MA-004-431-1

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA
MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY OF SAAD DAHLEB BLIDA
U.S.D.B
FACULTY OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

THESIS

SUBMITTED FOR THE FULLFILLMENT REQUIRMENT TO GET MASTER DEGREE IN COMPUTER SCIENCE

OPTION : SIR

IMPLEMENTATION OF A DISTRIBUTED ALGORITHM FOR THE BALANCING OF INTEGER INDIVISIBLE LOADS IN DISTRIBUTED SYSTEMS

Supervised by :
PR.OULD-KHAOUA M.

Presented by :
BENSAFI HIND

Board of Examiners :

DR.BENYAHIA M.	PRESEDENT	USDB
PR.OULD-KHAOUA M.	SUPERVISOR	USDB
DR.ZAHRA F.	EXAMINER	USDB
MME.HAMMAD A.	SUPERVISOR	

ACADEMIC YEAR : 2017/2018

MA-004-431-1

Abstruct

One of the important tools within the distributed computer systems is load balancing. The load balancing is a process of redistribution of tasks among multiple processors. This redistribution should be made so that each processor has approximately equal work to do. In this thesis we present a dynamic algorithm for balancing indivisible loads on a network. Our aim is to reach an equilibrium state of the network using a final number of steps.The algorithm works as follows: each node in the network communicate with its neighbors and balance their loads locally to reach the equilibrium state. The results of the experiments of various load balancing methods, considering two typical load balancing environement centralized and decentralized, indicate that the proposed greedy sort method has more advancements over the others, especially in decentralized than a centralized environement.

Key words: Load balancing, indivisible loads, dynamic load balancing.

Résumé

L'un des moyens très importants dans le domaine des systèmes distribués est l'équilibrage de charge. L'équilibrage de charge consiste à la redistribution des charges du travail sur différents processeurs. Cette redistribution doit garantir que chacun des processeurs existants aura presque la même charge du travail. Dans ce mémoire nous présentons un algorithme d'équilibrage des charges indivisibles dans un réseau. Notre but est d'arriver par la suite à un état d'équilibre global. L'algorithme fonctionne comme suit: chaque nœud dans le réseau communique avec ses voisins et équilibrent leur charges localement pour atteindre l'état d'équilibre global. Les résultats des expérimentations de différentes méthodes d'équilibrage, en considérant les deux cas centralisé et décentralisé, la méthode "*greedy sort*" proposée donne de bonnes résultats par rapport aux deux autres, dans un milieu décentralisé mieux que dans le centralisé.

Mot clés: Équilibrage de charge, charges indivisibles, équilibrage de charge dynamique.

ملخص

إنّ موازنة الحمُل يعتبر أحد أهم الوسائل المستعملة في النُظُم الموزّعة، حيثِ أنّه عبارة عن تقنيّة لإعادة توزيع الأعمال الموجودة على مختلف المعالجات المتوفرة. بيد أن هذه التّقنية تخضع لشروط منها أهمّيّة التوزيع الشبه متساوي للأعمال على كل المعالجات. لذلك فإنّ هدفنا من خلال هذه المذكّرة هو إنشاء و تقديم خوارزميّة ديناميكيّة لموازنة الحمُل لشبكة ما، إلّا اننّا ركّزنا من خلال هذا العمل على الحمُل الغير قابلة للتقسيم.

يمكن وصف خوارزمية الموازنة كالتّالي: كلّ جهاز في الشّبكة يقوم بتبادل المعلومات مع الأجهزة المجاورة لهُ و من ثمَّ موازنة الحُمُل محليّاً. إذّ نتائج التجارب لمختلف تقنيّات المُوازنة، آخذين بعين الإعتبار البيئتين المركزيّة و اللّامركزيّة، قد أظهرت أنّ التقنيّة المُقترحة *"greedy sort"* قد أعطت نتائج متقدّمة مقارنةً بالتّقنيّات الاخرى، و في بيئة لامركزيّة احسن من البيئة المركزيّة.

كلمات مفتاحيّة: مُوازنة الحُمُل ، حُمُل غير قابلة للتقسيم ، مُوازنة ديناميكيّة للحُمُل.

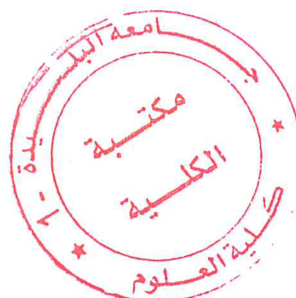
Acknowledgment

First and foremost, I would like to thank ALLAH Almighty for giving me the strength, knowledge, ability and opportunity to undertake this research study and to persevere and complete it satisfactorily. Without his blessings, this achievement would not have been possible.

Second I would like to thank my thesis advisor Prof.OULD–KHAOUA Mohamed of the Computer Science department at Blida university. The door to Prof.OULD–KHAOUA office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it.

I would also like to thank the head of the IT cell at "Algeria Airlines" Mme.HAMMAD Amina, without her passionate participation and input, this thesis could not have been successfully conducted.

Finally, I must express my very profound gratitude to my parents and to my brothers Adel and Nabil and to my freinds for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.



Contents

Abstract i

Acknowledgment iii

Introduction 7

1 Related works **10**

1 Static Load Balalncing 10

2 Dynamic Load Balalncing 12

3 Load Balancing in distributed systems 14

1.3.1 Load Balancing in cloud computing environments 14

1.3.2 Load Balancing in grid computing environments 15

4 Indivisible loads problem 16

1.4.1 Divisible loads 17

1.4.2 Indivisible loads 18

2 Intorduction to distributed systems and load balancing **20**

1 Definitions 20

2 Motivation 21

3 Parallel or distributed system 22

4 Types of distributed systems 23

2.4.1 Distributed Computing Systems 24

2.4.2 Distributed Information Systems 25

2.4.3 Distributed Pervasive Systems 26

5 Distributed systems challenges 29

2.5.1 Load balancing problem 29

3 Self-stabilizing algorithm for indivisible integer-valued loads **31**

1 Self stabilazing algorithm for indivisible loads 32

3.1.1 Self-stabilizing approach 32

3.1.2 The algorithm 34

3.1.3 Pairwise methods 36

4 Numerical results **40**

1 First Experiment: Brute force vs. Greedy Sort 40

2 Second Experiment: Centralized vs. Decentralized 42

3 Third Experiment: Decentralized algorithm, Extreme case 44

4 Fourth Experiment: Different topologies 46

5 Discussing 49

Conclusion **51**

Bibliography **52**

1

Appendix 56

List of Figures

0.1 classification of loads processing. [7] 8

0.2 Knapsack analogy. 9

2.1 A distributed system that connects processors by a communication network. 21

2.2 Difference between distributed and parallel system. 23

2.3 An example of a cluster computing system. 24

2.4 A nested transaction. 25

2.5 Interapplication communication. 26

2.6 Example of body-area network. 27

2.7 Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors. 28

2.8 Molecular dynamics simulation software. 30

3.1 Simple network of 8 nodes. 31

3.2 Network nodes before (right) and after (left) the balancing. 32

3.3 Self-stabilization approach. 33

3.4 The damped pendulum analogy. 33

3.5 A pair of nodes example. 38

4.1 Minimum (a) and Maximum (b) total load by iteration for 200 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods. 41

4.2 Delta by iteration for 200 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods. 41

4.3 Itrations by nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods. 42

4.4 Minimum (a) and Maximum (b) total load by iteration for 200 nodes: Centralized case (in purple) and Decentralized case (in green) using Greedy Sort. 43

4.5 Delta by iteration for 200 nodes: Centralized case (in purple) and Decentralized case (in green) using Greedy Sort. 43

4.6 Itrations by nodes: Centralized case (in purple) and Decentralized case (in green) using Greedy Sort. 44

4.7 Minimum (a) and Maximum (b) total load by iteration for 200 nodes: Decentralized case (in purple) and Decentralized with all loads in one node (in green) using Greedy Sort. 45

4.8 Delta by iteration for 200 nodes: Decentralized case (in purple) and Decentralized with all loads in one node (in green) using Greedy Sort. 45

4.9 Diffrent network topologeis. 47

4.10 Minimum (a) and Maximum (b) total load by iteration: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash–Net topology (in orange) using Greedy Sort. 48

4.11 Delta by iteration: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash–Net topology (in orange) using Greedy Sort. 48

0.12 Minimum (a) and Maximum (b) total load by iteration for 400 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods. 56

0.13 Delta by iteration for 400 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods. 57

0.14 Minimum (a) and Maximum (b) total load by iteration for 600 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods. 57

0.15 Delta by iteration for 600 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods. 58

0.16 Minimum (a) and Maximum (b) total load by iteration for 400 nodes: Comparaison of Centralized (in purple) and Decentralized (in green) cases. 58

0.17 Delta by iteration for 400 nodes: Comparaison of Centralized (in purple) and Decentralized (in green) cases. 59

0.18 Minimum (a) and Maximum (b) total load by iteration for 400 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash–Net topology (in orange) using Greedy Sort. 59

0.19 Delta by iteration for 400 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash–Net topology (in orange) using Greedy Sort. 60

0.20 Minimum (a) and Maximum (b) total load by iteration for 600 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash–Net topology (in orange) using Greedy Sort. 60

0.21 Delta by iteration for 600 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash–Net topology (in orange) using Greedy Sort. 61

List of Tables

4.1 Exection time of Brute Force and Greedy Sort. 49

4.2 Exection time of Centralized and Decentralized self-stab algorithm. 49

Introduction

To increase the performance of the processors, at first, the founders have hardly worked to increase the frequency of operation of their chips. However, it seems that today we have arrived at frequency that will be difficult to break. To increase performance, manufactors have turned to produce chips with multiple compute units.

Althoug, in robust systems, the use of multi-core or multi-processors machines is not yet sufficient. A recent tendance was to distribute computation among several physical processors. In front of the physical limits of the storage capacity and the speed of calculation, distributed systems appear as a competetive solution that can solve problems requiring large computing capabilities and handling large amounts of data. Several advantages characterize these distributed computer systems : high availibility, best fault tolerance, great flexibility and most importantly, they allow to share many ressources (processor, memory, disk, ect.) through a network.

The optimal use and sharing of computing ressources of distributed systems is a very important aspect that mobilize many researchs around the world. This optimality requires a distribution of the workload on the different computer nodes intelligently. It is necessary to avoid, as far as possible, situations where some nodes are overloaded while others are underloaded or complety free. Balancing the workload on the various available resources proves to be a real challenge.

In general, the underlying principle of load balancing is to distribute load units on a distributed network of ressources. This definition can encompress a broad class of problems such as performance, response time, scalability, overhead and many more. The notion of load can be another crucial problem, the load unit could take a sevral forms (see figure 0.1), not necessary related to a computer task :

- **Indivisible loads :** These loads are independent, indivisible, and, in general, of different sizes. This means that the load cannot be further subdivided, thus, they have to be processed in their entirety in a single processor.
- **Divisible loads :** A divisible load can be either modularly or arbiturly divised. A modularly divisible load is a priori subdivided into smaller modules based on some characteristics of the load or the system. The processing of a load is complete when all its modules are processed. Alternatively, an arbiturly divisible load has the property that all items in the load demand an identical type of processing. These loads have the characteristic that they can be arbitrarily partitioned into any number of load fractions. These load fractions may or may not have precedence relations[6].

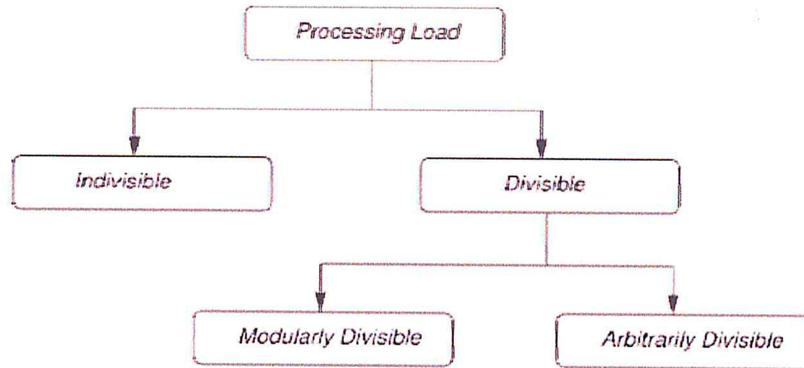


Figure 0.1: classification of loads processing. [7]

During the last years, load balancing became an important prerequisite for both indistrial and academic users to balance their work efficiently, so, load balancing is known to have applications in many domains : Clouds, scheduling, routing, numerical computaion, finite element computations and wireless sensor networks [5].

The deployment of load balancing algorithm involves many challenges. For example, the decentralized nature of the system where there are no base station, it means, that to reach the state of equilibrium, nodes have to communicate and exchange data localy. The second challenge is the indivisible nature of data, how can we stabilize the global workload without the subdivision of the load unit? This issue is known as indivisible integer-valued load balancing problem : Stabilizing all loads arrived to the system using only local communications while maintaining the integrity of the load unit.

Based on the previous motivation, in this report, we focus our research on the distributed load balancing of indivisible loads problem. The main contribution of the thesis is a self stabilizing algorithm inspired of the self stabilizing algorithm initially developed to tackle the problem of non negative real numbers of divisible loads (average consensus in wirless sensor networks). A good analogy of our approach is the Multiple integer Knapsack Problem (MKP)(figure 0.2). The MKP fits naturally to our problem if we consider nodes of the network as similar knapsacks all filled with items of different sizes. If the knapsack are communicating, then the system self stabilize to an equilibrium with almost the same weight in each knapsack.

These next sections are organized as follows. In section 1 we present a review of the existing load balancing algorithms in the various distributed systems. In section 2 we introduce the distributed systems and the load balancing problem. In section 3 the proposed solution is presented. In section 4 we detail the simulation and the experiments of our algorithm. In the last part we present our conclusions and the future work on this topic.

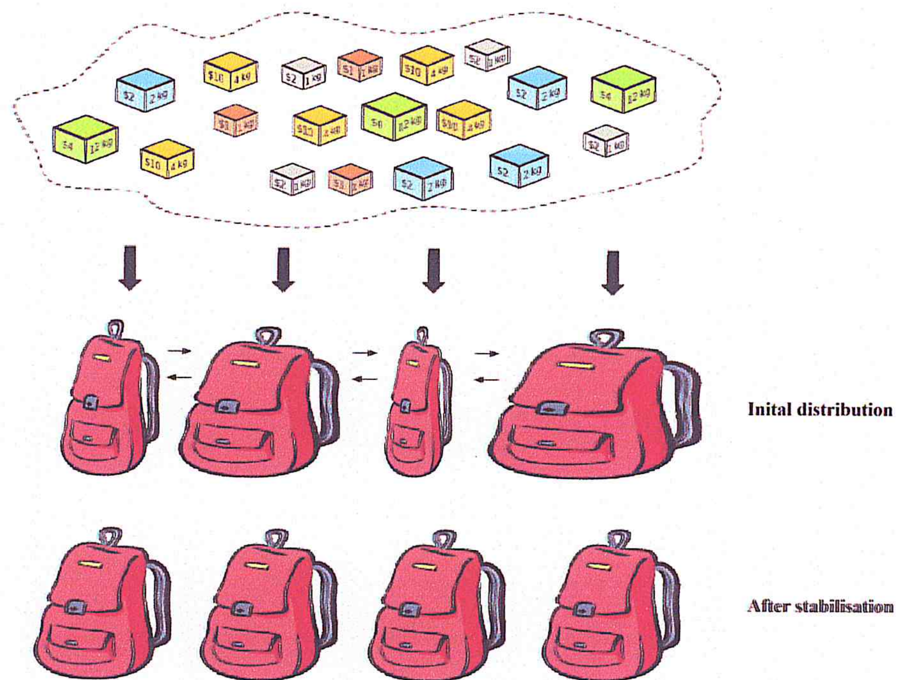


Figure 0.2: Knapsack analogy.

Chapter 1

Related works

Load Balancing (LB) is a method of increasing the performance of Distributed System (DS). It is a policy of transferring the load among various processors of DS to improve job response time and resource utilization while also ignoring a condition where few processors are overloaded while others are idle or doing under loaded work at any given instant of time in the system [1]. It can be classified into two subcategories : Static Load Balancing (SLB) and Dynamic Load Balancing (DLB).

The SLB algorithms are generally based on the informations about the behaviour of the system. The algorithm is executed by providing prior informations about the system. This means that the decisions related to load balance are made at compile time. Many hypotheses are assumed like the performance of a node or the resources needed to execute a task. Dynamic strategies, in the other hand, react to the current system state in making transfert decisions. The current status of the distributed network is analyzed to adapt the system dynamically.

The LB problem (static or dynamic) has been discussed in traditional distributed systems literature for more than two decades. Various strategies and algorithms have been proposed, implemented and classified in a number of studies . In this chapter, we will cite some articles in order to give an overviw of the problem.

1 Static Load Balalncing

In this method the performance of the processors is determined at the initial stage of process. Then based on their performance the work load is distributed by the master processor. The slave processors calculate their allocated work and then send their result to the master.In these method, a job is always executed on the processor to which it is assigned that is SLB techniques are non preemptive. The main intention of SLB method is to reduce the overall execution time of a concurrent program while minimizing the communication delays.

Many authors have been interested to study SLB strategies. For example, S. Ichikawa and S. Yamashita present in [22] a static load balancing scheme for partial differential equation solvers in a distributed computing environment. Their method considers both computing and communication time to minimize the total execution time with automatic data partitioning and processor allocation. The problem was formulated as a combinatorial optimization and solved by the branch-and-bound method for up to 20–24 processors. They present approximation algorithms that give good allocation and partitioning in practical time. The quality of the approximation was evaluated in comparison to the optimal solution or theoretical lower bounds.

Another interesting work was the balance of Nash [20]. D.Grosu and A.T.Chronopoulos formulate the SLB problem in heterogeneous distributed systems as a noncooperative game among users. For the proposed noncooperative LB game, they present the structure of the Nash equilibrium. Based on this structure they derive a new distributed LB algorithm. The performance of their noncooperative load balancing scheme is compared with that of other existing schemes (proportional scheme [9], global optimal scheme [23], and individual optimal scheme [24]). The main advantages of their load balancing scheme are the distributed structure, low complexity and optimality of allocation for each user.

Jie Li and Hisao Kameda studied also the SLB problem in a distributed computer system with the tree hierarchy configuration [28]. The problem was formulated as a nonlinear optimization problem. After studying the conditions that the solution to the optimization problem of the tree hierarchy network satisfies, It was demonstrated that the special structure of the optimization problem leads to an interesting decomposition technique. A new effective decomposition algorithm to solve the optimization problem was presented. The proposed algorithm was compared with two other well known algorithms: the Flow Deviation (FD) algorithm and the DafermosSparrow (D-S) algorithm. The experiments shows that both the proposed algorithm and the D-S algorithm have much faster convergence in terms of central processing unit (CPU) time than the FD algorithm.

The SLB in single channel and star network configurations was studied in [35, 34]. Tantawi and Towsley studied LB of single class jobs in a distributed computer system that consists of a set of heterogeneous host computers connected by single channel and star communications networks such as satellite networks and some LAN's. A key assumption of theirs was that the communication delay does not depend on the source-destination pair. They considered an optimal SLB strategy which determines the optimal load at each host so as to minimize the mean job response time, and derived two algorithms (called single-point algorithms) that determines the optimal load at each host for given system parameters.

Static policies use only the system statistical information in making LB decisions, and therefore have the advantages in mathematical analysis and in implementation because of their simplicity. Also, static strategies will suppress (or at least minimize) data redis-

tributions and control overhead during execution. Furthermore, static allocations seem to be necessary for a simple and efficient memory allocation[23]. Even though, a usual disadvantage of all static methods is that the final selection of a node for process allocation is made when the process is initially created and it cannot be changed during process execution in order to make variations in the system load [33]. This lead us to focus on another important and efficient type of LB: the dynamic load balancing.

2 Dynamic Load Balalncing

As the literature is vast with respect to DLB, we will see different works to dealing with this problem.

However, it is necessary to begin to clarify the meaning of DLB in the literature, with many possible interpretations. For example, we can talk about DLB when the load varies during the LB process [11].We also talk about DLB when the network topology varies [3]. In this example, J.Bahi, R.Couturier and F.Vernier see a dynamic network as a network with dynamic links. the edges in the network may vary at each time step. They suppose that no computer can be added or definitively retrieved in a dynamic network and at each time step, each node in a dynamic network knows which of its edges are alive (a dynamic network can be viewed here as a network in which some edges can be lost during the execution of the algorithm due to faulty communications or timeout). With these considerations, they propose three algorithms: Their first algorithm is a classical diffusion load balancing algorithm, requiring some adaptations because of the dynamicity of the network (the diffusion matrix integrates the information when a link is missing). GAE is a new algorithm. It constraints one node to exchange its load with at most one of its neighbors: at time t and for each node, all edges except one are unusable (the choice of the single neighbor of each node is done using strategies arbitrary, random or more sophisticated). The latter, which we called relaxed diffusion, introduces a relaxation parameter in the diffusion algorithm. The relaxation parameter may dramatically speed up the convergence. The authors then simulate different networks with a certain percentage of unusable edges (from 0% to 50%) to illustrate the behavior of these algorithms and to prove their convergence towards the uniform distribution of the workload. The algorithms of J. Bahi, R. Couturier and F. Vernier allow to balance the load on a dynamic network in which the communication links are not 100% reliable.

Another approach for dynamic load balancing is to "use the past to predict the future", that is, to use the computational speed observed for each processor or resource to decide the redistribution of the loads. Many authors [10, 40] have been interested in this approach, especially [40] where load balancing involves assigning to each processor a job proportional to its capabilities with minimizing the execution time of the program. The authors examine the different behaviors of load balancing strategies: global versus

local and centralized versus distributed on a network of workstations. They showed that different strategies are best for different applications under varying parameters such as the number of processors, data size, iteration cost, communication cost, etc.

A very particular view of dynamic load balancing is based on the redistribution of data among the participating processors during the execution of the algorithm. This redistribution is done by transferring data from the most loaded processors to the least loaded processors. This load balancing phase can be centralized by a single processor or distributed over all the processors.For example, in [21], the main problem is the permanent change in the load of each workstation (the authors worked on an architecture organized into clusters), which makes the execution time of parallel applications unpredictable. Simulation and experimental studies of their load balancing strategy were performed under various load situations and it was shown that it can effectively balance the workload among the workstations involved. Further, it was shown that a significant improvement in performance can be achieved when compared to the case where no load balancing is employed.

Another example of the many existing models for dynamic load balancing strategies in [38] where H.Willebeeck-Lemair and P.Reeves present five approaches. "Sender/Receiver Initiated Diffusion(SID/RID)" can be summarized as follows: all processors inform their nearby neighbors of their load level and update this information while the application is running, the "Hierarchical Balancing Method (HBM)" organizes the system into hierarchy of subsystems, the Gardient Model (GM) uses a routing map to migrate the data from the most heavily loaded processors to the least loaded and closest processors, and finally, the Dimension Exchange Method (DEM) which is method of exchange in the different dimensions of a hypercube that requires a synchronization phase to balance the load and then iteratively balance it again.

the main advantage of dynamic load balancing over static scheduling is that the system does not need any pieces of information about the behavior of the load for the tasks deployed on the distributed network. Without this constraint, these algorithms can be deployed in a more general way without prior knowledge of the deployed applications. This adaptability comes with a cost: the run-time overhead due to load transfer among the nodes and the complexity of the load balancing decision process. Nowday, the dynamic strategies are more used then the static ones.

In the following section we present a more specific view of dynamic load balancing, with more restrictive assumptions. Several authors are interested in dynamic load balancing for specific environments: parallel applications, clouds, ..ect. In the next section, we will present the load balancing problem in the different distributed systems.

3 Load Balancing in distributed systems

The problem of load distribution has been studied for more than thirty years in the areas of distributed systems [36, 16, 17]. In what follows, we focus on the main research work dealing with dynamic load balancing in particular contexts.

1.3.1 Load Balancing in cloud computing environments

Cloud computing is emerging technology which is a new standard of large scale distributed computing and parallel computing. It provides shared resources, information, software packages and other resources as per client requirements at specific time. As cloud computing is growing rapidly and more users are attracted towards utility computing, better and fast service needs to be provided. For better management of available good load balancing techniques are required. So that load balancing in cloud becoming more interested area of research. And through better load balancing in cloud, performance is increased and user gets better services [13].

we can define a cloud computing as the internet based computing in which the different services like storage, servers and application are provided to organizations computers and device using internet. It is a type of computing in which resources are shared rather than owning personal devises or local personal servers which can be used to handle applications on system. So as compared to this traditional "own and use" technique if we use cloud computing, the purchasing and maintenance cost of infrastructure is eliminated. It allows the users to use resources according to the arrival of their needs in real time.Thus, we can say that cloud computing enables the user to have convenient and on-demand access of shared pool of computing resource such as storage, network, application and services,...etc [13].

In 2010, S C. Wang et al. [37] presented a dynamic load balancing algorithm called load balancing Min-Min (LBMM) technique which is based on three level frameworks. This technique uses Opportunistic Load Balancing algorithm which keep each node busy in the cloud without considering execution time of node. Because of this it causes bottle neck in system. This problem is solved by LBMM three layer architecture. First layer request manager which is responsible for receiving task and assigning it to one service manager to second level. On receiving the request service manager divide it into subtasks. After that service manager will assign subtask to service node to execute task.

In [25], Dhinesh Babu L.D and P. Venkata Krishna proposed a Honey Bee Behavior inspired Load Balancing (HBB-LB) technique which helps to achieve even load balancing across Virtual Machine (VM) to maximize throughput. It considers the priority of task waiting in queue for execution in virtual machines. After that work load on VM calculated decides weather the system is overloaded, under loaded or balanced. And based on this

VMs are grouped. New according to load on VM the task is scheduled on VMs. Task which is removed earlier. To find the correct low loaded VM for current task, tasks which are removed earlier from over loaded VM are helpful.

In 2014, the load balancing model which was proposed in [32] is aimed at the public cloud which has numerous nodes with distributed computing resources in many different geographic locations. This model divides the public cloud into several cloud partitions. When the environment is very large and complex, these divisions simplify the load balancing. The cloud has a main controller that chooses the suitable partitions for arriving jobs while the balancer for each cloud partition chooses the best load balancing strategy.

The paper in [31] presented a dynamic load balancing algorithm for cloud computing based on an existing algorithm called weighted least connection (WLC) [27]. The WLC algorithm assigns tasks to the node based on the number of connections that exist for that node. This is done based on a comparison of the number of connections of each node in the cloud and then the task is assigned to the node with least number of connections. However, WLC does not take into consideration the capabilities of each node such as processing speed, storage capacity and bandwidth. The proposed algorithm is called Exponential Smooth Forecast based on Weighted Least Connection (ESWLC). ESWLC improves WLC, that is,the ESWLC builds the conclusion of assigning a certain task to a node after having a number of tasks assigned to that node and getting to know the node capabilities. ESWLC builds the decision based on the experience of the node's CPU power, memory, number of connections and the amount of disk space currently being used.

1.3.2 Load Balancing in grid computing environments

Contrary to the traditional distributed systems for which a plethora of algorithms have been proposed, few of which were focussed on grid computing. This is due to the innovation and the specific characteristics of this infrastructure.

Foster and Carl Kesselman proposed a distributed computing infrastructure for advanced science and engineering, which they called the Grid [29]. A computational grid is a distributed architecture of large numbers of computers connected that enables effective access to high performance computing ressources. The systems connected together by a grid might be distributed globally, running on mutiple hardware platforms, under different organizations. Servers or personal computers run independent tasks and are loosely linked by the internet or low-speed networks.

In 2006, Belabbas Yagoubi and Yahya Slimani [39] propose a load balancing algorithm based on a tree model representation of a grid architecture. Based on a tree model, their algorithm presents the following main features: it is layered, it supports heterogeneity

and scalability, and finally, it is totally independent from any physical architecture of a grid. The tree model presented in their paper allows the transformation of any grid architecture into an unique tree with at most four levels. The proposed strategy was implemented and evaluated on a grid simulator developed for the circumstance. The first results of the experimentations show that the proposed model can lead to a better load balancing without high overhead. It was observed also that significant benefit in mean response time was realized with a reduction of communication cost between clusters of the grid.

J. Cao, et. al. use 'intelligent agents' [8] to balance the load in grid. In this method, each agent acts as a local resource. These agents reduce the execution time by sharing and interchanging the information with each other. Having hierarchical structure is of the specification of these agents. By using this specification, as well as static and dynamic methods, the agents provide load balancing.

In [29], the algorithm which is based on ant colony optimization (ACO), the ants can move in the form of search-max or search-min. In the first case, an ant moves ahead at random to find a node with overload, then it switches in the form of search-min (underload), and it is the time that the ant makes balance between the heavy-load node and the light one. Using artificial life techniques is one of the methods which provide load balancing in Grid. This method utilizes two kinds of algorithms including genetic and Tabu search (TS) to solve the problem of Grid load balancing. Whenever the space of solution is broad the genetic algorithm can be used. This algorithm performs the job scheduling alternatively and continuously because the load balancing can be done periodically.

4 Indivisible loads problem

The interest in network-based computing has grown significantly in last years. In this environement, a number of workstations or computers are linked through a network to form a wide loosely coupled distributed system. One of the major advantages of such a distributed system, besides its role of storing information in a distributed manner and allowing the use of shared resources, is the capability that it offers to a user to exploit the considerable power of the complete network or the subnet of it by partitioning and transferring its own processing load to the other processors in the network.

This load distribution paradigm essentially concerns a single large load that comes from or arrives at one of the nodes of the network. The load is massive and requires a lot of time to process given the computing capacity of the node. The processor partitions the load into multiple fractions, retains one of the fractions to be processed, and sends the rest to its neighbors (or other nodes in the network) for processing. An important

problem here is to decide how to achieve a balance in the load distribution between the processors so that the calculation is completed in the shortest possible time. This balancing can be done early or dynamically depending the calculation progresses and the computational requirements. This strategy of division is suitable for applications that can independently process the processing load into smaller fractions so that partial solutions can be consolidated to build the complete solution to the problem.

Not so far, the problems discussed in the literature did not attempt to formulate scheduling policies based on the type of loads submitted by a user. Usually, the stress has been on designing new and more efficient parallel algorithms in place of classical sequential algorithms, which requires the parallelism of functions to be exploited in the algorithm. However, there is another type of parallelism that occurs in data and is called parallelism of the data, that is, to partition the date into optimally sized segments and assigned to several processors, in order to be processed in the shortest possible time. But, how this partitioning (or division of load) can be performed depends on its divisibility property, that is, on the property that determines whether a load can be decomposed into a smaller set of loads. This leads us directly to the domain of what is known as divisible and indivisible loads.

1.4.1 Divisible loads

The simplest definition of of a divisible load is that is the one that can be arbitrarily divisible, that is, one that can be divided into any number of segments of any desired fractional size and execute these parts independently in parallel on different processors. such loads are commonly encountered in applications involving image processing, signal processing, processsing of massive experimental data, and so on.

- **Image processing**

  In image processing for example, the first level of computation partitions the given image into many segments. Each of these segments is processed locally and independently on different processors. This is done to extract local features of the image from different segments. In the second level of computation, these local features from different processors are exchanged and processed to extract the desired feature. It is at the first level of computation that the load can be considered to be arbitarily divisible without any precedence relations. As before, the given image can be arbitrarily partitioned into several subframes of varying sizes (that is, each may contain a different number of pixels) and each of these subframes can be processed independently. A practical situation in which processing of such dara may frequently be necessary involves the space shuttle orbiter, which collects massive volume of image data that has to be communicated to the earth for processing .

This kind of data also has the potentiel of arbitrary divisibility. The data can be partitioned and sent directly for processing to a number of processors situated at various geographical points on the surface of the earth, in which case they incur considerable communication delays. Depending on the location of the processing units the communication delays will be different.

- **Signal processing**

  another example of divisibility is the signal processing. A simple application involves the poroblem of recovering a signal buried in zero-mean noise. The raw data consists of a large number of measurements that can be arbitrary partitionned and shared among several processors.

- **Massive experimental data**

  Another application involves passing a very long linear data file through a digital filter. This may be for frequency shaping purposes (this is, passing the data through a low pass filter) or for pattern matching (that is, passing the data through a matched filter designed to find a particular pattern). In either case the data file may be partitioned among a number of processors. Each processor runs the same filter on its segment of the data.

Several studies have been done to deal with problem of divisible loads. Theoretical analysis of balancing arbitrarily divisible loads using diffusion-based DLB schemes has been done using spectral analysis of Markov processes on graphs as in [30]. The analysis has also been extended to distributed gossip algorithms that reach a consensus [2].

1.4.2 Indivisible loads

These loads are independent, indivisible, and, in general, of different sizes. This means that a load cannot be further subdivided and has to be processed in its entirety in single processor. The problem of the indivisible loads is known to be NP-complete and hence only heuristic algorithms can be proposed to obtain suboptimal solutions in reasonable time [6]. These loads can be either real or integer valued. For example, a real-valued loads can be found in the scientific simulations, where the initial computational domain is decomposed into smaller subdomains whose sizes are fixed. At any given time, the cost of each subdomain is a real number, and subdomains cannot be subdivided. Or integer-valued, as the example of the google accounts. These accounts can be moved from a server to another in their entirely and cannot be subdivised.

In the following sections, we focus on DLB of indivisible integer-valued loads in arbitrary networks. Each load is defined by a constant integer number. Loads cannot be modified or subdivided, but only moved from one processor to another.

It is observed with through our study that load balancing algorithm works on the principle on which situation workload is assigned, during compile time or run time. Depending on the compile time or run time it may be static or dynamic. Static strategies have the advantages in mathematical analysis and in implementation because of their simplicity. But, in the other hand, dynamic load balancing policies have been believed to have better performance than static ones in case of distributed environments whereas they may have more overhead than the latter. In this chapter, some works on load balancing in the different distributed systems was presented, the study showed that the load balancing is an important issue in distributed systems that must be treated.

In the following sections, a novel dynamic load balancing algorithm will be proposed and implemented. Our algotrithm will be inspired from Bahi et al. and Lagacherie works.

Chapter 2

Intorduction to distributed systems and load balancing

Since the advent of the Internet in the 1970s [26], a recent trend in computer systems was to distribute computation among several physical processors. The advances in networking and hardware technology had made sensor networking and embedded systems a reality and an integral part of each person's life ˘ from the home network with the interconnected gadgets to the automobile communicating by GPS (Global Positioning System)–. For that, distributed computing was as the centerpiece of all computing and information access.

The distributed systems is a very important field that has his principles and challenges. In the following, we present the main principles of distributed systems, then a well known problem in this systems, the load balancing problem.

1 Definitions

A distributed system is a collection of independent entities that cooperate to solve a problem that cannot be individually solved. In computer science, a distributed system is defined as a set of autonomous processors communicating over a communication network (see figure 2.1) and having the following features [26]:

- **No common physical clock** This is an important assumption because it introduces the element of "distribution" in the system and gives rise to the asynchrony among the processors.
- **No shared memory** This is a key feature that requires message-transmission for communication.

- **Geographical separation** The more geographically distant processors are, the more representative is the distributed system. The processors are connected using a local or a wide area network (LAN/WAN).

- **Autonomy and heterogeneity** The processors are "loosely coupled" in that they have different speeds and each can be running a different operating system. They are not usually part of a dedicated system, but cooperate with one another by offering services or solving a problem together.

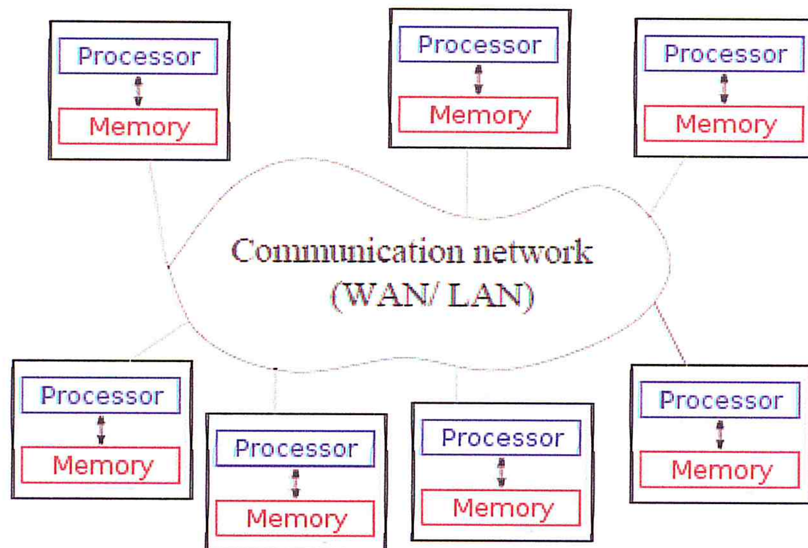


Figure 2.1: A distributed system that connects processors by a communication network.

2 Motivation

Generally, there are five major reasons for building distributed systems [23]. In the following, we briefly elaborate on each of them.

- **Resource Sharing:** If a number of different nodes are connected to each other, then a user at one node may be able to use the resources available at another. For example, a user at node A may be using a printer that is provided by node B. Meanwhile, a user in B may access a file that resides at A. Resources such as peripherals, complete data sets in databases, special libraries, as well as data (variable/files) cannot be fully replicated at all the sites because it is often neither practical nor cost-effective. Further, they cannot be placed at a single site because access to that site might prove to be a bottleneck. Therefore, such resources are typically distributed across the system.

- **Performance Improvement:** If a particular computation can be partitioned into a number of sub computations, that may allow us to distribute the computation among the various nodes. In addition, if a particular node is currently overloaded with jobs, some of them may be moved to other, lightly loaded nodes. This movement of jobs is called load balancing.

- **Reliability:** If one node fails in a distributed system. the remaining nodes can potentially continue operating. For example, If the system is composed of a number of computers, the failure of one of them should not affect the rest. In general, if enough redundancy exists in the system in both hardware and software, the system can continue with its operation, even if some of its nodes have failed.

- **Communication:** When a number of nodes are connected to each other via a communications network, the users at different nodes have the opportunity to exchange information.

- **Extensibility:** This is the ability to easily adapt changes without significant disruption of the system. The changes may include varying workloads, adding more hosts or replacing existing processors.

In addition to meeting the above requirements, a distributed system also offers the following advantages:

- **Inherently distributed computations:** In many applications such as money transfer in banking, or reaching consensus among parties that are geographically distant, the computation is inherently distributed.

- **Access to geographically remote data and resources:** In many scenarios, the data cannot be replicated at every site participating in the distributed system because it may be too large or too sensitive to be replicated. For example, payroll data within a multinational corporation is both too large and too sensitive to be replicated at every office/site. It is therefore stored at a central server. Similarly, special resources such as supercomputers exist only in certain locations, and to access such supercomputers, users need to log in.

3 Parallel or distributed system

The terms "parallel system" and "distributed system" have a lot of overlap, and no clear distinction exists between them [18]. The same system may be characterized both as parallel and distributed. For example, parallel system may be seen as a particular tightly coupled form of distributed system, and distributed system may be seen as a loosely

coupled form of parallel system. Nevertheless, it is possible to classify the systems as "parallel" or "distributed" using the following criteria:

- In parallel systems, all processors may have access to a shared memory to exchange information between them.
- In distributed systems, each processor has its own private memory (distributed memory). Information is exchanged by passing messages between the processors.

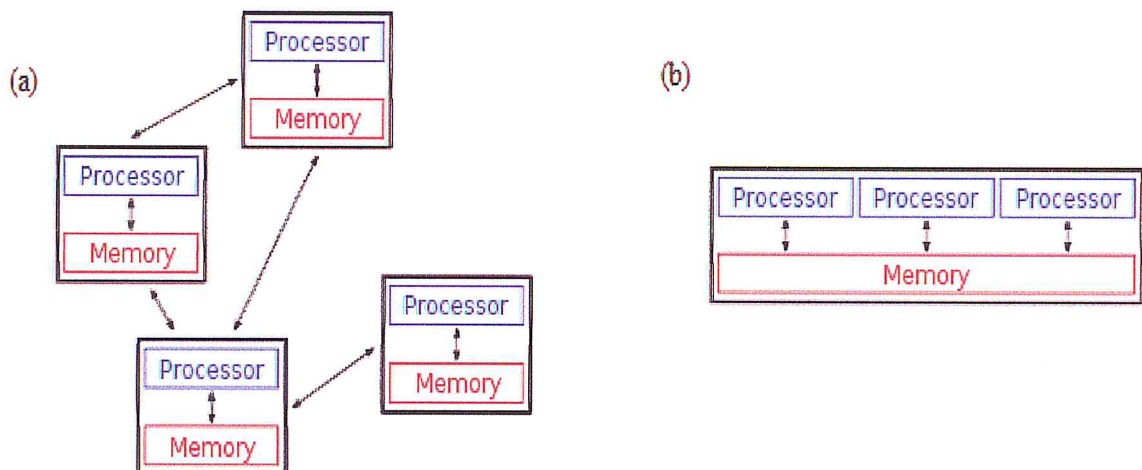


Figure 2.2: Difference between distributed and parallel system.

The figure 2.2 illustrates the difference between distributed and parallel systems. Figure (a) in the left shows a distributed system where each node has its own local memory, and information can be exchanged only by passing messages from one node to another by using a communication links. Figure (b), in the right side, shows a parallel system in which each processor has a direct access to a shared memory.

A very important point to mention here is that the definitions of distributed and parallel algorithm can not quite mutch to the same definitions of distributed and parallel systems. Nevertheless, as a rule, parallel system within a shared-memory processor uses parallel algorithms while the distributed system uses distributed algorithms.

4 Types of distributed systems

Distributed systems are used for many different applications. For that, sevral types distributed systems exist. In the following we make a distinction between distributed computing systems, distributed information systems, and distributed embedded systems.

2.4.1 Distributed Computing Systems

An important class of distributed systems is the one used for high-performance computing tasks. Basically, we can make a distinction between two subgroups : cluster computing and grid computing.

Cluster Computing Systems

In cluster computing, the hardware consists of a collection of similar workstations or computers, closely connected by a high−speed LAN. Cluster computing systems became popular when the price/ performance ratio of personal computers has improved. At a certain point, it became financially and technically interesting to build a supercomputer by simply connecting a series of simple computers with a high-speed network. In almost all cases, the cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines.

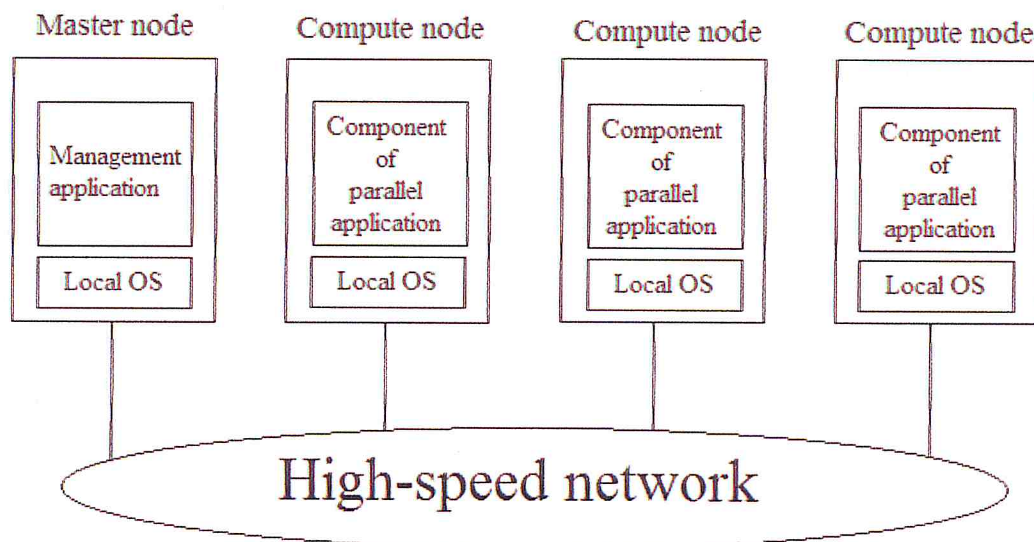


Figure 2.3: An example of a cluster computing system.

One well-known example of a cluster computer is formed by Linux-based Beowulf clusters (figure 2.3). Each Beowulf cluster consists of a collection of compute nodes that are controlled and accessed by means of a single master node. The master typically handles the allocation of nodes to a particular parallel program, maintains a batch queue of submitted jobs, and provides an interface for the users of the system.

Grid Computing Systems

A characteristic of cluster computing is its homogeneity. In most cases, the computers in a cluster are largely identical, they have the same operating system, and are all connected through the same network. In contrast, grid computing systems have a high degree of heterogeneity: no assumptions are made concerning hardware, operating systems, networks, etc.

2.4.2 Distributed Information Systems

Transaction Processing Systems

In practice, transactions on a database are usually performed in the form of transactions. A transaction may be nested and constructed from a number of sub—transactions. As shown in figure 2.4, the higher-level transaction can generate other transactions on different machines. Each of these children can also perform one or more sub—transactions.

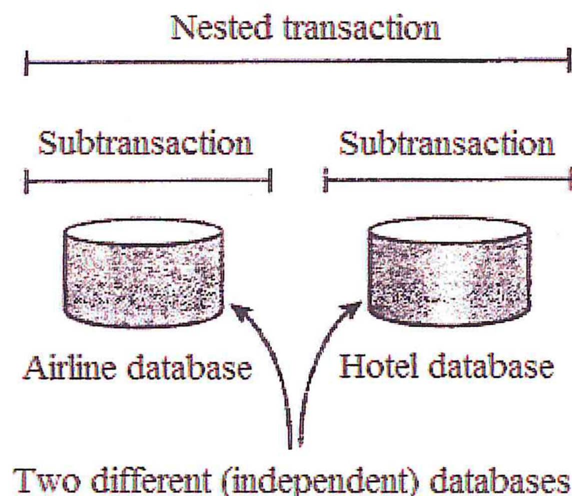


Figure 2.4: A nested transaction.

Nested transactions are important in distributed systems because they provide a natural way to distribute a transaction across multiple machines. For example, a transaction to plan a trip by which three different flights must be reserved can be divided into three sub—transactions. Each of these sub—transactions can be managed separately and independently of the other two.

Enterprise Application Integration

The need for communication between applications has led to many communication models. The main idea was that existing applications could exchange information directly, as shown in figure 2.5.

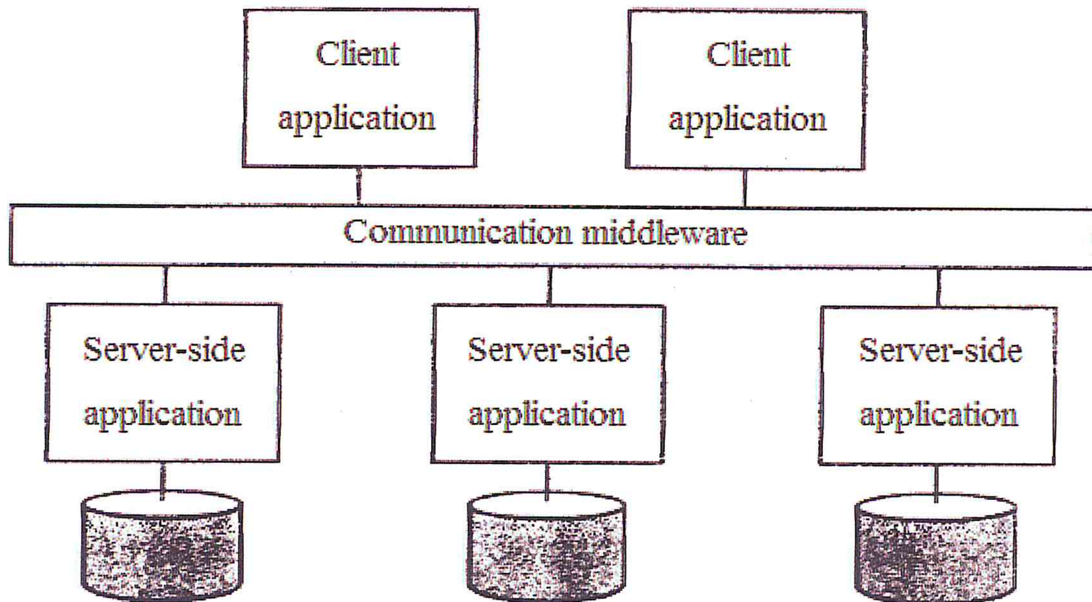


Figure 2.5: Interapplication communication.

Several types of communication middleware exist. With Remote Procedure Calls (RPC), an application can effectively send a request to another application by doing a local procedure call. A request is sent as message to the callee. Likewise, the result will be sent back and returned to the application as the result of the procedure call.

As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as Remote Method Invocations (RMI). An RMI is essentially the same as an RPC, except that it operates on objects instead of applications.

2.4.3 Distributed Pervasive Systems

The distributed systems we have discussed so far are largely characterized by their stability: the nodes are fixed and have a more or less permanent and high quality connection to a network. However, things have become very different with the introduction of mobile and embedded computing devices. We are now dealing with distributed systems in which

instability is the default behavior. The devices we call distributed pervasive systems, are often characterized by their small size, battery—powered, mobile and wireless connection.

Home Systems

These systems generally consist of one or more personal computers, but more importantly integrate typical electronics such as TVs, audio and video equipment, gaming devices, (smart) phones and other personal portable devices into a single system. In addition, we can expect that all kinds of devices such as kitchen appliances, surveillance cameras, clocks, controllers for lighting, and so on, will all be hooked up into a single distributed system.

The home systems are essentially intended to provide "services" to residents wherever they are. For example, the movie shown on the video can be viewed in any room. your phone calls should be automatically directed to the nearest phone, information should always be available on demand and displayed on a nearby screen.

Electronic Health Care Systems

Another important and future class of pervasive systems is electronic (personal) health care. With the increasing cost of medical treatments, new technologies are being developed to monitor the well-being of individuals. A major goal of these systems is to prevent people from being hospitalized.

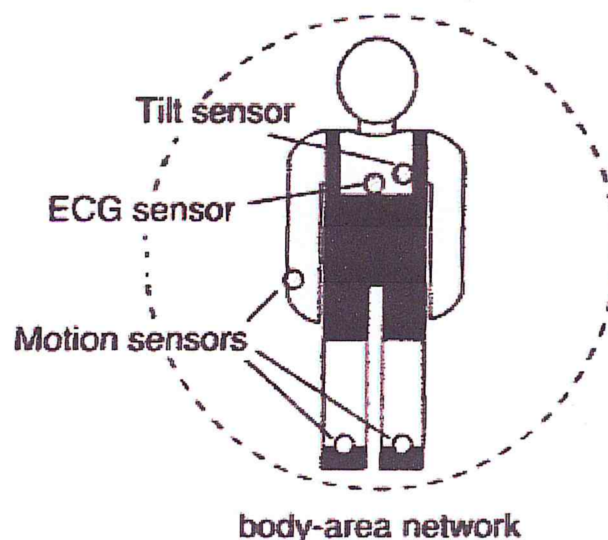


Figure 2.6: Example of body-area network.

Personal health care systems are often equipped with various sensors organized in (preferably wireless) body-area network (BAN) (figure 2.6). The BAN can be permanently

connected to an external network, via a wireless connection, to which it sends monitored data. Of course, other links with a doctor or other people may also exist.

Sensor Networks

Our last example of pervasive systems is sensor networks. A sensor network typically consists of tens to hundreds or thousands of relatively small nodes, each equipped with a sensing device. Most sensor networks use wireless communication, and the nodes are often battery powered. Their limited resources, restricted communication capabilities.

The relationship with distributed systems can be clarified by considering the sensor networks as distributed databases. This view is fairly common and easy to understand when many sensor networks are deployed for measurement and surveillance applications. In these cases, an operator would like to extract information from the network by simply issuing queries such as "What is the traffic load on Highway 1?" Such queries resemble those of traditional databases. In this case, the answer will probably be provided through a collaboration of many sensors located around Highway 1, while leaving other sensors untouched.

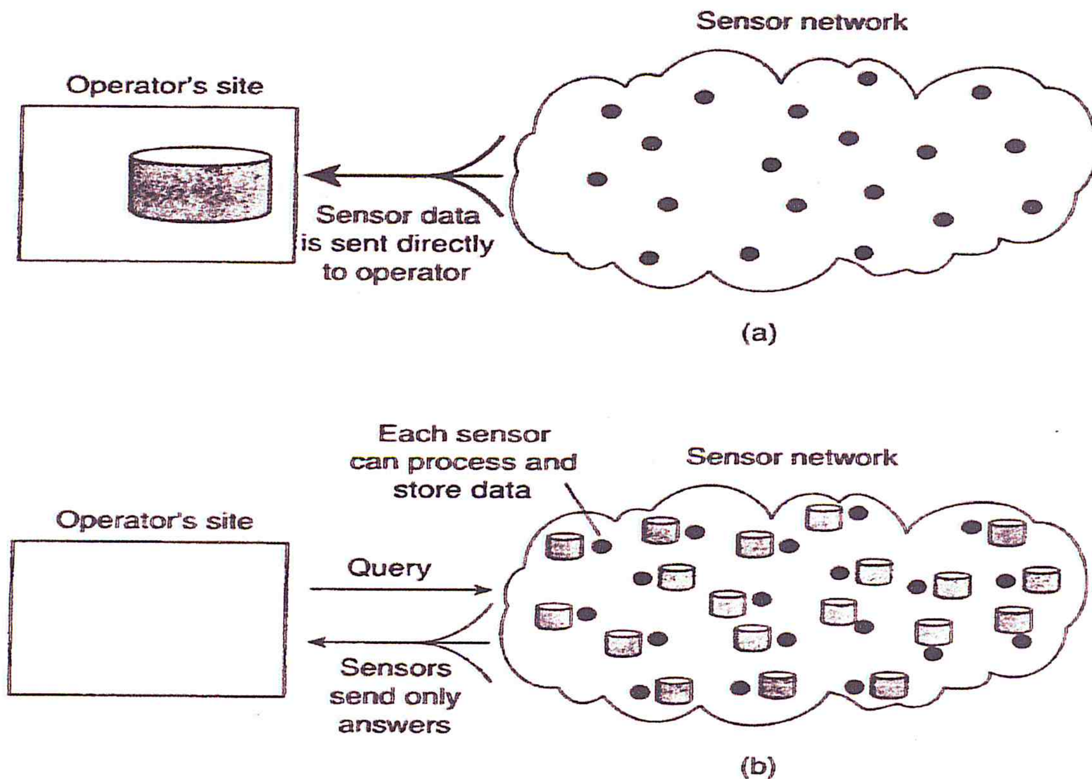


Figure 2.7: Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors.

To organize a sensor network as a distributed database, there are essentially two

extremes, as shown in figure 2.7. First, sensors do not process but simply send their data to a centralized database located at the operator's site. The other extreme is to let each node compute and send the answer.

5 Distributed systems challenges

Distributed computing systems have been in widespread existence since the 1970s when the internet came into being [26]. At the time, the primary issues in the design of the distributed systems included providing access to remote data in the face of failures, file system design and directory structure design. while these continue to be important issues, many newer challenges have surfaced as improving the performance of the system by balancing the load among nodes.

2.5.1 Load balancing problem

The load sharing problem is to develop algorithms to transfer jobs automatically from heavily loaded processors to lightly loaded processors. Its primary goal is to ensure that no processor is idle while there are processes waiting for services in other processors. Load balancing may be necessary because of a variety of factors such as high network traffic causing the network connection to be a bottleneck or high computational load.

Stativ vs. Dynamic

Load balancing algorithms can be broadly categorized into static and dynamic. Static load balancing algorithms distribute processes/loads to processors at compile time relying on a priori knowledge about the processes and the system on which they run, while dynamic algorithms distribute loads to processors at run-time.

Static load balancing algorithms can be very attractive. For example, in parallel programs, the satic strategies could be useful if the execution times of processes and their communication requirements can be predicted, or, in other cases, they can be the only choice because the size of the processes'state prevents migration of processes during run−time. However, the dynamic strategies seems to be more effective. While the static load balancing algorithms rely on the estimated execution times of processes and interprocess communication requirements, this may be not satisfactory for parallel or distributed programs that are of the dynamic and/or unpredictable kind. For example, in a parallel combinatorial search application, processes evaluate candidate solutions from a set of possible solutions to find one that satisfies a problem−specific criterion. Each

process searches for optimal solutions within a portion of the solution space. The solution space usually change as the search proceeds. Portions that encompass the optimal solution with high probability will be expanded and explored exhaustively, while portions that have no solutions will be deleted at run−time. Consequently, processes are generated and destroyed at run−time. To ensure efficiency, processes have to be distributed at run−time.

Another example of dynamic and unpredictable program behavior is parallel simulation of molecular dynamics (MD), see figure 2.8. An MD program simulates the dynamic interactions among atoms in a system of interest for a period of time. At each time step, the simulation calculates the forces between atoms, the energy of the whole structure and the movements of atoms. Assume that each process of the program is responsible for simulating a portion of the system domain. As atoms tend to move around the system domain, the computational requirements of the processes may change from step to step and create an imbalance. To improve efficiency, processes'loads have to be redistributed periodically at run−time.

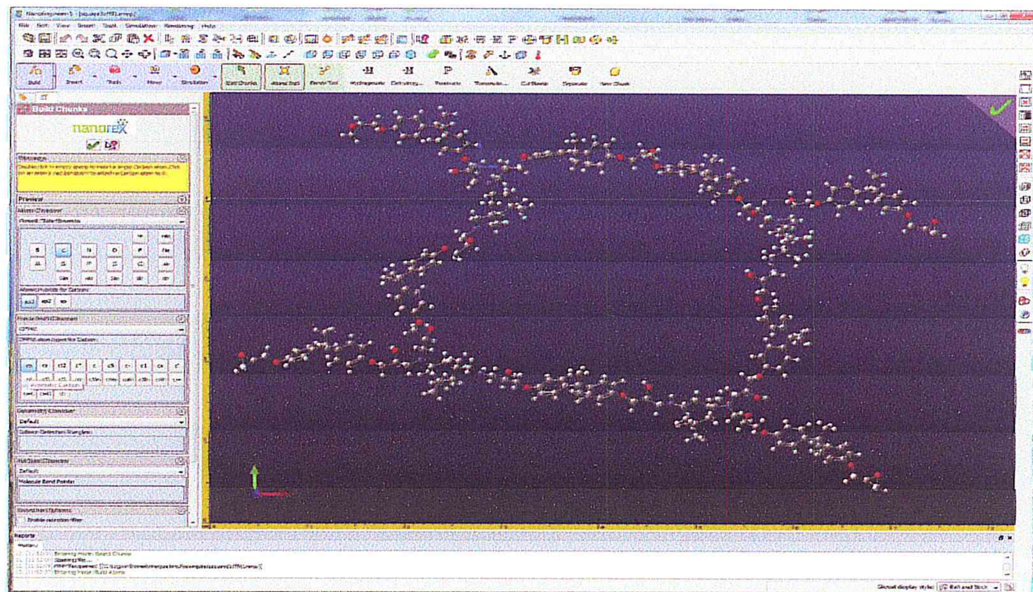


Figure 2.8: Molecular dynamics simulation software.

In the previous section we presented the main principles of distributed systems, then, we focused on one of their important challenges, the load balancing problem. In the next section, our intention is to introduce a general dynamic load balancing algorithm for a specific kind of loads, the indivisible loads.

Chapter 3

Self-stabilizing algorithm for indivisible integer-valued loads

In the following sections, the network is represented with a connected nonoriented graph $G = (V; E)$. V is the set of vertices of the graph representing the nodes, and E is the set of edges of the graph representing connections between nodes. Only static topology is considered assuming bi-directional communications for connected nodes.

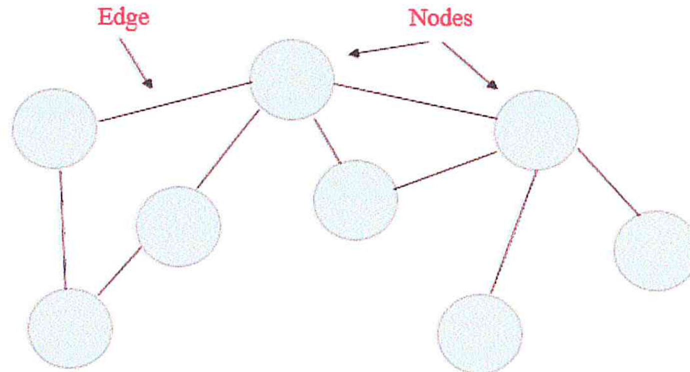


Figure 3.1: Simple network of 8 nodes.

The indivisible load balancing problem can be then stated as follows: Each node i in the network has an initial list of loads (this loads can not be divided). Then, when the imbalance is detected, each node maintains this list in coordination with his neighbors to reach the global legitimate state. The objective of the self stabilization algorithm is to ensure that, after a period of time, each node in the network has approximately the same sum of loads (figure 3.2).

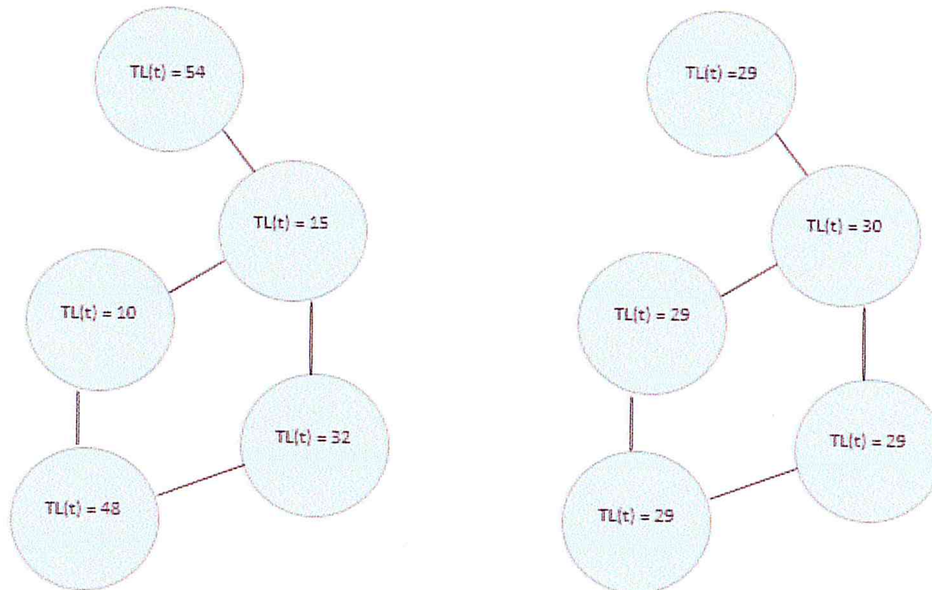


Figure 3.2: Network nodes before (right) and after (left) the balancing.

1 Self stabilazing algorithm for indivisible loads

In this section, we present a self stabilizing algorithm based on two balancing methods: Brute force and greedy sort. These two pairwise methods will work to balance the loads for each selected pair of nodes. The firt methods will serve as banchmark for our balancing method, the greedy sort method.

3.1.1 Self-stabilizing approach

A self-stabilizing system is a distributed system that can be started in any possible global state. Once started, the system regains its coherence by itself. After the faults occur, the system starts to converge to legitimate behavior. The self-stabilization property is very useful for systems in which processors may malfunction for a while, and then restart their operations. When there is a long enough period during which no processor malfunctions the system will stabilize.

The earlier study of self-stabilizing systems started with the fundamental paper of Dijkstra [15]. In 1974, Dijkstra introduced to computer science the notion of self-stabilization in the context of distributed systems. He defined a system as self-stabilizing when *«regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps»*. The best analogy to describe the definition of djikstra of self-stabilization notion could be the physical phenomena of the damped pendulum.

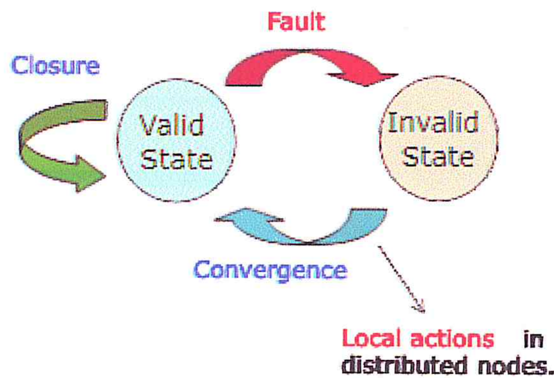


Figure 3.3: Self-stabilization approach.

Imagine a pendulum initially in position 1 (figure 3.4) in stable position, then, for a finite period of time the pendulum is perturbed (i.e someone can shake the pendulum or the wind could blow on the pendulum). After the perturbation, the pendulum will be in an arbitrary position (position 2). After a while, the pendulum will return to the stable position (position 3). This simple analogy helped in understading the notion of self-stabilization in the context of distributed systems.

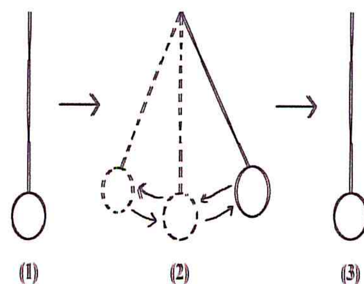


Figure 3.4: The damped pendulum analogy.

Usually, when designing distributed systems, there are always "legitimate configurations", that is, a configuration that are correct according to the specification of the system. In most cases, the set of "legitimate states" is stable, that is, if the system start from a legitimate configuration L and it can reach another configuration is also legitimate. However, the distributed system is a physical system, that is, it may be submitted to several kinds of faults that perturbes the system for a period of time, so, the system reachs an arbitrary configuration U which is likely not to be legitimate. This configuration is in some set of configuration that is likely to be larger then the set of legitimate configurations. In this case, saying that the system is self-stabilizing means that if there are no more perturbation, then, after a finite period of time , this system will reach the legitimate state and stay in it. This property of starting from an arbitrary configuration

and converting to a legitimate configuration and stay in it is called : self-stabilization.

$$L \xrightarrow[perturbation]{} \underbrace{U \supset L \xrightarrow{convergence} L}_{self-stabilization} \tag{3.1}$$

For this case of indivisible loads, the objective of self-stabilization is to arrive at a global ligitimate state using a set of rules that will transfer the node load greater than a threshold to neighbors using rules. These rules represent "atomic transactions" that will be used to move an atomic quantity from an overloaded node to an underloaded node within a number of steps until convergence. In our case, the legitimate state is a set of configurations of the network nodes loads that satisfies the following requirements :

1. The maximum load is non-increasing and the minimum load is non-decreasing.
2. The load difference between two nodes is minimized as much as possible in each lap.

In the figure 3.2, we notice that the maximum total load is decreasing from 54 to 30 and the minimum is increasing from 10 to 29. In the other hand, the delta between maximum and minimum has been reduced from 44 to 1.

3.1.2 The algorithm

Using the graph model $G = (V; E)$, each node i has only a partial view of the system. It is the local state that includes the state of the node and the state of its neighbors. The union of local states of the nodes is the global state of the distributed system.

In our algorithm, based on its local state, a node can execute a move which is a pre-defined rule. A node executes the set of rules as long as it is active (no equilibrium is reached).

For a node i, we have the following:

Variables :

$i.id$	Node identifier
$i.invit$	Invitation pointer
$i.accept$	Acceptation pointer

Macro :

$i.neighbors$	List of neighbors
$i.loads$	Current list of loads
$i.virtual - loads$	Virtual list of loads

Rules :

Invitation The first rule is the invitation rule. This rule checks the local state of the node to ensure that no invitation is pending. If applicable, the node checks the existence of neighbors with a total load difference superior to its smallest load. The invitation pointer is updated accordingly.

Accept invitation and begin transaction This rule checks, for a node i if a neighbor j has issued an invitation. And if the total load difference clause is valid, then the node i accept the invitation and update the pointer accordingly.

Confirm transaction This rule checks that invitation and acceptance pointers are paired. If true, then the virtual list of loads of both node i and j is updated accordingly with the pairwise method.

Commit transaction This rule checks the invitation and acceptance pointers, the current and future values then update the current list of i with its virtual list.

Release pointers after transaction This rule cleans the pointers states after an operation between nodes.

Correct pointers This rule checks pointer states and corrects them if applicable.

Algorithm 1 Self-Stabilizing algorithm for indivisible integer-valued loads

$R1$: Invitation //executed by the underloaded node
if $i.I = null$ and $i.A = null$ and $\exists j \in N(i)$ and $\exists l \in L(i)$: $l \leq TotalLoad[L(j)]) - TotalLoad[L(i)]$ **then**
  $(i.I \leftarrow j)$
end if
$R2$: Accept invitation and begin transaction //executed by the overloaded node
if $\exists j \in N(i)$ and $\exists l \in L(j)$:$l \leq TotalLoad[L(j)]) - TotalLoad[L(i)]$ and $j.I = i$ and $i.A = null$ and $i.I = null$ **then**
  $(i.A \leftarrow j)$
end if
$R3$: Confirm transaction
if $\exists j \in N(i)$ and $j.A = i$ et $i.I = j)$ and $\exists l \in L(j)$:$l \leq TotalLoad[L(j)]) - TotalLoad[L(i)]$ **then**
  $(Pair - Wise)$
end if
$R4$: Commit transaction
if $(\exists j \in N(i)$: $(j.A = i$ and $i.I = j)$ or $(j.I = i$ and $i.A = j)$ and $TotalLoad[V(i)]) = TotalLoad[V(j)]$and $TotalLoad[L(i)]) \neq TotalLoad[V(i)$ **then**
  $L(i) \leftarrow V(i)$
end if
$R5$: Release pointers after transaction
if $\exists j \in N(i)$: $(j.A = i$ and $i.I = j)$ or $(j.I = i$ or $i.A = j)$ and $TotalLoad[L(j)]) = TotalLoad[L(i)]$ **then**
  $(i.I \leftarrow null)$
  $(i.A \leftarrow null)$
end if
$R6$: Correct pointers
if $i.I = j$ and $(j \notin N(i)$ or $l > TotalLoad[L(j)]) - TotalLoad[L(i)]$ and $j.A \neq i$ and $L[j] = V[j])$ **then**
  $(i.I \leftarrow null)$
end if
if $(i.A = j$ and $(j \notin N(i)$ or $j.I \neq i$ or $l > TotalLoad[L(j)]) - TotalLoad[L(i)]$ and $L[j] = V[j])$ **then**
  $(i.A \leftarrow null)$
end if

3.1.3 Pairwise methods

As the imbalance is detected, the balance is applicated using the pairwise method. This method is used to distribute the loads between the pair of nodes. In the following are the outlines of this method :
For each pair $(i; j) \in E$:

1. Merge the two current lists of loads.

2. Set the current lists of loads to none.

3. For each item of load in the merged list do :
    - Select the least loaded node between i and j.
    - Assign the load item to the selected node.
    - Update the future list of the selected node.

In the previous method, we focus on the mutual exclusion property. In computer science, the mutual exclusion is property of concurrency control which is intuited for the purpose of preventing race condition. It is a requirement that one thread of execution never enter its critical section at the same time that another concurring thread of execution enter its own critical section. The requirement of metual exclusion was first indentified by Dijkstra in [14]. In our solution, The term "mutual exclusion" stems from the fact that, at each iteration (time-step), we may have parallel but independent pairs of neighbors involved in the balancing process.

In our implementation, we have choose to use 2 pairwise methods. Once the imbalance is detected (the load difference between a node i and its neighbor j is greater than the smallest load of j), the balance begins by executiong the choosen method. In the following, we will explain each one in details.

Brute-force search

One way of balancing load is the "brute-force search" method, also called "exhaustive search" or "generate and test". It's a technic that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. For example, to find the divisors of a natural number n, A force brute search method would enumerate all integers from 1 to n, and check whether each of them divides n without remainder. A brute-force approach for the eight queens puzzle would examine all possible arrangements of 8 pieces on the 64-square chessboard, and, for each arrangement, check whether each (queen) piece can attack any other.

The brute-force algorithms are known to be simple to implement, slow in exection, tedious to test, and will always find a solution if it exists. They are typically used in critical applications where any errors in the algorithm would have very serious consequences; or when the simplicity of implementation is more important than speed. Furthermore, the force-brute search method is also useful as a baseline method to which all others are compared. In the experimental results section, the brute-force search will be considered as a baseline [12].

obviously, our method will enemurate every possible combination of loads, and then select the combination with the smallest total load difference (figure 3.5).

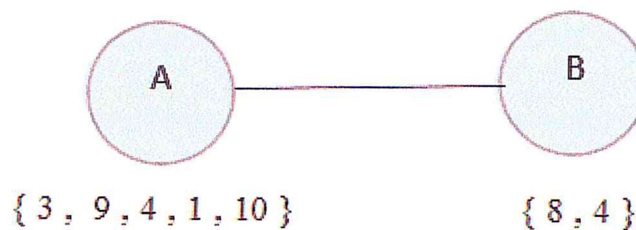


Figure 3.5: A pair of nodes example.

Greedy simple

A greedy method is simple method used to solve optimization problems. The problems that are solved using this method include finding the better order to execute a certain set of jobs on a computer, finding the shortest path in graph, etc. It is a method that follows the problem approach in making the locally optimal choice at each stage with the hope of finding a global optimum solution [4].

To slove an optimization problem, we look for a set of condidats constituting a solution that optimizes (minimizes or maximazes, as the case may be) the value of the objective function. A greedy algorithm proceeds step by step. Initially the set of condidates is empty. Then at each step, we try to add to this set the best remaining candidates, our choice being guided by the selection function. The selection function is dependent on the problem at hand. For example, the selection function in the case of minimum weight spanning tree picks an edge of minimum weight from the remaining edges, an object with maximum profit per unit weight out of the remaining objects is chosen for puting in the knapsack in the case of knapsack problem, or for our case, the node with minimum total load is selected to receive the item load.

Using the pair of nodes in figure 3.5, the application of greedy simple will be as follows:
Fusion list $=\{3, 9, 4, 1, 10, 8, 4\}$.
First step : $A=\{\}, B=\{\}$.
Distibuting the loads to the underloaded node in each step:

- $A=\{3\}, B=\{\}$
- $A=\{3\}, B=\{9\}$
- $A=\{3, 4\}, B=\{9\}$
- $A=\{3, 4, 1\}, B=\{9\}$
- $A=\{3, 4, 1\}, B=\{9, 10\}$

- $A = \{3\ ,\ 4\ ,\ 1\ ,\ 8\}, B = \{9\ ,\ 10\}$
- $A = \{3\ ,\ 4\ ,\ 1\ ,\ 8\ ,\ 4\}, B = \{9\ ,\ 10\}$

The greedy methods are usually known to be simple, easy to implement and run fast. However, the use of a greedy simple method do not always provide a globally optimum solution. In the following example we see how the greedy simple method makes a locally optimal choices, however, ends up to a non optimal global solution.

Greedy sort

Basing on the same concept as greedy simple, the greedy sort adds the property of sorted load. The same approach used in the method above will be applicated on a sorted fusion list. This method will garanty to the greedy method the optimal global solution.
Fusion list = $\{3\ ,\ 9\ ,\ 4\ ,\ 1\ ,\ 10\ ,\ 8\ ,\ 4\}$.
Fusion list sorted = $\{1\ ,\ 3\ ,\ 4\ ,\ 4\ ,\ 8\ ,\ 9\ ,\ 10\}$.
First step : $A = \{\}, B = \{\}$.
Distibuting the loads to the underloaded node in each step:

- $A = \{1\}, B = \{\}$
- $A = \{1\}, B = \{3\}$
- $A = \{1\ ,\ 4\}, B = \{3\}$
- $A = \{1\ ,\ 4\}, B = \{3\ ,\ 4\}$
- $A = \{1\ ,\ 4\ ,\ 8\}, B = \{3\ ,\ 4\}$
- $A = \{1\ ,\ 4\ ,\ 8\}, B = \{3\ ,\ 4\ ,\ 9\}$
- $A = \{1\ ,\ 4\ ,\ 8\ ,\ 10\}, B = \{3\ ,\ 4\ ,\ 9\}$

In this chapter, we have introduced a self stabilizing algorithm for load balancing in arbitrary networks. Dealing with indivisible loads, the idea of the algorithm was to use a pairwise method that allows a pair of neighbors to share and redestibute their loads. In the next chapter, the algorithm will be implemented and the experimantal results of the implementation of the algorithm and the pairwise methods will be discussed.

Chapter 4

Numerical results

In this section, we provide numerical results to show the impact of different parameters like network size, network topologie, centralized and decentralized approach ,and the pair-wise methods on our algorithm. To evaluate the algorithm, we conducted a series of experiments using codeblocks where the algorithm was implemented. The goals of these experiments are to study and measure the scaling properties and convergence behavior of the algorithm in several context.

In our experiments, the network is modeled as a graph, the edges representing the links between nodes. We considered different number of nodes for the network in these experiments: 200, 400 and 600. the graphs are generated following four possible topologies: grid, torus, hypercube and random (hash−net). The initial values and the number of loads of the network nodes are random values in the range $[1-10]$. For each size of graph, we consider 25 executions of the algorithm then the results are averaged.

1 First Experiment: Brute force vs. Greedy Sort

In first experiment we consider the maximum (MaxL) and minimum (MinL) total loads by iterartion using brute force and the greedy sort method (figure 4.1).
The results show that the two methods bring us to the same results, with more iterations to the sorted method. The experiments show also the evaluation of the difference value between that max total load and the minimum total load that is decreasing with each iteration to reach almost a negligible value (figure 4.2).

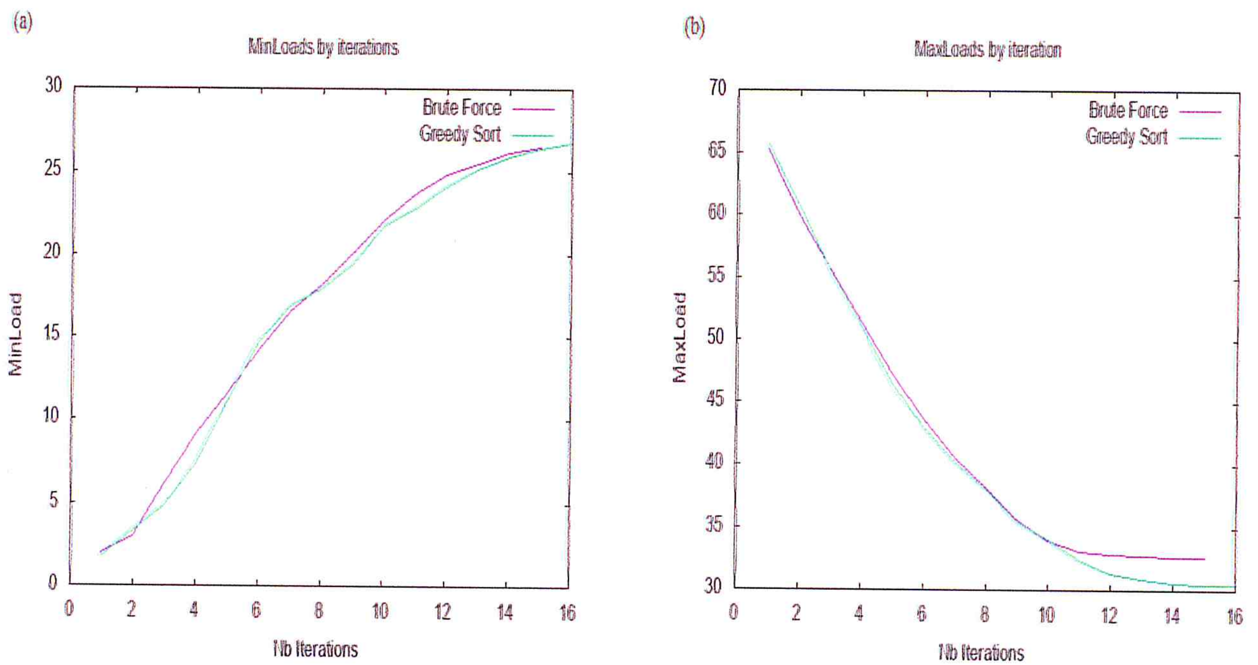


Figure 4.1: Minimum (a) and Maximum (b) total load by iteration for 200 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods.

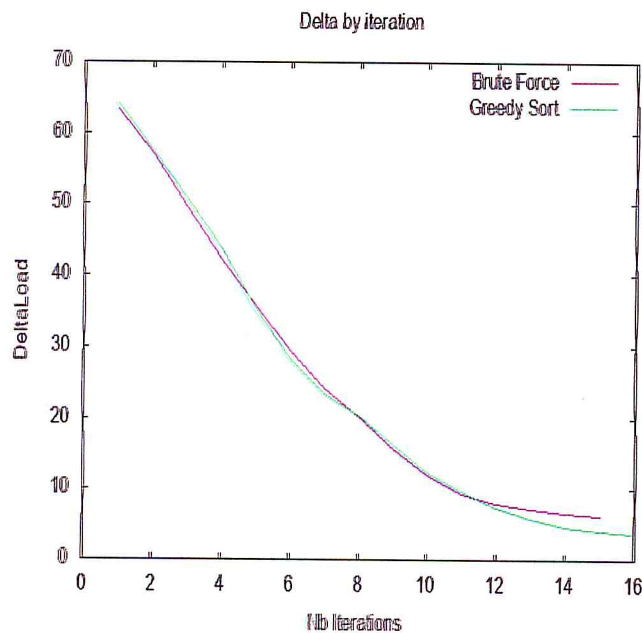


Figure 4.2: Delta by iteration for 200 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods.

To study the impact of the network size on our algorithm, the results of different sizes execution have been gathered in the following figure (4.3), the graph shows that the number of iterations increases whenever the network topology size increases.

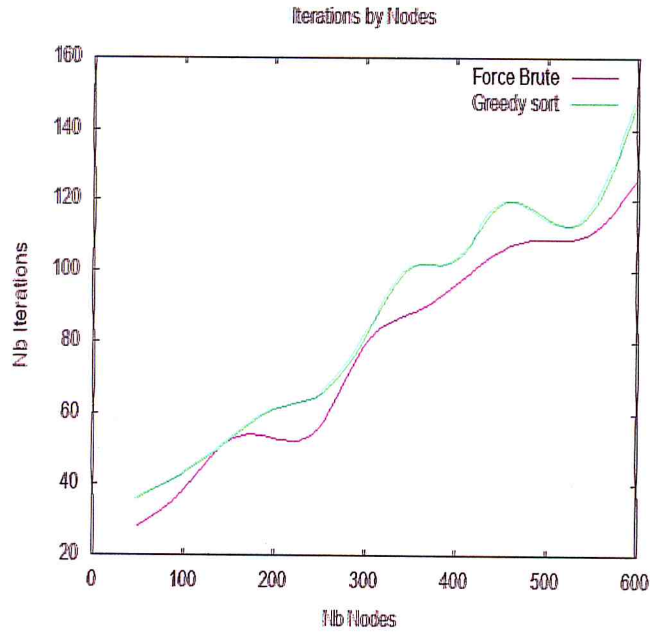


Figure 4.3: Itrations by nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods.

2 Second Experiment: Centralized vs. Decentralized

The load balancing algorithms can be divided into two groups: centralized and decentralized [19]. In this experiment, we present the behavior of self-stab (self–stabilizing) algorithm in both cases using the greedy sort method. In the centralized case, a master node collects the loads and distributes them among the different nodes of the network. However, in the decentralized one, the load balancing is performed locally between each node and his neighbors. As experiment 1, the delta, minimum and maximum total load are represented in figures 4.4 and 4.5

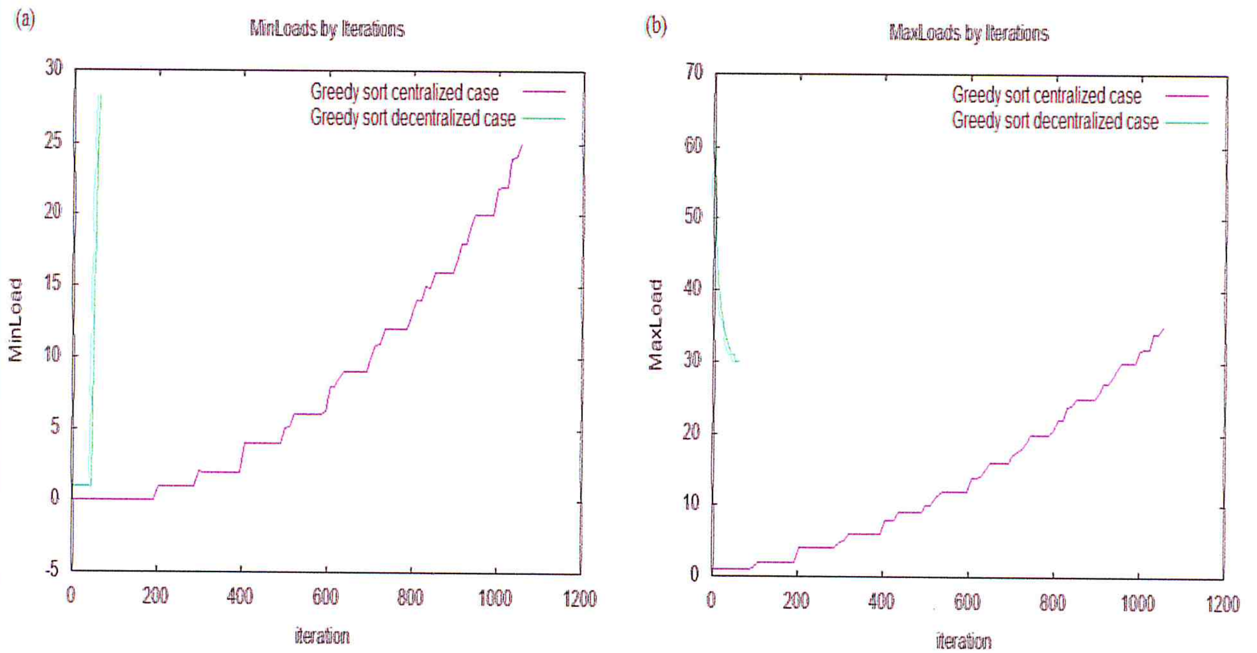


Figure 4.4: Minimum (a) and Maximum (b) total load by iteration for 200 nodes: Centralized case (in purple) and Decentralized case (in green) using Greedy Sort.

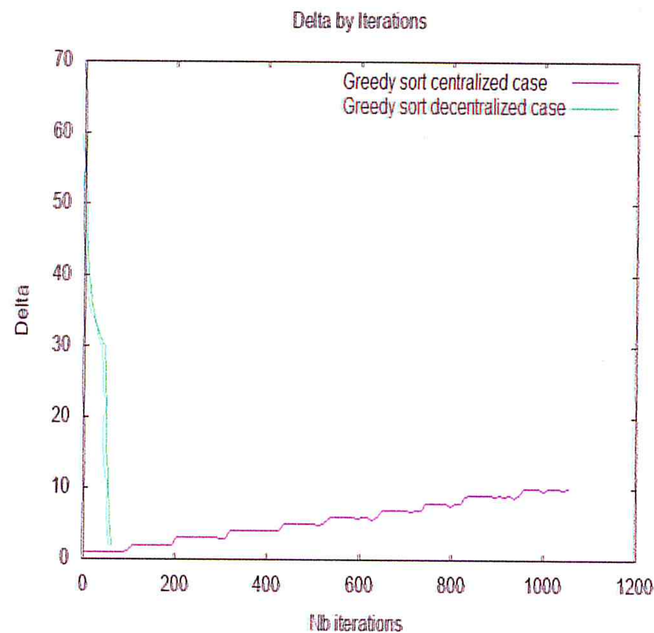


Figure 4.5: Delta by iteration for 200 nodes: Centralized case (in purple) and Decentralized case (in green) using Greedy Sort.

To study the impact of the network size on our algorithm, the results of different sized execution have been gathered in the following figure (figure 4.6), the graph shows that

the number of iterations increases whenever the network topology size increases in both cases and significantly in the centralized case.

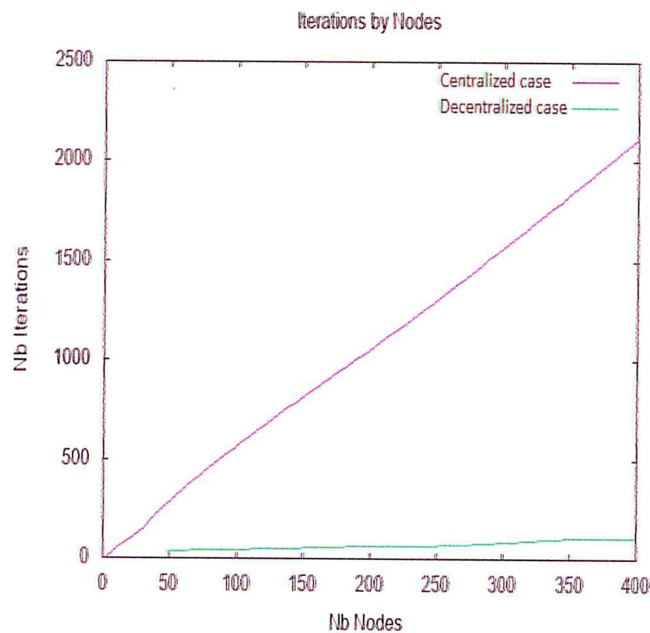


Figure 4.6: Itrations by nodes: Centralized case (in purple) and Decentralized case (in green) using Greedy Sort.

3 Third Experiment: Decentralized algorithm, Extreme case

The figures 4.7 and 4.8 show a comparison of the decentralized self-stab algorithm and decentralized self-stab algorithm with all loads in one node. This experiment is to show the behavior of our algorithm in the most delicate case of all loads arriving to a single node.

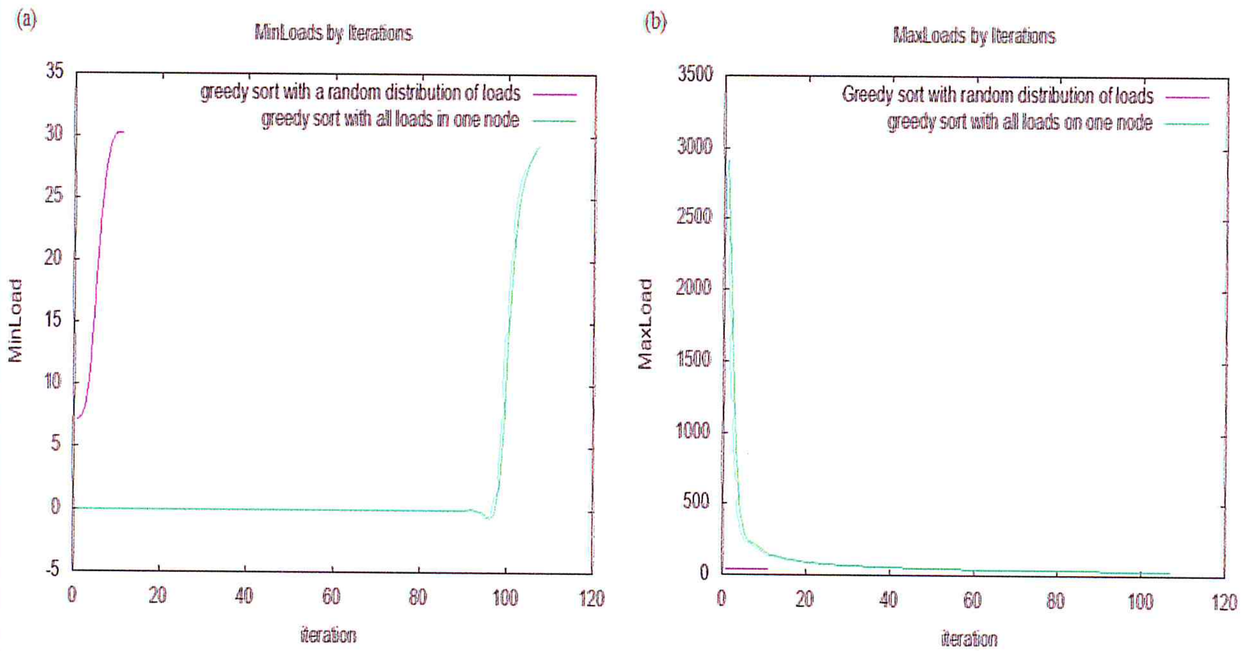


Figure 4.7: Minimum (a) and Maximum (b) total load by iteration for 200 nodes: Decentralized case (in purple) and Decentralized with all loads in one node (in green) using Greedy Sort.

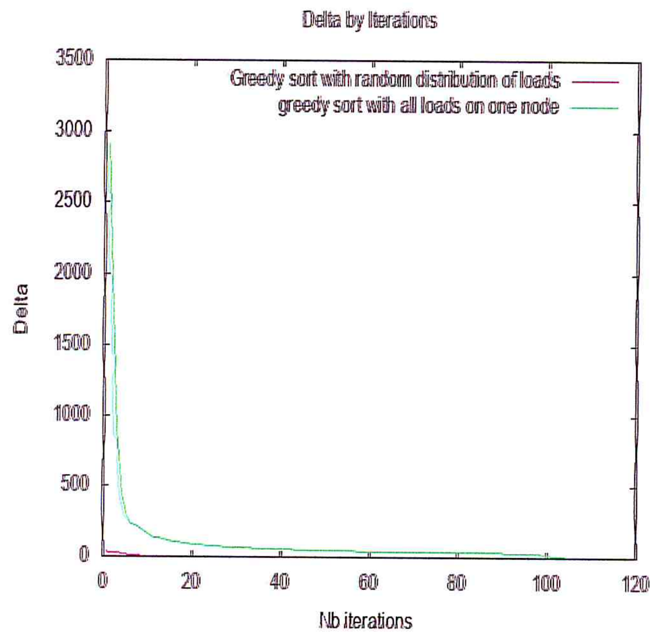


Figure 4.8: Delta by iteration for 200 nodes: Decentralized case (in purple) and Decentralized with all loads in one node (in green) using Greedy Sort.

4 Fourth Experiment: Different topologies

The nodes of a distributed system need to communicate by some means. This is effected by connecting the node to some form of network. A communications network may be either static or dynamic. Static networks have a fixed connections, while dynamic networks have switching connections which can change in time [1].

The mathematical term for the study of connectedness is 'topology'. Networks investigated in the literature includes: chains, grids, hypercubes, toruses, Hash—Nets (Random), ect; that may have many properties as diameter, multiplicity of paths or coupling. The choice of topology can determine the efficienty of an algorithm. In this section, we will execute the self—stab algorithm using different static topologies to study the impact of the coupling parameter.

To show the impact of the network topologies (loosely—coupled and strongly—coupled), we excute the self-stab algorithm in four different topologies.

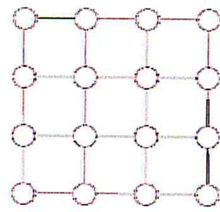
- **Grid.** In the grid topology, the nodes are stacked one on top of another, so the processors are connected both vertically and horizontally. Figure 4.9 shows the resulting topology which is called a 'grid' or a 'mesh'.

- **Torus.** Grids can be extended by connecting the loose ends together in the same way the chain is converted to a ring, see figure 4.9. This topology is called a 'toroidally connected grid' or 'torus'. All nodes in a torus have four neighbors nodes.

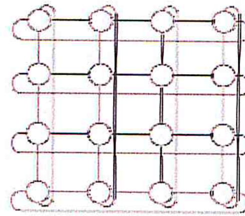
- **Hypercube.** Simple cubes have three dimensions, 'hypercubes' are produced by increasing the number of dimensions in a cube. The term hypercube refers to any cube with four or more dimensions, see figure 4.9.

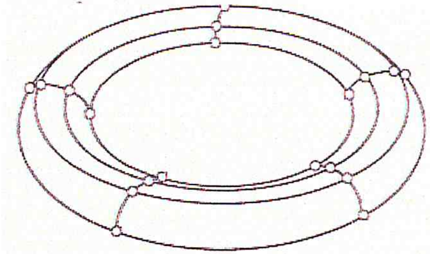
- **Hash-Nets/Random.** It's an anarchic approach to the choice of topology for a distributed system. It consists of connecting everything randomly, see figure 4.9.

[1] www.gigaflop.co.uk

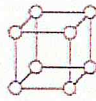


Grid

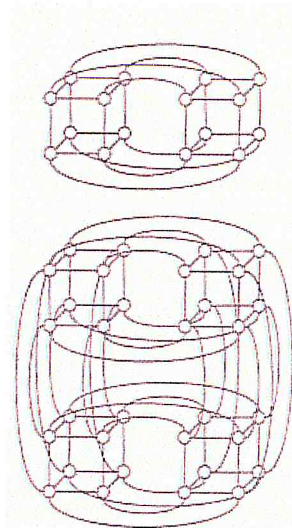




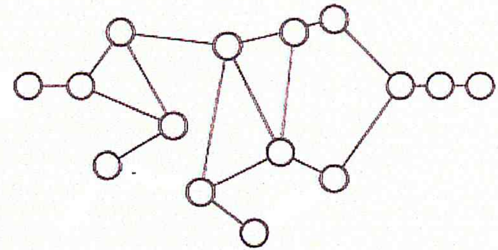
Torus



Cube



Hypercube



Hash-Net / Random

Figure 4.9: Diffrent network topologeis.

The results of executing the self-stab algorithm using greedy sort method in different topologies are presented in the following (figures 4.10 and 4.11).

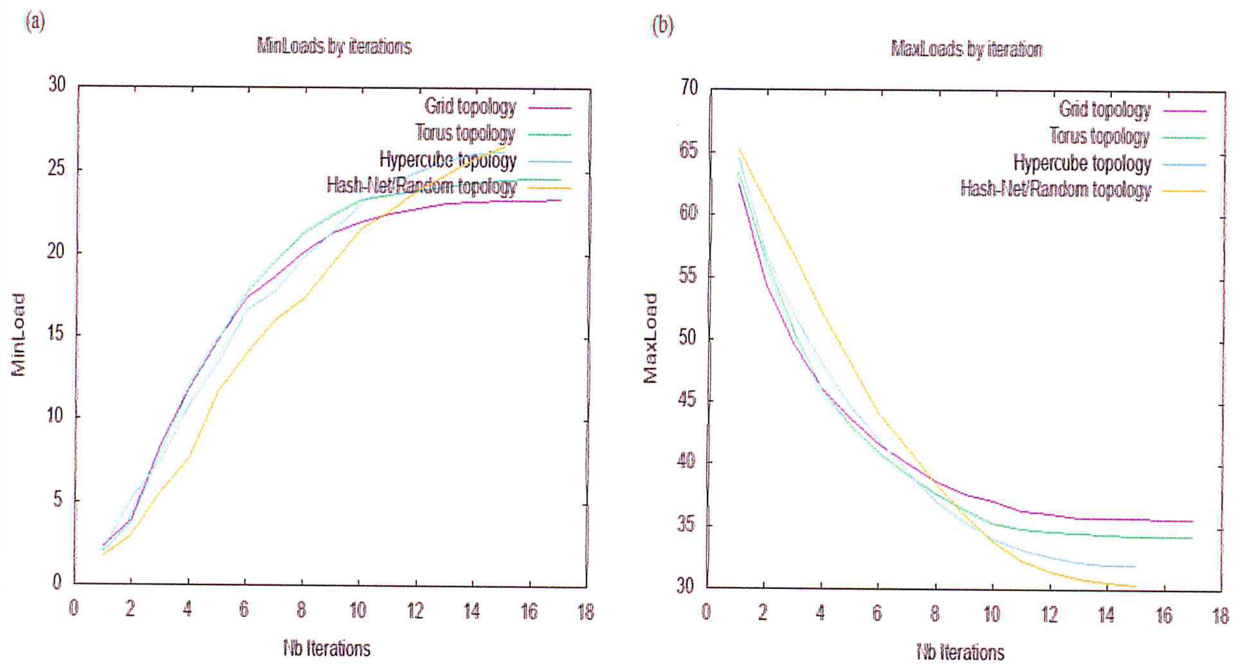


Figure 4.10: Minimum (a) and Maximum (b) total load by iteration: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash−Net topology (in orange) using Greedy Sort.

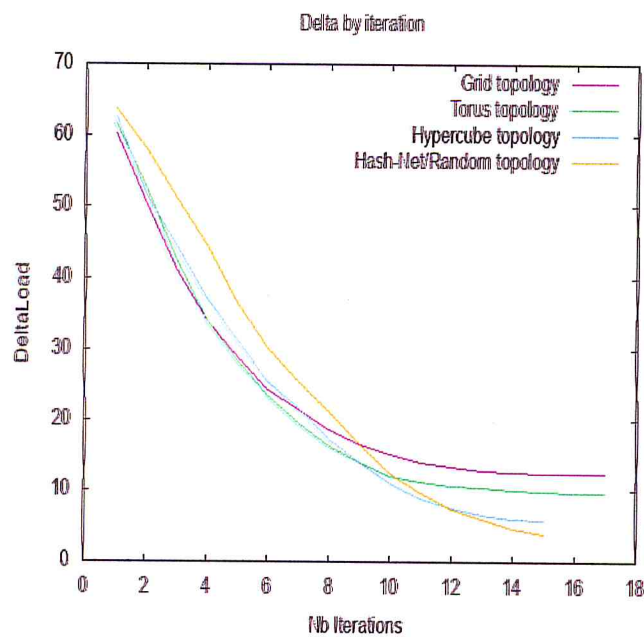


Figure 4.11: Delta by iteration: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash−Net topology (in orange) using Greedy Sort.

The study of the network size on the different topologies is provided in **Appendix** (figures 0.18, 0.19, 0.20 and 0.21).

5 Discussing

In the previous section, the experiments show the convergence of the self-stabilizing algorithm and offer a study of the differents pairwise methods that was implemented. The first experiment has evaluated the capability of the algorithm to stabilizing. In this expirement, the brute force and greedy sort show a similar behavior with more iterations for the sorted method. This can be justified by the fact that the sorted pairwise method affects an item load in each time step (iteration), otherwise, the brute force approach, besides of wastiong time in enumerating all combinations, the brute force shares load between the pair of nodes in a one time step. In table 4.1 the execution time (in seconds) of both methods is presented. These results show clearly that the sorted method is faster than the brute force method.

	Brute Force (s)	Greedy Sort (s)
200	171.491	70.416
400	427.285	276.688
600	849.754	570.19

Table 4.1: Exection time of Brute Force and Greedy Sort.

The second experiment involves the greedy sort method in two different contexts: Centralized and Decentralized. The previous results and the table 4.2 show that self–stab algorithm converges faster in distributed environement then a centrazlized with less iterations, that is due to the fact that in centralized algorithm one load is distributed at each step to a single node, this makes more iterations and most importantly more time.

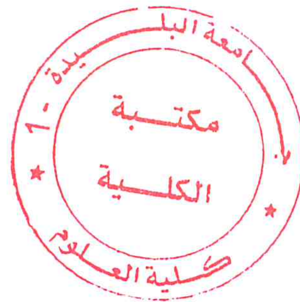
	Centralized (s)	Decentralized (s)
200	6811.24	70.416
400	87154	276.688
600	150862	570.19

Table 4.2: Exection time of Centralized and Decentralized self-stab algorithm.

Another experiment was to show the behavior of the distributed self–stab in the most delicated case, the case when all the loads arriving to the system arrive to a single node. The goal of this experiment is to compare to the centralized case. The results show that the extreme case takes more iteration then usual. This can be explained as in the extreme case all the network is empty, and the delta between the maximum and the minimum total load is huge at the beginning so that takes more iterations and more time. But in comparaison to the centralized case, the extreme self-stab algorithm reacts much better.

The fourth and the final experiment was the execution of self–stab in different levels of coupled networks, in topologies where nodes have more or less number of neighbors.

The self–stab algorithm in a "grid", then a more coupled topology the "torus", and then another more coupled topology "hypercube", and even more in an anarchic topology shows approximately the same behavior. The self–stab algorithm converges in the different topologies and for different number of nodes.



Conclusion and Future work

In this report, we have presented a load balancing algorithm for any abstarct network. The main challenge in building this algorithm was the indivisible nature of the loads coming to the network. The presented algorithm is based on a self–stabilizing approach in which each node can initiate transactions with its neighbors and reach a legitimate state with a limited number of iterations. To demonstrate the algorithm capabilities we have benchmarked the self-stab algorithm using the proposed balancing method "Greedy Sort" with the same algorithm using "Brute Force" method. Other experiments as distributed vs. non distribitued network, topology design impact was their to improve the expected behavior of the self– stab algorithm.

The analysis of the experiments results emphasize the advantages of the "Greedy Sort" self–stablizing algorithm especially in a distributed environement. Although, several ways of improvements and future research directions can be proposed. The analysis of the self–stab algorithm usisng the discrete event simulation would be very usufull to show the real behavior of the algorithm. Furthermore, the load balancing is known to be an NP-complete problem and the designng of an heuristic to solve it will be a very interesting future work.

Bibliography

[1] Mahfooz Alam and Zaki Ahmad Khan. Issues and challenges of load balancing algorithm in cloud computing environment. *Indian Journal of Science and Technology*, 10(25), 2017.

[2] Tuncer Can Aysal, Mehmet Ercan Yildiz, Anand D Sarwate, and Anna Scaglione. Broadcast gossip algorithms for consensus. *IEEE Transactions on Signal processing*, 57(7):2748–2761, 2009.

[3] Jacques Bahi, Raphaël Couturier, and Flavien Vernier. Synchronous distributed load balancing on dynamic networks. *Journal of Parallel and Distributed Computing*, 65(11):1397–1405, 2005.

[4] SK Basu. *Design methods and analysis of algorithms*. PHI Learning Pvt. Ltd., 2013.

[5] Petra Berenbrink, Colin Cooper, Tom Friedetzky, Tobias Friedrich, and Thomas Sauerwald. Randomized diffusion for indivisible loads. *Journal of Computer and System Sciences*, 81(1):159–185, 2015.

[6] Veeravalli Bharadwaj, Debasish Ghose, Venkataraman Mani, and Thomas G Robertazzi. *Scheduling divisible loads in parallel and distributed systems*, volume 8. John Wiley & Sons, 1996.

[7] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.

[8] Junwei Cao, Daniel P Spooner, Stephen A Jarvis, and Graham R Nudd. Grid load balancing using intelligent agents. *Future generation computer systems*, 21(1):135–149, 2005.

[9] Yuan-Chieh Chow et al. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, 100(5):354–361, 1979.

[10] Michal Cierniak, Mohammed Javeed Zaki, and Wei Li. Compile-time scheduling algorithms for a heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

[11] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301, 1989.

[12] Taz Daughtrey and Sue Carroll. *Fundamental concepts for the software quality engineer*, volume 2. ASQ Quality Press, 2007.

[13] Tushar Desai and Jignesh Prajapati. A survey of various load balancing techniques and challenges in cloud computing. *International Journal of Scientific & Technology Research*, 2(11):158–161, 2013.

[14] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[15] EW Dijkstra. Self-stabilizing systems in spite of distributed control. commua. *ACM 17, 11 (Nov. 1974)*, pages 643–644.

[16] Derek L Eager, Edward D Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *ACM SIGMETRICS Performance Evaluation Review*, 13(2):1–3, 1985.

[17] YT ëWM85ë. Wang and rjt morris. load sharing in distributed systems. *IEEE Transactions on Computers, C-34: 204í217*, 1985.

[18] Sukumar Ghosh. Distributed systems: An algorithmic approach, (chapman hall/crc computer and information science series), 2007.

[19] Sukalyan Goswami and Ajanta De Sarkar. A comparative study of load balancing algorithms in computational grid environment. In *Computational Intelligence, Modelling and Simulation (CIMSim), 2013 Fifth International Conference on*, pages 99–104. IEEE, 2013.

[20] Daniel Grosu and Anthony T Chronopoulos. Noncooperative load balancing in distributed systems. *Journal of parallel and distributed computing*, 65(9):1022–1034, 2005.

[21] Mounir Hamdi and Chi-Kin Lee. Dynamic load balancing of data parallel applications on a distributed network. In *Proceedings of the 9th international conference on Supercomputing*, pages 170–179. ACM, 1995.

[22] Shuichi Ichikawa and Shinji Yamashita. Static load balancing of parallel pde solver for distributed computing environment. In *Proc. 13th Int'l Conf. Parallel and Distributed Computing Systems*, pages 399–405, 2000.

[23] Hisao Kameda, Jie Li, Chonggun Kim, and Yongbing Zhang. *Optimal load balancing in distributed computer systems*. Springer Science & Business Media, 2012.

[24] Chonggun Kim and Hisao Kameda. Optimal static load balancing of multi-class jobs in a distributed computer system. *IEICE TRANSACTIONS (1976-1990)*, 73(7):1207–1214, 1990.

[25] P Venkata Krishna. Honey bee behavior inspired load balancing of tasks in cloud computing environments. *Applied Soft Computing*, 13(5):2292–2303, 2013.

[26] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[27] Rich Lee and Bingchiang Jeng. Load-balancing tactics in cloud. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2011 International Conference on*, pages 447–454. IEEE, 2011.

[28] Jie Li and Hisao Kameda. A decomposition algorithm for optimal static load balancing in tree hierarchy network configurations. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):540–548, 1994.

[29] Simone A Ludwig and Azin Moallem. Swarm intelligence approaches for grid load balancing. *Journal of Grid Computing*, 9(3):279–301, 2011.

[30] Yuval Rabani, Alistair Sinclair, and Rolf Wanka. Local divergence of markov chains and the analysis of iterative load-balancing schemes. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 694–703. IEEE, 1998.

[31] Xiaona Ren, Rongheng Lin, and Hua Zou. A dynamic load balancing strategy for cloud computing platform based on exponential smoothing forecast. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 220–224. IEEE, 2011.

[32] Siddharth Sonawane, Prathmesh Arnikar, Ankita Fale, Sagar Aghav, and Shikha Pachouly. Load balancing in cloud computing. *International Journal*, 4(2), 2014.

[33] P Beaulah Soundarabai, A Sandhya Rani, Ritesh Kumar Sahai, J Thriveni, and KR Venugopal. Comparative study on load balancing techniques in distributed systems [j]. *International Journal of Information Technology*, 6(1):53–60, 2012.

[34] Asser N Tantawi and Don Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM (JACM)*, 32(2):445–465, 1985.

[35] Asser N Tantawi and Donald F Towsley. A general model for optimal static load balancing in star network configurations. In *Proceedings of the Tenth International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 277–291. North-Holland Publishing Co., 1984.

[36] Alexander Thomasian. A performance study of dynamic load balancing in distributed systems. In *ICDCS*, pages 178–184, 1987.

[37] Shu-Ching Wang, Kuo-Qin Yan, Wen-Pin Liao, and Shun-Sheng Wang. Towards a load balancing in a three-level cloud computing network. In *Computer Science and information technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 1, pages 108–113. IEEE, 2010.

[38] Marc H Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on parallel and distributed systems*, 4(9):979–993, 1993.

[39] Belabbas Yagoubi and Yahya Slimani. Dynamic load balancing strategy for grid computing. *Transactions on Engineering, Computing and Technology*, 13:260–265, 2006.

[40] Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for a network of workstations. In *High Performance Distributed Computing, 1996., Proceedings of 5th IEEE International Symposium on*, pages 282–291. IEEE, 1996.

1

Appendix

Brute Force vs. Greedy Sort : Min, Max and Delta for 400 and 600 nodes

The following figures show the same previous results of min, max and delta of the different experiments for both 400 and 600 nodes.

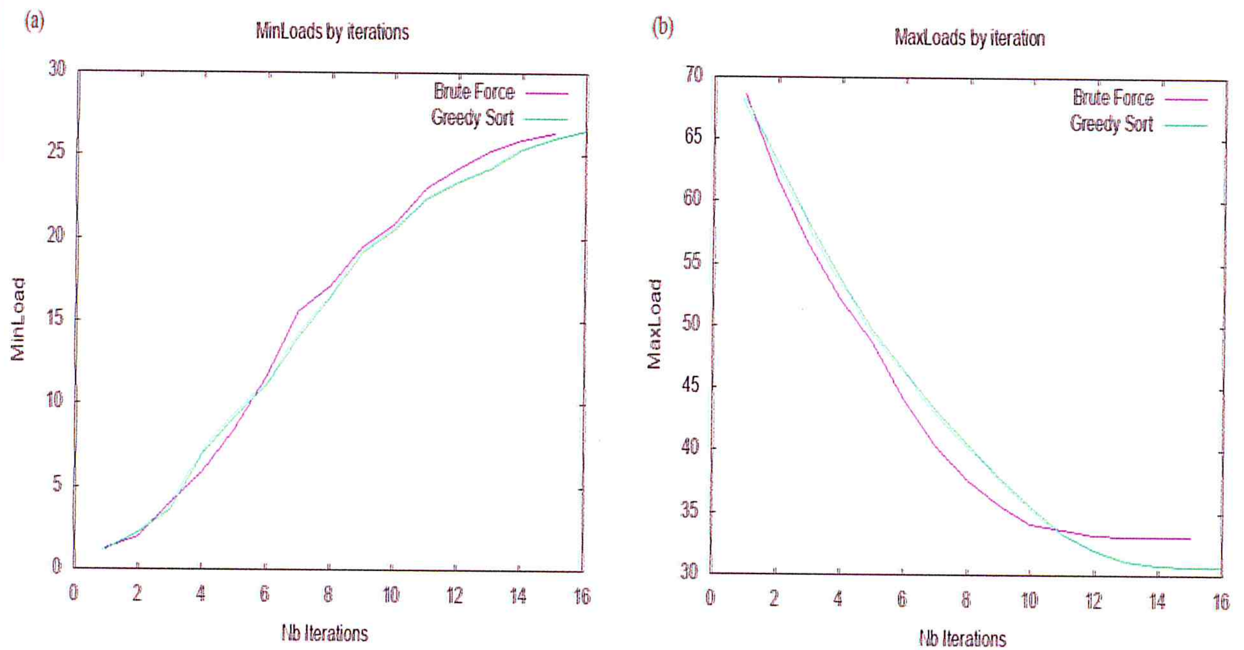


Figure 0.12: Minimum (a) and Maximum (b) total load by iteration for 400 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods.

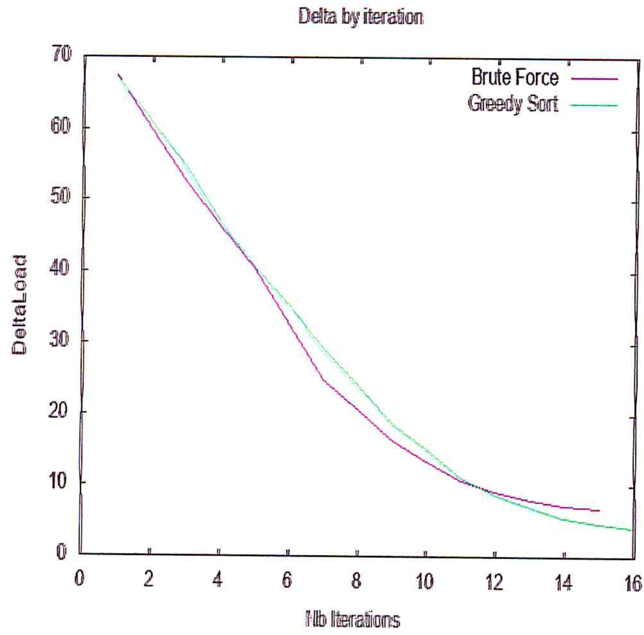


Figure 0.13: Delta by iteration for 400 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods.

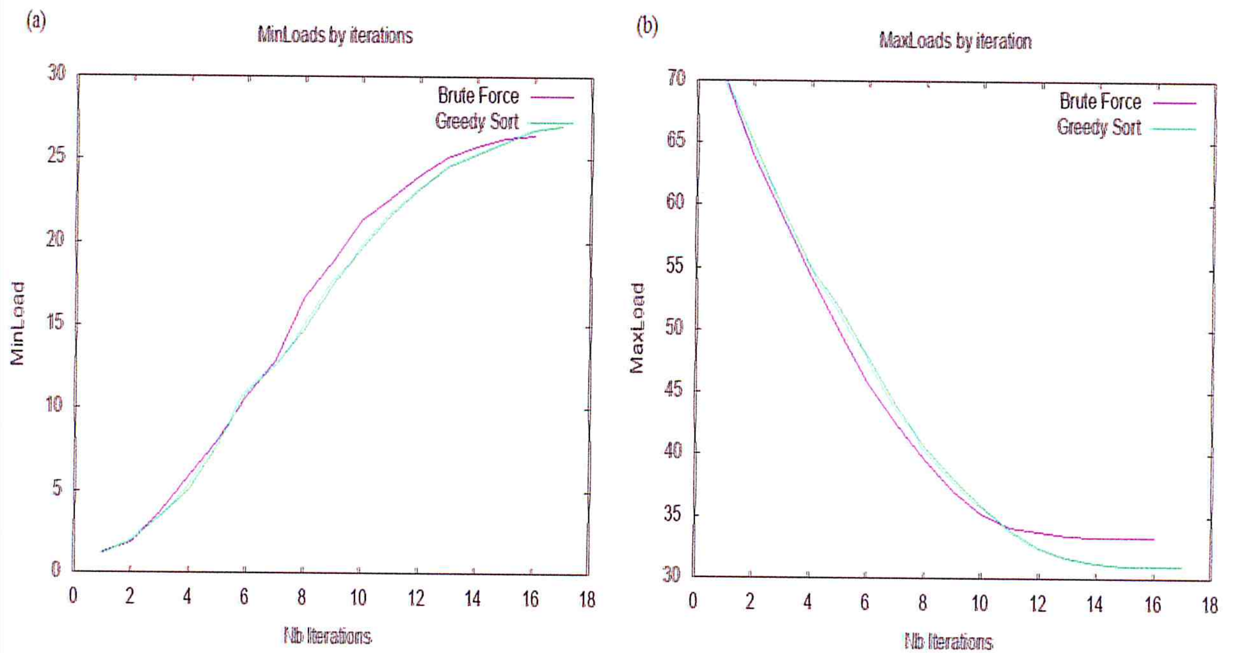


Figure 0.14: Minimum (a) and Maximum (b) total load by iteration for 600 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods.

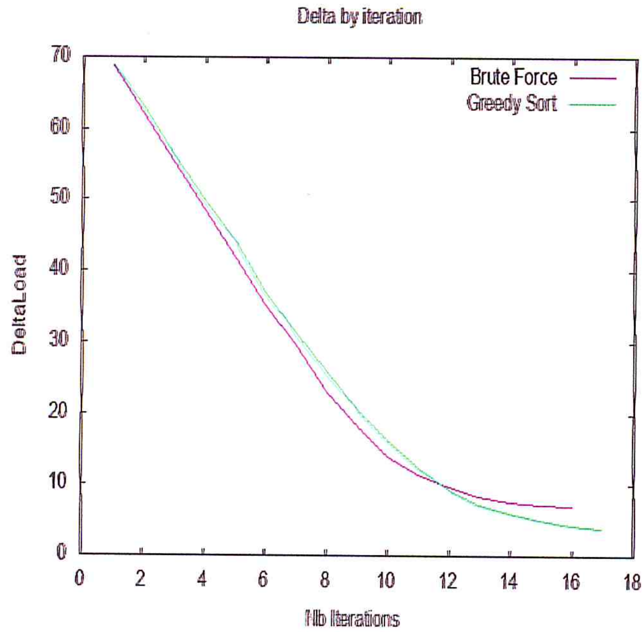


Figure 0.15: Delta by iteration for 600 nodes: Comparaison of Brute Force (in purple) and Greedy sort (in green) pairwise methods.

Centralized vs. Decentralized : Min, Max and Delta for 400

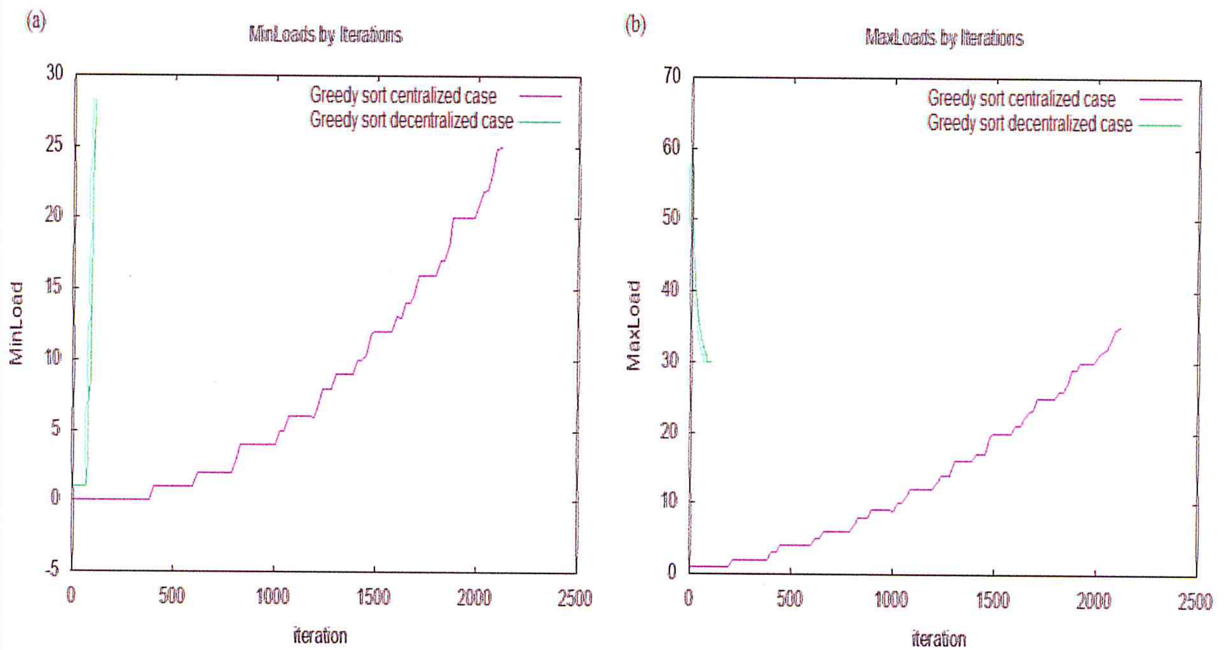


Figure 0.16: Minimum (a) and Maximum (b) total load by iteration for 400 nodes: Comparaison of Centralized (in purple) and Decentralized (in green) cases.

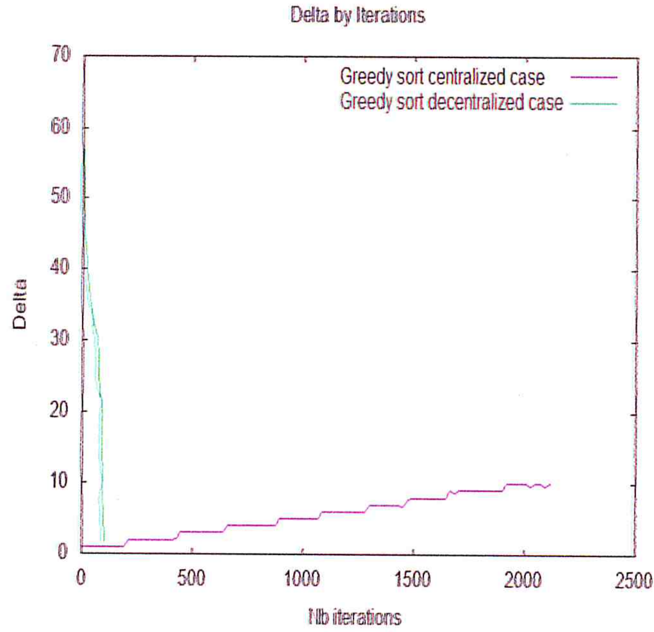


Figure 0.17: Delta by iteration for 400 nodes: Comparaison of Centralized (in purple) and Decentralized (in green) cases.

Different topologies : Min, Max and Delta for 400 and 600 nodes

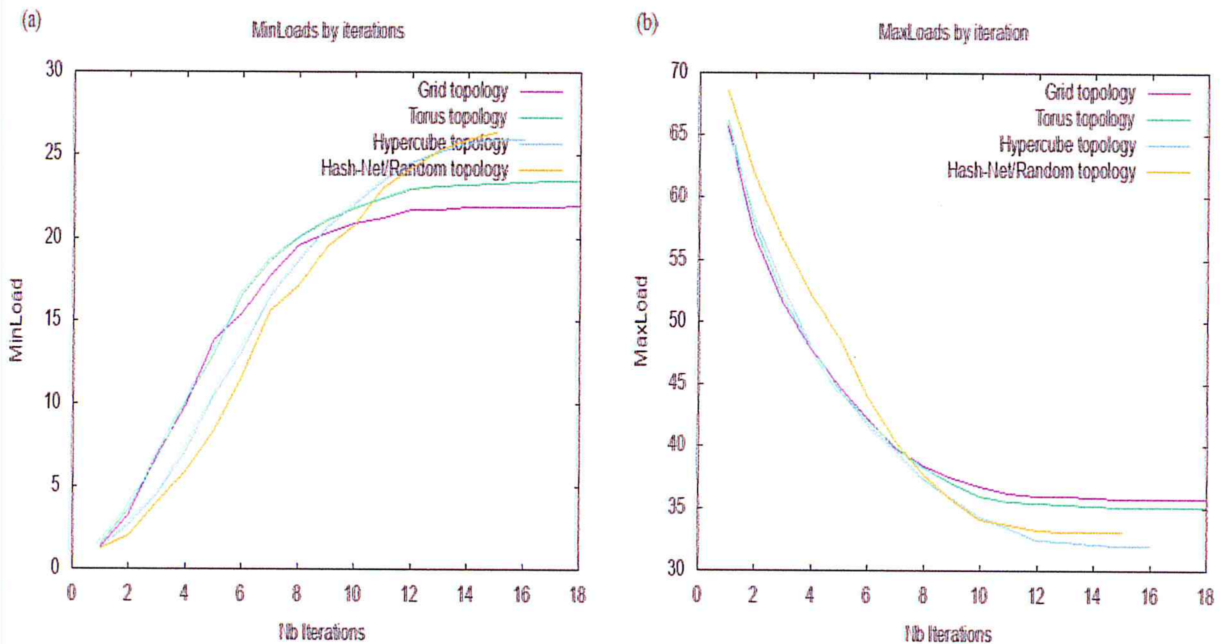


Figure 0.18: Minimum (a) and Maximum (b) total load by iteration for 400 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash–Net topology (in orange) using Greedy Sort.

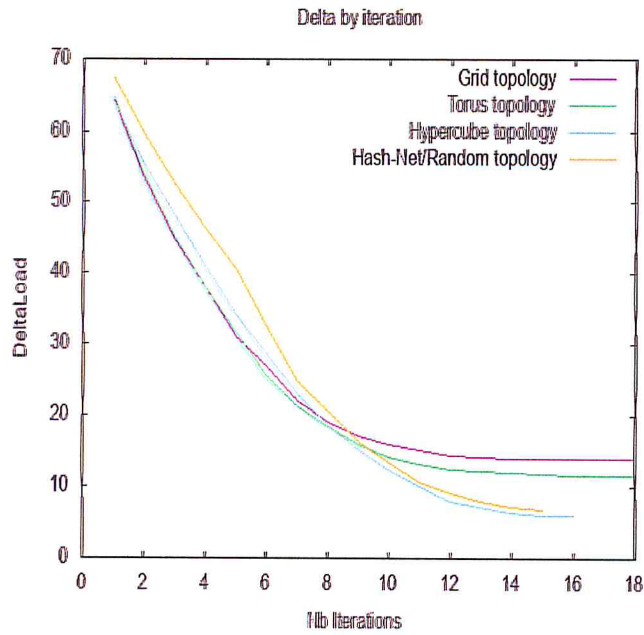


Figure 0.19: Delta by iteration for 400 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash−Net topology (in orange) using Greedy Sort.

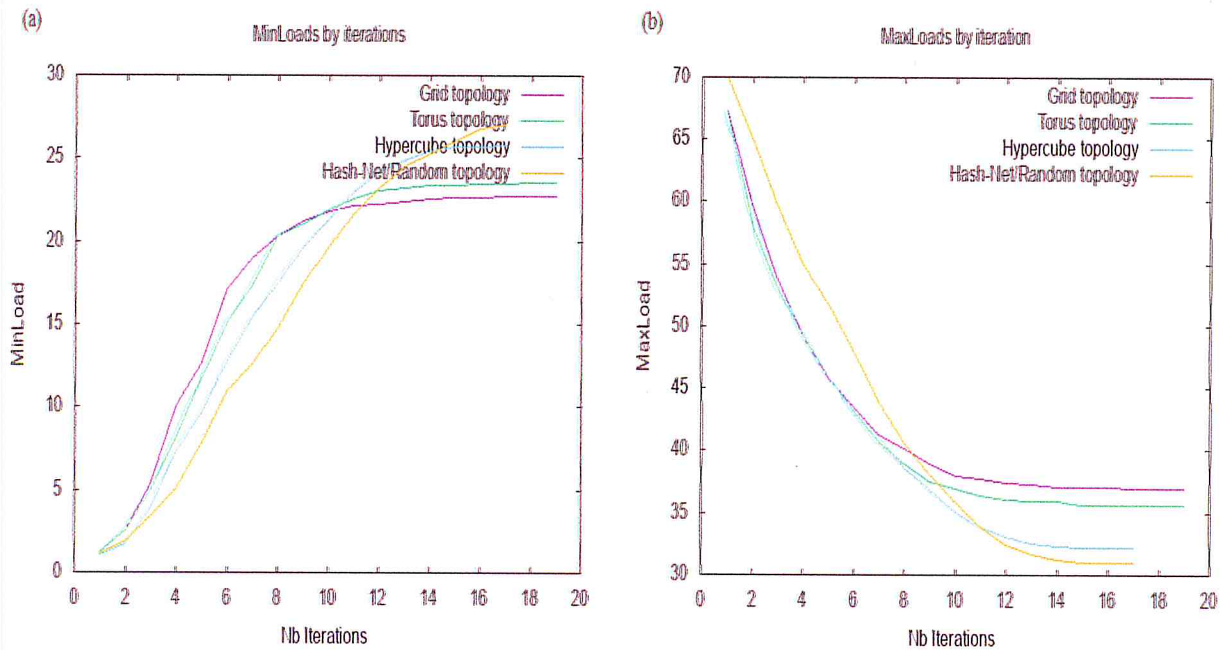


Figure 0.20: Minimum (a) and Maximum (b) total load by iteration for 600 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash−Net topology (in orange) using Greedy Sort.

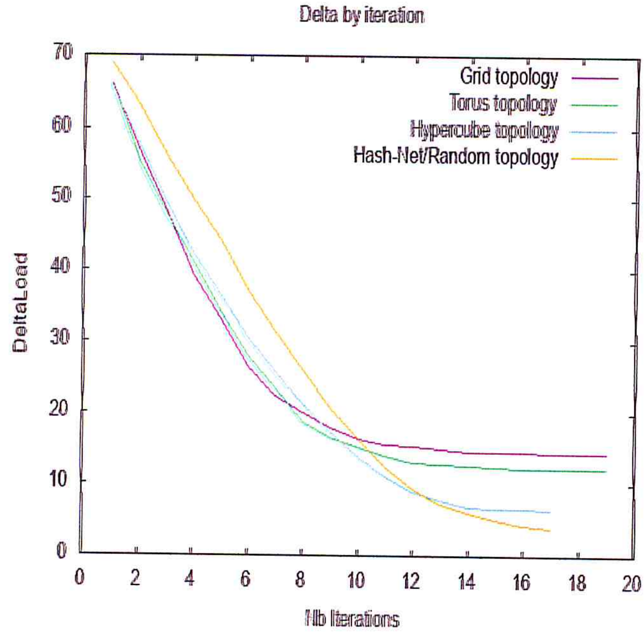


Figure 0.21: Delta by iteration for 600 nodes: Grid topology (in purple), Torus topology (in green), Hypercube topology (in blue) and Hash−Net topology (in orange) using Greedy Sort.