *i*: $w_{vi}^{max}(t)$,

d) minimum load value of neighbouring processors of processor *i*: $w_{vi}^{min}(t)$.

The maximum load difference through the domain of processor *i* is evaluated ($w_i^{max}(t)$ - $w_i^{min}(t)$) in order to detect whether its domain is unbalanced or not. We recall that N_i is balanced at instant *t* if $w_i^{max}(t) - w_i^{min}(t) \leq 1$. If the domain is not balanced FLD block will be executed. Otherwise, if the domain is balanced, the PIM block will be executed.

Fine Load Distribution (FLD) block

If the domain is unbalanced ($w_i^{max}(t)$ - $w_i^{min}(t)$>1) then one of the two following actions can be carried out according to the values of the four parameters evaluated in the previous block.

- *Action 1:* If processor *i* is the processor with maximum load of its domain ($w_i(t) = w_i^{max}(t)$) and all its neighbours have the same load ($w_{vi}^{max}(t) = w_{vi}^{min}(t)$), then processor *i* will distribute α units of load to its neighbours one by one. The value of α is computed as $\alpha = (w_i^{max}(t) - w_i^{min}(t) - 1)$, in order to maintain the load value of processor *i* lower-bounded by the load average of its domain. The distribution pattern coincides with the neighbours order kept in memory by processor *i* ($j_1 < j_2 < \ldots < j_r$), so processor *i* will send one load unit to processors: $j_1,\ j_2\ \ldots\ j_\alpha$. Note that the value of α will always be smaller than the number of neighbours $(\alpha < r)$, otherwise, the first stage of DASUD would perform some load movements, and this part would not start up.

- *Action 2*: If processor *i* is the processor with maximum load of its domain ($w_i(t) = w_i^{max}(t)$) but not all the processor belonging to the domain have the same load ($w_{vi}^{max}(t) \neq w_{vi}^{min}(t)$), then one unit of load is sent to one of the less loaded neighbour processors denoted by j_{min}^i, which is obtained as follows, $j_{min}^i = min\ \{j \in N_i \mid w_{ij}(t) = w_i^{min}(t)\}$

If action 1 or action 2 have produced some load movements, then the second stage of DASUD has finished. Otherwise, the next block to be executed will be the SIM block.

A preliminary proposal of DASUD, which only incorporated the two actions described above, was reported in [Luq95].

Sending Instruction Message (SIM) block

This block is related to the possibility of commanding load movements between non-directed connected processors. If the domain of processor *i* is not balanced ($w_i^{max}(t) - w_i^{min}(t) > 1$) but processor *i* is not the most loaded processor ($w_i(t) \neq w_i^{max}(t)$), then processor *i* will command one of its neighbours with maximum load, j_{max}^i, to send a load unit to one of its neighbours with minimum load, j_{min}^i. The values of j_{max}^i and j_{min}^i are obtained as follows:

$$j_{min}^i = min\ \{j \in N_i \mid w_{ij}(t) = w_i^{min}(t)\}$$
$$j_{max}^i = min\ \{j \in N_i \mid w_{ij}(t) = w_i^{max}(t)\}$$

and the message $\left(i,\ j_{min}^i,\ t,\ w_{ij_{max}^i}(t)\right)$ is sent from processor *i* to processor j_{max}^i, where *i* is the index of the processor that sends the messages, j_{min}^i is the index of the target processor, *t* is the time instant at which the message is sent and $w_{ij_{max}^i}(t)$ is the estimation load of j_{max}^i processor that processor *i* keeps in memory when the message is sent. As a consequence of this action, processor *i* commands the movement of one unit of load between two processors belonging to its domain. Note that these two processors can be non-directed connected processors. The PIM block will subsequently be executed.

Processing Instruction Messages (PIM) block

This block will always be executed after the SIM block and when no load movements have been performed by the first stage of DASUD, and the underlying domain is balanced. The block is related to the management of the instruction messages received from processors belonging to the underlying domain. Processor *i* will consider the received instruction messages which have the following content: $(j,\ j',\ t',\ w_{ji}(t'))$. All messages whose fourth component accomplishes that $w_{ji}(t') = w_i(t)$ are sorted according to the following criteria:

- Firstly, in descending order of sending time, *t'*.
- Second, in ascending order of index *j*.
- Lastly, in ascending order of target processor *j'*.

The first element of the sorted list is chosen, and processor *i* sends one load unit to the processor *j'*, crossing processor *j*.

Finally, at the end of each iteration of DASUD, processor *i* proceeds with the elimination of all received instruction messages from processors belonging to its domain. This action is always carried out independently of which previous stage has been executed.

As a consequence of applying the second stage of DASUD, some load movements can eventually be carried out to correct unbalanced situation. This amount of load is denoted by $\delta_i(t)$, and it can be equal to {0,1, α}. Notice that when no load movements are produced by the first stage of DASUD ($s_{ij}(t) = 0 \quad \forall j \in P$), the value of $\delta_i(t)$ could be a non-negative value ($\delta_i(t) \geq 0$). Otherwise, if processor *i* sends a portion of its excess load as a consequence of applying the first stage of DASUD, no extra load movements will be carried out by the second stage of DASUD ($\delta_i(t)=0$). Notice that the first stage of DASUD coincides with the SID algorithm. Then, bearing in mind that $s_{ij}(t)$ and $\delta_i(t)$ identify the amount of load sent by processor *i* to its neighbours *j* at time *t*, and denoting by $r_{ij}(t)$ the amount of load received by processor *j* from processor *i* at time *t*, the load of processor *i* at time *t+1* can be expressed by the following formula:

$$w_i(t+1)=w_i(t)-\sum_{j=1}^{n}s_{ij}(t)-\delta_i(t)+\sum_{j=1}^{n}r_{ji}(t) \qquad (1)$$

Finally, the complete behaviour of DASUD is summarised in figure 3.4 where the LBA block of DASUD is provided.

```
LBA block
evaluate $\overline{w}_i(t)$
if ($w_i(t) > \overline{w}_i(t)$)
{
    $w_i(t+1)=w_i(t)-\sum_{j=1}^{n}s_{ij}(t)-\delta_i(t)+\sum_{j=1}^{n}r_{ji}(t)$
}
```

Figure 3.4 LBA block of the DASUD algorithm in processor i

3.3 An example of DASUD execution

This section illustrates the behaviour of the DASUD load-balancing algorithm by considering a 3-dimensional hypercube with an initial load distribution denoted by the load vector: $w(0)=(4,\ 3,\ 5,\ 3,\ 2,\ 1,\ 3,\ 8)$. In figures 3.5 the numbers inside the nodes denote the load of each processor and the subindexes are the corresponding processor index.

The load movement between two processors is indicated with an arrow between them and the label of each arrow represents the amount of load to be moved. The actions carried out at each stage of DASUD, as well as the most relevant parameters evaluated for each processor during each iteration of the load balancing process, are summarised in table 3.1. Particularly, column 3 in the table illustrates the received but not processed instruction messages of each processor. The fourth and fifth columns include the value of the load average within the underlying domain and the load excess of the underlying processor, respectively. The load movements generated by the first stage of DASUD are shown in the sixth column. When no load movements are performed from processor *i* to any of its neighbours as a consequence of applying this stage, the content of the corresponding cell is $s_{ij}(1)=0$.

Otherwise, the subindexes indicate the source and destination processors of a particular load movement. The remaining columns correspond to the different blocks of the second stage of DASUD. A discontinuous line in the cell indicates that the corresponding action is skipped. The word *no* indicates that no load movements are performed by that block. In the FLD block column, the amount of load to be moved and the destination processor are indicated by *i to j*. For example, for processor 3 the expression "*1 to 2*" in the column *Action 1* of the FLD block, indicates that one load unit is moved to processor 2. Finally, the SIM block column reflects the message to be sent and the destination processor.

In this example, two iterations of DASUD are needed to achieve the final balanced load distribution. During the first iteration, processors 1 and 8 perform load movements as a consequence of executing the first stage of DASUD. Therefore, stage two of DASUD is skipped. Processors 2 to 7 do not perform load movements as a consequence of applying the first stage of DASUD ($s_{ij}(1) = 0$), stage 2 is then carried out. All these processors detect that their domains are not balanced, so the load-balancing algorithm goes on through the FLD block. Processor 3 observes that its load is the largest within its domain and all its neighbours have the same load value. Then, action 1 of the FLD block is executed by processor 3 and 1 load unit will be sent to processor 2. Processors 2, 4, 5, 6 and 7 produce no load movements when the FLD block is executed, each one then executes the SIM and PIM blocks. The instruction messages sent by each one of these processors are reported in the SIM block column of table 3.1. As the current iteration is the first one, no processor has received messages, therefore, no processor will perform any load movements by the PIM block .

At the beginning of the second iteration of the DASUD algorithm, processors 2, 3 and 8 have instruction messages to be processed. Processors 2 and 5 have enough load excess to distribute among their neighbours by applying stage 1 of DASUD then stage 2 will be skipped. Processor 1, 6 and 7 detect that their domains are unbalanced. As they have performed no load movements at the first stage of DASUD, their load-balancing process goes to stage 2. Processor 7 sends 1 load unit to processor 6 as a consequence of executing action 2 of the FLD block, the SIM and PIM blocks will then be skipped by this processor. Processors 1 and 6 each send one instruction message to processor 5 and neither of them perform load movements by the PIM block because they have not received messages from the previous iteration. Processors 3, 4 and 8 perform no load movements by the first stage of DASUD and their domains are balanced, their computations then, go on directly to the PIM block. Processor 4 has not received instruction messages so no action can be carried out by the PIM block. The received-instruction messages of processors 3 and 8 are discarded because the load value of these processors have changed since the messages have been sent. At the end of the load-balancing iteration, each processor deletes its received instruction messages independently of which previous stage has been executed.

The final maximum load difference obtained throughout the whole system is one unit of load, and so the system is balanced.

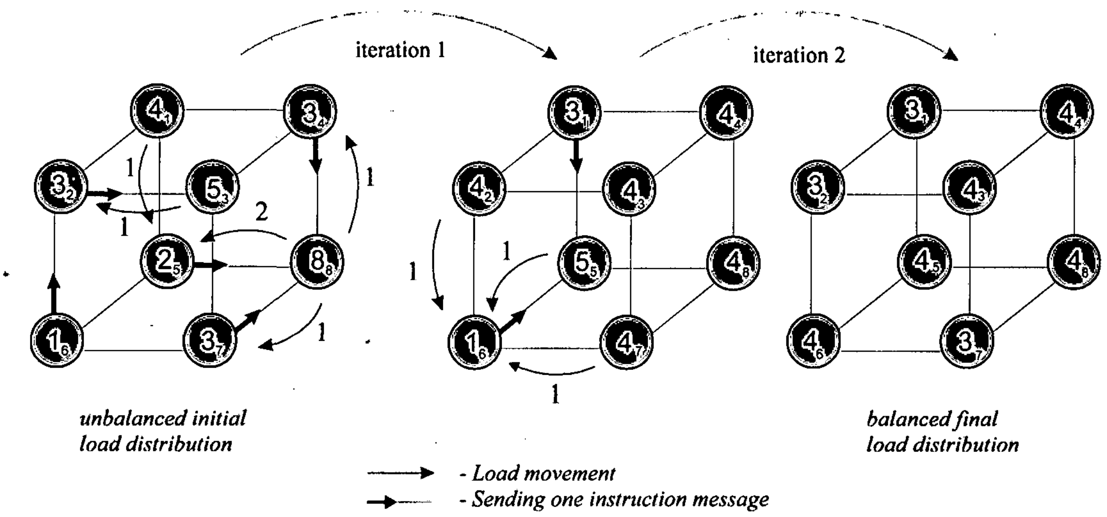

iteration 1
iteration 2
1
1
2
1
1
1
1
1
unbalanced initial load distribution
balanced final load distribution
- Load movement
- Sending one instruction message

Figure 3.5 An example of DASUD's execution

DASUD										
itert.(t)	*Proc. i*	*received instruction messages*	$\overline{w}_i(t)$	$d_{ii}(t)$	*Stage 1*	*Stage 2*				
						Unbalanced domain?	*FLD block*		*SIM block*	*PIM block*
							Action 1	*Action 2*		
1	1	no	3	-1	$s_{15}(1)=1$	---	---	---	---	---
	2	no	3.75	0.75	$s_{ij}(1)=0$	yes	no	no	(2,6,1,5) to 3	no
	3	no	3.5	-1.5	$s_{ij}(1)=0$	yes	1 to 2	no	---	---
	4	no	5	2	$s_{ij}(1)=0$	yes	no	no	(4,4,1,8) to 8	no
	5	no	3.75	1.75	$s_{ij}(1)=0$	yes	no	no	(5,6,1,8) to 8	no
	6	no	2.25	1.25	$s_{ij}(1)=0$	yes	no	no	(6,6,1,3) to 2	no
	7	no	4.25	1.25	$s_{ij}(1)=0$	yes	no	no	(7,6,1,8) to 8	no
	8	no	4	-4	$s_{84}(1)=1$ $s_{85}(1)=2$ $s_{87}(1)=1$	---	---	---	---	---
2	1	no	4	1	$s_{ij}(2)=0$	yes	no	no	(1,1,2,5) to 5	no
	2	(6,6,1,3)	3	-1	$s_{26}(2)=1$	---	---	---	---	---
	3	(2,6,1,5)	4	0	$s_{ij}(2)=0$	no	---	---	---	discard messages
	4	no	3.75	-0.25	$s_{ij}(2)=0$	no	---	---	---	no
	5	no	3.25	-1.75	$s_{56}(2)=1$	---	---	---	---	---
	6	no	3.5	2.5	$s_{ij}(2)=0$	yes	no	no	(6,6,2,5) to 5	no
	7	no	3.25	-0.75	$s_{ij}(2)=0$	yes	no	1 to 6	---	---
	8	(4,4,1,8) (5,6,1,8) (7,6,1,8)	4.25	0.25	$s_{ij}(2)=0$	no	---	---	---	discard messages

Table 3.1 Evolution of the DASUD execution for the example of figure 3.5

3.4 DASUD's complexity

Now that we have described the behaviour of a single DASUD iteration, we shall now analyse the complexity involved there in.

Before starting the DASUD execution, all processors should sort the indexes of the neighbours using the same criteria. However, the cost of this can be neglected because it is insignificant if the global computational cost is considered. One iteration of DASUD in a processor *i*, involves different operations depending on which stage or block produces load distribution among its neighbours. The pseudo-code introduced in figure 3.6 shows how the different blocks of DASUD are organised, and table 3.2 summarises the maximum number of operations that should be performed at each one (we recall that the number of neighbouring processors of processor *i* is denoted by *r*). From its overall analysis we can conclude that the computational complexity of DASUD is dominated by the complexity of the PIM block (O($r \log r$)). This means that the overall complexity of the computational period of DASUD is low.

```
{       evaluate $\overline{w}_i(t)$, $d_{ii}(t)$;
        /* FIRST STAGE */
        if ($d_{ii}(t) < 0$) evaluate $s_{ij}(t)\ \forall j \in N_i \setminus \{i\}$;
        if (no load movements are performed by the FIRST STAGE)
        /* SECOND STAGE */
        {       SUD block;
                if (unbalanced domain)
                {       /* LFD block */
                        if (($w_i(t) = w_i^{max}(t)$) && ($w_{vi}^{max}(t) = w_{vi}^{min}(t)$)) Action 1;
                        if (($w_i(t) = w_i^{max}(t)$) && ($w_{vi}^{max}(t) \neq w_{vi}^{min}(t)$)) Action 2;
                        if (no load movement performed by Action1 and Action2)
                        {       SIM block;
                                PIM block;
                        }
                } else
                        PIM block;
        }
        Deleting Instruction Messages;
}
```

Figure 3.6 Pseudo-code of one single DASUD iteration.

DASUD's operations				
Actions			***kind of operation***	***quantity***
Iteration Initial Actions		*check load estimations*	*memory accesses*	*r*
		evaluate $\overline{w}_i(t)$	*division*	*1*
			addition	*r+1*
		evaluate $d_{ii}(t)$	*subtraction*	*1*
First Stage		*evaluate* $d_{ij}(t)$	*subtraction*	*r*
		evaluate D	*addition*	*r*
		evaluate $P_{ij}(t)$	*division*	*r*
		evaluate $s_{ij}(t)$	*multiplication*	*r*
			round	*r*
Second Stage	***SUD***	*evaluate* $w_i^{max}(t)$, $w_i^{min}(t)$, $w_{vi}^{max}(t)$, $w_{vi}^{min}(t)$	*subtraction*	*2(r+1)*
		evaluate $w_i^{max}(t)$-$w_i^{min}(t)$:	*subtraction*	*1*
	LFD	*compare* $w_{ii}(t)$ *with* $w_i^{max}(t)$	*subtraction*	*1*
	SIM	*send one message*	*transmission*	*1*
	PIM	*sort a list of r elements depending on 3 parameters*	*comparisons*	*O(*$r\log r$*)*
Iteration Completion Actions		*delete r messages*		

Table 3.2. All possible operations performed by one single DASUD iteration.

In the following sections, formal aspects of DASUD are theoretically studied by firstly regarding its convergence.

3.5DASUD's convergence

In this section we will demonstrate DASUD convergence. For this purpose we assume the *partially asynchronous* assumption which was introduced by Bertseka and Tsitsiklis in [Ber89]. In that work, the authors divided asynchronous algorithms into two groups: totally asynchronous and partially asynchronous. To paraphrase them, totally asynchronous algorithms "can tolerate arbitrarily large communication and computation delays", but partially asynchronous algorithms "are not guaranteed to work unless there is an upper bound on those delays". This bound is denoted by a constant *B* called *asynchronism measure.* That assumption has been modified to be applied to the realistic load model assumed by DASUD, and to consider some particular characteristics of the same. The formal description of this assumption is provided below:

Assumption 1. (*Partially Asynchronism*) There exists a positive integer *B* such that:

a) for every $i \in P$ and for every t ∈ **N**, $\{t, \quad t+1, \quad \ldots \quad ,t+B-1\} \ \cap \ T_i \neq \varnothing$

b) for every $i \in P$, for every t ∈ **N** and for every $j \in N_i$, $t-B < \tau_{ij}(t) \leq t$

c) the load sent from processor *i* to processor *j* at time *t* is received by processor *j* before time *t+B*.

d) the instruction message sent by processor *i* to processor *j* at time *t*, is received by processor *j* before time *t+B.*

Part (a) of this assumption postulates that each processor performs a step of the load balancing process at least once during any time interval of length *B*; *part (b)* states that the load estimations kept in memory by any processor at a given time *t* were obtained at any time between *t-B* and *t*; *part (c)* postulates that load messages will not be delayed more than *B* time units as well as the instruction message sent by processor *i* to processor *j*, as has been stated in *part (d)* of this assumption.

Under assumption 1, to prove the DASUD's convergence consists of proving the following theorem.

Theorem 1. *Under assumption 1, DASUD is a finite algorithm.*

For that purpose, we introduce the following three definitions and two lemmas.

Definition 1. $m_1(t) = min \quad \{w_i(t') \mid i \in P, \quad t-3B < t' \leq t\}$ is the minimum load value among the total system at a given interval of time of length three times *B*

Definition 2. $m_k(t) = min \quad \{w_i(t') \mid i \in P, \quad t-3B < t' \leq t \quad w_i(t') > m_{k-1}(t)\}$, for any $k > 1$. The value of $m_k(t)$ is the minimum load value that occupies the *k-st* place if loads are sorted in ascending order at a given interval of time of length *3B*.

Definition 3. An agreement is: *min* $\varnothing$ = *L*+1

> **Lemma 1.** *The sequence* $(m_1(t))_{t\geq 0}$ *is non-decreasing and upper bounded. Furthermore, there exists a non-negative integer* t_1 *such that,*
>
> *a)* $m_1(t) = m_1(t_1)$ *for all* $t \geq t_1$, *and*
>
> *b)* $r_{ji}(t) = s_{ij}(t) = \delta_i(t) = 0$, $\forall j \in P$, $\forall t \geq t_1$ $\forall i \in P$ *such that* $w_i(t_1) = m_1(t_1)$.

Lemma 1 states that there is a time t_1 in which the minimum load value becomes stable (under DASUD algorithm) and all the processors with that minimum load value neither send nor receive any amount of load.

Proof. Let fix a processor $i \in P$ and a time $t \in$ **N**. We shall prove that, $w_i(t+1) \geq m_1(t)$.

If $t \notin T_i$ then processor *i* does not execute the load balancing process, so it can receive load from some neighbour processor, but it has not sent any amount of load. Thus

$$w_i(t+1) = w_i(t) + \sum_{j=1}^{n} r_{ji}(t) \geq w_i(t) \geq m_1(t)$$

If $t \in T_i$ then two different situations can be found depending on which block of DASUD produces the load movements:

case 1: No load movements are generated by the *FLD and PIM blocks* of DASUD ($\delta_i(t)=0$). Two different scenarios can be analysed:

a) Processor *i* is an underloaded processor ($d_{ii}(t)\geq 0$). Then $s_{ij}(t)=0$ for all neighbours and so that $w_i(t+1)\geq w_i(t)\geq m_1(t)$.

b) Processor *i* is an overloaded processor ($d_{ii}(t)<0$). Then,

$$w_i(t+1)=w_i(t)-\sum_{j=1}^{n}s_{ij}(t)-\delta_i(t)+\sum_{j=1}^{n}r_{ji}(t)\underset{(1)}{\geq} w_i(t)\ -\ \sum_{j=1}^{n}\left(-P_{ij}(t)d_{ii}(t)\right)+\sum_{j=1}^{n}r_{ji}(t)\underset{(2)}{=}$$

$$=w_i(t)+d_{ii}(t)+\sum_{j=1}^{n}r_{ji}(t)=\overline{w}_i(t)+\sum_{j=1}^{n}r_{ji}(t)\geq\overline{w}_i(t)$$

The inequality (1) is obtained by taking into account that: $floor\left(-P_{ij}(t)d_{ii}(t)\right)\leq -P_{ij}(t)d_{ii}(t)$. Moreover, the equality (2) is true because $\sum_{j=1}^{n}P_{ij}(t)=1$. Since $\overline{w}_i(t)$ is the load average of some processors at a given time between *t-B* and *t,* it is obvious that, $\overline{w}_i(t)\geq m_1(t)$. Hence $w_i(t+1)\geq m_1(t)$.

case 2: Some unit of load can be moved by applying FLD and PIM blocks of DASUD ($\delta_i(t)>0$). In this case processor *i* has not generated load movements by the first stage of the DASUD algorithm ($s_{ij}(t)=0$ for all neighbours *j*). Two different situations can be found:

a) Action 1 or action 2 of the FLD block of DASUD are performed. In this case $w_{ii}(t)=w_i^{max}(t)$, and $w_i(t)-w_i^{min}(t)>\delta_i(t)$. Thus

$$w_i(t+1)=w_i(t)-\delta_i(t)+\sum_{j=1}^{n}r_{ji}(t)\geq w_i(t)-\delta_i(t)>w_i^{min}(t)\geq m_1(t).$$

b) The PIM block of DASUD is executed, then processor *i* services a message such as: $\left(j,\ \ j',\ \ t',\ \ w_{ji}(t')\right)$. In this case $w_i(t)=w_{ji}(t')$,i.e., the *estimation load* that processor *j* keeps in its memory about processor *i* at the time which this message was sent, coincides with the current load value of processor *i.* Furthermore, $t-2B<t'\leq t$ because all messages sent by a processor *j* to processor *i* before

time $t-2B$, will be received by this processor before $t-B$ and deleted before *t*. The meaning of the message $\left(j,\ \ j',\ \ t',\ \ w_{ji}(t')\right)$ is the following:

$$w_{ji}(t') - w_{jj'}(t') > 1$$

so, there is a time τ, with $t-3B < t'-B < \tau \le t' \le t$, such that $w_{jj'}(t') = w_{j'}(\tau)$. Therefore, $w_i(t) - w_{j'}(\tau) > 1$ and we can conclude that $w_i(t+1) \ge w_i(t) - 1 > w_{j'}(\tau) \ge m_1(t)$.

As a conclusion, we have proved that $w_i(t+1) \ge m_1(t)$ $\forall i \in P$ and $\forall t \in \mathrm{N}$. So, $m_1(t+1) \ge m_1(t)$ $\forall t \in \mathrm{N}$.

Since $\left(m_1(t)\right)_{t\ge 0}$ is an upper bounded non-decreasing integer sequence, (*L* is an upper bound) there exists a nonnegative integer t_0 such that $m_1(t) = m_1(t_0)$ $\forall t \ge t_0$. *Then, part (a) of lemma 1 is proved for any time t that* $t \ge t_0$.

We shall now prove part (b) of lemma 1. For this purpose, we set $P_1(t) = \{i \in P \ \mid \ w_i(t) = m_1(t)\}$ as the set of processor with minimum load value of the system at time *t* and we shall see that: $P_1(t_0) \supseteq P_1(t_0+1) \supseteq P_1(t_0+2) \supseteq \ldots$, i.e., the sequence of sets of processors with minimum load is a non-increasing sequence beyond t_0.

Let $t \ge t_0$ and $i \in P \setminus P_1(t)$*. We shall see that $i \notin P_1(t+1)$, i.e., if processor *i* is not a processor with minimum load at time *t*, then it will never be a processor with minimum load value at time *t+1*.

* Note that: $P \setminus P_1(t) = \overline{P_1(t)}$

- If $t \notin T_i$, then we have seen that $w_i(t+1) \geq w_i(t) > m_1(t) = m_1(t+1)$, so $i \notin P_1(t+1)$.
- If $t \in T_i$, some different situations can be found:

  ⇒ if $\delta_i(t) > 0$ then, as we have seen before $w_i(t+1) > m_1(t) = m_1(t+1)$, so, $i \notin P_1(t+1)$.

  ⇒ if $\delta_i(t) = 0$ and processor *i* is an underloaded processor ($d_{ii}(t) \geq 0$), then it has been proved that $w_i(t+1) \geq w_i(t) > m_1(t) = m_1(t+1)$, thus $i \notin P_1(t+1)$.

  ⇒ if $\delta_i(t) = 0$ and processor *i* is an overloaded processor ($d_{ii}(t) < 0$), then we have that $w_i(t+1) \geq \overline{w}_i(t) > m_1(t) = m_1(t+1)$ and $i \notin P_1(t+1)$.

So, we can conclude that $P_1(t_0) \supseteq P_1(t_0+1) \supseteq P_1(t_0+2) \supseteq \ldots$

Since $P_1(t_0)$ is a finite set, there exists an integer $t_1 \geq t_0$ such that $P_1(t) = P_1(t_1)$ $\forall t \geq t_1$.

Note that if processor *i* has the minimum load value at time t_1 ($i \in P_1(t_1)$), then $\delta_i(t) = 0$ $\forall t \geq t_1$. Furthermore, this processor becomes an underloaded processor ($d_{ii}(t) \geq 0$ $\forall t \geq t_1$). In such situation, $s_{ij}(t) = 0, \forall j \in P, \forall t \geq t_1$. Finally, if the load of processor *i* keeps constant and it sends no load to any of its neighbours, obviously, no load can be received, i.e. $r_{ji}(t) = 0, \forall j \in P, \forall t \geq t_1$. Then, part (b) of lemma 1 is proved.

Since part (a) has been proved for any *t* such that $\forall t \geq t_0$ and $t_1 \geq t_0$, then lemma 1 is proved ■

Lemma 2. *Let us define* $P_k(t) = \{ i \in P \mid w_i(t) = m_k(t) \}$. *Then, there exists an increasing sequence* $t_1, t_2 \ldots$ *of positive integers such that,*

a) $m_k(t) = m_k(t_k)$ *for all* $t \geq t_k$, *and*

b) $r_{ji}(t) = s_{ij}(t) = \delta_i(t) = 0$, $\forall j \in P$, $\forall t \geq t_k$ *and* $\forall i \in \bigcup_{h=1}^{k} P_h(t_k)$

This lemma states that there exists a time t_k beyond which the minimum load value of order *k* becomes stable, and all the processors that belong to the sets of processors with load values equal to or less than that load value neither send nor receive any amount of load.

Proof. We prove the result by induction on *k*.

For *k* = 1 we have lemma 1.

Assume that *k* > 1 and the result is true for *k*-1.

Let fix a time $t > t_{k-1} + 3B$ and a processor $i \in P \backslash \bigcup_{h=1}^{k-1} P_h(t_{k-1})$, i.e., the load of processor *i* at time t_{k-1} is bigger than minimum load value of order *k-1*. We shall see that $w_i(t+1) \geq m_k(t)$.

If $t \notin T_i$ then processor *i* does not execute the load balancing process, so it can receive load from some neighbour processor, but it has not sent any amount of load. Thus

$$w_i(t+1) = w_i(t) + \sum_{j=1}^{n} r_{ji}(t) \geq w_i(t) \geq m_k(t).$$

If $t \in T_i$ then two different situations can be found, depending on which block of DASUD produces the load movements:

***case 1*:** No load movements are generated by the *FLD* and *PIM* blocks of DASUD ($\delta_i(t) = 0$). Three different scenarios can be analysed:

a) Processor *i* is an underloaded processor ($d_{ii}(t) \geq 0$). Then $s_{ij}(t) = 0 \quad \forall j \in P$ and so $w_i(t+1) \geq w_i(t) \geq m_k(t)$.

b) Processor *i* is an overloaded processor ($d_{ii}(t) < 0$) and there exists a neighbouring processor *j* whose load is less than or equal to the minimum load value of order *k-1* ($\exists j \in \bigcup_{h=1}^{k-1} P_h(t_{k-1})$ such that $j \in N_i$). Then the portion of excess load to be moved to this neighbour processor will be bigger than the portion of any other neighbour *j'* whose load value is bigger than the minimum load value of order *k-1* ($P_{ij'}(t) < P_{ij}(t) \quad \forall j' \in P \setminus \bigcup_{h=1}^{k-1} P_h(t_{k-1})$ and $j \in N_i$). Therefore, $s_{ij'}(t) \leq s_{ij}(t) = 0$ and then $w_i(t+1) \geq w_i(t) \geq m_k(t)$.

c) Processor *i* is an overloaded processor ($d_{ii}(t) < 0$) and there does not exist any neighbouring processor *j* whose load is less or equal to the minimum load value of order *k-1* ($\nexists j \in \bigcup_{h=1}^{k-1} P_h(t_{k-1})$ such that $j \in N_i$) then $\overline{w}_i(t) \geq m_k(t)$. Bearing in mind the proof of lemma 1, we see that: $w_i(t+1) \geq \overline{w}_i(t)$, so $w_i(t+1) \geq m_k(t)$

case 2: Some unit of load can be moved by applying FLD and PIM blocks of DASUD ($\delta_i(t) > 0$). In this case processor *i* has not generated load movements by the first stage of the DASUD algorithm ($s_{ij}(t) = 0$ for all neighbours *j*). Two different situations can be found:

a) Action 1 or action 2 of the FLD block of DASUD are performed. In this case $w_i(t) - w_i^{min}(t) > \delta_i(t)$. Therefore, $w_i(t+1) \geq w_i(t) - \delta_i(t) > w_i^{min}(t) \geq m_k(t)$. The last inequality is true because a neighbour *j* of processor *i* that has minimal load cannot belong to $\bigcup_{h=1}^{k-1} P_h(t_{k-1})$. If processor *j* belonged to this set it could not receive load from *i*, but $\delta_i(t) > 0$.

b) The PIM block of DASUD is executed, then processor *i* services a message like this: $(j, j', \tau, w_{ji}(\tau))$. Then $w_{ji}(\tau) = w_i(t)$, $\quad t - 2B < \tau \leq t$,i.e., the *estimation* *load* that processor *j* keeps in its memory about processor *i* at the time which this message was sent, coincides with the current load value of processor *i*. Then , the *estimation load* of the target processor (*j'*) that processor *j* keeps in its memory is smaller than the current load of processor *i* minus one ($w_{jj'}(\tau) < w_i(t)-1$). The target processor has a load value bigger than the minimum load value of order k-1 ($j' \in P \setminus \bigcup_{h=1}^{k-1} P_h(t_{k-1})$). Now, $w_{jj'}(\tau)=w_{j'}(\tau')$, where $t_{k-1} < t-3B < \tau - B < \tau' \leq \tau \leq t$.

Then $w_i(t+1) = w_i(t) - 1 + \sum_{j=1}^{n} r_{ji}(t) > w_{j'}(\tau') \geq m_k(t)$.

With the above steps, we have proved that:

$$w_i(t+1) \geq m_k(t) \qquad \forall\, i \in P \setminus \bigcup_{h=1}^{k-1} P_h(t_{k-1}) \qquad \forall\, t > t_{k-1} + 3B$$

Hence, the sequence of integers $(m_k(t))_{t > t_{k-1}+3B}$ is a non-decreasing integer sequence upper bounded by *L*, then there exists a time t'_k with $t'_k \geq t_{k-1} + 3B$ such that $m_k(t) = m_k(t'_k) \quad \forall\, t \geq t'_k$. We have now proved the part (a) of *lemma 2.*

Now, we shall prove part (b) of lemma 2. For this purpose, we shall prove that $P_k(t'_k) \supseteq P_k(t'_k+1) \supseteq P_k(t'_k+2) \supseteq \ldots$ i.e., the sequence of sets of processors with minimum load of order *k* is a non-increasing sequence beyond t'_k.

Let $t \geq t'_k$ and let $i \in P \setminus \bigcup_{h=1}^{k} P_h(t)$. We shall see that if processor *i* is a processor with a load value bigger than the minimum load value of order *k* at time *t*, then its load will remain lower bounded by the minimum load value of order *k* beyond *t+1* ($i \in P \setminus \bigcup_{h=1}^{k} P_h(t+1)$).

- If $t \notin T_i$, then as we have already seen before, $w_i(t+1) \geq w_i(t) > m_k(t) = m_k(t+1)$ and, therefore, $i \in P \setminus \bigcup_{h=1}^{k} P_h(t+1)$.
- If $t \in T_i$, then some different situations can be found:

⇒ If $\delta_i(t) > 0$, as we have already seen, we have that $w_i(t+1) > m_k\ (t) = m_k\ (t+1)$, and, therefore, $i \in P \setminus \bigcup_{h=1}^{k} P_h(t+1)$.

⇒ If $\delta_i(t) = 0$ and the processor *i* is an underloaded processor ($d_{ii}(t) \geq 0$) then $w_i(t+1) \geq w_i(t) > m_k\ (t) = m_k\ (t+1)$ and, therefore, $i \in P \setminus \bigcup_{h=1}^{k} P_h(t+1)$.

⇒ If $\delta_i(t) = 0$ and processor *i* is an overloaded processor ($d_{ii}(t) < 0$) and there exists a neighbouring processor *j* with load value less than or equal to the minimum load value of order *k-1* ($\exists j \in \bigcup_{h=1}^{k-1} P_h(t)$ such that $j \in N_i$) then $s_{ij}(t) \leq \sum_{\tau=t}^{t+B} r_{ij}(\tau) = 0$.

Then, as $P_{ij'}(t) < P_{ij}(t)$ $\forall j' \in P \setminus \bigcup_{h=1}^{k-1} P_h(t)$ with $j' \in N_i$, we have that $s_{ij'}(t) \leq s_{ij}(t) = 0$, thus $w_i(t+1) \geq w_i(t) > m_k\ (t) = m_k\ (t+1)$. Therefore $i \in P \setminus \bigcup_{h=1}^{k} P_h(t+1)$

⇒ If $\delta_i(t) = 0$ and processor *i* is an overloaded processor ($d_{ii}(t) < 0$) and there does not exist any neighbouring processor with load value less or equal to the minimum load value of order *k-1* ($\nexists j \in \bigcup_{h=1}^{k-1} P_h(t)$ such that $j \in N_i$) then $\overline{w}_i(t) > m_k\ (t) = m_k\ (t+1)$ and $w_i(t+1) \geq \overline{w}_i(t)$. Therefore, $i \in P \setminus \bigcup_{h=1}^{k} P_h(t+1)$.

Hence, we can conclude that $P_k(t'_k) \supseteq P_k(t'_k+1) \supseteq P_k(t'_k+2) \supseteq \dots$

Since $P_k(t'_k)$ is finite, $\exists\, t_k \geq t'_k$ such that $P_k(t) = P_k(t_k) \quad \forall t \geq t_k$.

Note that if processor i has the minimum load value of order *k* at time t_k ($i \in P_k(t_k)$) then $\delta_i(t) = 0 \quad \forall t \geq t_k$.

Moreover, if $\exists j \in \bigcup_{h=1}^{k-1} P_h(t)$ with $j \in N_i$ then $s_{ij'}(t) = 0 \quad \forall j' \in P$ and $\forall t \geq t_k$, and if $\nexists j \in \bigcup_{h=1}^{k-1} P_h(t)$ with $j \in N_i$ then $d_{ii}(t) \geq 0 \quad \forall t \geq t_k$ and, therefore, $s_{ij'}(t) = 0 \quad \forall j' \in P$ and $\forall t \geq t_k$. Hence, we also have $r_{j'i}(t) = 0 \quad \forall j' \in P$ and $\forall t \geq t_k$, part (b) of lemma 2 is proved. ■

Proof (of the Theorem). Let $t_1, t_2, \dots$ the increasing sequence of positive integers of Lemma 2.

Since *P* is finite, there is a positive integer *k* such that $\bigcup_{h=1}^{k} P_h(t_k) = P$.

Therefore, beyond time t_k the DASUD algorithm does not perform any additional load movement (a sketch of this proof was presented in [Cor98]). ■

Finally, it should be noted that this algorithm operates without the processors knowing the value of *L*. If the value of *L* varies, the algorithm is able to adapt to those changes during the global load-balancing process.

The DASUD's convergence proof provides the basis for developing a general convergence proof for realistic iterative and totally distributed load-balancing algorithms. A general model for realistic load-balancing algorithms is developed and the convergence of this realistic load-balancing model is proved. This important contribution is included in appendix A.

3.6 DASUD's convergence rate

As has already been commented, DASUD is an iterative load-balancing algorithm, therefore, after having established its convergence, the next logical path would be to determine the number of iterations required to obtain a stable state. Given the difficulty of obtaining this number in any exact way, an upper bound for the convergence rate is conjectured.

We follow this proposal by an analysis of the worst initial load distribution where all load is concentrated in one processor. Bearing in mind that *n* is the number of processors of the system and *L* is the total load, the maximum amount of load to be moved among the system should be (n-1) blocks of $\left\lceil \frac{L}{n} \right\rceil$ units of load. If one multiplies this amount of load by the maximum number of edges to cross in the worst case, one obtains

$$\left\lceil \frac{L}{n} \right\rceil * (1 + 2 + \ldots + (n-1)) = \frac{n(n-1)}{2} * \left\lceil \frac{L}{n} \right\rceil \le \frac{n(n-1)}{2n} * (L+1) = \frac{n-1}{2} * (L+1).$$

From this inequality the following conjecture is derived:

Conjecture A. *DASUD achieves the stable situation with at most* $\frac{n-1}{2} * (L+1)$ *steps.*

A more accurate upper bound can be conjectured by considering the topology diameter (*d*) and the maximum initial load difference (D_0).

Conjecture B. *DASUD achieves the stable situation with at most* $\frac{d}{2} * (D_0 + 1)$ *steps.*

As its name indicates, both proposed upper bound conjectures are valid up to the point that no counter example is found. Since no mathematical proof is provided for these conjectures, we decided to experimentally validate them by comparing the experimental results with the theoretical conjectured value. In particular, we only validate *Conjecture B* for being more precise than *Conjecture A*. This experimantal validation is provided in chapter 5 of this work.

3.7 Perfect local balance achieved by DASUD

As we have reported in the general assumptions introduced in chapter 2, a domain is considered to be unbalanced, as is the entire system, when the maximum load difference between any two processor belonging to it is bigger than one load unit. Recalling from section 3.2 that one iteration of DASUD in a processor *i* is composed by two load-balancing stages where the first stage performs a coarse load distribution of the load excess of the underlying processor, whereas the second stage produces a more accurate excess distribution in order to achieve the perfect load balance. In section 2.4.1 we have provided the description of the SID algorithm, as well the unbalanced situations that SID was not able to balance. The same examples, used to exemplify SID's problems will be used in this section to show how DASUD is able to solve them. These cases are shown together in figures 3.7.

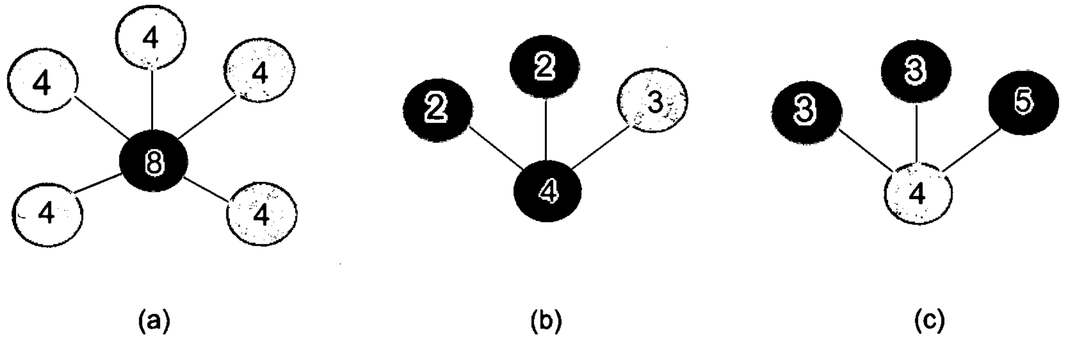


Figure 3.7 Three unbalanced load distributions achieved by the SID algorithm.

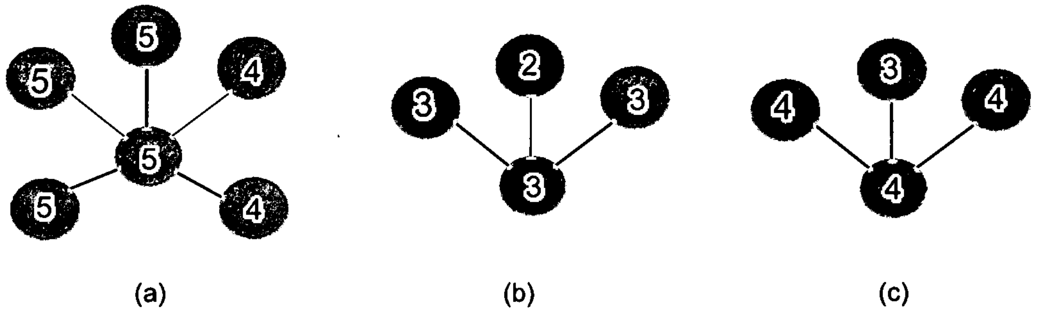


Figure 3.8 Balanced load distributions acieved by DASUD.

Let us analyse each one of these three problems individually by starting with example 3.7(a). In this case, since the local load average evaluated by the red processor is equal to 4.6, its load excess is 3.4. Since the first stage of DASUD results in no load moving, the second stage must be executed, more precisely, action 1 of the FLD block. The red processor distributes 3 load units individually amongst certain neighbouring processors. The final load distribution obtained after the DASUD iteration in the red processor is illustrated in figure 3.8(a).

Let us now consider the unbalanced situation depicted in figure 3.6(b). In this case, not all processor neighbouring the red one have the same load value, but the maximum load value corresponds to this one. As happens with the previous example, the second stage of DASUD should be executed because the first one generates no load movements, but now the action 2 of the FLD block will be executed instead of action 1. Action 2 generates the red processor sending one load unit to one of its less loaded neighbours. In particular, to the processor which is identified with the smaller index in the sorted list of the underlying processor. Therefore, the load distribution achieved in this case is the one depicted in figure 3.8(b).

Finally, the unbalanced load distribution shown in figures 3.7(c) is treated. In this case, the processor in yellow detects that its domain is not balanced because the maximum load difference within it is bigger than one load unit, but it has no excess load to move. Therefore, it commands the red processor to send 1 load unit to one of the blue processors, specifically the one whose index is the smallest. This commanding action is performed by sending an instruction message from the yellow processor to the red one. The block responsible for sending this is the SIM block. This instruction message will be received by the red processor in a posterior time instant and it will be stored to be processed when required. The DASUD's execution in the red processor goes directly to its second stage because this processor has no load excess. Since the underlying domain is detected to be balanced, the PIM block is executed and, as a consequence, one load unit is sent from the red processor to the selected blue one via de yellow processor. The final load distribution after this balancing process is depicted in figures 3.8(c).

In conclusion, we have seen that DASUD has the ability of detecting unbalanced domains and guiding the necessary load movements to achieve a local load distribution where the maximum load difference is one load unit. However, this fact does not ensure the capability of reaching an even global load distribution. The following section deals with this fact.

3.8 Global balance degree achieved by DASUD

As we have just seen, on DASUD's completing its execution, each processor's domain is balanced. Since the balance condition is locally but not globally assured, situations such as the one shown in figure 3.9 may be attained.

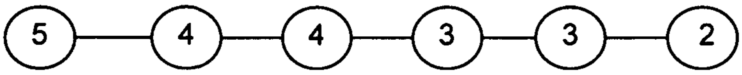


Figure 3.9 Global state unbalanced, local domain balanced

As we can observe in the figure, each domain is balanced because its maximum load difference is one load unit, but the existence of overlapped domains stems from having a global unbalance load distribution since the global maximum load difference is 3 load units. Notice that in this example, each processor observes its domain as balanced. However, each processor is not able to control the balance degree between processors outside its domain, although their domains overlap with the underlying domain. All that can be assured is that the load from the relevant processors of the 2 overlapped domains will differ at most in one load unit of the processor load common to both domains. However, this fact does not apply between the non-common processors of two overlapped domains as is observed from figure 3.9 where the maximum load difference between a pair of non-common processors belonging to two overlapped domains is sometimes 2 load units. In the worst case, this effect would be spread by the shortest path between two processors located at the maximum distance, i. e. by a path with a distance equal to the diameter of the architecture (*d*) driving the system to a final global unbalanced load distributions. We call such effect a "*platform effect*". Thus, an upper bound for the final maximum global load difference should be delivered.

Let t_f be the instant at which DASUD finishes in all processors, then the maximum load difference for any domain is upper bounded by one, which is formally denoted as the following:

$$\left|w_j(t_f)-w_k(t_f)\right| \leq 1 \quad \forall j,k \in N_i \ and \ \forall i \in P .$$

Therefore, if $i,j \in P$ and $i_o = i,\ i_1,\ i_2,\ \ldots, i_r = j$ is a minimum length way between processors *i* and *j*, then

$$\{i_o, i_1\},\ \{i_1, i_2\}, \ldots, \{i_{r-1}, i_r\} \in E$$

and as $i_k, i_{k+2} \in N_{i_{k+1}}$, we have:

$$\left|w_i(t_f)-w_j(t_f)\right| \leq \left|w_i(t_f)-w_{i_2}(t_f)\right| + \left|w_{i_2}(t_f)-w_{i_4}(t_f)\right| + \ldots + \left|w_{i_{2l}}(t_f)-w_{(2l+2)}(t_f)\right| + 1$$

where $l = floor\left(\frac{r}{2}\right)-1$. The previous inequality corresponds to the formal description of the *platform effect* described above. Therefore, if *d* is the diameter of the architecture graph *G*, then

$$\left|w_i(t_f)-w_j(t_f)\right| \leq floor\left(\frac{d}{2}\right)+1 \quad \forall\, i,j \in P ,$$

i.e., the maximum load difference for processors at the end of DASUD is upper bounded by β, which is defined as follows:

$$\beta = floor\left(\frac{d}{2}\right)+1$$

Chapter 4

Comparative study of nearest-neighbour load-balancing algorithms

Abstract

In this chapter, the proposed load-balancing algorithm DASUD is compared to the three nearest-neighbour load-balancing algorithms described in chapter 2: SID, GDE and AN. The simulation framework has been designed including different interconnection networks as hypercube and torus, as well as a wide set of system sizes which range from 8 to 128 processors and for different load distributions patterns which vary from situations which exhibit a light unbalance degree to high unbalance situations. The comparison has been carried out in terms of stability and efficiency. The stability concerns the goodness of the final stable load distribution achieved for each one of the tested algorithms and efficiency measures the cost incurred in achieving such a final situation in terms of the number of simulation steps and the amount of load movements performed during the global load-balancing process.

4.1 Simulation framework

Recalling from chapter 1 that in a totally distributed load-balancing framework, each processor in the system alternates its execution time between computational operations from the underlying application and load-balancing operations. These load-balancing operations are divided into three bocks: the LM (Load Manager) block, the LBA (Load-Balancing Algorithm) block and the MM (Migration Manager) block. This decomposition of the load-balancing process into three blocks allows to experiment in a "plug & play" fashion with different strategies at each one of the blocks. As mentioned in chapter 2, we are interested in analysing different nearest-neighbour strategies with respect to the behaviour of their LBA (Load-Balancing Algorithm) block. For that purpose, we have developed an LBA simulator, which allow us to:

- test the behaviour of different load-balancing algorithms under the same conditions;
- evaluate the behaviour of the load-balancing algorithms for different processor networks;
- evaluate the behaviour of the algorithms for different load situations;

This simulation environment hs been used to evaluate the effectiveness of the load-balancing algorithms analysed in chapter 2 (SID, GDE and AN) and 3 (DASUD).

A consideration was adopted in the load-balancing simulation process in order to simplify programming and make the results more comprehensible. We assumed that all the simulated algorithms were globally synchronised. All processors perform the while-loop introduced in section 2.3 in global *simulation steps* where no processor proceeds with the next iteration of its load-balancing process until the current one has been finished in all processors of the system. Therefore, the load-balancing simulation process is performed by consecutive *simulation steps* understanding by simulation step the execution of an iteration of the load-balancing operations in as many system processors as possible in a simultaneous manner. Although the simulation process is performed in a synchronous way for all of them, each simulated load-balancing algorithm (DASUD, SID,GDE and AN) has different synchronisation requirements. Let us describe for each algorithm what is identified as step of the load-balancing simulation process.

DASUD and SID have no synchronisation requirements between processors, therefore the definition of one *simulation step* of the load-balancing simulation process coincides for both algorithms and consists of executing the algorithm simultaneously in all processors of the system once. This definition does not apply either to GDE nor to AN because they have some particular synchronisation requirements that are not required either for SID or DASUD.

On the one hand, the GDE algorithm superimposes an order in the communication steps guided by the number of dimensions of the underlying topology. Therefore, if the dimension of the underlying topology is equal to *c* (recalling from chapter 2 that dimension and edge-coloured is assumed to be the same) then one step of the load-balancing simulation process consists of concurrently executing the algorithm in all processors that have one edge in the inspected dimension once. Notice that one *step* does not coincide with the term sweep (introduced in section 2.4.2) which corresponds to executing as many load-balancing steps as dimensions exist in the underlying system. In this case, as has already been observed (above) the all-port communication model assumed in this discussion is not fully exploited.

On the other hand, the restriction imposed by the AN algorithm lies in the impossibility of executing the load-balancing process simultaneously in processors whose domains overlap, giving rise to the creation of groups called serialisation sets described in section 2.4.3. Therefore, we identify one step of the load-balancing simulation process as the simultaneous execution of the load-balancing operations in all processors belonging to the same serialisation set.

Figure 4.1 indicates in red those processors that execute load-balancing during a given simulation step for each one of the simulated algorithms where the underlying topology is a 3x3 torus.

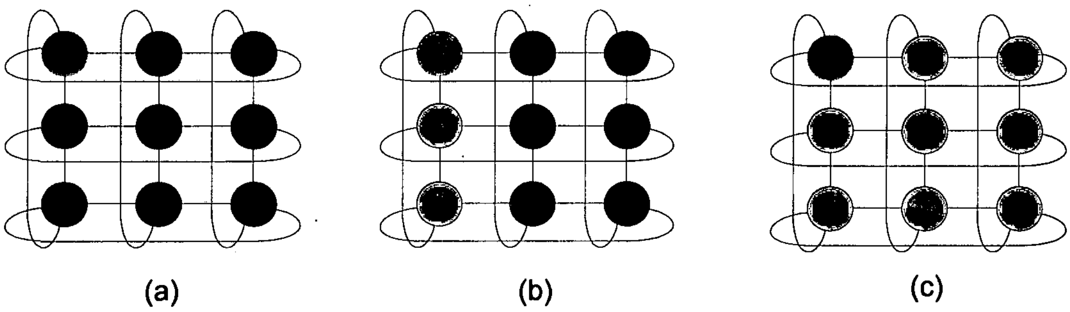


Figure 4.1 Processors that execute load-balancing simultaneously in a certain simulation step under SID and DASUD (a), GDE (b) and AN (c).

The load-balancing simulation process was run until global termination detection was accomplished. This termination condition can be a limit on the number of *simulation steps* set beforehand, or the detection that no load movements have been carried out from one step to the next, i.e., the algorithms have converged. Specifically, in our experiments, simulations were stopped when no new load movements were performed during two consecutive steps of the load-balancing simulation process. We refer to the *simulation step* at which the load-balancing simulation finishes as *last_step*. Although the simulation did not mimic the truly asynchronous behaviour of some algorithms, their results can still help us to understand the performance of the algorithms since the final load imbalances are the same whether the algorithm is implemented synchronously or asynchronously. The main difference is in the convergence speed.

Subsequently, we shall describe the complete simulation framework by firstly describing the different kinds of interconnection networks used, and following this with the set of initial load distributions applied.

4.1.1 Interconnection Networks

The load-balancing simulator has been designed to execute iterative load-balancing algorithms in arbitrary networks. In our experimental study, the following *k*-ary *n*-Cube topologies have been used: 2-ary *n*-Cube (hypercube) and *k*-ary 2-Cube (2-dimensional torus). The sizes of these communication networks were: 8, 16, 32, 64 and 128 processors. However, in order to have square *k*-ary 2-Cube, instead of 8, 32 and 128 processors, the sizes of these topologies have been changed by 9 (3x3), 36 (6x6) and 121 (11x11) , respectively.

4.1.2 Synthetic Load distributions

In our simulations, the problem size is known beforehand and all the experiments included in this chapter are performed for a fixed problem size *L* equal to 3000 load units. Therefore, the expected final load at each processor, i.e., the global load average, can be evaluated *a priori* to be $\lceil L/n \rceil$ or $\lfloor L/n \rfloor$ *n* being the size of the topology. We generated an initial set of synthetic load distributions that were used as inputs to the simulator. The set of initial load distributions were classified into two main groups: *likely distributions* and *pathological distributions. Likely distributions* cover all the situations that are assumed to appear in real scenarios where most of the processors start from an initial load that is not zero. In this case, each element $w_i(0)$ of the initial global load distribution denoted by $w(0)$, has been obtained by random generation from one of four uniform distributions patterns. These four distribution patterns cover a wide range of *likely* configurations: from highly balanced initial situations to highly unbalanced initial situations. The four patterns used in *likely distributions* were the following:

- Initial load distributions *varying 25%* from the global load average:
$$\forall i\ w_i(0) \in [L/n - 0.25 * L/n, L/n + 0.25 * L/n]$$
- Initial load distributions *varying 50%* from the global load average:
$$\forall i\ w_i(0) \in [L/n - 0.50 * L/n, L/n + 0.50 * L/n]$$
- Initial load distributions varying 75% from the global load average:
$$\forall i\ w_i(0) \in [L/n - 0.75 * L/n, L/n + 0.75 * L/n]$$
- Initial load distributions varying 100% from the global load average:
$$\forall i\ w_i(0) \in [L/n - L/n, L/n + L/n]$$

The 25% variation pattern corresponds to the situation where all processors have a similar load at the beginning, and these loads are close to the global average, i.e., the initial situation is quite balanced. On the other hand, the 100% variation pattern corresponds to the situation where the difference of load between processors at the beginning is considerable. 50% and 75% variation patterns constitute intermediate situations between the other two. For every likely distribution pattern, 10 different initial load distributions were used.

The group of *pathological distributions* was also used in order to evaluate the behaviour of the strategies under extreme initial distributions. In these distributions a significant amount of processors has a zero initial load. These scenarios seem less likely to appear in practice, but we have used them for the sake of completeness in the evaluation of the strategies. The pathological distributions were classified in four groups:

- A spiked initial load distribution, where all the load is located on a single processor: $w(0) = (L, \ 0, \ \ldots, \ 0)$, i.e., there are n-1 idle processors in the system.
- 25% of idle processors, a quarter of the processors have an initial load equal to 0.
- 50% of idle processors, half of the processors start with an initial load equal to 0.
- 75% of idle processors, a quarter of the processors have all the initial load.

In addition to the above mentioned load distributions, each one was scattered using two different shapes: a *single mountain* shape and a *chain* shape defined as follows:

- *Single Mountain (SM)*, where load values from the initial load distribution have been scattered by drawing a single mountain surface, i.e., there is a localised concentration of load around a given processor in the network. Therefore, the unbalance is concentrated and it is not easily recognisable in its real magnitude with a simple overwiew of the system (see figure 4.2(a)).

- *Chain*, where load values from the initial load distribution have been scattered by drawing multiple mountain surfaces, i. e., there are several processors that have a local concentration of load and as a consequence, a homogeneous distribution on the unbalance in the system is obtained (see figure 4.2(b)).

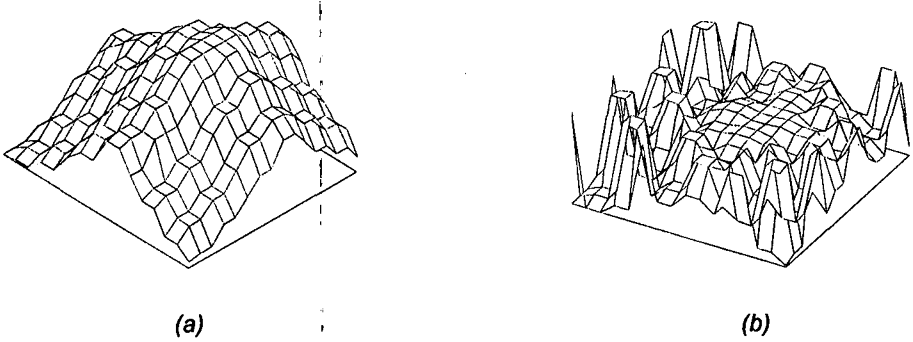


Figure 4.2. Two shapes: Single Mountain (a) and Chain (b)

As a consequence, we have evaluated not only the influence of the values of initial load distribution, but also the influence of how these values are collocated onto the processors.

To sum up, the total number of distributions tested for a given processor network was 87, which were obtained in the following way: 10 *likely distributions* * 4 patterns * 2 shapes + 3 *pathological distributions* * 2 shapes + 1 spiked *pathological distribution.*

The study outlined below is oriented to compare the simulated algorithms (DASUD, SID, GDE and AN) according to their stability and efficiency. For this purpose, in the following section, we shall introduce the quality indexes measured to perform the study, and in the subsequent sections, the stability and efficiency analysis of the simulated load-balancing algorithms are provided. Finally, the last section of this chapter provides a summary of all results and the main conclusions of the comparative study are reported.

4.2 Quality metrics

Stability measures the goodness of the final load distribution achieved by any load-balancing algorithm. Therefore, bearing in mind that we are dealing with integer load values, the final balanced state will be the one where the maximum load difference between any two processors of the topology should be zero or one depending on *L* and the number of processors. If *L* is an exact multiple of *n,* the optimal final balanced state is the one where the maximum load difference between any two processors of the system is zero. Otherwise, it should be one. In our experiments, two different indexes have been measured to evaluate the stability of the compared algorithms:

- *dif_max:* maximum load difference between the highest loaded processor and the least loaded processor throughout the whole system;
- σ : global standard load deviation.

Since efficiency reflects the cost incurred in arriving at the equilibrium state, the following two indexes were evaluated to have a measure of this cost for all strategies:

- *steps*: is the number of *simulation steps* needed to reach a final stable distribution;
- *load units* (*u*): this measures the quantity of load movements incurred in the global load-balancing process. For a given *step s* of the simulation process the maximum amount of load moved from any processor to one of its neighbours is called *max_load(s)*. According to our synchronous simulation paradigm, step *s* will not end until *max_load(s)* units of loads have been moved from the corresponding processor to its neighbour. Therefore the duration of each step depends directly on the value of *max_load(s)*. The underlying communication model is the all-port one as has been commented on section 2.2, therefore, the value of *u* might be evaluated as follows:

$$u = \sum_{s=1}^{s=last\_step} max\_load(s)$$

According to the definitions presented above, whereas the *steps* index determines the number of simulation steps needed to achieve the final load distribution without considering the duration of each one these, the index *u* is seen to be a good measure for dealing with this aspect. Furthermore, *u* is a representative index of the "time" incurred by each step of the simulation process because the global time is directly related to the amount of load that should be moved at each load-balancing step.

As has been mentioned throughout the description of our simulation framework, the global load-balancing process may be iterated until no load movements are carried out throughout the entire system or can be stopped at a predetermined step. We have selected the first of these two alternatives. Therefore, since we are evaluating the complete load-balancing process the quality indexes described above are measured at the end of the load-balancing simulation process.

Following this, the simulation results for stability and efficiency are reported starting with the first.

4.3 Stability analysis

Stability is the ability of a load-balancing algorithm to coerce any initial load distribution into an equilibrium state. As has been previously mentioned, the final balance degree achieved by each one of the compared load-balancing algorithms has been evaluated by measuring the *dif_max* and the σ indexes at the end of the load-balancing simulation process. For both indexes, the influence of the following three parameters are individually considered:

- the initial load distribution pattern (% variation from the global load average in likely initial load distributions, and % of idle processors in pathological distributions);
- the system size (number of processors);
- the shape of the initial load distribution (single mountain versus chain);

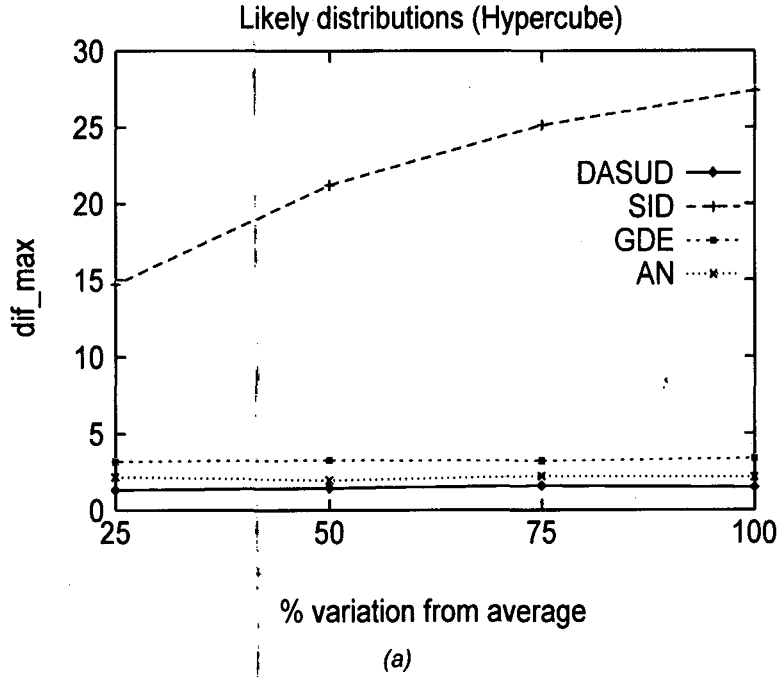
The results obtained for each one of these parameters are outlined in the following sections, and for all of them the influence of the underlying interconnection network (hypercube and torus), as well as the two groups of initial load distributions (*likely* and *pathological*) have been individually considered by showing the results in different graphics or tables. Finally, at the end of the stability analysis, the conclusions extracted from this experiment are outlined.

4.3.1 Influence of the initial load distribution pattern in *dif_max*

Figures 4.3 and 4.4 show the results obtained on average in the stability comparison in terms of *dif_max* for hypercube and torus respectively. In particular, for hypercube interconnection networks, the influence for *likely* and *pathological* initial load distributions is depicted in figure 4.3(a) and 4.3(b), and for torus topologies the influence of both load patterns is shown in figure 4.4(a) and 4.4(b), respectively.

The maximum load difference obtained by SID is always greater than the one obtained by DASUD, GDE and AN independently of the initial load distribution and the underlying topology. DASUD, GDE and AN have the quality of keeping the maximum load difference nearly constant for any load distribution pattern. Nevertheless, DASUD outperforms GDE and AN because it obtains a better final balance degree in all cases. On average, GDE obtained a maximum difference of 3.2 for torus and 3.3 for hypercubes. AN keeps bounding its maximum load difference by 2.2 and 2.5 for hypercube and torus topologies, respectively. Finally, DASUD obtained a maximum difference of 1.4 for hypercube and 1.8 for torus. An unappreciably slight increase in the maximum difference was obtained on average by these strategies for *pathological* distributions, but their relative situation is maintained.

Notice that the behaviour of all simulated algorithms is very similar whatever topology is used, and only a slight increase in the final maximum load difference is observed for torus topologies.



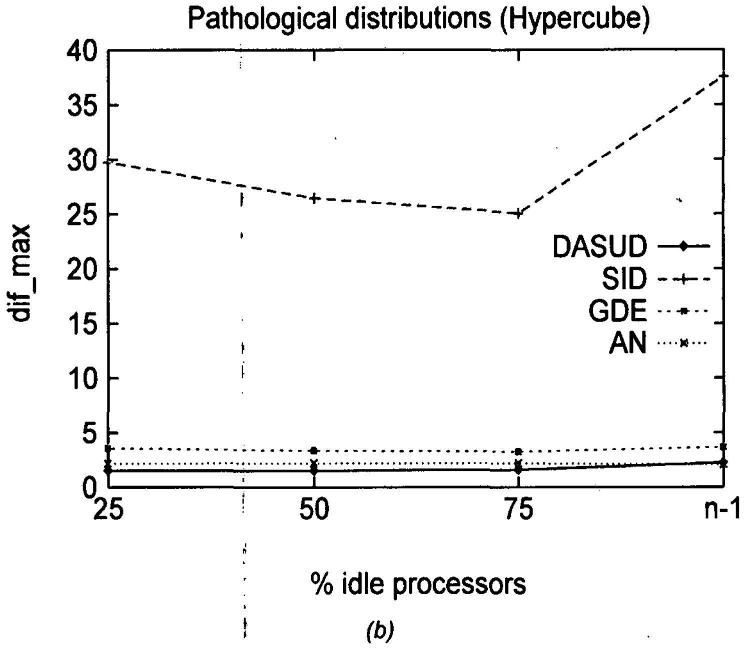
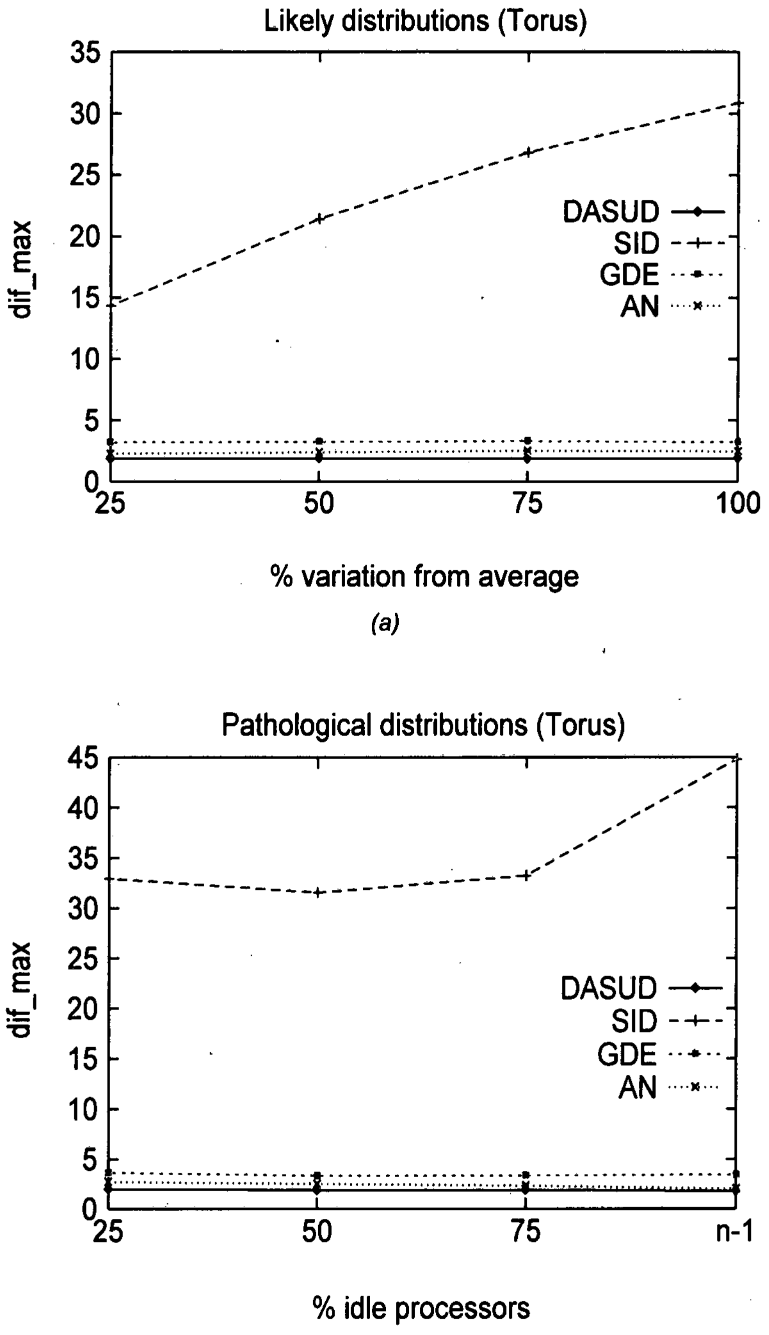


Figure 4.3 Maximum load difference for DASUD, SID, GDE and AN algorithms considering (a) likely and (b) pathological initial load distributions for hypercube topology with respect to the initial load patterns.



(a)

(b)

Figure 4.4 Maximum load difference for DASUD, SID, GDE and AN algorithms considering (a) likely and (b) pathological initial load distributions for torus topology with respect to the initial load patterns.

4.3.2 Influence of the system size in *dif_max*

Tables 4.1(a) and 4.1(b) show for hypercubes and torus topologies respectively, and for *likely* and *pathological* initial distributions, the influence of the size of the architecture on the final balance for all simulated algorithms. The values included in those tables are the mean values for all initial load distribution patterns within each group of initial load distribution. In tables B.1 and B.2 in appendix B, the individual values for each load patterns are included for hypercube and torus respectively.

From the analysis of the results shown in tables 4.1, we can extract that as the number of processors increases, the maximum difference obtained at the end likewise increases for DASUD, GDE and AN. The increment of the maximum difference observed is not very significant for the three algorithms; for instance, on average, the maximum difference was always less than 3 for DASUD, 5 for GDE and 4 for AN when the number of processors was 128 for both *likely* and *pathological* distributions and for hypercube interconnections schemes. When DASUD is considered for torus topologies, the maximum load difference obtained is 4, whereas for GDE and AN it is 5, as with the case of the biggest size (121 processors).

In contrast, the SID algorithm exhibits a different behaviour because, for large system sizes (121 or 128 processors), the maximum load difference that it is able to achieve slightly decrease instead of increasing. Since the problem size remains constant for any system size, the global load average decreases as the number of processors increases. Therefore, initial load distributions for large architecture sizes provide less unbalance distributions than initial load distributions for small system sizes. Consequently, SID does not improve its balancing ability as the number of processors increases, but it takes advantage of the more balanced distribution at the beginning of the load-balancing process.

By comparing the results obtained for both topologies, we can conclude that all strategies perform slightly better for hypercube schemes than for torus schemes. The size of the diameter that is directly related to the domain's size is the main reason explaining this phenomenon. Since the diameter of the torus is bigger than the hypercubes' diameter for the same (or similar) number of processors, a load gradient effect appears, along overlapped domains, that has more incidence in torus than in hypercubes.

Hypercube (*dif_max)*								
Number	*likely distributions*				*pathological distributions*			
of Procs.	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
8	0.4	8,76	2.32	0.8	0.74	3.75	2.5	0.25
16	1	23.73	2.65	1.27	1	18.25	3	1.5
32	1.56	31.27	3.2	1.95	1.75	43.73	3.37	2.12
64	1.98	27.01	3.74	3.25	2.25	49.37	3.87	2.62
128	2.28	18.37	4.08	3.77	2.37	32.37	4	4.12

(a)

Torus (*dif_max)*								
Number	*likely distributions*				*pathological distributions*			
of Proc.	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
9	0.85	8.1	1.77	1.07	1	6	2	1
16	1	24.01	2.52	1.4	1.25	17.5	2.25	1
36	1	34.48	2.92	2.37	1	54.87	3.25	2
64	2	29.71	4.18	3.05	2	56	4.12	2.87
121	3.05	20.01	4.87	4.25	3.75	43.12	4.5	4.12

(b)

Table 4.1 Maximum load difference for DASUD,SID, GDE and AN considering likely and pathological initial load distributions for hypercubes (a) and torus (b) attending to the architecture size.

4.3.3 Influence of the initial load distribution shape in *dif_max*

As a final consideration for stability analysis with reference to the maximum load difference within the system (*dif_max*), we have observed the results obtained according to the original shape used in the initial load distribution. For all the experiments we have always considered two different shapes for every initial load distribution: a *Single Mountain* (SM) shape and a *Chain* shape. For all topologies we have observed that the final maximum load difference depends on how the load distribution was scattered through the system. Tables 4.2(a) and 4.2(b) shows this dependency for hypercubes and torus respectively and, additionally, for both *likely* and *pathological* distributions. Each number is the mean value for all distribution patterns, which are reported in tables B.3 and B.4 in appendix B. One can observe that, on average, for the *chain* shape initial scattering, the final state obtained is slightly more balanced than the final state obtained when the initial scattering corresponds to the single mountain shape for both topologies and for all simulated algorithms. This behaviour can be explained because with the *single mountain* shape there is a high load gradient effect on the whole system. As a consequence, since the maximum local load difference that can be achieved, in the best case, is 1 load unit, the existence of a high initial load gradient favours maintaining a global load gradient at the end of the load-balancing process. This effect has a remarkable influence on the SID algorithm because it is not able to detect unbalanced domains, and in the case of scattering the load using the *single mountain* shape, all domains are initially unbalanced.

With the *chain* shape, the load is scattered onto various high-load areas surrounded by low-load areas. As a consequence, the initial load gradient effect that appears is lower than in the *single mountain* shape and, therefore, it is easier for all strategies to arrange global unbalance.

Hypercube (*dif_max by shapes*)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	1.725	32.17	3.35	2.17	1.9	38.3	3.65	2.93
Chain	1.185	17.39	3.07	2.16	1.06	16.9	3.25	2.7

(a)

Torus (*dif_max by shapes*)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	1.59	34.08	3.35	2.42	1.75	42.15	3.49	2.47
Chain	1.56	12.56	3.17	2.42	1.73	23.9	3.3	2.6

(b)

Table 4.2. Maximum load difference for DASUD,SID, GDE and AN considering likely and pathological initial load distributions for hypercubes (a) and torus (b) with respect to the shape on the initial load distribution.

4.3.4 Influence of the initial load distribution pattern in the σ

Tables 4.3(a) and 4.3(b) show the results obtained on average in the stability comparison in terms of *global standard deviation (σ)* for hypercube and torus respectively by considering the influence of *likely* and *pathological* initial load distributions in both cases.

As can be observed from the results included in both tables, all simulated load-balancing algorithms have a similar behaviour whatever initial load distribution group (*likely* or *pathological*) is applied. Only SID denotes a slight difference between the standard global load deviation achieved for likely and pathological initial load distributions. The rest of the simulated algorithms (DASUD, GDE and AN) are always able to drive the system into the same degree of balance whatever initial load distribution the load-balancing process starts from.

From the individual analysis of each algorithm, we can extract the following conclusions: the AN algorithm exhibits the best final degree of balance followed by DASUD and GDE in this order. As was expected, the balance degree achieved by the SID algorithm is the worst.

In previous sections, we saw that the maximum load difference achieved by DASUD was less than the maximum load difference achieved by AN. Therefore, we can deduce that AN has the ability of obtaining, in most of the processors, a final load value equal to the global load average, whereas DASUD has the ability of driving all processors into a final situation where their load values are very close to the global load average. However, since the maximum load difference obtained by AN is larger than the one obtained by DASUD, this means that at the end of the load-balancing process when AN is applied, there is a small number of processors whose load values differs from the global load average by a significant value.

Hypercube (*standard deviation - σ)*								
	likely distributions				*pathological distributions*			
	25%	50%	75%	100%	25%	50%	75%	n-1
DASUD	0.21	0.2	0.2	0.21	0.2	0.2	0.2	0.2
SID	3.8	5.11	5.9	6.2	6.32	5.85	5.66	7.66
GDE	0.78	0.7	0.75	0.8	0.86	0.8	0.82	0.88
AN	0.1	0.11	0.1	0.11	0.1	0.1	0.1	0.1

(a)

torus (*standard deviation - σ)*								
	likely distributions				*pathological distributions*			
	25%	50%	75%	100%	25%	50%	75%	n-1
DASUD	0.3	0.31	0.31	0.31	0.345	0.34	0.34	0.35
SID	3.89	5.49	6.63	7.56	8.43	8.82	8.48	10.89
GDE	0.81	0.8	0.83	0.85	0.87	0.86	0.86	0.88
AN	0.14	0.14	0.14	0.14	0.17	0.17	0.17	0.17

(b)

Table 4.3 Standard deviation for DASUD, SID, GDE and AN considering likely and pathological initial load distributions for hypercube (a) and torus (b), for all system sizes and for all shapes with respect to the initial distribution patterns.

4.3.5 Influence of the system size in the σ

Tables 4.4(a) and 4.4(b) show the global standard load deviation obtained by all the strategies with respect to the system size. The values included in each table are the mean values for all initial load distribution patterns. The separate values are included in tables B.5 and B.6 from appendix B.

As can be seen, DASUD, GDE and AN achieve a deviation that is very low, for all topologies and distributions. In particular it is less than 1.5 for all cases. In contrast, SID exhibits a higher standard deviation for all cases. SID obtains, on average, more than 10 times the deviation obtained by the other three algorithms DASUD, GDE and AN. However, all strategies have a common behaviour as the system size increases. The balance degree obtained for all load-balancing algorithms worsens as the number of processors grows. Such a characteristic reflects the fact that totally distributed load-balancing algorithms are affected by architecture size and, in particular, by the diameter of the topology.

We also observe that there is a slight difference between the results obtained for each topology. The final balance degree attained in torus is worse than the balance degree obtained in hypercube topology. The reason for this difference is the diameter of each topology, since for the same diameter the final standard deviation reached is very similar whatever interconnection network is used. For example, the 4-dimensional hypercube and the 4x4 torus have the same diameter, which is equal to 4, and their final standard deviations coincide.

Hypercube (*standard load deviation* - σ)								
Number of Proc.	*likely distributions*				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
8	0.03	2.6	0.6	0.02	0.0	1.2	0.7	0.0
16	0.5	5.2	0.7	0.21	0.5	4.7	0.8	0.21
32	0.0	5.8	0.8	0.0	0.0	8.4	0.8	0.0
64	0.01	6.3	0.9	0.01	0.01	8.5	1	0.01
128	0.5	6.4	0.9	0.3	0.5	9.3	1	0.3

(a)

Torus (*standard load deviation* - σ)								
Number of Proc.	*likely distributions*				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
9	0.35	2.5	0.5	0.15	0.47	2.1	0.5	0.18
16	0.5	5.7	0.6	0.2	0.5	4.4	0.6	0.21
36	0.01	5.8	0.8	0.01	0.01	11.3	0.8	0.01
64	0.01	7.4	1	0.02	0.01	13.6	1	0.01
121	0.72	8.2	1.2	0.33	0.75	14.4	1.5	0.43

(b)

Table 4.4 Global standard deviation for DASUD, SID, GDE and AN considering likely and pathological initial load distributions for hypercubes (a) and torus (b) with respect to the architecture size.

As a final consideration, we observe that there is not such a significant difference between the final balance degree achieved for the two groups of initial load distributions (*likely* and *pathological*). This means that the final balance degree obtained by these load-balancing algorithms does not depend so much on the initial load distribution.

4.3.6 Influence of the initial load distribution shape in the σ

The last parameter evaluated concerning stability analysis is how the final balance degree can be affected by the initial load distribution shape. The values depicted in tables 4.5(a) and 4.5(b) are the mean values of the final balance standard deviation for all initial load patterns in the corresponding initial load distribution group (*likely* or *pathological*). The individual values for each load pattern are included in tables B.7 and B.8 in appendix B.

As happens in the analysis performed for the maximum load difference (*dif_max*), the global standard deviation is very similar whatever load scattering is applied in the case of DASUD, GDE and AN. Only for SID is a significant increment observed when the initial load distribution shape is Single Mountain. This fact confirms that the presence of a high unbalance gradient throughout the whole system favours the existence of final balanced domains, but favours unbalanced when were compared to its overlapped domains.

4.3.7 Conclusions of the stability analysis

In this section, we summarise the main conclusion extracted from the stability analysis outlined previously. In table 4.7 these conclusions are exposed taking into account each one of the analysed parameters. Notice that, in general, DASUD, GDE and AN exhibit a common behaviour when the ability of driving the system into a stable state as close to the even load distribution is considered. DASUD and AN obtain the best results for all analysed parameter, therefore, to emphasise this fact, the name of both algorithms is wrote in red in table 4.7.

Hypercube (*σ by shapes*)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	0.2	7.43	0.76	0.1	0.2	8.06	0.84	0.14
Chain	0.21	3.07	0.75	0.07	0.2	3.7	0.82	0.1

(a)

Torus (*σ by shapes*)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	0.32	8.45	0.82	0.14	0.345	10.74	0.87	0.17
Chain	0.32	3.33	0.82	0.14	0.34	6.45	0.86	0.16

(b)

Table 4.5. Global standard deviation for DASUD, SID, GDE and AN considering likely and pathological initial load distributions for hypercubes (a) and torus (b) with respect to the shape on the initial load distribution.

Stability Summary				
	dif_max		***standard deviation (σ)***	
patterns	**DASUD**	*remains low and invariant for any topology, any initial load group and any initial load distribution pattern.*	**DASUD**	*remains low and invariant for any topology, any initial load group and any initial load distribution pattern.*
	GDE		GDE	
	AN		**AN**	
	SID	*Very high, and depends on the initial load distribution group*	SID	*Very high, and depends on the initial load distribution group*
system size	**DASUD**	*remains low and increases as the system size increases, but not greatly. It is higher for torus than for hypercube topologies.*	**DASUD**	*remains low and increases as the system size increases, but not greatly. It is higher for torus than for hypercube topologies*
	GDE		GDE	
	AN		**AN**	
	SID	*Very high and increases as the system size also increases.*	SID	*Very high and increases as the system size also increases.*
shape	**DASUD**	*remains low and is slightly larger for the SM shape than for the Chain shape*	**DASUD**	*remains low and is slightly larger for the SM shape than for the Chain shape.*
	GDE		GDE	
	AN		**AN**	
	SID	*Very high and is larger for the SM shape than for the Chain*	SID	*High and is larger for the SM shape than for the Chain shape*

Table 4.6. Summary of the results from the comparative study with respect to the stability analysis

4.4 Efficiency analysis

In this section, the comparison is focused on evaluating the costs incurred by the load-balancing process of all simulated algorithms in terms of the number of load-balancing *simulation steps* needed to reach the stable state and the quantity of load units moved throughout the system during the complete load-balancing process (*u's*). We recall from section 4.1 that one *simulation step* is defined as the execution of the load-balancing operations in as many processors as is possible to do so simultaneously, and that *u* (*load units*) measures the maximum amount of load moved at each *simulation step* for all load-balancing process. For the sake of simplicity, in the rest of this section we refer to *simulation steps* as *steps*.

The order of efficiency result exposure follows the same scheme as that followed for stability analysis. Both the efficiency indexes *load units* (*u*) and *steps* have been analysed by considering the influence of the initial load distribution pattern, the system size and the shape of the initial load distribution. As happens in the stability case, all studies have treated the underlying topology (hypercube and torus), independently as well as the initial load distribution group (*likely* and *pathological*). Finally, the conclusions of the efficiency analysis are reported.

4.4.1 Influence of the initial load distribution pattern in *u*

Table 4.8 summarises the amount of load moved throughout the system for all simulated algorithms by taking into account the initial load distribution pattern. Each number is the mean value obtained for all system sizes. It can be observed that all strategies exhibit a similar behaviour for both initial load distribution groups (*likely* and *pathological*). As the initial unbalance degree increases, the amount of load movements required to achieve the final load distribution increases as well. However, the amount of load moved for each individual algorithm presents some important differences. The two algorithms that generated less effort in arranging the initial unbalance in terms of *u*'s are the SID and the DASUD algorithm. Both of these behave, on average, in a similar way, independently of the underlying topology. AN has a common behaviour for both topologies but exhibits a significant increment in the load moved throughout the system compared to SID and DASUD. Finally, GDE clearly depends on the underlying topology, obtaining worst results for torus interconnection networks than for hypercubes. The main reason for these differences is the degree of concurrency allowed by each algorithm in the execution of the load-balancing operations among all system processors (simultaneous running processors). Since SID and DASUD maximally exploit this capacity, they are able to overlap more load movements than their counterparts and, by contrast, GDE and AN restrict this concurrence degree to a subset of processors within the system, and thus the total load movements are propagated throughout more time. Notice, however, that GDE hardly depends on the underlying topology. This algorithm does not take advantage of the all-port communication model because at each step a given processor can only perform load movements between itself and one immediate neighbour. However, when it is executed in a hypercube topology, since for all inspected dimensions all processor have a link with it, all processors execute the load-balancing process simultaneously. This fact does not apply to torus topologies because the number of links for each processor does not coincide with the number of dimensions (colours in this case). For that reason, the total load units moved in the entire load-balancing process exhibits a considerable increment.

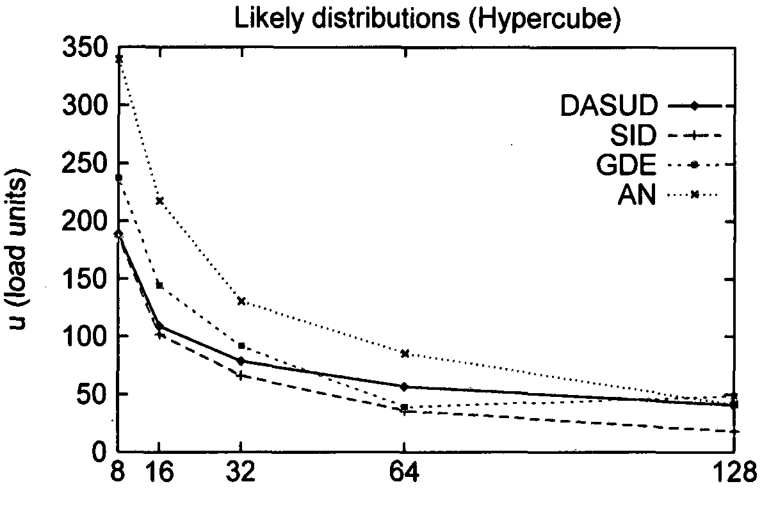
Hypercube (*load units – u's*)								
	likely distributions				*pathological distributions*			
	25%	50%	75%	100%	25%	50%	75%	n-1
DASUD	38.62	75.75	108.17	155.64	117	180.2	301.7	781.4
SID	29.41	61.92	93.66	140.47	99.7	160.9	284.4	753.8
GDE	45.1	91.26	82.49	183.92	263.12	389.3	444.8	1222.2
AN	65.16	129.24	180.42	264.18	289.9	425.8	517.5	1261.4
Torus (*load units – u's*)								
DASUD	37.53	75.88	121.42	139.77	118.8	169.1	312.4	1041.2
SID	27.11	61.3	41.21	123.68	93.8	142.8	278.4	1012
GDE	168.9	247.5	337.2	459.9	365.76	750.9	1567.0	2957.6
AN	53.74	98.65	129.37	188.14	239.3	370.2	494.3	1152.8

Table 4.8 Load units moved (u's) for DASUD, SID, GDE and AN algorithms considering likely and pathological initial load distributions for hypercube and torus topology with respect to the initial load distribution patterns

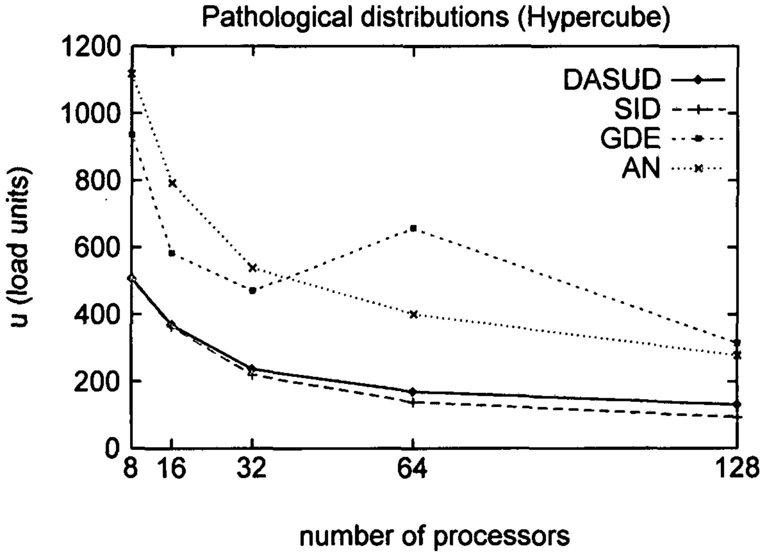
4.4.2 Influence of the system size in *u*

Figures 4.5 and 4.6 give more detailed information about the movements of *load units (u)* for hypercube and torus topologies, respectively. From their analysis we can extract the following observations. First, DASUD and SID result in being independent of the underlying topology because, starting from the same load distribution, either *likely* or *pathological,* both strategies generate similar results for hypercube and for torus. Secondly, the total amount of load moved during the load-balancing process by DASUD was slightly higher than the quantity moved by SID for any initial load distribution (*likely* or *pathological*) and for any topology (*hypercube* or *torus*). In contrast, GDE and AN obtain, in general, worse results. The amount of load moved during the LB process is, on average, twice that moved by DASUD and SID for likely initial load distributions and, furthermore, when we consider pathological patterns, this difference could be more than three times.

It is worth observing that GDE exhibits a topology dependence that is detected when a more detailed analysis of its behaviour is performed. This dependence is easily detected if the relative situation with the AN's result is observed. Whereas the GDE algorithm has better results, on average, than AN for hypercube topologies, for torus AN is faster than GDE, i.e., the movement measure is smaller for AN than for GDE. But notice that, since AN does not significantly vary its behaviour whatever topology is analysed (the *u*'s are nearly the same for torus of hypercube), GDE stems from an underlying topology dependence (the *u*'s for torus are twice the *u*'s for hypercube). One important reason for this fact is that GDE is based on DE algorithm as has been described in chapter 2, which was originally developed for hypercube interconnection networks. Under this topology, at each step of the simulation process, all processors execute load-balancing operations because they all have an edge in all dimensions of the hypercube. Therefore, in spite of not exploiting the all-port communication model, for hypercubes the maximum concurrence in load-balancing operations is exploited. However, for the torus topologies, GDE represents a considerable loss in efficiency. This is due to the fact that in each simulation step, not all processors in the system will be executing load-balancing operations, since in this case the number of dimensions is obtained by minimally colouring the graph associated with the topology.

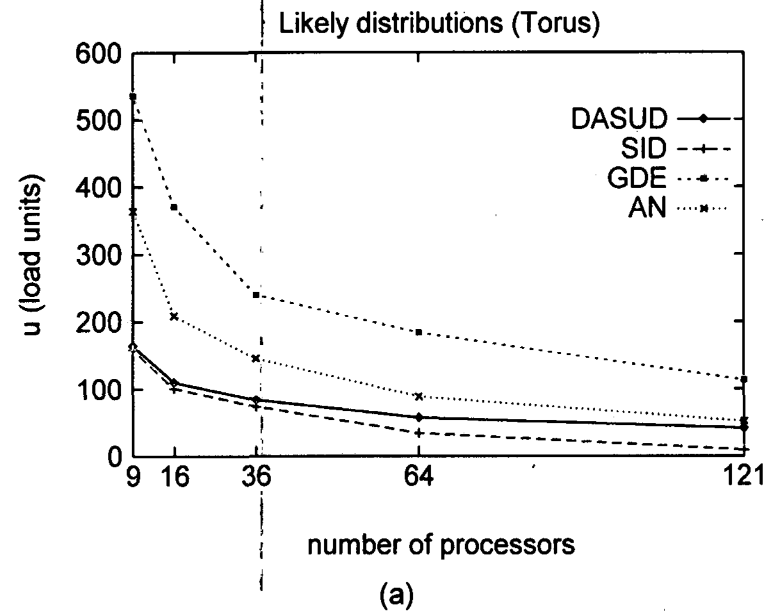


(a)



(b)

Figure 4.5 Efficiency in terms of movement measure (u's) for hypercube topologies, and considering (a) likely and (b) pathological initial load distributions.



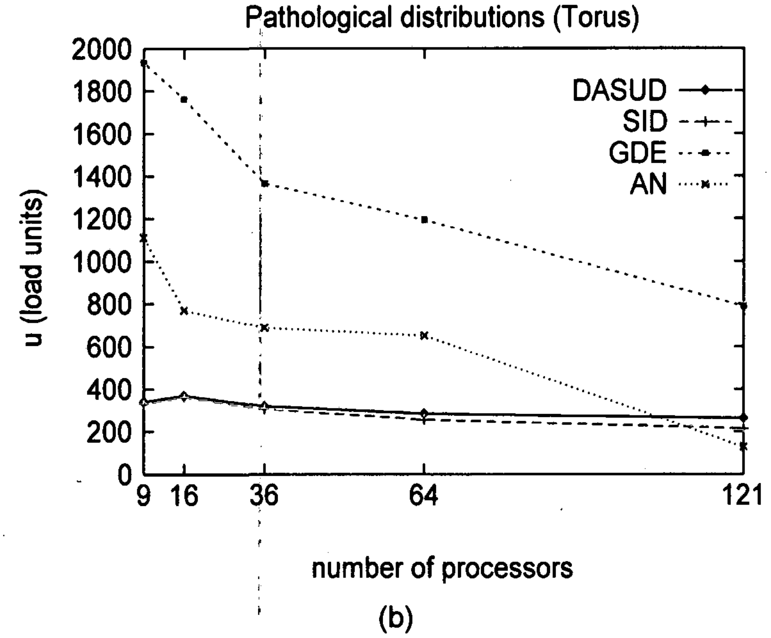


Figure 4.6 Efficiency in terms of movement measure (u's) for torus topologies and considering (a) likely and (b) pathological initial load distributions.

4.4.3 Influence of the initial load distribution shape in *u*

The effect of how the initial load distribution is scattered through the processors has also been investigated in order to observe whether it has any influence on the evaluated indexes, or not, for all algorithms. In table 4.9, we show the amount of load moved on average by each algorithm to obtain a stable state depending on how the initial load distribution is scattered to processors. Each value is the mean value for all sizes of hypercubes and torus, and for all initial load patterns. The separate values are included in tables A.5 and A.6 in appendix A.

From the analysis of tables 4.9(a) and 4.9(b) we extract that the value of *u* for *single mountain* scattering is larger than that obtained when *chain* scattering is applied for all strategies when the initial load distribution group is the *pathological* group. This is attributable to the kind of load movements generated. When *single mountain* scattering is applied all local load movements have the same global direction, from heavily loaded processors to lightly loaded ones, because this kind of load scattering generates a local load gradient equal to the global one. Consequently, all local load movements are productive movements. Furthermore, since in *single mountain* shape all load is evenly concentrated in the same region of the system keeping the rest of processors idles, load redistribution becomes difficult. Non-idle processor belonging to the limit of the loaded region are continuously receiving load from processors within the loaded area and sending it to the idle region, this fact implies an increment in the load moved around the system. On the other hand, when *chain* scattering is used, idle processors are surrounded by loaded processors and, consequently the load distribution is faster and not so tedious.

When we analyse the values obtained for *likely* initial load distributions only GDE and AN behave in the same way as for *pathological* cases. DASUD and SID change their behaviour by providing more load movement for *chain* shape than for *single mountain* shape. In this case, both algorithms are penalised for their capacity of having all processors concurrently executed load-balancing operations. The following scenario appears: there are some processors that can see themselves as locally load-maximum while not being globally-maximum and, therefore, some load thrashing is generated. GDE and AN do not exhibit this problem, because only a few processors are simultaneously executing the load-balancing process, and the thrashing effect is avoided.

Hypercube (load units- u's - by shapes)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	88.34	69.94	110.5	192.21	344.05	320.45	629.07	705.35
Chain	100.75	92.79	90.87	127.29	201	187.33	300.1	302.13

(a)

Torus (load units - u's - by shapes)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	88.46	69.48	332.3	118.63	420.45	385.75	1482.6	580.6
Chain	92.99	84.1	274.5	116.32	140	346	798.13	346

(b)

Table 4.9 Load units moved (u's) for DASUD, SID, GDE and AN considering likely and pathological initial load distributions for hypercube (a) and torus (b) topologies with respect to the shape on the initial load distribution.

4.4.4 Influence of the initial load distribution pattern in *steps*

Finally, in the following sections we deal with the number of simulation steps needed to achieve the final stable load distribution. In this section, we analyse the influence of the initial load distribution pattern on the number of total steps performed during the load-balancing simulation process. As happens for the *load units*, the number of steps incurred by all simulated algorithms increases as the initial load imbalance increases, whatever initial load distribution group is observed. From an individual analysis of each load-balancing algorithm, the algorithm that incurs in the least number of steps is the SID algorithm. DASUD occupies the second place in the ranking followed by AN and, finally by GDE. However, all strategies need more steps to reach the final load distribution when they are executed in a torus topology than in a hypercube topology. The highest connectivity degree exhibited by hypercube interconnection networks is the reason for such difference. GDE has obtained the worst results because it does not exploit the all-port communication model, and it takes more steps to be aware of load changes at each domain.

Hypercube (*steps*)								
	likely distributions				*pathological distributions*			
	25%	50%	75%	100%	25%	50%	75%	n-1
DASUD	9.56	13.47	15.26	16.78	15.7	17.2	17	24.2
SID	5.35	7.51	9.03	9.79	8.2	9.9	9.5	12
GDE	34.11	34.73	35.9	45.11	44.5	47	38.4	51.5
AN	29.56	33.6	35.6	38.64	38.8	39.6	38	51.2

(a)

torus (*steps*)								
	likely distributions				*pathological distributions*			
	25%	50%	75%	100%	25%	50%	75%	n-1
DASUD	22.5	28.5	33.02	38.16	38.8	47.8	48.2	68
SID	5.25	9.09	11.25	14.11	11.6	19.1	22.4	38.6
GDE	47.76	49.24	50.9	52.9	66.1	67.8	69.5	70.4
AN	22.43	25.76	28.18	30.28	32.1	35.9	33.9	42.4

(b)

Table 4.10 Steps for DASUD, SID, GDE and AN algorithms considering likely and pathological initial load distributions for hypercube (a) and torus (b) topologies with respect to the initial load distribution patterns

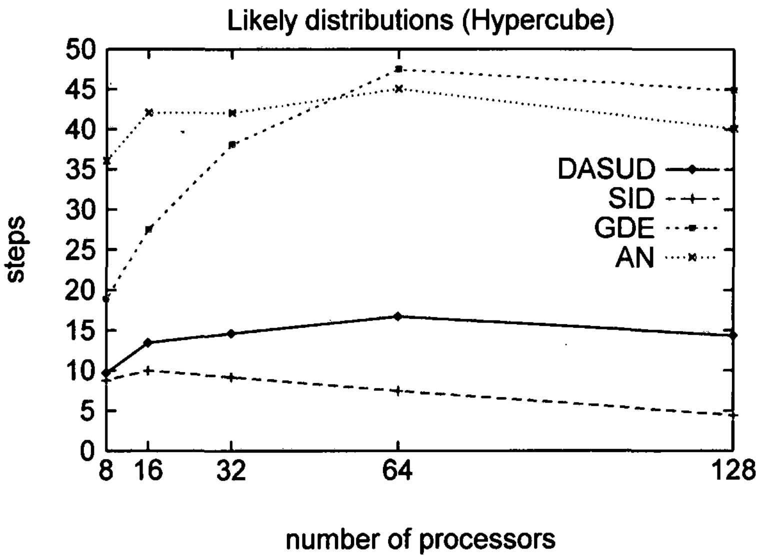
4.4.5 Influence of the system size in *steps*

Figures 4.7 and 4.8 show the number of simulation *steps* needed for the whole simulation process on average for hypercubes and torus, respectively, *for* DASUD, SID, GDE and AN strategies. The individual values corresponding to each initial load distribution patterns are included in Appendix B.

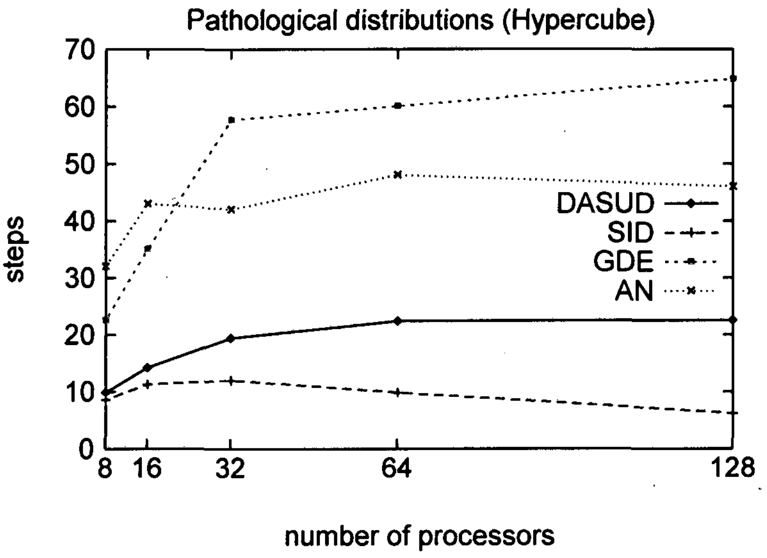
SID is the strategy that needs the smallest number of *steps* for all topologies and the smallest number of processors to stop the load balancing-process. This fact stems from the incapacity of SID to evenly attain load distributions at the end of the load-balancing process as it has been extracted from the stability analysis. For that reason, the low number of SID steps is not an important point in this analysis. However, that is not the case for the other simulated strategies. We continue by analysing the results obtained for each one in further detail.

We begin with the GDE load-balancing algorithm. If we focus globally on GDE behaviour for hypercubes and torus, we will see that, whilst for the hypercubes the number of simulations steps increases as the system size also increases, in the case of the torus, the number of simulation steps remains fairly constant. The main reason for this fact is the connectivity degree of a given processor, i.e., the number of processors directly connected to it. In hypercube topologies, the size of the domain of a given processor increases as the system size increases, whereas, for torus the domain size remains constant as the number of processors grows. In particular, for hypercube topologies the size of the domain coincides with the diameter of the system, whereas the number of immediate neighbours for a given processor in torus interconnection networks is always 4.

We shall now analyse the results obtained by the AN algorithm. In this case, we can also highlight a distinct behaviour for the topologies analysed. When the load-balancing process is performed in hypercube topologies, the number of steps spent in the global load-balancing process exhibits a homogeneous behaviour, whatever the system size might be. By contrast, when the underlying topology is torus, a more irregular behaviour is obtained. The degree of connectivity exhibited by each topology has an important relevance in these results. Since the AN strategy has the constraint of not allowing the simultaneous execution of the load-balancing algorithm in processors whose domains are overlapped, large domain sizes stem from being less overlapped in load movements because fewer concurrent load-balancing operations may be performed in the system. Therefore, the number of *steps* increases.

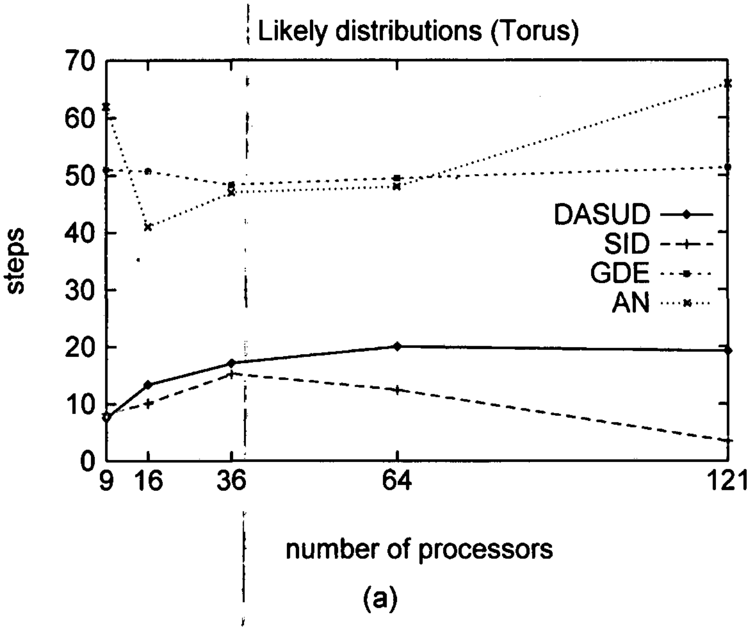


(a)



(b)

Figure 4.7 Steps for DASUD, SID, GDE and AN algorithms considering (a) likely and (b) pathological initial load distributions for hypercube topology with respect to the system size.



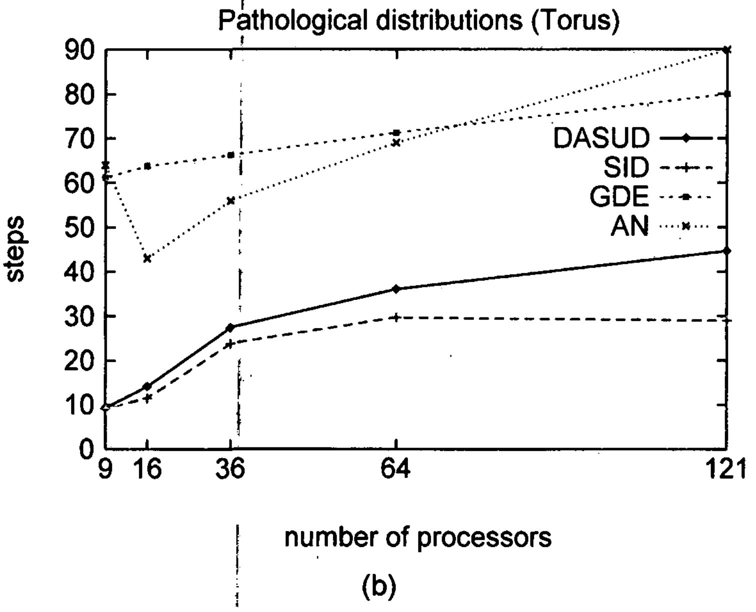


Figure 4.8 Steps for DASUD, SID, GDE and AN algorithms considering (a) likely and (b) pathological initial load distributions for torus topology with respect to the system size.

4.4.6 Influence of the initial load distribution shape in *steps*

We have also investigated the influence of the initial load scattering on the number of steps needed by the load-balancing process to reach the termination condition. Tables 4.11(a) and 4.11(b) show the average of such number of steps for all sizes of hypercubes and torus, respectively. More detailed information can be found in tables B.15 and B.16 in appendix B.

For single mountain shapes the number of steps is higher than for *chain* shapes than for *single mountain* shape This characteristic is independent of the initial unbalanced degree and of the initial load distribution group (*likely* or *pathological*). For all *likely* distributions, the number of steps required *for single mountain* shapes is approximately twice, on average, the number of steps required for *chain* distributions. This feature has a high similarity to the behaviour exhibited by the total amount of load moved when the load distribution shape is considered (section 4.4.3). The local unbalance gradient observed by each processor coincides with the global one, therefore, a number of productive steps are performed to coerce the imbalance into a load distribution as close to the even one as possible.

HypercubeTopology (steps by shapes)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	17.79	9.22	40.64	36.56	21.6	11.55	48.19	44.8
Chain	9.74	6.62	34.28	32.14	12.53	7	39.55	34.9

(a)

Torus Topology (steps by shapes)								
	likely distributions				*pathological distributions*			
	DASUD	SID	GDE	AN	DASUD	SID	GDE	AN
SM	39.6	13.34	52.71	28.55	58.05	27.5	69.57	37.37
Chain	21.49	6.5	47.72	24.77	35.13	11.6	66.32	32.26

(b)

Table 4.11 Steps for DASUD, SID, GDE and AN considering likely and pathological initial load distributions for (a) hypercube and (b) torus topologies with respect to the initial load distribution shape

	Efficiency Summary			
	load units (u's)		simulation steps	
patterns	DASUD	increases as the initial unbalance degree also increases	DASUD	increases as the initial unbalance degree increases
	GDE		GDE	
	AN		AN	
	SID		SID	
system size	DASUD	decreases as the system size increases	DASUD	increases as the system size increases but not greatly
	GDE		GDE	
	AN		AN	
	SID		SID	
shape	DASUD	likely: larger for Chain than for SM shape pathological: larger for SM than for Chain.	DASUD	larger for SM shape than for Chain shape
	SID		GDE	
	AN	larger for SM shape than for Chain shape	AN	
	GDE		SID	

Table 4.12. Summary of the results from the comparative study with respect to the stability analysis

4.4.7 Conclusions of the efficiency analysis

Finally, in this section, we summarise the main conclusion extracted from the efficiency analysis. In table 4.12 these conclusions are exposed by taking into account each one of the analysed parameters. For each parameter (pattern, system size and shape) the two algorithms that provides the best results are written in red. Notice that for all parameters and quality indexes the load-balancing algorithm that requires less time to reach the final load distribution is the SID algorithm. However, DASUD's behaviour remains very close to SID for the whole experimentation.

It is worth noting that the best efficiency results obtained by the SID algorithm denote its incapacity to coerce the system into a balance load distribution. The load-balancing process applying the SID algorithm lasts a short time period because, although it allows all processors to work simultaneously, the performed load movements slightly arrange the system imbalance. In contrast, DASUD incurs within a similar load-balancing time, but performs more productive load-balancing movements. AN and GDE both are penalised for their synchronisation requirements. This restriction forces them to execute more load-balancing steps and load movements cannot be overlapped, as happens when SID or DASUD algorithms are applied. This drawback implies that AN and GDE take a large time period in achieving the final load distribution.

4.5 Summary and conclusions of the comparative study

Table 4.13 includes a summary of all experimental studies performed in this chapter. Since the detailed analysis of the stability and efficiency results have already been carried out in the corresponding sections, in table 4.13 we use very simple terms to denote the goodness of each algorithm with respect to each study. A trade-off column is also included to expose the main conclusions of this chapter. On the one hand, with respect to the stability analysis, DASUD, AN and GDE obtain the best results. In particular, DASUD is the best. On the other hand, by considering the efficiency analysis, SID and DASUD are the two algorithms that arrive first at the end of the load-balancing process. Therefore, DASUD is the load-balancing algorithm that exhibits the best trade-off between the final balance degree and the cost incurred in achieving it.

Summary of the comparative study			
	Stability	***Efficiency***	***Trade-off***
DASUD	*Very Good*	*Low Cost*	*The best trade-off between stability and efficiency*
AN	*Very Good*	*High Cost*	*Medium trade-off*
GDE	*Good*	*High Cost*	*Bad trade-off*
SID	*Very Bad*	*Low Cost*	*Bad trade-off*

Table 4.13. Summary of the results from the coparative study with respect to the stability analysis

Chapter 5

Scalability of DASUD

Abstract

In this chapter, using the same simulation framework reported in chapter 4, we analyse the scalability of DASUD with respect to problem size as well as system size.

5.1 Introduction

An important aspect of performance analysis in load-balancing algorithms is the study of how well the algorithm adapts to changes in parameters such as problem size and system size.

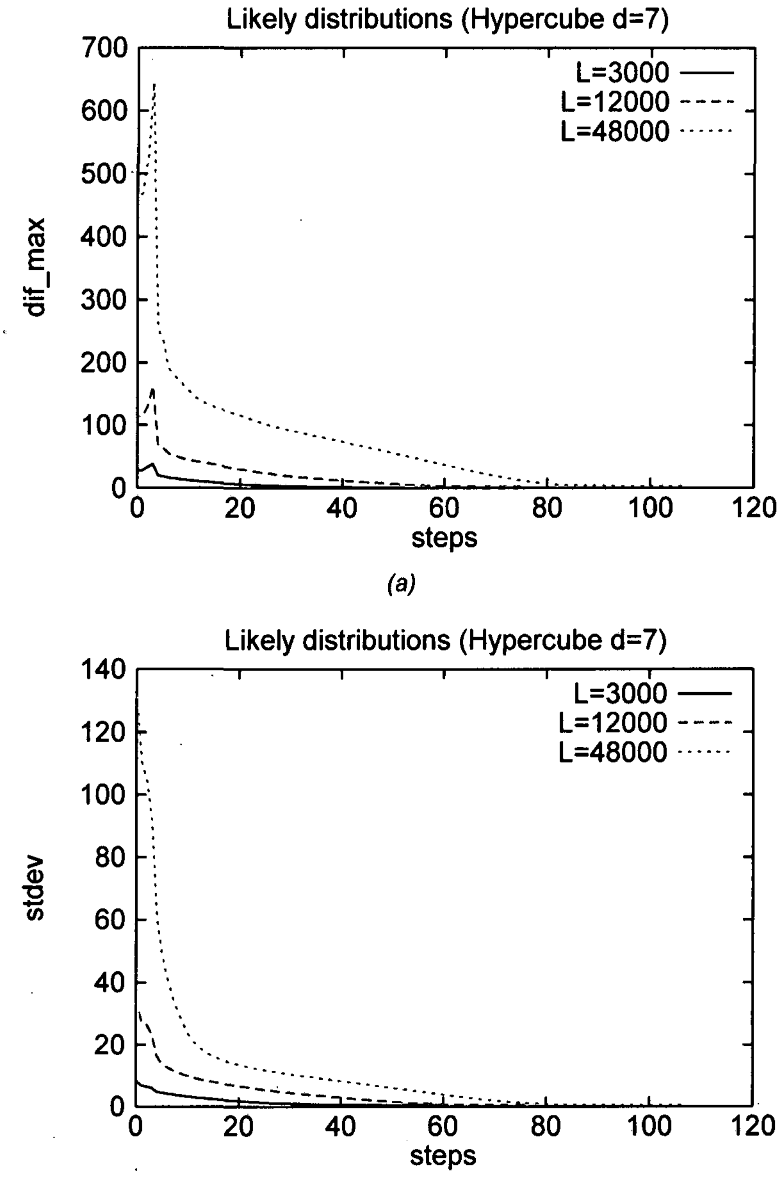
The experimental study outlined in chapter 4 has been undertaken considering different systems sizes and topologies but the problem size keeps constant for all experiments. The aim of that chapter was to compare different load-balancing algorithms with respect to their ability to achieve the final stable state, and the cost incurred to reach it. From that analysis, we have concluded that DASUD results in the best trade-off between the final balance degree and the cost incurred to achieve it. In this chapter, we perform a more precise analysis of DASUD's behaviour under different problem sizes. The distributed nature of DASUD make us suppose that DASUD will react in a similar way for small problems as well as for large ones. Furthermore, we believe that DASUD will behave similarly as the problem size increases.

The results provided in the following sections have been obtained by setting the total amount of load distributed among the whole system (L) by one of the following values: 3000, 6000,12000, 24000 and 48000 load units. Two experimental studies are outlined below. Firstly, we analyse how DASUD is able to adapt to different problem sizes when the underlying topology and system size do not change. For that purpose, we do not simulate the load-balancing process for all commented problem sizes, and we only consider the two extreme and the medial values (3000, 12000 and 48000). Secondly, the influence of the system size is considered. Since in chapter 4 this parameter has been evaluated for a fixed problem size, here, we analyse what happens when the problem size changes as the system size changes as well. In this case, all above mentioned problem sizes (L) have been used to obtain a similar initial load imbalance for all system sizes.

5.2 DASUD's scalability with respect to the problem size

This section is focused on the evaluation of the DASUD's ability to work independently on problem size for a fixed topology, and on system size. Since we have executed the load-balancing simulation process for the three problem sizes mentioned (*L*): 3000, 12000 and 48000, the final global load average will be different for each problem size. At each step of the simulation process, the maximum load difference (*dif_max*) throughout the system and the global standard deviation (σ), denoted as *stdev* in the graphics, have been evaluated in order to plot their evolution as the load-balancing simulation process progresses, both indexes have been previously introduced in chapter 4. The graphics for all topologies and system sizes considering likely and pathological initial load distributions are reported in appendix C to this work. Since DASUD exhibits a similar behaviour for all of them, we only include in this section the analysis of the figures corresponding to the largest system size for each topology (7-dimensional hypercube and 11x11 torus) and for likely initial load distributions.

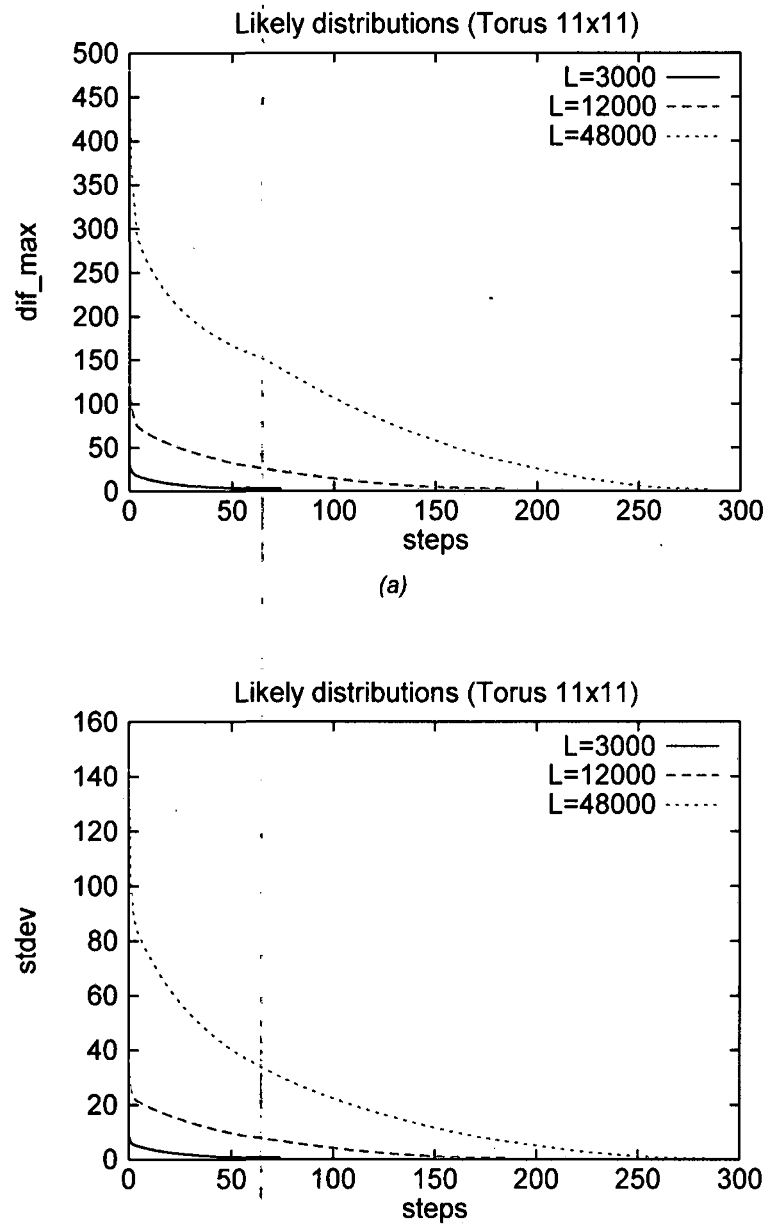
Figures 5.1(a) and 5.1(b) show the global maximum load difference and the global load standard deviation as the load-balancing simulation process progresses for a 7-dimensional hypercube. Each plot for those figures is the mean value of the global maximum load difference for all initial *likely* load distributions at the same *simulation step.* Figures 5.2(a) and 5.2(b) depict the same information described above but in this case for a 11x11 torus topology. As can be observed in figures 5.1(a) and 5.1(b), the balance rate has an insignificant degradation as the problem size increases for a 7-dimensional hypercube topology. Consequently, for hypercubes, DASUD's ability to have a high decreasing gradient in the global unbalance during the initial iterations of the load-balancing process is independent of the problem size. From a more accurate analysis of figure 5.1(a) we note that the global maximum load difference shows irregular behaviour throughout the initial load-balancing *simulation steps.* However, the global standard deviation (figure 5.1(b)) shows constant decreasing throughout the whole load-balancing simulation process. This situation is not usual, but it sometimes appears at the very beginning of the load-balancing simulation process as a consequence of some small load thrashing. However, this fact is not very relevant, since the global load standard deviation shows no fluctuations during this small period. Therefore, the load-balancing simulation process in hypercubes seems to be less influenced by the growth of the problem size.



(a)

(b)

Figure 5.1 Influence of the problem size in (a) the global dif_max and (b) in the global load stdev as the load-balancing process progresses for a 7-dimensional hypercube and for likely initial load distributions.



(a)

(b)

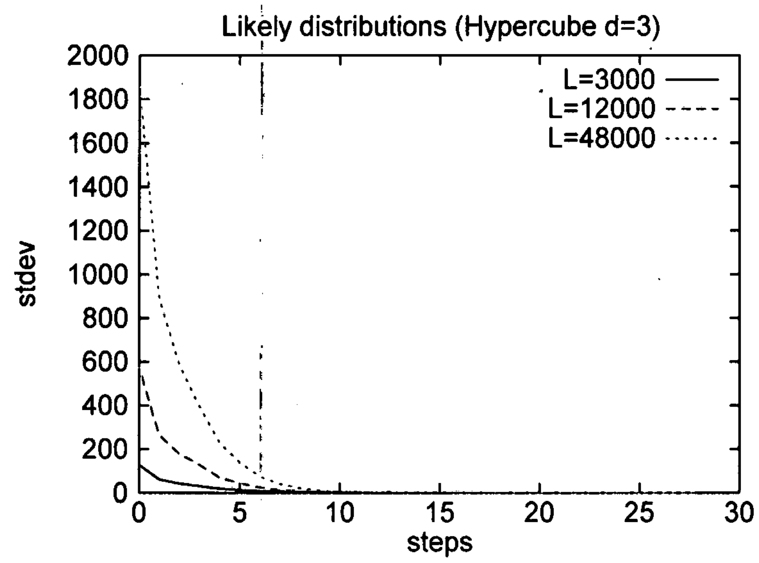
Figure 5.2 Influence of the problem size (a) in the global dif_max and (b) in the global load stdev as the load-balancing process progresses for a 11x11 torus and for likely initial load distributions

The analysis of DASUD behaviour in torus topologies when the problem size changes exhibits some important characteristics. As happens for hypercubes, the global maximum load difference and the global load standard deviation monotonically decreases as the load-balancing simulation process progresses. However, DASUD's response rate in 11x11 torus decreases slower than in hypercubes, i.e., while the problem size increases the response rate slightly degrade, as can be observed in figures 5.2(a) and 5.2(b). These results can be explained by the enlarged diameter of this particular interconnection network. The diameter of an 11x11 torus is equal to 10, whereas the diameter of a 7-dimensional hypercube is 7. Therefore, as we only consider local load movements, the larger topology diameter is, the slower the load distribution is performed. With regard to the results obtained for both topologies with similar diameter, the response rate remains very close, as is shown in figure 5.3. This figure shows DASUD's behaviour with respect to the global standard deviation (*stdev*) for a 3-dimensional hypercube (5.3(a)) whose diameter is equal to 3 and for a 3x3 torus (5.3(b)) whose diameter is 2, for likely initial load distribution (these graphics are extracted from appendix C). It is easy to observe that the behaviour depicted in both figures is practically the same.

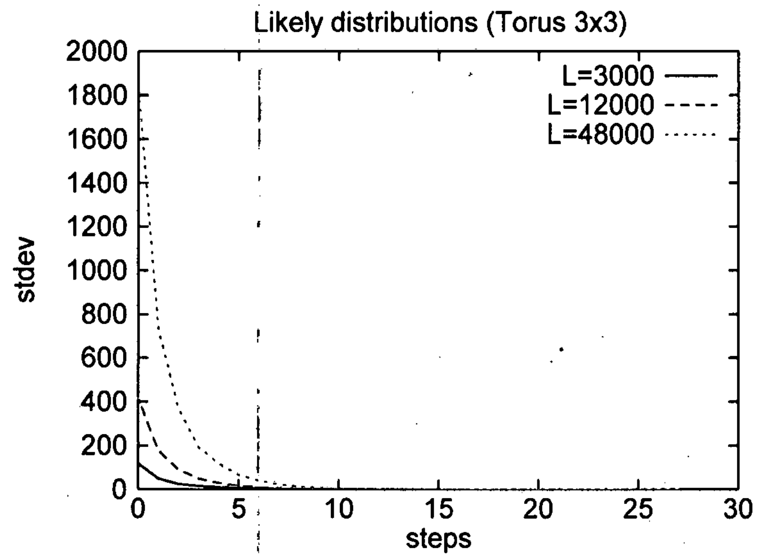
In conclusion, the ability of DASUD to reach a good balance degree depends minimally on the problem size. Additionally, this load-balancing algorithm is also able to act similarly for different configurations of the architecture by exhibiting a slight dependence on the topology's diameter.

5.3 DASUD's scalability with respect to system size

In this section, we analyse DASUD's response rate as the number of processors increases for a given interconnection pattern. For this purpose we have varied the number of processors from 8 to 128. As in the previous section, the study has been performed for hypercubes and torus, starting the load-balancing process from any initial load distribution including either *likely* or *pathological* patterns. The results included below correspond to those obtained when likely initial load distributions are applied. Since the observed behaviour for *pathological* initial load distributions is the same as for *likely* patterns, the graphics for *pathological* distributions are not included in this section ; however, they may be consulted in appendix C.



(a)



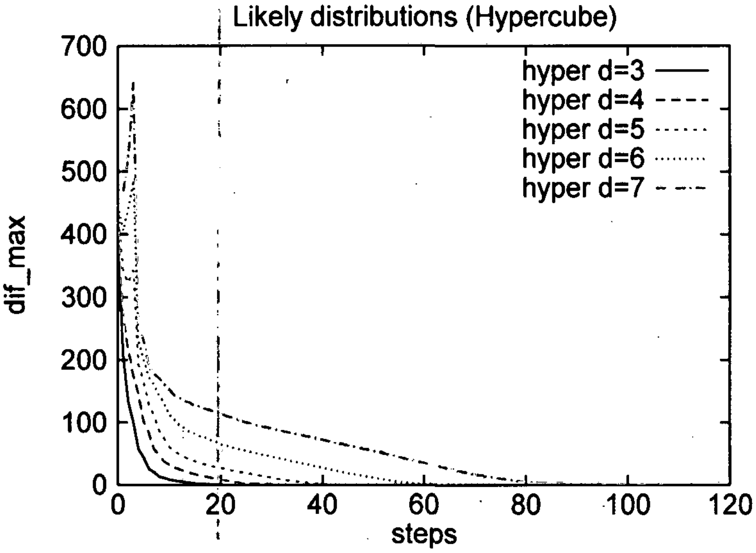
(b)

Figure 5.3 Influence of the problem size in the global standard deviation as the load-balancing process progresses for (a) a hypercube with d=3, and (b) a torus with d=2 for likely initial load distributions.

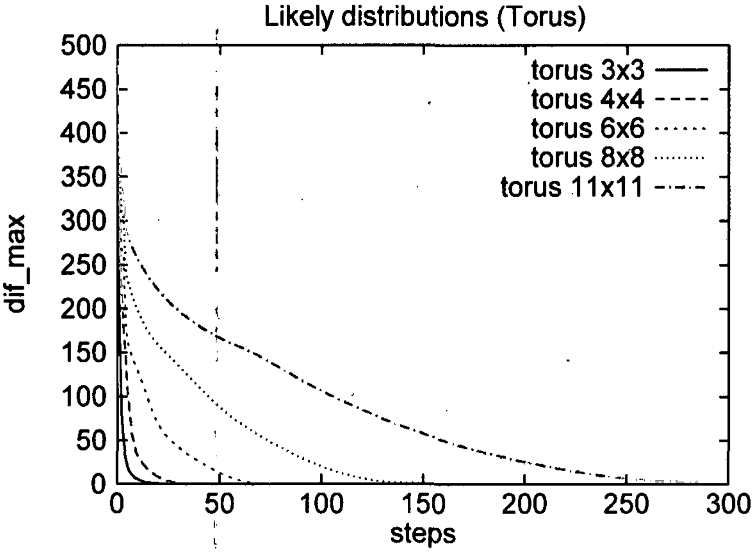
The evolution of the global maximum load difference (*dif_max*) for hypercubes and torus is depicted in figures 5.4(a) and 5.4(b) respectively. The results for global load standard deviation (*stdev*) are shown in figures 5.5(a) and 5.5(b). As has been mentioned previously, DASUD's capacity for moving load among the system is limited by the diameter of the underlying interconnection network. Larger diameters yield slower load propagation, since load movements are performed locally. The results obtained for torus topologies seems, at first sight, worse than the results obtained for hypercubes, however, what is happening is that for the same system size, the diameter of each topology has different values. In torus interconnection schemes, the directed connected processors for all processor remains constant as the system size increases whereas in hypercube architectures the size of the domain increases as the system size also increases. Therefore, for the same number of processors, hypercube exhibits a higher connectivity degree than torus and, consequently, the value of the diameter is, at most, the same for hypercube and for torus, but in most cases, it is smaller for the same system size.

Notice that the larger system sizes are 128 and 121 processors for hypercube and torus respectively, whereas the corresponding diameters are 7 and 10. But, if we observe the results for topologies with the same diameter value as happens for a 4-dimensional hypercube and 4x4 torus, for instance, we observe that there is a slight difference in the obtained response rate, even if we analyse the global maximum load difference or the global standard deviation.

Finally, a similar conclusion as for scalability analysis with respect to problem size can be extracted. The ability of DASUD to reach a good balance degree depends minimally on the system size, more precisely, DASUD exhibits a slight dependence on the topology's diameter whatever interconnection pattern we have.

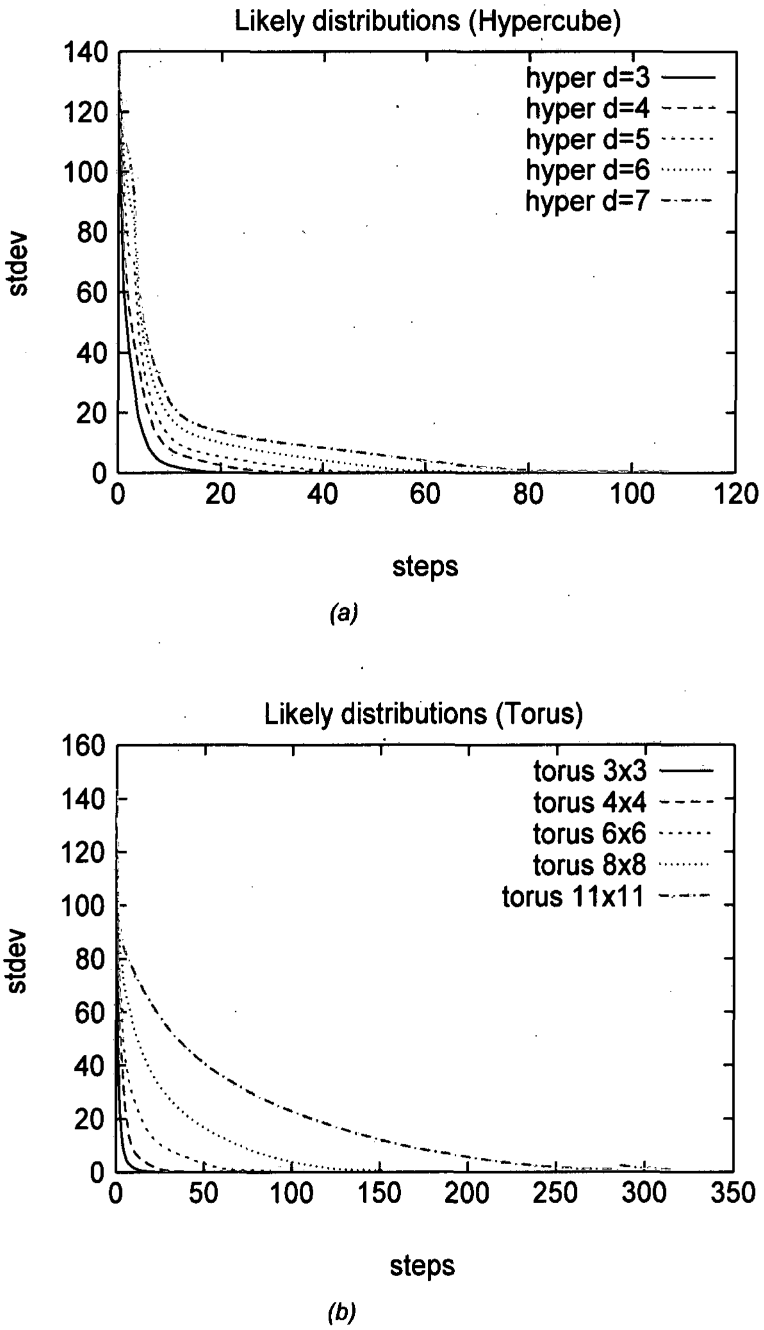


(a)



(b)

Figure 5.4 Influence of the system size in the global maximum load difference as the load-balancing process progresses for (a) a hypercube and (b) torus for likely initial load distributions.



(a)

(b)

Figure 5.5 Influence of the system size in the global standard deviation as the load-balancing process progresses for (a) a hypercube and (b) torus for likely initial load distributions.

5.4 Conclusion about DASUD's scalability

From the conclusion obtained in the two previous scalability analyses where the influence of the problem size and the system size have been studied, we conclude that DASUD is a load-balancing algorithm whose balance properties slightly depend upon the diameter of the underlying topology. The main reason for this fact is its totally distributed nature. Since load movements are performed locally, larger diameters yield slower load propagation.

Chapter 6

Enlarging the domain (d_S-DASUD)

Abstract

In this chapter, the following question is analysed: is it possible to accelerate the load-balancing process enlarging the DASUD's domain to include non-directed connected processors? For this purpose, an extended system model is provided. The influence of enlarging the domain into the time incurred in transferring messages beyond one link and in the extra computational cost incurred by the extended version of DASUD (d_s-DASUD) has been analysed. From this analysis we determine which enlargement provides the best trade-off between balance improvement and load-balancing time spent.

6.1 Introduction

Up to this point, the DASUD algorithm identifies the domain of a given processor *i* with processors with which it has a direct neighbourhood relation. A question that arose during the evaluation of DASUD was: *how would DASUD work if it was able to collect more load information than only that of its immediate neighbours*? Intuitively, we expect that if DASUD had the capability of using more load information to take load-balancing decisions, globally, it should be able to reach a better final balance situation spending less time. We also expect that this effect be proportional to the enlargement of the domain. For instance, if each processor were able to collect the load information from the whole system, i.e., DASUD working as a totally distributed load-balancing algorithm with global information, DASUD should then, be very fast in correcting unbalanced load distribution, and the final state should be evenly. However, we know that this DASUD extension incurs an extra time cost due to communication and computational cost increments. Thus, we were interested in studying the influence of the domain enlargement in the trade-off between improvement in the balance degree and the time incurred during the load-balancing process.

We refer to the extended version of DASUD as d_s-DASUD where d_s is the *domain scope* which is a non-negative integer value that represents the minimum number of links needed to cross from the underlying processor to the furthest processor belonging to its domain. We notice that when d_s is equal to1, the algorithm coincides with the original DASUD algorithm. Although d_s-DASUD is still a totally distributed load-balancing algorithm, the implementation of d_s-DASUD introduces some changes in the underlying system model described in chapter 2. Therefore, an extended system model is described in the following section. Subsequently, the time metrics evaluated to determine whether it is worth enlarging the domains or not are introduced. Finally, we report the experimental study performed to analyse how DASUD works when the domain of each processor is enlarged.

6.2 Extended system model

We recall from chapter 2 that we represent a system by a simple undirected graph *G=(P,E)*, i.e. without loops and with one or zero edges between two different vertices. The set of vertices P={*1,2,....,n*} represents all processors in the system. One edge *{i,j}* $\in$ E if there is a link between processor *i* and processor *j*. A processor *i* has a *domain* which is defined as the set of processors from which processor *i* maintains load information including itself and is defined as follows $N_i = \{j \in P \mid \{i,j\} \in E \cup \{i\}$. The *size* of a given domain is the number of processors belonging to that domain and is denoted by $\# N_i$. The number of direct neighbours for a given processor is denoted by *r*, which coincides with $\# N_i - 1$.

In the new definition of DASUD (*d_s*-DASUD) when the value of d_s is bigger than one (d_s>1), some *virtual links* can be considered to provide a way for connecting the underlying processor to non-directly connected processors belonging to its domain. Then, a new set of edges E_v can be considered and *{i,j}* $\in E_v$ if there is a real or virtual link between processor *i* and processor *j*. The set of processors belonging to the domain of processor *i*, including itself for a certain value of d_s, is denoted by $N_i^{d_s}$ The resultant topology is referred to as *virtual topology* and, consequently, a *virtual diameter* (*d_v*) should be defined. Figures 6.1(a) and 6.1(b) show the domain of a given processor (red colour) for a virtual 3-dimensional hypercube topology with d_s equal to 1 and 2, respectively. Notice that, when the value of d_s is equal to 1, the set of processors belonging to the domain of the underlying processor match directly with its direct neighbourhood (processors directly connected to it), therefore, the virtual topology coincides with the real one.

The *size of the virtual domain* of processor *i* depends on the value of d_s and is denoted by $\#N_i^{ds}$, and the number of virtual neighbours will be denoted by r_v. In order to simplify notation, in the rest of this chapter when we refer to neighbour processors, we consider both neighbours, real and virtual.

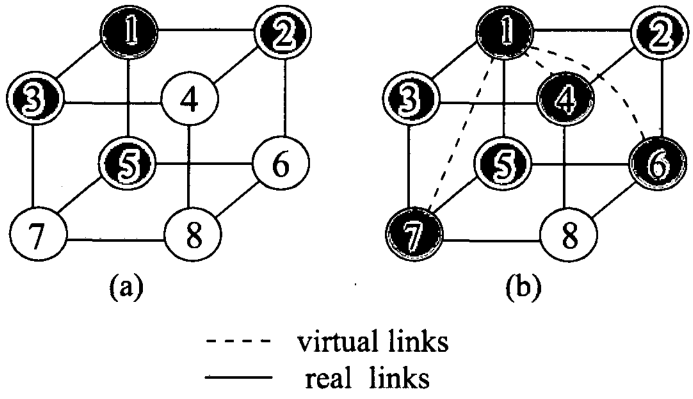


Figure 6.1 Virtual hypercube topology for (a) 1-DASUD and (b) 2-DASUD

The original version of DASUD has been theoretically analysed in chapter 3. In that chapter, some upper bounds have been provided for the *final balance degree* and the *balance rate*. Both of these have been derived under the assumption that a given processor was restricted to use load information from its directly connected processors. However, if this condition is relaxed as a consequence of enlarging the domain of each processor, these upper bounds should be updated in order to be applied to the extended version of DASUD (d_s-DASUD). Since this modification affects the diameter of the underlying topology (d) which should be changed by its virtual version d_v (virtual diameter), the original formulas for the above mentioned upper bound, should be updated. The notation of these extended upper bounds is shown in table 6.1.

Final balance degree	$\beta = floor\left(\frac{d_v}{2}\right) + 1$
Convergence rate	$B = \frac{d_v}{2} * (D_0 + 1)$

Table 6.1 Extended upper bounds for d_s-DASUD

The extended version of DASUD seems to be an interesting alternative for increasing the convergence rate of the original algorithm. Since larger domains allow performing more accurate balance decisions, we are interested in evaluating such a possibility. Therefore, we analyse the influence of the domain scope enlargement in the balance degree with the aim of determining a domain scope (d_s) that exhibits the best trade-off between balance degree improvement and the time overhead introduced when the scope of the domain is enlarged. For this purpose, in the following section, we describe all times involved during the load-balancing process, and how these times have been evaluated by taking into account the influence of the domain scope. Subsequently, these metrics will be used to experimentally analyse whether or not it is worth enlarging the domain scope of the DASUD algorithm.

6.3 Metrics

Iterative load-balancing algorithms improve the load imbalance of a system by successive iterations of the load-balancing operations. In such approaches the total load-balancing time depends on the number of iterations performed. Since the load-balancing process has been evaluated by simulation using the same load-balancing simulator described in chapter 4, we will use the term simulation step (or step for simplicity) as in chapter 4, instead of iteration.

Under the synchronous simulation paradigm, the total load-balancing overhead incurred by any iterative load-balancing algorithm (T_{bal}) can be obtained as follows,

$$T_{bal} = \sum_{s=1}^{last\_step} T_{bal}^{s}$$

where T_{bal}^{s} is the time required to execute one simulation step of the load-balancing process in the whole system, and *last_step* denotes the last step of the load-balancing simulation process. More precisely, the duration of the *s*-step of the load-balancing simulation process (T_{bal}^{s}) can be divided into two *communication periods* (information collection and transfer periods) and the *computational period*:

- Communication periods:
  - ⇒ Information collection period (T^s_{col}): Interval of time required to gather all load information needed by all processors for executing the load-balancing algorithm.
  - ⇒ Transfer period (T^s_{trf}): Time required to perform all load movements from source processors to destination processors.
- Computational period ($T^s_{c\_bal}$): Period of time dedicated by the load-balancing strategy to evaluate the load movements in all processors.

Consequently, and bearing in mind the synchronous paradigm, the total time overhead introduced by the global load-balancing process (T_{bal}) can be obtained as follows:

$$T_{bal} = \sum_{s=1}^{last\_iteration} \left(T^s_{col} + T^s_{c\_bal} + T^s_{trf} \right)$$

Notice that the duration of these load-balancing periods is directly affected by the domain scope (d_s), i.e., as the domain scope increases, those periods will likewise extend. Therefore, it is necessary to be able to evaluate this overhead in order to obtain reliable results. We now describe how the *communication and computational periods* have been evaluated. Subsequently, these time are used to introduce a goodness index called the *trade-off factor,* which allow us to determine which d_s provides the best results.

6.3.1 Communication periods

Since in direct networks, the blocking time of a message (which is defined as the time spent waiting for a channel currently being used by another message [Ni93]) cannot be pursued, to evaluate the communication times incurred by the extended version of DASUD (d_s-DASUD) during the *collection information* and *transfer periods*, the interconnection network functional simulator NETSIM has been used [Fra99]. This simulator considers a *wormhole* routing as routing technique and it takes into account the resource contentions that a message encounters in its path. In this routing technique the original message is broken into small units called *flits.* The header flit(s) of a message contains all the necessary routing information and all the other flits contain the data elements. The flits of the message are transmitted through the network in a pipelined fashion. Since only the header flit(s) has(have) the routing information, all the trailing flits follow the header flit(s) contiguously. Flits of two different messages cannot be interleaved at any intermediate node [Moh98]. The communication latency for a *wormhole* routing technique is obtained by considering the following times:

- *Start-up time* (t_s): time needed to prepare a message. This delay is incurred only once for a single message.
- *Per-hop time* (t_h): the time taken by the header of a message to travel between two directly-connected processors in the network.
- *Per-flit transfer time* (t_w): the time taken by a flit of the message to traverse one link.

If we consider a message that is traversing a path with *l* links, then the header of the message takes lt_h time to reach the destination. If the message is *m* flits long, then the entire message will arrive in time mt_w after the arrival of the header of the message. Therefore, the total communication time (t_{comm}) for a *wormhole* routing, i.e. the time taken by one message to go from source processors to the destination one, is given by

$$t_{comm} = t_s + lt_h + mt_w \qquad (1)$$

The injection of messages into the network has been simulated by using the worst pattern in which all processors start the injection process simultaneously. This injection pattern has been used for both communication periods, *collection information period* and *transfer period.* From the execution of the NETSIM simulator we have obtained the total simulation time incurred for the whole communication process by including the injection of the messages by the source processors, the resource contentions and deadlock detection and recovery. Subsequently, the particular usage of this network simulator for each communication period is described.

The information collection period (T_{col}^{s})

Remember that the *information collection period* is the time spent in gathering load information for all processors in the system. Since the domain scope is a fixed value, which does not change at each load-balancing step, the time incurred by the information collection period will be the same for each load-balancing step. Therefore, the simulation of load messages travelling around the system only needs to be executed once. This simulation has been performed by injecting 2-flit messages from all processors to all of its neighbours, and the total NETSIM simulation time is considered as the time spent by the *information collection period.*

The transfer period (T_{trf}^{s})

In contrast to the *information collection period*, the duration of the *transfer period* depends on the current step of the load-balancing simulation process. At each load-balancing step, the execution of the load-balancing algorithm at each individual processor provides different load movements. Therefore, the time spent in sending and receiving those messages must be evaluated at each load-balancing step. For that purpose, for each experiment a trace file is generated, where all load movement decision generated for all processors at each load-balancing simulation step are recorded. The information stored in this trace file is used as input to the NETSIM simulator in order to evaluate the NETSIM time spent for each load-balancing simulation step in performing the corresponding load transfer movements. In particular, for a given load transfer of size M, the length of the corresponding messages that was injected in the networks was 2M.

6.3.2 Computational period ($T_{c\_bal}^{s}$)

The *computational period* for *d_s-DASUD* has been evaluated using the formula of DASUD's complexity derived in chapter 3. Since that formula was derived by considering the domain of a given processor as its immediate neighbours, it should be adapted to take into account the extended model of DASUD. This updating consists of substituting the number of direct neighbours (*r*) by the number of virtual neighbours (*r_v*) as follows:

$$O(r_v \log r_v)).$$

Since the domain size is fixed for a given d_s, and it the same for all processors, the computational time incurred by each processor at each load-balancing step remains constant during the whole load-balancing simulation process.

6.3.3 Trade-off factor ($t\_off(k)$)

Finally, we introduce an index to measure the trade-off between the global standard deviation in the balance iteration *k* ($\sigma(k)$) and the time incurred to achieve this ($T_{bal}(until\_k)$). This goodness index is called the *trade-off factor* and is denoted by $t\_off(k)$. For this purpose, let us introduce the term $T_{bal}(until\_k)$. In the previous section, we have described how the three time periods involved in the load-balancing process (*information collection period,* the *computational period* and the *transfer period*) are individually evaluated. We evaluate the duration of these periods at each load-balancing iteration in the way described in the previous section. By adding together all these times from step 1 until step *k*, we can obtain the time spent for the load-balancing process throughout these *k* steps. Figure 6.3 graphically shows the serialisation of these times, and formula (2) formally describes it. We observe that the terms T^s_{col} and $T^s_{c\_bal}$ from formula (2) remains constant for all steps, whereas T^s_{trf} depends on the current simulation step, as has previously been commented. Furthermore, when *k* coincides with the *last_step* the evaluated time is the time spent in the global load-balancing simulation process.

$$T_{bal}(until\_k) = \sum_{s=1}^{k}\left(T^s_{col} + T^s_{c\_bal} + T^s_{trf}\right) \tag{2}$$

Finally, the *trade-off* factor is obtained by the multiplication of both parameters involved $\sigma(k)$ and $T_{bal}(until\_k)$, as shown in formula (3).

$$t\_off(k) = T_{bal}(until\_k) * \sigma(k) \tag{3}$$

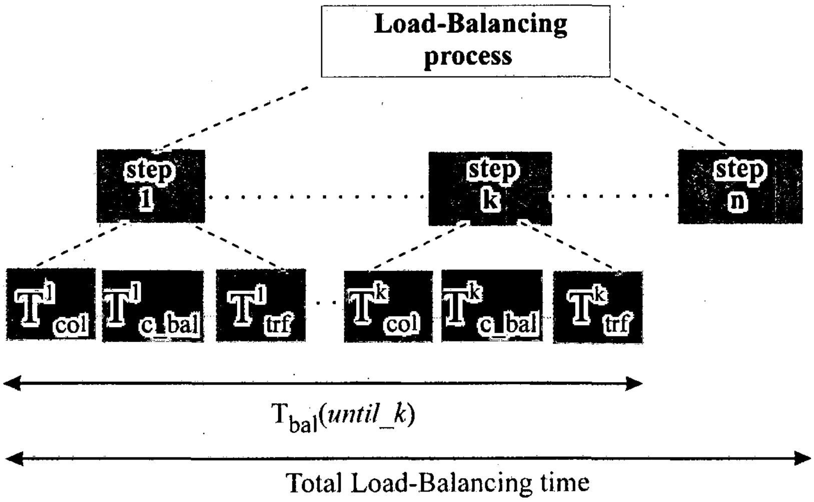


Figure 6.3 Time evolution as the load-balancing process progresses.

In particular, this trade-off factor would tend to be optimal for small global load standard deviations values, and for small load-balancing times as well. But, furthermore, the trade-off factor will be small when one of the two operands is also small.

6.4 The experimental study of d_s-DASUD

This section is aimed to analysing the influence of domain enlargement on balance improvement by taking into account the extra time incurred as a consequence of sending and receiving messages beyond immediate neighbours, and spending more time in executing the load-balancing algorithm. The experimental study outlined below has been carried out using the same simulation framework described in chapter 4. We recall that the set of initial load distributions used is divided into two groups (*likely* and *pathological*), and that each initial load distribution has been scattered by following two different shapes (Single Mountain and Chain). Moreover, the load-balancing simulation process has been executed under two different interconnection networks (hypercube and torus), and the system size ranges from 8/9 to 121/128 processors. The problem size varies, as does the system size, in order to obtain in all cases a similar initial load imbalance.

In order to analyse the existent relation between balance improvement throughout the load-balancing simulation process, and the time spent during that process, the global load standard deviation has been evaluated at each *simulation step k* (σ(*k*)), as well as the corresponding $T_{bal}(until\_k)$. This parameter has been evaluated for all *simulation steps* until the *last_step* is achieved. As in the previous experimental studies, the simulation process has been run until no load movements were produced from one iteration to the next. However, we have superimposed a maximum number of *simulation steps* at which the simulation process will be stopped, although the final stable load distribution has not been achieved. This *simulation step limit* has been chosen as 2000.

The experimental study outlined in the following section is aimed at determining which domain scope (d_s) provides the best trade-off between the final balance degree and the time incurred to reach it. In the subsequent section, the experimental study reported is focused on deriving the d_s that provides the best balance improvement at the beginning of the load-balancing process without executing the load-balancing process until its completeness.

6.4.1 The best degree of final balance

Figures 6.4 and 6.5 show the evolution of the global load standard deviation (*stdev*) through simulation time (*time*) for *likely* initial load distributions for a 5-dimensional hypercube and for a 6x6 torus, respectively, where all possible d_s have been considered. These two examples have been chosen as representative for both topologies because the rest of system sizes exhibit a similar behaviour. Nevertheless, complete simulation results may be found in Appendix D. Each plot of the depicted curves shows the mean value of the global load standard deviation at a given step *k* (σ(*k*)) of the load-balancing simulation process for all initial load distributions versus the mean time needed to achieve such situations ($T_{bal}(until\_k)$). Although these values are obtained for the entire balance simulation process, in order to make the analysis of the curves easier, in figures 6.4 and 6.5 only the time interval where the relevant variations are detected have been plotted. However, this fact does not affect the comprehension of the following discussion.

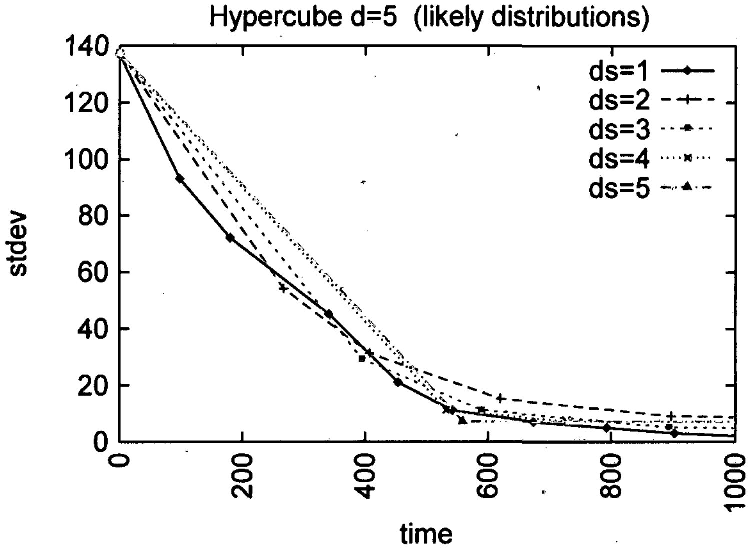


Figure 6.4 Global load standard deviation versus time for a 5-dimensional hypercube varying ds from 1 to 5 for likely initial load distributions.

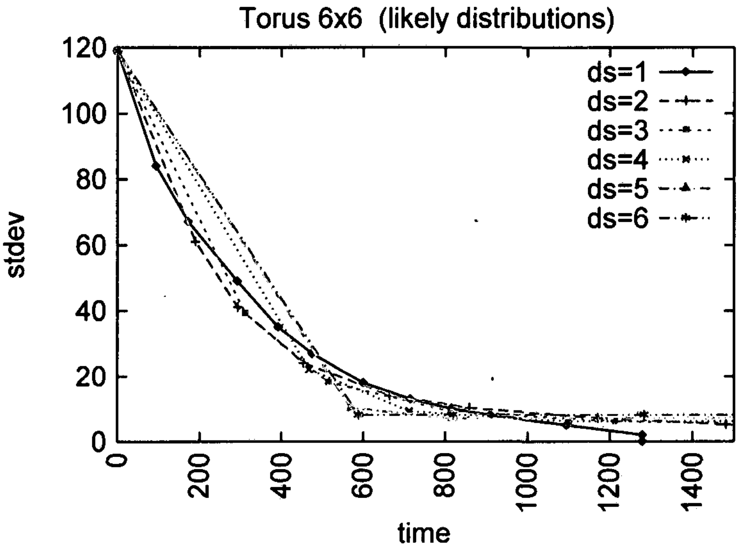


Figure 6.5 Global load standard deviation versus time for a 6x6 torus varying ds from 1 to 6 for likely initial load distributions.

From a preliminary analysis of figures 6.4 and 6.5 we can conclude that there seems to be no difference in the global balance rate, whichever domain scope is applied. However, we observe that there is a time beyond which the balance rate of larger d_S's is slowed down reversing their behaviour with respect to small domain scopes. We call this atypical phenomenon *inversion rate effect*. The magnitude of this effect is more appreciable in tables 6.2 and 6.3 where the mean values for the final global load standard deviation, the total load-balancing time and the total number of load-balancing *simulating steps* are shown for hypercube and torus respectively. It is interesting to observe that the final balance degree achieved for all domain scopes is approximately the same, but only a slight improvement is obtained for larger d_S's. Notice that in the case of the largest d_S, the perfect final balance situation is achieved as was expected, i.e., *stdev* equal to 0 when *L* (problem size) is an exact multiple of the underlying number of processors, and is very close to 0 otherwise. In the case of the largest system sizes for both torus and hypercube topologies (121 and 128 processors), the values included in the tables do not represent the real final situation because in both cases the *simulation step limit* previously commented on has been achieved.

With respect to the load-balancing *simulation steps* needed to reach the final load distribution, one can observe that, on average, this follows a sequence of values that exhibits a global minimum at approximately d_S equal to 3, on average, for torus topologies, whilst for hypercube topologies the minimum number of steps alternates between d_S equal to 1 and 2 (yellow cells). In contrast, the time incurred in attaining this final situation significantly increases as the domain scope also increases. For instance, for all topology and system sizes, the biggest domain scope, instead of being the fastest in reaching the final stable situation, becomes the slowest. Unexpectedly, the minimum time is obtained with d_S equal to 1 (green cells).

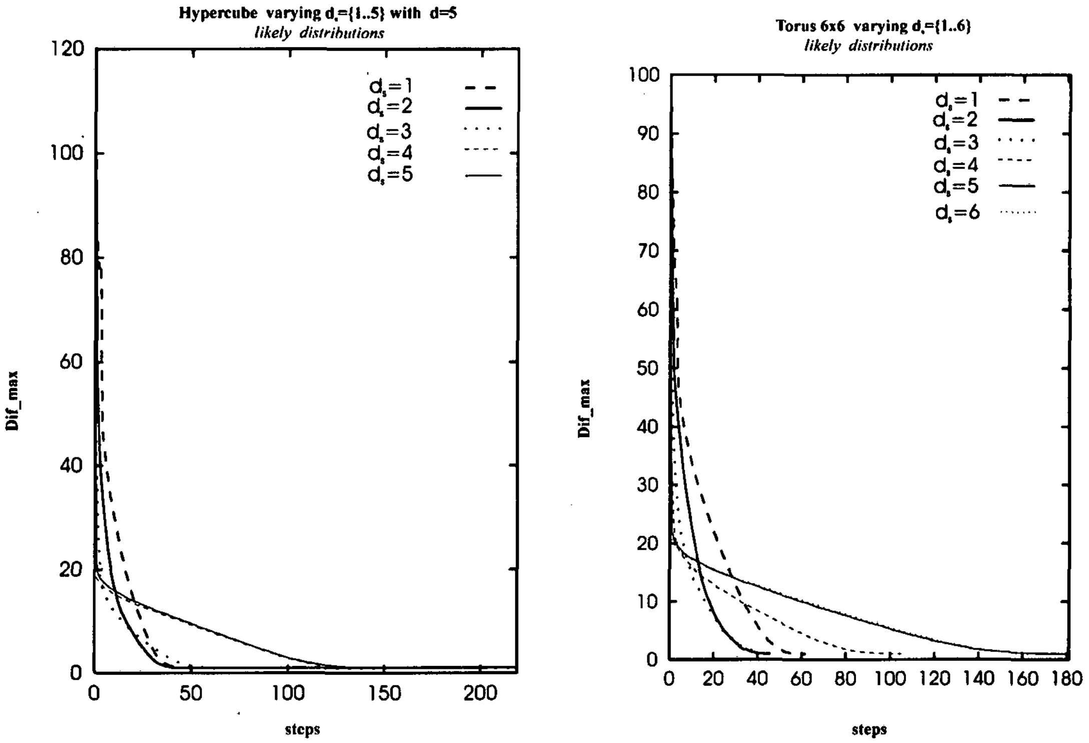
Hypercube		*Likely distributions*			*Pathological distributions*		
d	*ds*	*std. deviation*	*time*	*steps*	*std. deviation*	*time*	*steps*
3	1	0.031	555,41	25	0.000	887,88	21
	2	0.000	619,75	15	0.000	1346,13	14
	3	0.000	705,51	13	0.000	1370,25	14
4	1	0.219	962,80	37	0.283	1610,10	35
	2	0.000	1824,56	31	0.044	3207,63	28
	3	0.000	3149,16	37	0.000	4570,88	39
	4	0.000	4118,74	58	0.000	6588,03	44
5	1	0.204	1672,36	57	0.250	2330,63	49
	2	0.000	5769,09	61	0.000	8425,50	62
	3	0.000	9128,08	69	0.356	12096,88	55
	4	0.000	25166,94	148	0.000	22614,70	116
	5	0.000	29631,30	170	0.000	27511,97	141
6	1	0.267	2364,61	78	0.262	3752,75	71
	2	0.000	11573,58	94	0.159	15896,00	86
	3	0.000	26762,59	107	0.000	30394,03	98
	4	0.002	104888,93	310	0.002	95003,13	263
	5	0.000	270014,80	737	0.663	215714,55	573
	6	0.000	274344,53	738	0.000	208615,75	545
7	1	0.349	3435,90	108	0.340	5325,84	93
	2	0.012	23684,14	144	0.000	32172,28	131
	3	0.000	86784,95	194	0.000	84834,18	163
	4	0.000	404814,09	599	0.000	403714,55	580
	5	0.000	833958,06	1045	0.000	722162,75	890
	6	0.056	1697818,60	1999	7.130	1709215,9	1999
	7	0.011	1695999,18	1999	10.459	1707831,1	1999

Table 6.2 Global load standard deviation, load-balancing time and steps at the end of the load-balancing process in hypercube topologies for all domain scopes.

Torus		*Likely distributions*			*Pathological distributions*		
nxn	*ds*	*sigma*	*time*	*steps*	*sigma*	*time*	*steps*
3x3	1	0,34	525,85	20	0,47	787,13	20
	2	0,34	668,14	15	0,47	981,38	15
4x4	1	0,32	964,76	30	0,36	1621,25	30
	2	0,00	1606,69	30	0,29	2450,25	25
	3	0,00	2874,39	40	0,00	3881,88	40
	4	0,00	4108,43	50	0,00	6293,13	50
6x6	1	0.471	1279,19	75	0.700	2074,500	86
	2	0,47	4027,48	50	0.471	5739,45	50
	3	0,47	8485,56	50	0.471	8511,38	51
	4	0.471	17894,25	95	0.471	17330,90	82
	5	0.471	40792,40	182	0.471	42592,13	182
	6	0.471	38434,11	166	0.471	39734,25	164
8x8	1	0.484	3263,99	153	0.484	6233,38	150
	2	0.060	5352,39	79	0.075	8824,32	79
	3	0.007	12840,13	77	0.000	16625,25	76
	4	.0.000	29084,70	98	0.000	32997,63	92
	5	0.000	76693,15	185	0.000	65757,00	148
	6	0.000	183512,14	349	0.000	177608,38	331
	7	0.000	58602,48	631	0.232	62835,00	573
	8	0.000	60232,61	738	0.000	64456,13	545
11x11	1	0.725	5939,61	318	2.280	12637,25	286
	2	0.461	10488,06	144	0.703	17633,93	134
	3	0.461	24203,06	121	0.471	31838,30	116
	4	0.436	50564,72	133	0.461	58815,21	132
	5	0.436	129053,01	199	0.461	131420,25	191
	6	0.436	349087,78	358	0.461	236162,13	233
	7	0.436	126472,75	765	0.461	136483,00	567
	8	0.436	149904,51	901	0.461	160236,75	736
	9	0.436	169677,32	1446	0.817	157790,75	1446
	10	0.436	185572,73	1999	1.416	195105,63	1999

Table 6.3 Global load standard deviation, load-balancing time and steps at the end of the load-balancing process in torus topology for all domain scopes.

Let us analyse more precisely the reasons for this atypical behaviour. For this purpose figures 6.6(a) and 6.6(b) must also be studied. These figures show the mean value of the global maximum load difference at each load-balancing step for all *likely* initial load distributions for a 5-dimensional hypercube and a 6x6 torus topology respectively and for all domain scopes. The results for the remaining system sizes and initial load distributions patterns can be found in Appendix D. Note that the global maximum load difference also suffers from the *inversion effect* which is caused by the existence of a step beyond which larger d_S slowed down the reduction of the global maximum load difference, as for standard deviation. The reasons for such behaviour are, although it apparently contradictory, the ability of DASUD to evenly balance unbalanced domains and certain convergence requirements. We now analyse these motives in detail.



(a) *(b)*

Figure 6.6 Global maximum load difference versus load-balancing steps for (a) a 5-dimensional hypercube and (b) a 6x6 torus for all possible domain scopes.

In chapter 4 we observed that the original DASUD algorithm has the ability of reaching, in most cases, even load distribution or, at most, in keeping the unbalance degree bounded by half the value of the diameter of the topology plus one. The extended version of DASUD, d_s-DASUD, also exhibits this ability. In particular, when the value of d_s coincides with the topology diameter (*d*) we can assert that the final stable state will be evenly. This characteristic stems from the capability of this load-balancing algorithm to search unbalanced domains, and to equilibrate them (maximum load difference available within the domain 0 or 1). Since the domain of each processor becomes the whole system when d_s is equal to *d*, d_s-DASUD is iterated until the perfect final balance situation is achieved. Attaining the balance state implies a great effort by d_s-DASUD for large d_s in terms of load-balancing steps, as is clearly depicted in figures 6.6(a) and 6.6(b). However, this effort is spent in moving small load quantities throughout the system, since the maximum load difference is not greatly reduced at each load-balancing step. The main reason for such an anomalous situation is certain convergence requirements needed by DASUD and by d_s-DASUD as well. We recall some implementation features from chapter 3 of DASUD which are directly involved in the *inversion rate* effect. During the execution of one iteration of the DASUD algorithm in a given processor, load movement decisions may be performed as a consequence of executing the named PIM block. This DASUD block processes all the received instruction messages and, sometimes, as a consequence of attending one of them, the movement of one load unit can be ordered. In this case, the rest of the instruction messages are deleted. Bearing in mind such behaviour, let us analyse what occurs in the following example. Assume 5 processors connected in a linear array, as shown in figure 6.7. The real diameter of this interconnection network is 4 (*d*=4) and, since the extended version of DASUD used in this example is 4-DASUD, its virtual diameter is 1 (d_v=1). Therefore, the domain of each processor includes all system processors, and the virtual topology corresponds to a full-connected system. The initial load distribution is that depicted in figure 6.7(a), where the maximum load difference in the whole system, coinciding with each processor domain, is 2 load units. Thus, it is detected as unbalanced by 4-DASUD, and some actions are carried out to arrange it. In particular, all processors detect their domains as unbalanced, because the global load average is equal to 9.2, but only processors 4 and 5 really perform certain actions. These actions are derived from the execution of the second stage of 4-DASUD. Since both processors observe that the maximum load difference within their domain is bigger than 1 load unit, and that their load values do not correspond to the biggest in the corresponding domain, both processors execute the SIM block of 4-DASUD. Consequently, both processors send one instruction message to processor 1 commanding 1 load unit to be sent to processor 4. When processor 1 processes this message, it will only attend to one of them by performing the load movement shown in figure 6.7(a). The resulting load distribution is depicted in figure 6.7(b). The unattended message will be deleted. Subsequently, the execution of the next iteration of 4-DASUD generates processors 1, 4 and 5 to send, one instruction message each to processor 2, commanding send 1 load unit to be sent to processor 5. Finally, only one of these messages will be attended to, and the final load distribution achieved is that shown in figure 6.7(c).

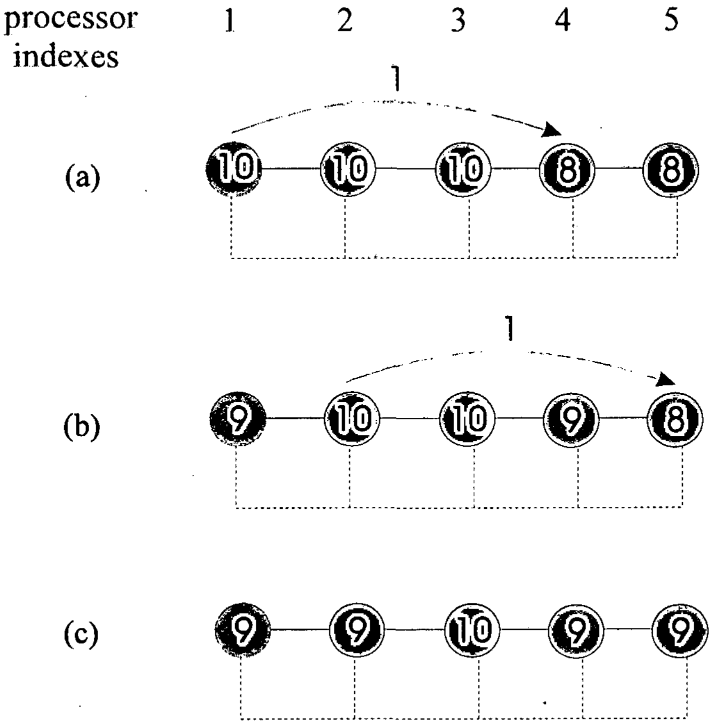


Figure 6.7 Load-balancing process applied using the 4-DASUD algorithm to a linear array with 5 processors

This slow way to balance a domain that includes all processors in the system becomes slower as the system grows. Notice that all instruction messages sent at each iteration of the load-balancing process are driven to the same processor instead of being sent to different processors. This design characteristic is needed to ensure the convergence of the algorithm. Therefore, we can conclude that the convergence requirements and the perfect local balance always achieved by DASUD whatever d_s is used, are the causes of the slow convergence rate for larger d_s.

At this point of the analysis of ds-DASUD, a preliminary conclusion arises: although intuitively the extended version of DASUD is supposed to be more effective in reaching a final stable load distribution, we have experimentally demonstrated that the best trade-off between the ability to stop the balance process and the time incurred to achieve it was exhibited by the original DASUD. There are two reasons that explain this anomalous behaviour. First, as the domain scope increases, the communication and computational costs significantly increase as well. Secondly, once a certain balance degree has been achieved in the system, the process to further reduce the existing imbalance is slower for large domains than for small domains. Therefore, we reoriented the current study to find an alternative solution if the load-balancing process is not carried out to its completeness. In particular, we were interested in the possibility of setting *a priori* a number of balancing steps to stop the load-balancing process with the certainly of having reached a substantial unbalanced reduction.

6.4.2 Greater unbalancing reduction

Tables 6.4 and 6.5 show the global standard deviation (*stdev*), the trade-off factor (*t_off*) and the percentage of unbalance reduction (*%*) for some particular steps from the very beginning of the load-balancing process (4 and 7). Each cell includes the mean value of the results obtained for *likely* initial load distributions. Since *pathological* initial load distributions exhibit a similar behaviour, their results are omitted here but they are also included in table D.1 and D.2 in appendix D. In order to put the initial load unbalance at the same level for all system sizes, problem size *L* has been varied proportionally to the number of processors. This fact is shown in the column denoted by step 0, where the global load standard deviation before executing the load-balancing process is included. Although we only show the values for the global load standard deviation, and the trade-off factor at steps 4 and 7, the same information has been evaluated and recorded throughout the whole load balancing process. However, we have chosen these initial balancing steps because we detected that during this period d_s-DASUD was able to reduce the initial load unbalance up to 90% and, in most cases improvement could be even greater. We indicate in green, for each system size, the biggest unbalance reduction and also the corresponding load standard deviation. Furthermore, yellow indicates the best trade-off factor for each system size too.

We can observe that for most of the cases both colour indices coincide in the same domain scope value. More precisely, for hypercube topologies, it is obvious that d_s equal to 3 is, on average, the best choice. However, this criterion does not apply to the torus topology. In this case we clearly observe that the best domain scope depends on system size. In particular, we can experimentally derive an optimal d_s in order to obtain a fast unbalance reduction by the following formula

$$d_s = \frac{d}{2} + 1$$

where d is the diameter of the topology. The optimal d_s values for both topologies are indicated in red in both tables.

Hypercube		*Likely distributions*						
step		*0*	*4*			*7*		
d	*d*$_s$	*stdev*	*stdev*	*%*	*t_off*	*stdev*	*%*	*t_off*
3	1	127,99	18,32	85,7	4238,11	5,89	95,3	1755,51
	2	127,99	1,19	99	371,03	0,29	99,9	117,55
	3	127,99	1,31	98,9	437,03	0,65	99,5	288,51
4	1	128,22	34,74	72,95	8671,45	14,37	89	4934,18
	2	128,22	5,94	95,4	2757,79	3,55	97,2	2247,51
	3	128,22	2,86	97,76	1604,60	2,36	98,2	1854,34
	4	128,22	3,55	97,2	2063,98	3,17	97,5	2598,79
5	1	137,27	45,51	66,84	15537,66	21,18	84,5	9627,37
	2	137,27	15,68	88,5	9725,52	9,83	92,8	8817,02
	3	137,27	5,61	95,9	5014,57	4,82	96,5	6316,25
	4	137,27	7,29	94,7	7781,99	7,07	94,8	11133,57
	5	137,27	7,48	94,5	8088,31	7,27	94,7	11651,27
6	1	134,42	56,01	58,3	21277,53	28,70	78,6	14691,53
	2	134,42	23,49	82,5	18218,26	15,19	88,7	17723,31
	3	134,42	9,56	92,8	13593,65	7,86	94,15	17136,59
	4	134,42	12,98	90,3	24239,50	12,75	90,5	36776,42
	5	134,42	14,95	88,8	29492,99	14,84	88,8	45690,51
	6	134,42	14,89	88,9	29766,60	14,79	89	46154,22
7	1	133,03	60,65	54,4	24584,05	35,63	73,2	19635,76
	2	133,03	31,07	76,6	30523,17	21,57	83,7	32617,08
	3	133,03	16,31	87,7	37467,53	13,49	89,8	49445,57
	4	133,03	23,25	82,5	72206,99	23,01	82,7	118297,1
	5	133,03	26,01	80,2	99923,61	25,88	80,5	161719,6
	6	133,03	28,68	78,5	116302,1	28,62	78,5	189485,1
	7	133,03	28,13	78,8	114600,3	28,07	78,8	186383,8

Table 6.4 Standard deviation, unbalance reduction percentage and trade-off between balance degree and time for hypercube topologies and for all possible d_s at different load-balancing steps.

Torus		*Likely distributions*						
step		*0*	*4*			*7*		
nxn	d_s	*stdev*	*stdev*	*%*	*t_off*	*stdev*	*%*	*t_off*
3x3	1	117.15	9,22	92.12	2102,85	3,36	97,1	992,80
	2	117.15	1,60	98,63	462,30	0,97	99,1	387,53
4x4	1	128.22	34,74	72,9	8671,97	14,41	88,7	4949,65
	2	128.22	5,95	95,3	2379,11	3,57	97,2	1967,83
	3	128.22	2,87	97,76	1537,10	2,36	98,1	1745,13
	4	128.22	3,55	97,27	2222,30	3,17	97,5	2729,41
6x6	1	119.48	49,61	58,47	14525,16	35,21	70,5	13802,32
	2	119.48	24,33	79,63	11059,08	14,48	87,8	9619,57
	3	119.48	9,38	92,14	7643,76	6,86	94,2	8317,24
	4	119.48	6,71	94,4	7376,39	6,24	94,7	10334,29
	5	119.48	7,32	93,8	9276,27	7,11	94	13801,85
	6	119.48	8,44	92,8	10829,25	8,26	93,1	16329,50
8x8	1	134.42	70,53	47,5	20539,18	61,38	54,3	23998,05
	2	134.42	46,50	65,4	23741,51	33,86	74,8	25527,80
	3	134.42	27,35	79,6	29022,10	17,44	87	28260,42
	4	134.42	14,64	89,1	24971,08	11,46	91,4	29982,51
	5	134.42	11,10	91,7	25216,00	9,97	92,5	35227,37
	6	134.42	13,67	89,8	37270,91	13,45	89,9	58003,30
	7	134.42	15,70	88,3	45863,81	15,59	88,4	72697,34
	8	134.42	14,89	88,9	44589,42	14,79	88,9	70755,54
11x11	1	144.14	83,75	41,9	21557,25	79,19	45,1	27818,50
	2	144.14	68,08	52,7	37041,99	58,83	59,2	48572,11
	3	144.14	56,75	60,6	68063,11	44,33	69,2	84999,45
	4	144.14	36,84	74,4	78724,50	26,62	81,5	89771,03
	5	144.14	25,91	82	88097,55	20,80	85,5	112173,61
	6	144.14	22,93	84,1	111291,33	20,33	85,9	158905,38
	7	144.14	21,66	84,9	116659,14	20,89	85,5	191785,35
	8	144.14	24,27	83,2	153230,30	24,07	83,3	260083,33
	9	144.14	27,53	80,9	220648,41	27,44	80,9	358625,85
	10	144.14	26,80	81,4	225854,32	26,74	81,4	373368,61

Table 6.5 Standard deviation, unbalance reduction percentage and trade-off between balance degree and time for torus topologies and for all possible d_s at different load-balancing steps.

Let us analyse in more detail what occurs with the torus topology. The torus topology exhibits a low connectivity degree because the number of directed connected processors remains constant as the system size increases. Therefore, the size of the domain rises slowly, increasing the number of iterations needed to significantly reduce the initial unbalance as well as the time incurred for that. However, the proposal d_s value exhibits the best unbalance reduction for all system sizes, excluding 11x11 torus, without a substantially increase in the trade-off factor at step 4. At step 7 the unbalance reduction is very close, as in step 4, but the trade-off factor is increased by more than 40%, which supposes a very high increment in load-balancing time. Although this penalty is not so important for hypercube topology, step 4 also exhibits the best compromise between balance reduction and time. Therefore, iterating the load-balancing process 4 times by using the proposal optimal d_s, the unbalanced reduction obtained will be more than 90% in most cases.

6.5 Conclusions to this chapter

We now summarise the main conclusions that arise from the above study. In scenarios where the best final balance degree is desired, the best choice is to execute the original DASUD algorithm until it finishes. Typical frameworks that suit this choice well are the execution of parallel applications such as Image thinning, N-body, Gaussian reduction, Branch&Bound,... where the load units can be expressed in terms of data. These applications are in need of synchronisation in some particular execution times. These points are good time candidates to execute the load-balancing process for evenly distributing the data. The load-balancing process should be executed simultaneously in all processors with the aim of achieving the best balance degree.

In contrast, frameworks where load units can be identified with independent processes without dependency restrictions best fit the second approach, where no termination detection technique is needed, but the number of balance steps and the domain scope should be set before starting, depending on the underlying topology and system size. In particular, for hypercube topology, the best d_s is equal to 3, whereas for torus topology the best d_s depends on the underlying diameter topology in the following way:

$$d_s = \frac{d}{2} + 1$$

However, for both topologies, in order to obtain a balance improvement larger than 90% with respect to the initial load imbalance, it is enough to execute the load-balancing process no more than 10 times.

Chapter 7

Conclusions and future work

Abstract

This chapter presents the conclusions obtained from this thesis and the current work and work plan to be undertaken in the future in order to continue research on load-balancing algorithms.

7.1 Conclusions and main contributions

After working for the last few years on the development of this thesis, one has to think about what the initial goals were, and what degree of achievement has been attained for them. We shall now describe each of the main goals in this thesis and how each one of them has been satisfactory achieved.

First of all, our work was aimed at developping an overview of the dynamic load-balancing problem in parallel computing by summarising the main issues that must be considered in this problem. The exhaustive analysis of the state of the art in load-balancing algorithms led to the elaboration of a load-balancing algorithm taxonomy where algorithmic design aspects, as well as generic implementation features of load-balancing strategies are considered. This load-balancing classification is included in the following book:

[Sen00] M.A.Senar, A. Cortés, A. Ripoll et alter.
Chapter 4: Dynamic Load Balancing,
Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments, editors José C. Cunha, Peter Kacsuk & Stephen C. Winter, Ed. Nova Science. (to be published)

The next goal was to develop a dynamic load-balancing algorithm that drives any initial load distribution in parallel system into the even load distribution by treating load as indivisible units. We focused on the development of a totally distributed load-balancing algorithm,because it arose as the most popular approach in the literature, as opposed to centralised schemes. In particular, we were interested in nearest-neighbour strategies for their apparent simplicity, effectiveness and scalability. Our preliminary proposal of load-balancing algorithm can be found in:

[Luq94] E.Luque, A.Ripoll, A. Cortés and T.Margalef, *A Distributed Diffusion Method for Dynamic Load-Balancing on Parallel Computers*, Actas de las V Jornadas de Paralelismo, (Malaga) Spain, September 1994, pp. 74-85.

[Luq95] E. Luque, A. Ripoll, A. Cortés, T. Margalef, *A Distributed Diffusion Method for Dynamic Load Balancing on Parallel Computers*, Proceedings of the Euromicro Workshop on Parallel and Distributed Processing, January 1995, pp. 43-50.

This preliminary DASUD proposal incorporated two important features. One of them was its ability to detect most of the local unbalanced load distributions and to arrange them by only performing local load movements, and the other one, lay in its asynchronous implementation. Later, a more accurate analysis of these problems led to the incorporation of actions complementary to DASUD, facilitating load movements between non-directed connected processors. With the incorporation of these actions DASUD was always able to reach optimal local load distributions and, consequently in most of the cases, global balanced load distributions were achieved.

The analysis of the balancing problems exhibited by discrete nearest-neighbour load-balancing algorithms can be found in the following publications:

[Cor97] A. Cortés, A. Ripoll, M. A. Senar and E. Luque, *Dynamic Load Balancing Strategy for Scalable Parallel Systems*, Parallel Computing: Fundamentals, Applications and New Directions (ParCo97), 1998 Elsevier Science B.V., pp.735-738.

[Rip98] A. Ripoll, M.A. Senar, A. Cortés and E. Luque, *Mapping and Load-Balancing Strategies for Parallel Programming*, Computers and Artificial Intelligence, Vol. 17, Nº 5, 1998, pp. 481-491.

Since DASUD was executed in an iterative way, one of the most important issues that arises was to ensure that the load-balancing process provides a final stable load distribution, i.e., the algorithm globally converges. As far as we are aware, there is no proof, in the literature, that demonstrates the convergence of realistic iterative load-balancing algorithms. For this reason, a new important goal arose: *to describe DASUD in a formal way that allows us to prove its convergence*. This goal was completely accomplished. Furthermore, having a formal description of DASUD allowed us to derive theoretical upper bounds for the final balance degree achieved by DASUD and for the number of iterations of the algorithm needed to obtain it. The formal description of DASUD, as well as a sketch of its convergence proof were reported in the following publication:

[Cor98] A. Cortés, A. Ripoll, M. A. Senar, F.Cedó and E. Luque, *On the Stability of a Distributed Dynamic Load Balancing Algorithm*, IEEE Proc. Int'l Conf. Parallel and Distributed Systems (ICPADS), 1998, pp. 435-446.

The full convergence demonstration of DASUD is provided in the following technical report:

[Cor98e] A.Cortés, A. Ripoll, M. A. Senar, F.Cedó and E.Luque, *On the convergence of SID and DASUD load-balancing algorithms*, PIRDI-8/98, Technical Report, Computer Science Department, *Universitat Autònoma de Barcelona*, 1998. http://pirdi.uab.es/

The proposed algorithm was compared by simulation to three well-known load-balancing algorithms within the nearest-neighbour category: SID (Sender-Initiated Diffusion), GDE (Generalised Dimension Exchange) and AN (Average Neighbourhood). For this purpose, a simulation environment was developed where different load-balancing strategies could be simulated under the same conditions. This load-balancing simulator was designed to easily change parameters such as topology (hypercube and torus), system size (from 8 to 128 processors) and initial load distribution. In particular, the set of initial load distributions simulated included situations which vary from load distributions that exhibit a light initial imbalance to load distributions with a high initial unbalance degree.

The comparative analysis was performed in terms of stability and efficiency. On the one hand, stability was evaluated by measuring the *dif_max* and σ (global standard deviation) quality metrics. On the other hand, the number of simulation steps needed to reach the final stable load distribution (*steps*) and the quantity of load movements incurred in the global load-balancing process (*load units – u -)* was evaluated to measure efficiency. The whole comparative study was performed for each quality metric and, for all of them, the influence of the initial load distribution pattern, system size and the shape of the initial load distribution were individually considered. The results show that our algorithm obtains the best trade-off between the final balance degree and the time incurred to achieve it.

This simulation environment also provided a framework to experimentally validate the theoretical upper bound for the final balance degree achieved by DASUD, as well as the upper bound for the number of iterations needed to coerce the system into a stable state.

The most relevant results from this experimental evaluation have been published in:

[Cor98b] A. Cortés, A. Ripoll, M. A. Senar, P.Pons and E. Luque, *Un algoritmo para balancear dinámicamente las tareas de un programa en sistemas paralelos*, Actas del IV Congreso Argentino Internacional de Ciencias de la Computación (CACIC'98), 1998, pp. 707-721.

[Cor98c] A.Cortés, A. Ripoll, M. A. Senar and E.Luque, *Evaluation of the Balancing Qualities of the DASUD algorithm*, Actas de las IX Jornadas de Paralelismo, San Sebastian (Spain), September 1998, pp. 381-388.

[Cor98d] A.Cortés, A. Ripoll, M. A. Senar and E.Luque, *Performance Comparison of Dynamic Load-Balancing Strategies for Distributed*, PIRDI-9/98, Technical Report, Computer Science Department, *Universitat Autònoma de Barcelona*, 1998. http://pirdi.uab.es/

[Cor99] A. Cortés, A. Ripoll, M. A. Senar and E. Luque, *Performance Comparison of Dynamic Load-Balancing Strategies for Distributed Computing*, IEEE Proc. Hawaii Int'l Conference on Systems Sciences (HICSS-32), 1999

[Cor99b] A. Cortés, A. Ripoll, M. A. Senar, P. Pons and E. Luque, *On the Performance of Nearest-Neighbours Load Balancing Algorithms in Parallel Systems*, IEEE Proc. of the Seventh Euromicro Workshop on Parallel and Dsitributed Processing (PDP'99), 1999, pp. 170-177.

Once the goodness of DASUD was established with respect to other algorithms in the literature, we focused on studying different aspects of DASUD. In particular, we experimentally tested the scalability of DASUD with respect to problem size and also the system size. From the experiments performed with this aim, we can conclude that DASUD is well-scalable, exhibiting the same behaviour for large sized systems as well as for large problem sizes, with only a slight dependence on the topology's diameter being shown.

Finally, the DASUD's ability to always reache a local balance load distribution by only considering load information from its immediate neighbours caused us to raise the issue of what would happen if we increased the DASUD scope so as to provide it, as near neighbours, not only with processors at distance 1 from any processor, but also those at distance 2 or 3, and so on.... We analysed this effect of enlarging the locality scope of the algorithm, and an extended version of the original DASUD was proposed (d_s-DASUD). Although we intuitively sensed that disposing of more load information would produce the best results, from the experimental study we concluded that the increase in communication costs incurred in collecting load information and transferring load beyond immediate neighbours, as well as the increment in the computational time of the load-balancing algorithm, degrades the response rate of DASUD.. Therefore, under execution environments where perfect balance distribution is needed (maximum load difference in the system should be smaller or equal to one load unit), the original DASUD algorithm where the domain of each processor is restricted to the direct neighbourhood obtains the best trade-off between the final balance degree and the time spent in attaining it with respect to any d_s-DASUD. However, in execution environments where it is sufficient to highly reduce initial load imbalance, the extended version of DASUD provides better results if the load-balancing process is stopped before reaching its conclusion. In particular, we have observed that for decreasing the original imbalance more than 90%, it is enough to iterate the load-balancing process 10 times. The d_s values that exhibit this ability are 3 for hypercube topologies, and half the diameter plus one for torus. For example, for a 4x4 torus topology whose diameter is equal to 4, the best d_s corresponds to 3 and, for a 6x6 torus we will chose d_s equal to 4. These experiments are currently in process of publication.

7.2 Current and future work

From the experience obtained throughout the development of this work, as explained in this thesis, new ideas have emerged, some of which are practically concluded, whilst others are still being worked on. We now outline all current and future lines of work, as well as their current degree of development.

An important contribution deriving from of this work, is the development of a general convergence proof for realistic iterative and totally distributed load-balancing algorithms. DASUD's convergence proof provides the basis for this new demonstration. A general model for realistic load-balancing algorithms has been developed and the convergence of this realistic load-balancing model is proved. As far as we are aware, it is the first convergence proof for load-balancing algorithms that treat loads as indivisible. Therefore, since most of the realistic strategies from the literature fit well with this load-balancing model, their convergence is, in this way, fully established. This proof is reported in:

[Ced00] F.Cedó, A. Ripoll, A.Cortés, M. A. Senar and E.Luque, *On the convergence of distributed load-balancing algorithms with integer load*, PIRDI-2/00, Technical Report, Computer Science Department, *Universitat Autònoma de Barcelona*, 2000. http://pirdi.uab.es/ (submitted to SIAM Journal of Computing).

The load-balancing algorithm proposed in this work, DASUD, has been validated by simulation for a wide set of initial load distributions that have been considered as static load situations, where load is neither created nor destroyed during the load-balancing process. Therefore, in order to evaluate the impact of application dynamicity on different load-balancing algorithms, some changes should be introduced in the current load-balancing simulator. For this purpose, the load-balancing simulator used in the experimental study described in this thesis has been changed to incorporate the capacity of updating the load values for each individual processor during the load-balancing simulation process.

Given the theoretical nature of this work, an immediate challenge was to carry out a real implementation of our DASUD algorithm. Currently, a preliminary real asynchronous version of DASUD has been developed. This real implementation was implemented on a cluster of PC's under Linux. In order to be able to execute DASUD for different interconnection patterns (hypercube, torus ...) a mechanism to logically configure the underlying platform such as different interconnections schemes is provided. As a result of this work the following degree project is derived [Her00].

Once DASUD has been implemented in a real platform, the following step should be to incorporate DASUD to a real application whose parallel execution in a multiprocessor environment generates a significant computational unbalance between processors as the application computation progresses.

Currently, we are in a process that joins the DASUD load-balancing algorithm with a parallel version of the *image thinning* application [Pla00]. Since this is an application with data parallelism, the observed imbalance is caused by the data distribution throughout the processors.

We recall that in chapter 1 of this thesis we introduce the idea that to design a load-balancing strategy three functional blocks should be considered: the *Load Manager block,* the *Load-Balancing Algorithm block* and the *Migration Management block.* In this thesis we focused on the development of a Load-Balancing Algorithm (DASUD) by skipping the other blocks. However, in execution environments where load consists of independent task, DASUD could be applied directly. In this case, it would simply require having a tool to facilitate moving tasks between processors. But, *what it happen when the underlying parallel application is defined as a set of co-operating tasks, i.e., when there are dependency relations between the tasks*?. A new line of work arises from this question, whose principal objective is to provide a framework for distribution strategies to obtain the necessary data in estimating costs and profits from possible migrations, and then decide whether or not to migrate such tasks.

References

[Ahm91] Ishfaq Ahmad and Arif Ghafoor, Semi-Distributed Load Balancing For Massively Parallel Multicomputer Systems, IEEE Transactions on Software Engineering, Vol. 17, No. 10, October 1991, pp. 987-1004.

[Ara96] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey and P. Stephan, Dome: parallel programming in a distributed computing environment. In Procc. of the International Parallel Processing Symposium (IPPS-96), 1996, pp. 218-224.

[Bak96] M.A. Baker, G.C. Fox and H.W. Yau, A Review of Commercial and Research Cluster Management Software, Northeast Parallel Architectures Center, Syracuse University, Technical Report, June 1996

[Bar90] C. Barmon, M.N. Faruqui and G.P. Battacharjee, Dynamic Load Balancing Algorithm in a Distributed System, Microprocessing and Microprogramming 29, 1990/91, pp. 273-285.

[Bau88] K. M. Baumgartner and R. Kling and B. Wah. A Global Load Balancing Strategy for a Distributed System Proc. Int. Conf. on Future Trends in Distr. Comp. Syst., pp. 93-102, 1988.

[Bau95] Joey Baumgartner, Diane J. Cook, Behrooz Shirazi. Genetic Solutions to the Load Balancing Problem International Conference on Parallel Processing ICPP Workshop on Challenges for Parallel Processing",, pp. 72-81, CRC Press, August 1995.

[Ber89] D.P. Bertsekas and J. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Boi90] J.E. Boillat, Load Balancing and Poisson Equation in a Graph, Concurrency: Practice and Experience, Vol. 2(4), December 1990, pp. 289-313.

[Bru99] R.K. Brunner and L.V. Kaé, Handling application-induced load imbalance using parallel objects, Technical Report 99-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999.

[Cha95] Hua Wu David Chang and William J. B. Oldham. Dynamic task allocation models for large distributed computing systems. IEEE Transactions on Parallel and Distributed Systems, 6(12), pp. 1301-1315, December 1995.

[Cas95] J. Casas et alter, MPVM: A migration transparent version of PVM, Technical Report CSE-95-002, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, February, 1995.

[Ced00] F.Cedó, A. Ripoll, A.Cortés, M. A. Senar and E.Luque, On the convergence of distributed load-balancing algorithms with integer load, PIRDI-2/00, Technical Report Computer Science Department, Universitat Autònoma de Barcelona, 2000. http://pirdi.uab.es/

[Cha85] K.M. Chandy and L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Trans. Comput. Syst. Vol. 3, 1985, pp. 63-75.

[Cor96] Antonio Corradi, Letizia Leonardi, Diffsusive Algorithms for Dynamic Load Balancing in Massively Parallel Architectures, DEIS Technical Report Nº DEIS-LIA-96-001, LIA Series Nº 8, April 1996. http://www-lia.deis.unibo.it/Research/TechReport.html

[Cod97] Codine: Computing in Distributed Networked Environments, GENIAS Software, URL: /www.genias.de/genias/english/codine.html, 1997.

[Cor97] A. Cortés, A. Ripoll, M. A. Senar and E. Luque, Dynamic Load Balancing Strategy for Scalable Parallel Systems, Parallel Computing: Fundamentals, Applications and New Directions (ParCo97), 1998 Elsevier Science B.V., pp.735-738.

[Cor98] A. Cortés, A. Ripoll, M. A. Senar, F.Cedó and E. Luque, On the Stability of a Distributed Dynamic Load Balancing Algorithm, IEEE Proc. Int'l Conf. Parallel and Distributed Systems (ICPADS), 1998, pp. 435-446.

[Cor98b] A. Cortés, A. Ripoll, M. A. Senar, P.Pons and E. Luque, Un algoritmo para balancear dinámicamente las tareas de un programa en sistemas paralelos, Actas del IV Congreso Argentino Internacional de Ciencias de la Computación (CACIC'98), 1998, pp. 707-721.

[Cor98c] A.Cortés, A. Ripoll, M. A. Senar and E.Luque, Evaluation of the Balancing Qualities of the DASUD algorithm, Actas de las IX Jornadas de Paralelismo, San Sebastian (Spain), September 1998, pp. 381-388.

[Cor98d] A.Cortés, A. Ripoll, M. A. Senar and E.Luque, Performance Comparison of Dynamic Load-Balancing Strategies for Distributed Computing, PIRDI-9/98, Technical Report Computer Science Department, Universitat Autònoma de Barcelona, 1998. http://pirdi.uab.es/

[Cor98e] A.Cortés, A. Ripoll, M. A. Senar, F.Cedó and E.Luque, On the convergence of SID and DASUD load-balancing algorithms, PIRDI-8/98, Technical Report Computer Science Department, Universitat Autònoma de Barcelona, 1998. http://pirdi.uab.es/

[Cor99] A. Cortés, A. Ripoll, M. A. Senar and E. Luque, Performance Comparison of Dynamic Load-Balancing Strategies for Distributed Computing, IEEE Proc. Hawaii Int'l Conference on Systems Sciences (HICSS-32), 1999

[Cor99b] A. Cortés, A. Ripoll, M. A. Senar, P. Pons and E. Luque, On the Performance of Nearest-Neighbours Load Balancing Algorithms in Parallel Systems, IEEE Proc. of the Seventh Euromicro Workshop on Parallel and Dsitributed Processing (PDP'99), 1999, pp. 170-177.

[Cor99c] Antonio Corradi, Letizia Leonardi and Franco Zambonelli, Diffusive Load-Balancing Policies for Dynamic Applications, IEEE Concurrency Parallel Distributed & Mobile Computing, January-March 1999, pp. 22-31.

[Cyb89] George Cybenko, Dynamic Load Balancing for Distributed Memory Multiprocessors, J.Parallel Distributed Compt. 7, 1989, pp. 279-301.

[Dan97] S.P. Dandamudi and M. Lo, "A Hierarchical Load Sharing Policy for Distributed Systems", IEEE Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomm. Systems (MASCOTS), Haifa, Israel, 1997, pp. 3-10.

[Die99] Ralf Diekmann, Andreas Frommer, Burkhard Monien, Effiecient schemes for nearest neighbors load balancing, Parallel Computing, 25, (1999), pp. 789-812.

[Dij83] E.W. Dijkstra, W.H.J. Feijen and A.J.M. van Gasteren, Derivation of a termination algorithm for distributed computations, Inform. Processing Lett. Vol. 16, 1983, pp. 217-219.

[Dim98] B. Dimitrov and V. Rego, Arachne: a portable threads system supporting migrant threads on heterogeneous network farms, IEEE Trans. On Parallel and Distrib. Syst., Vol. 9(5), pp. 459- 469, 1998.

[Dou91] F. Douglis and J. Ousterhout, Transparent process migration: design alternatives and the Sprite implementation, Software – Practice and Experience, 21 (8), pp. 757-785, August, 1991.

[Eag85] Derek L. Eager, Edwuard D. Lazowska and John Zahorjan, A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing, ACM SIGMETRICS, Conference on Measurement and Modelling of Computer Systems, 1985, pp. 1-3.

[Eri88] O. Eriksen, A termination detection protocol and its formal verification, Journal of Parallel and Distributed Computing, 5, 1988, pp. 82-91.

[Eva94] D.J. Evans, W.U.N. Butt, Load Balancing with Network Partitioning Using Host Groups, Parallel Computing 20 (1994), pp. 325-345.

[Fer87] Ferrari D. and Zhou S.,An empirical investigation of load indices for load balancing applications, Proc. of Performance' 87, pp 515-528, 1987.

[Fio78] S. Fiorini and R.J. Wilson, Edge-coloring of graphs, In L. W. Beineke and R.J. Wilson editors, Selected Topics in Graph Theory, Academic Press 1978, pp. 103-125.

[Fox89] G.C. Fox, W.Furmanski,J.Koller and P.Simic, Physical optimization and load balancing algorithms, In Proceedings of Conference on hypercube Concurrent Computers and Applications, pp. 591-594, 1989.

[Fra82] N. Francez and M. Rodeh, Achieving distributed termination without freezing, IEEE Trans. Software Eng. Vol. SE-8, May 1982, pp. 287-292.

[Fra99] D. Franco, I. Garcés and E. Luque, "A new method to make communication latency uniform: Distributed Routing Balancing", Proc. of ACM International Conference on Supercomputing (ICS99), 1999, pp. 210-219.

[Her00] Jaime Herrero Sánchez, Diseño e implementación de un algoritmo asíncrono de balanceo de carga en un sistema distribuido, Enginyeria Superior en Informàtica (E.T.S.E),Universitat Autònoma de Barcelona, Sep. 2000.

[Hor93] G. Horton, A Multi-Level Diffusion Method for Dynamic Load Balancing, Parallel Computing 19, 1993, pp. 209-218.

[Hos90] S.H. Hosseini, B. Litow, M.Malkawi, J.McPherson, and K. Vairavan, Analysis of a Graph Coloring Based Distributed Load Balancing Algorithm, Journal of Parallel and Distributed Computing 10, 1990, pp. 160-166.

[Hu99] Y.F. Hu, R.J. Blake, An improved diffusion algorithm for dynamic load balancing, Parallel Computing 25 (1999), pp. 417-444.

[IBM93] IBM. IBM LoadLeveler: General information. IBM, September 1993.

[Jon97] J.P. Jones and C. Brickell, Second Evaluation of Job Queieng/Scheduling Software: Phase 1 Report, Nasa Ames Research Center, NAS Tech. Report NAS-97-013, June, 1997.

[Kal88] L.V. Kale, Comparing the Performance of Two Dynamic Load Distribution Methods, Proceeding of the 1988 International Conference on Parallel Processing, Vol. 1, pp. 8-12.

[Kal96] L.V. Kalé and S. Krishnan, Cham++: Parallel programming with message-driven objects, in Gregory V. Wilson and Paul Lu, editors, Parallel Programming using C++, pp. 175-213, MIT Press, 1996.

[Kon97] R.B. Konuru, S.W. Otto and J. Walpole, A Migratable User-level Process Package for PVM, Journal of Parallel and Distributed Computing, 40, pp. 81-102, 1997.

[Kum92] D. Kumar, Development of a class of distributed termination detection algorithms, IEEE Trans. Knowledge and Data Eng. Vol. 4, Nº 2, April 1992, pp. 145-155.

[Kun91] Kunz T., The influence of different workload descriptions on a heuristic load balancing scheme, IEEE Trans. on Software Engineering, 17 (7) pp 725-730,1991

[Lin87] Frank C. H. Lin and Robert M.keller, The Gradient Model Load Balancing Method, IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January 1987, pp. 32-38.

[Lin92] Hwa-Chun Lin and C. S. Raghavendra, A Dynamic Load-Balancing Policy with a Central Job Dispatcher (LBC), IEEE Transactions on Software Engineering, Vol. 18, No. 2, February 1992, pp. 148-158.

[Lül91] R. Lüling B. Monien, and F. Ramme, Load balancing in large networks: A comparative study. In Proceedings of 3th. IEEE symposium on parallel and distributed processing, pp. 686-689, December 1991.

[Luq94] E.Luque, A.Ripoll, A. Cortés and T.Margalef, A Distributed Diffusion Method for Dynamic Load-Balancing on Parallel Computers, Actas de las V

Jornadas de Paralelismo, Las Alpujarras (Malaga) Spain, September 1994, pp. 74-85.

[Luq95] E. Luque, A. Ripoll, A. Cortés, T. Margalef, A Distributed Diffusion Method for Dynamic Load Balancing on Parallel Computers, Proceedings of the Euromicro Workshop on Parallel and Distributed Processing, January 1995, pp. 43-50

[Mas96] E. Mascarenhas and V. Rego, Ariadne: Architecture of a portable threads system supporting thread migration, Software practice and Experience, vol. 26(3), pp. 327-357, 1996.

[Mat87] F. Mattern, Algorithms for distributed termination detection, Distributed Computing, 2, 1987, pp. 161-175.

[Mil93] D.S. Milojicic, W. Zint, A. Dangel and P. Giese, Task migration on top of the Mach microkernel, in Mach III Symposium Proceedings, pp. 273-289, Santa Fe, New Mexico, April, 19-21, 1993.

[Mun95] F.J. Muniz, E.J. Zaluska, Parallel load-balancing: An extension to the gradient model, Parallel Computing, 21 (1995), pp. 287-301.

[Mut98] S.Muthukrishnan, B. Ghosh and M.H. Schultz, First- and Second-Order Diffusive Methods for Rapid, Coarse, Distributed Load Balancing, Theory of Computing Systems, 31, 1998, pp. 331-354.

[Mur97] Tina A. Murphy and John G. Vaughan, On the Relative Performance of Diffusion and Dimension Exchange Load Balancing in Hypercubes, Procc. of the Fifth Euromicro Workshop on Parallel and Distributed Processing, PDP'97, January 1997, pp. 29-34.

[Ni93] L.M. Ni and P.K. McKinley, "A survey of wormhole routing techniques in direct-networks", IEEE Computer 26 (2), 1993, pp. 62-76.

[Ove96] B.J. Overeinder, P.M.A. Sloot, R.N. Heederick and L.O. Hertzberger, A dynamic load balancing system for parallel cluster computing, Future Generation Computer Systems, 12(1), pp. 101-115, May, 1996.

[Pap84] C. Papadimitrious, Computational Complexity, Addison-Wesley, 1994.

[Pet98] S. Petri, M. Bolz and H. Langendörfer, Migration and rollback transparency for arbitrary distributed applications in workstation clusters, Proc. of Workshop on Run-Time Systems for Parallel Programming, held in conjunction with IPPS/SPDP'98, 1998.

[Pla00] Mercedes Planas Sánchez, Diseño e implementación de una aplicación paralela en un sistema distribuido: Algoritmo de Thinning, Enginyeria Superior en Informàtica (E.T.S.E),Universitat Autònoma de Barcelona, Sep. 2000.

[Pru95] J. Pruyne and M. Livny, Providing resource management services to parallel applications, in J. Dongarra and B. Tourancheau de., 2nd Workshop on Environments and Tools for Parallel Scientific Computing, pp. 152-161, 1995.

[Ran83] S.P. Rana, *A distributed solution of the distributed termination problem*, Inf. Process. Letters, Vol. 17, 1983, pp. 43-46.

[Ran88] S. Ranka, Y.Won and S. Sahni, Programming a hypercube multicomputer, IEEE Software, 5, September 1988, pp. 69-77.

[Rip98] A. Ripoll, M.A. Senar, A. Cortés and E. Luque, Mapping and Load-Balancing Strategies for Parallel Programming, Computers and Artificial Intelligence, Vol. 17, Nº 5, 1998, pp. 481-491.

[Rön90] S. Rönn and H. Haikkonen, Distributed termination detection with counters, Information Processing Letters, Vol 34, 1990, pp. 223-227.

[Rus99] S.H. Russ et alter, Hector: An Agent-Based Architecture for Dynamic Resource Management, IEEE Concurrency, pp. 47-55, April-June, 1999.

[Sal90] Vikram A. Saletore, A Distributed and Adaptive Dynamic Load Balancing Scheme for Parallel Processing of Medium-Grain Tasks, In Proc. of the 5th Distributed Memory Comput. Conf., pp. 994-999, 1990.

[San94] J. Sang G. Peters and V. Rego, Thread migration on heterogeneous systems via compile-time transformations, Proc. Int'l Conf. Parallel and Distributed Systems (ICPADS), pp. 634-639, 1994.

[Sav96] Serap A. Savari, Dimitri P. Bertsekas, Finite Termination of Asynchronous Iterative Algorithms, Parallel Computing, vol.22, 1996, pp. 39-56.

[Sen00] M.A.Senar, A. Cortés, A. Ripoll et alter.
Chapter 4: Dynamic Load Balancing,
Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments, editors José C. Cunha, Peter Kacsuk & Stephen C. Winter, Ed. Nova Science. (to be published)

[Shi89b] Y. Shih and J. Fier, Hypercube systems and key applications, In K. Hwang and D. Degroot, editors, Parallel Processing for Supercomputers and Artificial Intelligence, McGraw-Hill Publishing Co. 1989, pp. 203-243.

[Shu89] Shu W. and Kale L.V. , A dynamic scheduling strategy for the Chare-kernel system, In Proceedings of Supercomputing 89, November 1999.

[Sil94] L. Silva, B. Veer and J. Silva, Checkpointing SPMD applications on transputer networks, Procc. of the Scalable High Performance Computing Conference, pp. 694-701, 1994.

[Sin97] P. K. Sinha, Distributed Operating Systems. Concepts and Design, IEEE Press, 1997.

[Smi97] P. Smith and N. C. Hutchinson, Heterogeneous Process Migration: The Tui System, Tech. Rep., Department of Computer Science, University of British Columbia, March, 14, 1997.

[Son94] Jianjian Song, A Partially Asynchronous and Iterative Algorithm for Distributed Load Balancing, Parallel Computing 20 (1994), pp. 853-868.

[Sta84] J.A. Stankovic and I.S. Sidhu, An adaptive bidding algorithm for processes, clusters and distributed groups. In Proceedings of 4th. International Conference on Distributed Computer Systems, pp. 49-59, May 1984.

[Ste95] B. Steensgaard and E. Jul, Object and native code thread mobility among heterogeneous computers, Proc. ACM Symp. Operating Systems Principles, pp. 68-78, 1995.

[Str98] V. Strumpen and B. Ramkumar, Portable Checkpointing for Heterogeneous Architectures, in Fault-Tolerant Parallel and Distributed Systems, Eds. Dimiter R. Avresky and David R. Kaeli, chapter 4, pp. 73-92, Kluwer Academic Press, 1998.

[Sub94] Raghu Subramain, Isaac D. Scherson, An Analysis of Diffusive Load-Balancing, In Proceedings of 6th ACM Symposiummon Parallel Algorithms and Architectures, 1994.

[Szy85] B. Szymanski, Y. Shi and S. Prywes, Synchronized distributed termination, IEEE Transactions on Software Engineering, SE-11(10), October 1985, pp. 1136-1140.

[Tan95] T. Tannenbaum and M. Litzkow, The Condor distributed processing system, Dr. Dobb's Journal, pp. 40-48, 1995.

[The85] M. M. Theimer, K. A. Lantz and D. R. Cheriton, Preemptable remote execution facilities for the V System, Proceedings of the 10th ACM Symposium on Operating Systems Principles, Pp. 2-12, Oscas Islands, Washington, December 1-4, 1985.

[Top84] R.W. Topor, Termination detection for distributed computations, Inform. Process. Lett. Vol.18, 1984, pp. 33-36.

[Wat98] Jerrell Watts and Stephen Taylor, A Practical Approach to Dynamic Load Balancing, IEEE Transaction on Parallel and Distributed Systems, vol. 9, No. 3, March 1998, pp. 235-248.

[Wil93] Marc H. Willebeek-LeMair, Anthony P. Reeves, Strategies for Dynamic Load Balancing on Highly Parallel Computers, IEEE Transactions on Parallel and Distributed Systems, vol. 4, No. 9, September 1993, pp. 979-993

[Wu96] Min-You Wu and Wei Shu, The Direct Dimension Exchange Method for Load Balancing in k-ary n-cubes, IEEE Symposium on Parallel and Distributed Processing, October 1996, pp. 366-369.

[Xu92] C. Z. Xu and F. C. M. Lau, Analysis of the Generalized Dimension Exchange Method for Dynamic Load Balancing, Journal of Parallel and Distributed Computing, 16, 1992, pp. 385-393.

[Xu93] C.-Z. Xu and F.C.M. Lau, Optimal Parameters for Load Balancing Using the Diffusion Method in k-ary n-Cube Networks, Information Processing Letters 47,1993, pp.181-187.

[Xu94] Chengzhong Xu and Francis C. M. Lau, Iterative Dynamic Load Balancing in Multicomputers, Journal of Operational Research Society, Vol. 45, N° 7, July 1994, pp. 786-796

[Xu95] Cheng-Zhong Xu and Francis, C. M. Lau, The Generalized Dimension Exchange Method for Load Balancing in k-ary n-Cubes and Variants, Journal of Parallel and Distributed Computing 24, 1995, pp.72-85.

[Xu97] Chengzhong Xu, Francis C. M. Lau, Load Balancing in Parallel Computers. Theory and Practice, Kluwe Academic Publishers, 1997.

[Yua90] Shyan-Ming Yuan, An Efficient Periodically Exchanged Dynamic Load-Balancing Algorithm, International Journal of Mini and Microcomputers, Vol. 12, No. 1, 1990, pp. 1-6.

[Zho88] Songnian Zhou, A Trace-Driven Simulation Study of Dynamic Load Balancing, IEEE Transactions oon Software Engineering, Vol. 14, No. 9, September 1988, pp. 1327-1341

[Zna91] T.F. Znati, R.G. Melhem, and K.R. Pruhs, Dilation-based bidding schemes for dynamic load balancing on distributing processing systems, In Proceedings of 6th Distributed Memory Computing Conference, pp. 129-136, April 1991.

Appendix A

DASUD load-balancing algorithm: *experimental and theoretical annexes*

In chapter 3 a full description of DASUD's behaviour and a theoretical study about it has been performed. From that analysis some theoretical upper bound have been derived for the final the balance degree and the convergence rate of the proposed algorithm. In this appendix, we include an experimental validation of these bounds using the same experimental framework introduced in chapter 4. Finally, a general load-balancing model and its convergence proof are provided.

A.1 Experimental validation of DASUD's final balance degree

Recall that one of the relevant characteristics of DASUD is its ability of searching and balancing unbalanced domains. As the domain of a given processor *i* coincides with its immediate neighbours one can obtain local balance load distributions, but not always globally balanced. However, this fact is controlled by the existence of an upper bound for the maximum global load difference achieved at the end of the load-balancing process which has been provided in chapter 3. This bound preserves DASUD from reaching poor final balanced situations. Recalling from that chapter that this bound is referred as β and it is defined as

$$\beta = floor\left(\frac{d}{2}\right) + 1$$

In this section, we experimentally validate this upper bound for the final balance degree by comparing the final maximum load difference obtained by simulation versus the theoretical value. Firstly, we show in table A.1 the value of the diameter (*d*) for hypercube and torus topologies for all simulated system sizes and the corresponding β value, which has been theoretically evaluated.

Table A.2 shows the maximum value for the maximum load difference obtained by DASUD in all our tests. As can be seen, in the worst case DASUD always achieves a maximum difference lower than the corresponding value of β. This means that, even for highly pathological initial distributions, DASUD is able to obtain a final maximum difference bounded by half of the diameter of the architecture plus one.

n=nº processors		*8/9*		*16*		*32/36*		*64*		*121/128*	
Topol.	*Diam.*	*d*	β	*d*	β	*d*	β	*d*	β	*d*	β
Hyper	*log n*	3	3	4	3	5	4	6	4	7	5
Torus	$2*\lfloor \sqrt{n}/2 \rfloor$	2	2	4	3	6	4	8	5	10	6

Table A.1 Diameter of some topologies and its corresponding β bound.

	Likely distributions						Pathological distributions)					
	% load variation	8/9	16	32/36	64	121/128	idle proc.	8/9	16	32/36	64	121/128
Hyper.	25%	1	1	2	3	3	25%	0	1	2	2	3
	50%	1	1	2	2	3	50%	2	1	2	2	3
	75%	1	1	2	3	3	75%	0	1	2	3	3
	100%	1	1	2	2	3	n-1	2	1	2	3	3
Torus	25%	1	1	2	3	4	25%	1	1	2	3	4
	50%	1	1	2	3	4	50%	1	2	1	2	4
	75%	1	2	2	3	4	75%	1	1	2	2	4
	100%	1	1	2	3	4	n-1	1	1	1	2	4

Table A.2. Maximum dif_max on average for likely and pathological distributions.

Following, a similar experimental validation is reported for the number of iterations needed by DASUD to achieve the final stable load distributions.

A.2 Experimental validation of DASUD's convergence rate

We have seen that DASUD achieves a good global balance degree at the end of the load-balancing process by validating the theoretical results with the ones obtained by simulation. In this section, the same comparison is performed by attending to the "time" needed to reach the final stable situation. Since DASUD is and iterative load-balancing algorithm, we measure this "time" in terms of simulating steps. In chapter 3, two upper bounds for the number of steps needed to complete the load-balancing process are conjectured. We validated only the referred as *Conjecture B* por ser la más precisa de ambas, and its definition is provided by following. Remember that *d* denotes the topology diameter and D_0 is the maximum initial load difference.

$$B = \frac{d}{2} * (D_0 + 1)$$

It was proved by simulation that this bound was attained for all distributions. As an example, we show the data corresponding to situations in which a grater number of steps were consumed, and this is contrasted with theoretical values in tables A.3 and A.4 where the results for likely and pathological load distributions are distributions are shown repectively. On one hand, column *Bound B* from both tables shows the value of the bound obtained according to *Conjecture B*. Each value has been computed using the biggest initial load difference for a given set of distributions with the same load variation and the same number of processors. On the other hand, column *Max. steps* contains the maximum number of steps spent by the DASUD algorithm for that particular distribution.

Likely Distributions											
		8/9		16		32/36		64		121/128	
	% load variation	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*
Hyper.	25%	11	282	14	189	17	119	16	73	11	44
	50%	13	564	16	377	21	236	25	143	19	85
	75%	14	845	17	564	24	354	27	213	24	126
	100%	14	1125	19	752	27	471	30	284	28	164
Torus	25%	10	167	14	189	18	127	19	97	16	66
	50%	11	333	17	377	26	252	28	191	30	128
	75%	11	499	18	564	31	377	35	285	33	190
	100%	12	666	20	750	33	499	40	379	45	252

Table A.3 Maximum number of steps by simulations againts Conjecture B for hypercube and torus topologies using likely initial load distributions

Patological Distributions											
		8/9		16		32/36		64		121/128	
	idle proc.	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*	*Max. steps*	*Bound B*
Hyper.	25%	12	751	15	502	21	315	21	192	18	112
	50%	11	1126	18	752	21	472	27	285	27	168
	75%	15	2251	17	1502	26	940	27	567	33	332
	n-1	12	4500	17	6002	25	7502	32	9003	35	10503
Torus	25%	9	430	13	502	25	339	28	256	31	170
	50%	10	600	17	752	33	504	41	380	49	255
	75%	9	1000	17	1502	38	1005	53	756	67	490
	n-1	11	3001	17	6002	40	9003	59	12004	82	15005

Table A.4 Maximum number of steps by simulations againts Conjecture B for hypercube and torus topologies using pathological initial load distributions

A.3 A realistic load-balancing model

Each processor *i* keeps in its memory an estimate $w_{ij}(t)$ of the load carried by each neighbouring processor and some other non-immediate neighbour processor *j* at time *t*. The load values from neighbour processors are periodically updated because each processor periodically sends information about its load to its neighbours. Furthermore, each processor *i* sporadically receives information about the load of some other non-directed connected processor through its neighbours. Due to communication delays and asynchronism, these estimates can be outdated, and we assume that

$$w_{ij}(t) = w_j(\tau_{ij}(t))$$

where $\tau_{ij}(t)$ is a certain time instant satisfying $0 \le \tau_{ij}(t) \le t$.

As the set of processors whose load values are kept in the memory of a given processor *i* is not a static set because it depends on time, we call it the *temporal domain* and we refer to it as the *t*-domain of the processor *i* at time *t*. Formally, the *t*-domain of processor *i* is defined as

$$D(i,j) = \{j \in i \mid i \textit{ has an estimate of the load of the processor } j \textit{ at time } t\}.$$

Periodically each processor *i* compares its load with the estimate of the load of the processors of its *t*-domain. We say that processor *i* detects its *t*-domain unbalanced at this time *t* if there is $j \in D(i,j)$ such that $\left|w_i(t) - w_{ij}(t)\right| > 1$.In this case, processor *i* transfers a non-negative amount of load, $s_{ij}(t)$, to processor *j*. Note that $s_{ij}(t)$ is an integer variable.

So there is a set of times T_i at which the processor *i* sends information about its load to its neighbours and compares its load with the load of the processors of its *t*-domain, and if it finds that it is overloaded, it transfers some of its loads to some underloaded processor in its *t*-domain by following *assumption* 1.

Assumption 1. *For all non negative integer t and all i,j ∈ P, $s_{ij}(t)$ is a non-negative integer. Furthermore, we have:*

$$s_{ij_0}(t) > 0 \Rightarrow t \in T_i, j_0 \in D(i,t) \text{ and } w_i(t) - \sum_{j \in P} s_{ij}(t) \geq w_{ij_0}(t) + s_{ij_0}(t).$$

This assumption is needed in order to prohibit processor *i* from transferring a very large amount of load and creating a load imbalance in the opposite direction, and precludes the possibility that two processors keep sending load to each other back and forth, without ever reaching equilibrium. More precisely, if processor *i* is an overloaded processor a time *t* and it decides to distribute a portion of its load among several processors belonging to its *t*-domain at that time *t*, then the remaining load of processor *i*, after carrying out those load movements, must be bigger or equal than the load of any of the processors that have received some load.

As we have previously mentioned, Bertsekas and Tsitsiklis [Ber89] divide asynchronous algorithms into two groups: totally asynchronous and partially asynchronous. To paraphrase them, totally asynchronous algorithms "can tolerate arbitrarily large communication and computation delays", but partially asynchronous ones "are not guaranteed to work unless there is an upper bound on those delays". This bound is denoted by a constant *B* called asynchronism measure. We assume the *partially asynchronous* assumption which is described by *assumption* 2.

Assumption 2 (partially asynchronism). *There exists a positive integer B such that:*

(a2) For every i and for every $t \geq 0$,

$$\{t, t+1, \ldots, t+B-1\} \cap T_i \neq 0$$

(b2) For all i and t, and all $j \in D(i,t)$,

$$t - B < \tau_{ij}(t) \leq t$$

(c2) The load $s_{ij}(t)$ sent from processor i to processor j at time $t \in T_i$ is received by processor j before time t+B.

Part (a2) of *assumption* 2 postulates that each processor performs a step of the load balancing process at least once during any time interval of length *B*; *part (b2)* states that the load estimations kept in memory by any processor at a given time *t* were obtained at any time between *t-B* and *t*; and *part (c2)* postulates that load messages will not be delayed more than *B* time units.

Note that synchronous algorithms also fulfil this assumption because they are a particular case of the partially asynchronous one where the *asynchronism measure B* becomes zero.

Finally, *assumption 3* describes the named static situation where no load is created or consumed during the LB process and therefore load is conserved.

Assumption 3. *The total load L of the system is constant. More precisely: Let* $v_{ij}(t)$ *be the amount of load that has been sent from processor i to processor j before time t, but that has not been received by processor j before time t. Let* $r_{ij}(t)$ *be the load received by processor i at time t. Then we have*

$$w_i(t+1) = w_i(t) - \sum_{j \in P} s_{ij}(t) + \sum_{j \in P} r_{ji}(t),$$

$$v_{ij}(t) = \sum_{\tau=0}^{t-1} \left(s_{ij}(\tau) - r_{ij}(\tau)\right),$$

$$v_{ij}(0) = 0,$$

$$\sum_{i=1}^{n} w_i(0) = L$$

and so

$$\sum_{i=1}^{n} \left(w_i(t) + \sum_{j \in P} v_{ij}(t) \right) = L \quad \mathit{for\ all}\ t \geq 1$$

The aim of the rest of this section is to postulate and prove that this iterative distributed load-balancing (IDLB) model is finite. For that purpose *Theorem A.1* is stated and proved.

Theorem A.1.*Under Assumption 1,2 and 3, the load balancing algorithm described is finite. That means that there exists a time* $\bar{t} > 0$ *such that for all* $t \geq \bar{t}$ *and all* $i, j \in P$, $s_{ij}(t) = 0$.

Proof. For notational convenience, we define $w_i(t) = w_i(0)$ for all $t < 0$.

Let

$$m_1(t) = min \quad \{w_i(\tau) \quad | \quad i \in P, \quad t - B < \tau \leq t\}$$

this means that $m_1(t)$ is the minimum load value among the total system at a given interval of time of length *B*. The minimum load value that occupies the *k*-st place if loads are sorted in ascending order at a given interval of time of length *B* is defined as

$$m_k(t) = min \quad \{w_i(\tau) \quad | \quad i \in P, \quad t - B < \tau \leq t \quad w_i(\tau) > m_{k-1}(t)\}$$

for all integer *k>1*.

For notational convenience, we define

$$min\ \varnothing = L+1$$

Let

$$P_k(t) = \{ i \in P \quad | \quad w_i(t) = m_k(t)\}$$

for al integer $k \geq 1$ represents the set of processors whose load value is equal to the minimum load value of order k.

By induction on *k*, we shall prove that there exists an increasing sequence $t_1 < t_2 < t_3 < \ldots$ of positive integers such that for all $k \geq 1$.

(1) $m_k(t) = m_k(t_k)$, $\forall\, t \geq t_k$,

(2) $P_k(t) = P_k(t_k)$, $\forall\, t \geq t_k$,

(3) $s_{ij}(t) = r_{ji}(t) = 0$, $\forall\, t \geq t_k$, $\forall\, i \in P_k(t_k), \forall j \in P$.

The previous three items postulate that there exists a time t_k beyond which the minimum load value of order *k* keeps constant (1), the set of processors holding that load value will not change (2), and no load will be received/sent from/to any of these processors (3). Note that $P = \bigcup_{k\geq 1} P_k(t)$ for all *t*. Since this is a disjoint union and *P* is a finite set, by definition of $P_k(t)$, we see that

$$P = \bigcup_{k=1}^{n} P_k(t)$$

for all *t*. Thus, with $\bar{t} = t_n$, the theorem follows. As the model's convergence is based on the three items outlined above, let us prove each one of them separately. Since the proofs are performed by induction on *k*, we firstly include the proofs of items (1),(2) and (3) for *k=1* and, following, the induction step for *k>1* is provided for each item.

In order to see (1) for *k=1*, we fix a processor *i* and a time *t*. If $s_{ij}(t) = 0$ for all $j \in P$, then

$$w_i(t+1) = w_i(t) + \sum_{j\in P} r_{ji}(t) \geq w_i(t) \geq m_1(t).$$

If $s_{ij_0}(t) > 0$ for some $j_0 \in P$, then

$$\begin{aligned} w_i(t+1) &= w_i(t) - \sum_{j\in P} s_{ij}(t) + \sum_{j\in P} r_{ji}(t) \\ &\geq w_{ij_0}(t) + s_{ij_0}(t) && \text{(by assumption 1)} \\ &= w_{ij_0}(\tau_{ij_0}(t)) + s_{ij_0}(t) \\ &> m_1(t) && \text{(by (b2))} \end{aligned}$$

Thus $m_1(t+1) \geq m_1(t)$ for all *t*. Since $(m_1(t))_{t\geq 0}$ is a non-decreasing sequence of integers and $m_1(t) \leq L$, there exists a positive integer $t_1^{'}$ such that

$$m_1(t) = m_1(t_1^{'}), \qquad \forall\, t \geq t_1^{'}.$$

So (1) follows for *k=1*.

In order to prove (2), we shall see that

$$P_1(t_1^{'}) \supseteq P_1(t_1^{'}+1) \supseteq P_1(t_1^{'}+2) \supseteq \ldots \qquad (*)$$

Let $t \geq t_1^{'}$ and let $i \in P \setminus P_1(t)^{*}$. If $s_{ij}(t) = 0, \forall j \in P$, then

$$w_i(t+1) \geq w_i(t) > m_1(t) = m_1(t+1)$$

If $s_{ij}(t) > 0$ for some $j_0 \in P$, then, as above,

$$w_i(t+1) > m_1(t) = m_1(t+1)$$

Thus $i \in P \setminus P_1(t+1)$ and this prove (*). Since $P_1(t_1^{'})$ is a finite set, there exists $t_1 \geq t_1^{'}$ such that

$$P_1(t) = P_1(t_1), \quad \forall\, t \geq t_1 .$$

So (2) is true for *k=1*.

In order to prove (3), let $t \geq t_1$, let $i \in P_1(t_1)$ and let $j_0 \in P$. If $j_0 \in D(i,t)$ then since

$$w_i(t) = m_1(t) \leq w_{ij_0}(t),$$