Load Balancing Pragmatics *

Erol Gelenbe and Ferhan Pekergin
EHEI,
UFR de Mathématiques et d'Informatique,
Université de Paris V,
45 rue Saint Peres, 75006 Paris (France).

Abstract. We examine the main practical issues concerning load balancing in distributed systems. Two complementary orientations are examined: an application oriented approach for High Performance Transaction Processing, and a low level process oriented approach.

Key Words: *Load Balancing, Distributed Operating Systems, High Performance Transaction Processing, Process Oriented Load Balancing.*

1 Introduction

Load balancing in a distributed or parallel system has the objective of furnishing the best performance by an appropriate distribution of active work and data among the set of available processors and memories. It must avoid overloading certain processors and leaving other processors idle. It must also avoid excessive communications, and excessive overhead due to other causes such as frequent process context switching.

Load balancing may be viewed at the "static" level, in which processes or sets of processes are assigned to a processor when they are created; or it may be viewed as a dynamic activity which can change the assignment as the load conditions change.

It may also treat processes in a static manner, while data is moved around the system dynamically as the need arises, or vice-versa, or it may treat both processes and data in a dynamic manner.

Load balancing may act at a "low" process level, just by measuring and observing the manner in which processes are being generated in the system, and the manner in which they interact. Or it may be viewed at the "high" level, which takes into account explicit knowledge of the workload, such as the structure of a database system which runs in a parallel or distributed environment.

Load balancing should clearly not be an essential component for the system to function, i.e. the system should be operational even without the load balancing algorithm. Thus a crash of the load balancing facilities should not cause radical consequences, but should only affect the load sharing performance. Since QMIPS does not cover system reliability, these important issues will not be discussed in this paper.

*This work was partially supported by the European Economic Community ESPRIT BRA Project QMIPS.

1.1 The Architecture

The architecture considered is composed of a set of N processors, where each processor is assumed to be a complete computing unit with main memory and disk. However, in some cases the disk unit at each of these processors may not have a large capacity, and it is then reasonable to assume that only a small number of these will be equipped with a very large secondary memory, suitable for containing a complete database. All processors will be interconnected via a high speed local network. Thus we will talk about *Ordinary Processors (OP)* which have main memory and a smaller secondary memory, and *Disk Processors (DP)* which may be composed of the same basic processing unit and have a very large disk. In some system configurations, DP's may not exist at all, in which case we would be dealing with a small database, as for instance in a small business application.

This architecture can be illustrated by two quite different specific examples:

- *A network of compatible high-end PC's with 386 or 486 type processors, interconnected via a local area network such as ETHERNET.* In this case there may (or may not) be one or more large disks which are accessible from some of the PC's.
- *A set of high-end workstations, interconnected via a very fast specialized network (Banyan network, or Hypercube, etc.), as in the EDS platform (ICL Ltd) or in the CM-5 (Thinking Macines), with a very large disk server accessible from some - or all - of the workstations.*

2 Load Balancing for Transaction Processing

In this section we discuss load balancing under the assumption that we are dealing with a specific application: high performance distributed database management and transaction processing.

We consider an application software which is composed of the following basic modules:

- User Modules (UM)
- Database Server Modules (DSM)
- Global Lock Management Modules (GLM)
- Mass Memory Modules (MMM)

We are assuming that multiple instances of each of these modules exist, though in fact in some cases there would be only one copy of each of these modules. We voluntarily do not use terminology identical to that of any commercial Database Management ystem, in order to stress concepts rather than a specific commercial system (e.g. ORACLE). However it is clear that such modules can be found in various commercial systems.

In certain systems, the MMM will be directly integrated into the underlying operating system, while in others the MMM may be a specific File Management System attached to the Database.

Let us briefly describe each of these modules' functions, and then discuss the manner in which the modules would be assigned to processors:

- The *UM* is the software module which contains a user application; typically it will contain copies of editors, query language processors, and all other software which is local to a user. A UM is created for each active user. It provides the user with the facilities for creating a transaction, then for monitoring its execution, and informing the user of its successful termination. Each user module has a cache memory which contains the a user's active blocks.

- A *DSM* contains the software which assures the correct and consistent execution of a tranaction. It keeps track of memory units (blocks, pages or files) which are assigned to each active user, and manages local user buffers. It communicates with the GLM's and with the user. It carries out commits (for instance, using some kind of shadow memory technique) into the mass memory.

- The *GLM* receives requests for memory units (blocks, pages or files) from the DSM's. It recognizes "read/write" or only "read" memory units. It assigns locks to memory units, and authorizes the transfer of copies of memory units to the DSM. It removes locks at the end of transactions.

- The *MMM* manage the access to the physical memory units. They are aware of physical addresses and provide access to disks and other fast secondary memory.

2.1 High Level Load Balancing for Distributed Transaction Processing

Let us now consider the manner in which these modules would be distributed in the type of architectures considered.

2.1.1 Load Balancing by Module Assignment

In either one of the two above architectures described above, load balancing can be carried out at a "high" level in the following manner. Though this is not an "unique" manner for doing so, it is simply based on the principle that all modules of the same type are in a certain sense equivalent.

- Assign the *MMM's to the DP's.* This is a fixed static assignment.

- Assign the *UM's, as they arise to the OP's.* As long as there are free OP's, assign a new UM to a free OP; when there are no longer free OP's left, assign UM's so as to distribute them equally among OP's.

- As long as there are free OP's, assign *DSM's to free OP's.* When there are no free OP's, assign a DSM to each OP, associating the DSM to the related UM.

The case of the Global Lock Management Module is somewhat specific, since its assignment should depend on the approach - *Centralized or Distributed* - which is being adopted for lock management. In the case of centralized lock management, it is clear that the lock manager should be assigned to a single DP.

In the case of distributed lock management, things become more complex. For instance if a "virtual ring" type protocol is being used for distributed lock management, the GLM should probably be distributed in close connection to each of the UM's. Another approach would be to associate the GLM with each MMM, and hence to assign a distributed GLM with each of the DP's.

2.1.2 Memory Management and Data Migration

Here let us assume, in order to fix our ideas, that all data is organized in individually adressable *blocks*. These are independent of pages in the following manner: blocks relate to the database system, while pages relate to virtual memory management.

Thus for certain blocks, for instance those which relate to the indexes of the database, their size can be so small that several blocks are stored on one page — which can be a source of conflict between processes attempting an access to neighbouring indexes.

Other blocks may be large enough to span several pages – though certain systems may limit the size of a block to one page.

We assume that blocks can migrate freely throughout the distributed system under the control of the GLM. Thus each UM will be able to maintain in its cache the blocks which have been assigned to that particular user by the GLM, and which have not been removed by the Cache Management Algorithm from the UM's cache.

We can assume that the *Cache Management Algorithm* of a UM will be based on the Least-Recently-Used (LRU) replacement algorithm, or some similar technique; when the cache is full, blocks will be returned to the MMM according to the LRU rule.

3 "Process Level" Load Balancing

The objective of load balancing is the improvement of system performance on a general basis or for some applications. The design of the load balancing component depends closely on the objective being considered.

To share the load between machines which constitute the system, certain tasks are executed remotely, and load balancing may not be the only reason for the remote execution of a process. Constraint satisfaction in real time computing or access to special facilities available only on some machines can constitute reasons for remote execution. Remote execution requests interact with the performance oriented process assignment algorithm, and make the implementation of the load balancing component more difficult.

In the present discussion, real time computing is not of central interest and the architecture is assumed to be homogeneous. So there is no *a priori* reason to consider the particular assignment of requests with supplementary constraints.

However the individual use of workstations by disabling remote access to these machines during certain periods - i.e. the withdrawal of certain processors from the distributed computation - is compatible with the design issues of the target distributed system. This mode of operation can be supported as an option in order to increase dynamic scalability of the system and its adaptability to different working conditions.

Static load balancing, or *load sharing*, is one often used major option. Indeed, in many cases dynamic load balancing is not efficient because of the important overhead due to migration.

Nevertheless, it can provide additional improvement if the system workload presents certain characteristics such as heavy global load, or great variability of the execution times and of the process generation intervals. Even under these conditions the improvement due to dynamic load balancing remains moderate.

On the other hand, the implementation of the dynamic process migration mechanism requires substantial extra effort. As a result, the implementation of a dynamic load balancing algorithm can become plausible only if a careful workload characterization shows clearly all the necessary conditions.

The fundamental property expected from a load balancing system component is efficiency. Load balancing must not consume more than a few percentage points of CPU time of workstations, or of network bandwidth. Generally, 1 % is considered an admissible value. These measures concern the resource utilization of the load balancing mechanism, but does not assure the successful scheduling of the processes in the system. The load data used to decide remote execution and to choose the target machine must be up-to-date, otherwise even intensive CPU consumption will not guarantee good balancing. Clearly, the objective of having freshly updated data at the same time as low cost communication cost is difficult to achieve.

In the design of the load balancing component, we recommend the use of a light asynchronous load-monitoring scheme in order to avoid overloading the communication subsystem. Broadcasting is used only occasionally and replies to a broadcast message are not allowed. This load monitoring scheme contributes to scalability of the load balancing component. On the other hand, the simplicity of the proposed algorithm helps maintain the CPU utilization for load balancing within reasonable bounds.

3.1 Static Load Balancing

The load balancing algorithm we discuss here is a *sender initiated algorithm with a two level decision mechanism.*

When a process is to be created, the workstation where the process appears must decide whether a remote execution request to the server should be submitted or not. This local decision is based on the comparison of the local load index with the global average load measure. If the workstation finds itself more heavily loaded than the average, it verifies the eligiblity of the process for remote execution by using a look-up table. If this process is remotely executable, a request for a remote machine is addressed to the server; otherwise the process will run where the need for creation is noticed.

The server which receives the remote execution demand can return a machine identity or can simply refuse this request after evaluation of the system load by using the data that it maintains and by the communicated load index of the sender. Thus, a remote execution request made locally is necessary but not sufficient to obtain remote execution.

This algorithm is a variant of the well known centralized decision algorithm with a request filtering mechanism on the load and on the type of process bases. The local filtering mechanism increases the efficiency especially when the system is heavily loaded. Under heavy load, each workstation can estimate that its load is high enough to ask for remote execution if there is no global measure of load. Thus the global average load measure constitutes a parameter for adaptive control. It is a slowly varying measure so it does not require frequent updates; however the efficiency of the algorithm depends on this parameter when the system load changes substantially. Its value can be easily calculated by the server and can be communicated to all workstations by broadcasting if a sunstantial load change is observed. This broadcasting does not cause any problem from the performance point of view, while there will be no reply to this message.

To enable a personal use of the workstations when an efficient interactive working mode is desired for some machines, the server must be informed about this fact. Therefore, workstations can declare themselves completely or partially disabled for remote access. In this case, the server considers these workstations as being *"more loaded"* than they really are; the server continues to keep track of their true load for possible and occasional participation in load sharing. Of course, this is not possible for a workstation which is completely disabled for remote access. On the other hand these workstations can continue, if they wish, to profit from the remote execution facility as in the normal case.

3.2 Load Monitoring

The load monitoring scheme in EPOCH is a workstation initiated asynchronous algorithm. The server is not allowed to ask about the load of workstations, since a mechanism based on polling would be costly from the communication point of view and a broadcasting scheme would imply buffer saturation by creating a *hot point* on the machine where the server resides.

The workstations communicate their load index to the server with their remote execution request. If the machines do not address such a demand for a long time, they should periodically inform the server about their load index in order to update the centralized load information. This update interval can be taken relatively large as long as significant load changes are urgently indicated to the server. Thus a long period of absence of messages from a workstation will be interpreted by the server as a quasi constant load. Thus, the server always concludes that the most recently received load index is still valid.

The number of messages concerning load information may be easily kept low. First, the calculation of the load index, as it will be seen in the sequel, does not reflect the short time fluctuations in the load. Only important and confirmed modification of the average load must be signaled. Moreover, there is no urgency to signal a decrease in the load. The effect of this "suboptimal" policy may be negligible. Signaling increasse of the load is much more important because of the nonlinear effect of the load on the waiting time of processes. However, if this increase is not sustained for a certain period of time, the number of messages which should indicate this fact will be limited; otherwise the load information will be communicated to the server with the remote execution requests.

It seems that the best index used in load balancing is the *Berkeley Unix Load Index* which consists of the average number of processes in the system, calculated in an interval of few seconds. This can be weighted with the CPU utilization as in certain existing systems.

3.2.1 Load Information Update and Storage

Load information may be kept in the server data region in the main memory as a linked list structure. Identities of target machines are organized as a queue depending on their load in a decreasing order. Particular attention must be paid to privilege the machines whose load information is more recent, among the ones having similar load. Load balancing information must be used on-line, since it becomes quickly out of date; thus there is no particular interest to keep it in a file. Moreover, it will be accessed only by the server. Thus there is no reason to share the memory used for its storage. Only some particular parameters especially useful in restarting the system must be put in a system file.

4 Bibliography

We provide below a bibliography containing some of the most useful references on load balancing of distributed systems.

References

[1] A. Barak, A. Shiloh
A Distributed Load Balancing Policy for a Multicomputer
Software Practice and Experience, Vol. 15, No. 9, Sept. 1985

[2] A. Barak, R. Wheeler
MOSIX: An Integrated Multiprocessor UNIX
Proc. Winter 89 USENIX Conference, pp.101-112, San Diego, CA., Jan. 1989

[3] G. Bernard, M. Simatic
A Decentralized and Efficient Algorithm for Load Sharing in Networks of Workstations
Proc. EurOpen Spring 1991 Conf., pp.139-148,Tromso, Norway, May 1991

[4] G. Bernard, D. Steve, M. Simatic
Placement et migration de processus dans les systèmes répartis faiblement couplés
Technique et Science Informatiques, Vol. 10, No. 5, pp. 375-392, 1991.

[5] F. Bonomi
On Job Assignement for a Parallel System of Processor Sharing Queues
IEEE Trans. on Computers, Vol. 39, No. 7, July 1990

[6] F. Bonomi, A. Kumar
Adaptive Optimal Load Balancing in a Nonhomogeneous Multiserver System with a Central Job Scheduler
IEEE Trans. on Computers, Vol. 39, No. 10, Oct. 1990

[7] F. Douglis, J. Ousterhout
Process Migration in the Sprite Operating System
Proc. 7th Int. Conf. on Distributed Computing Systems, Berlin, Sept. 1987

[8] M. R. Eskicioglu
Process Migration in distributed Systems : A Survey of Design Issues
submitted to Journ. of Parallel and Distributed Comp.

[9] D. Ferrari, S. Zhou
A Load Index for Dynamic Load Balancing
Proc. 1986 Fall Joint Computer Conference, Dallas, Texas, Nov. 1986

[10] W. Joosen, B. Vandenborre, P. Verbaeten
Desing and Implementation of an Experimental Load Balancing Environment
Proc. EurOpen Spring 1991 Conf., pp.139-148,Tromso, Norway, May 1991

[11] C. Kim, H. Kameda
An Algorithm for Optimal Static Load Balancing in Distributed Computer Systems
IEEE Trans. on Computers, Vol. 41, No. 3 , March 1992

[12] H. C. Lin, C. S. Raghavendra
A Dynamic Load Balancing Policy with a Central Job Dispatcher (LBC)
IEEE Trans. on Software Eng. Vol. 18, No. 2, Feb. 1992

[13] D. M. Nicol, P. F. Reynolds Jr.
Optimal Dynamic Remapping of Data Parallel Computations
IEEE Trans. on Computers, Vol. 39, No. 2, Feb. 1990

[14] J. A. Stankovic
An Application of Batesian Decision Theory to Decentralized Control of Job Scheduling
IEEE Trans. on Computers, Vol. 34, No. 2, Feb. 1985

[15] M. M. Theimer, K. A. Lantz
Finding Idle Machines in a Workstation Based Distributed System
IEEE Trans. on Software Eng. Vol. 15, No. 11, Nov. 1989

[16] B. Walker, G. Popek, R. English, C. Kline, G. Thiel
The LOCUS Distributed Operating System
Proc. 9th ACM Symp. on Operating System Principles, Nov. 1983

[17] Y. Wang, R. J. T. Morris
Load Sharing in Distributed Systems
IEEE Trans. on Computers, Vol. 34, No. 3, March 1985

[18] J. Wong, R. B. Jorgensen
Distributed Processing on Unix Based Workstations
Iowa State Univ. Tech. Rep. No. 88 – 13, June 1988

[19] E. R. Zayas
Attacking the Process Migration Bottleneck
Proc. 11th ACM Symp. Operating Systems Principles, Austin, Texas, Nov. 1987

[20] S. Zhou
A Trace Driven Simulation Study of Dynamic Load Balancing
IEEE Trans. on Software Eng., Vol. 14, No. 9, Sept. 1988