From: Proceedings of the International Conference on Parallel Processing; Chicago, IL; August, 1989.

IMPLEMENTATION OF GAMMON: AN EFFICIENT LOAD BALANCING STRATEGY FOR A LOCAL COMPUTER SYSTEM

Katherine M. Baumgartner[1], *Ralph M. Kling*[2] *and Benjamin W. Wah*[2]

[1] Digital Equipment Corporation, 146 Main Street, Maynard, MA 01754

[2] Computer Systems Group, University of Illinois at Urbana-Champaign, CSL, 1101 W. Springfield Avenue, Urbana, IL 61801

ABSTRACT

This paper shows the realization of GAMMON (Global Allocation from Maximum to Minimum in cONstant time), an efficient load balancing algorithm we described at ICPP 1987, on a network of Sun workstations. GAMMON uses the available broadcast capability of multiaccess networks to implement an efficient search technique for finding hosts with maximal and minimal loads. The search technique has average overhead which is independent of the number of participating stations. The transition from the theoretical concept to a practical, reliable, and efficient implementation is described.

1. INTRODUCTION

Global allocation, or *load balancing*, uses communication facilities on distributed computer systems to support remote job execution in a user transparent fashion to improve resource utilization and reduce response time. A decision to load balance is made if the job is likely to be finished sooner when executed remotely than when executed locally.

We have reported an innovate and efficient load balancing strategy at the 1987 International Conference on Parallel Processing [2, 3]. The strategy is based on an efficient distributed search of the extremum of a set of numbers, which reflect the workloads of the computers, and on the broadcast capability of local multiaccess networks. Using a recursive formulation of the search process, it was found that the identification of the computers with the maximum and the minimum workload can be found in constant time, independent of the number of computers connected on the network. The algorithm was developed without hardware modification to existing networks.

This paper is an update on our recent efforts in implementing the load balancing strategy we proposed in 1987 [2]. A tradeoff is made in substantially reducing the synchronization overhead of our previous strategy, while achieving comparable (but slightly degraded) performance in identifying the target computers for load balancing. The subsequent section gives an overview of the GAMMON strategy, outlining the strategy steps and briefly describing a distributed extremum search. The following two sections describe the transition from the theoretical concept to a practical implementation, and discuss the details of the implementation. Performance results are presented in Section 5 followed by conclusions.

2. GLOBAL SCHEDULING STRATEGY

Fig.1 shows a diagram of the system under consideration in this study. Multiple identical processors with finite queues are connected by a broadcast bus. Each processor can have arrivals external to the system or from the bus. Jobs are modeled as independent tasks. If jobs are migrated to a processor across the bus, the results must be returned to the originating processor when execution is completed.

2.1 Scheduling Operations

Since the common bus allows only one transmission at a time, an efficient strategy is to send jobs from the maximally loaded processor to the minimally loaded processor. This will reduce the time that a job is waiting for execution at one processor while another processor is idle (called the idle-while-waiting condition). Obviously, the occurrence of the idle-while-waiting condition should be decreased as well as the occurrence of any state that makes idle-while-waiting more likely. In multiprogrammed systems, assuring that processors are busy is not sufficient to minimize the occurrence of the idle-while-waiting condition. Since the response time for each job is degraded as more jobs are added to the active queue, jobs need to be distributed evenly. GAMMON considers the state of both the source and destination processors. Further, required status information is minimal; only the load extremes need to be identified.

Three scheduling operations are required: identification of the load extremes, job migration, and result return. Migrating jobs and returning results are straightforward and are described in Section 4. Identifying processors with the load extremes efficiently is more difficult. In order for the algorithm's complexity to be independent of the number of processors connected to the bus, any centralized scheduling algorithm, such as polling, is not suitable.

The priority ordering for tasks using the bus network is (1) regular message transfer, (2) result return, (3) job migration, and (4) max/min identification. The intuitive reasons behind this choice are that the network was designed for message traffic, hence it has the highest priority. Early return of results will contribute to reduction in average response time, while early

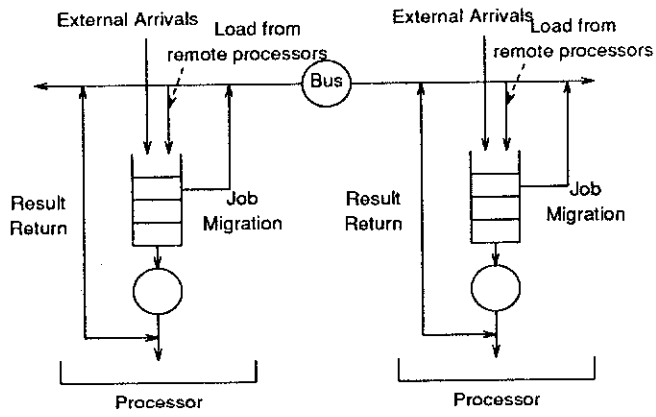

Fig.1 Queuing diagram of distributed processors

Acknowledgements: This research was supported in part by the National Aeronautics and Space Administration Contract NCC 2-481 and the National Science Foundation Grant MIP 85-19649.

migration of jobs may not because jobs may be queued at the remote processor; hence return is given higher priority to job migration. Lastly, identifying candidates for load balancing should be given a lower priority relative to job migration because the bus network is shared among all competing jobs; therefore, migrating multiple jobs simultaneously will delay the initiation times of all jobs concerned. GAMMON consists of repeatedly determining which of the current tasks has the highest priority, and then executing that task.

2.2 Distributed Extremum Search

The extremum-search strategy used by GAMMON is for the processors to cooperatively search ranges of load values to find a load extreme. Starting with an interval (L,U], each processor has search parameter x_i reflecting its load. The processors maintain a global window (or sub-interval) on the interval. Processors with parameters within the window attempt to broadcast their search parameters, and if there are one or more parameters in the window, there will be a contention resolution followed by a broadcast of one the the search parameters. In that case the upper bound of the interval under consideration is updated to the value broadcast. If there are no parameters in the window (no attempted broadcasts), the lower bound of the interval is updated to the upper bound of the global window. The minimum is identified when the lower bound of the window is equal to the upper bound, or the global window is sufficiently small that there is low probability of more that one search parameter existing in the window. Two-state network information is available to a processor to indicate the outcome of an attempted broadcast; (a) a station successfully broadcasts in a contention iteration, and (b) no station broadcasts in the broadcast slot. A broadcast slot consists of the time for all processors to contend for the channel and the time for the winner to broadcast its message. The search for the maximum is similar.

As an example, consider a case with five processors participating with search parameters of x_1=0.48, x_2=0.90, x_3=0.35, x_4=0.3, and x_5=0.75. The initial interval is (0,1], and let the initial window value chosen be 0.51. Processors 1, 3 and 4 attempt to broadcast their parameters. Suppose that processor 3 is the winner and transmits. The next interval to be searched is $(0,x_3)$. Let the next window value chosen be 0.33. Only processor 4 tries to transmit its parameter, so it is successful. The search has not concluded even though x_4 is the minimum because the fact that it was the only processor broadcasting is not available to processor 4, or the other processors. The next interval searched is $(0,x_4)$, and the window chosen is 0.25. There is, of course, no broadcast. This process will continue until the bounds of the window isolate x_4, and the minimum is globally known.

The window choices are based on the probabilities of the two outcomes, transmission and idle. Details of the window selection process, including alternate selection strategies are shown elsewhere [2]. The strategy described here allows one-broadcast per broadcast slot. Other strategies investigated include a two-broadcast strategy and a combined one- and two-broadcast strategy. Simulation results have demonstrated that the one-broadcast strategy is superior. Appropriate window values can be precomputed using a greedy algorithm. Simulation results show that an average of 2.7 broadcast slots is needed to identify the station with the minimum parameter, independent of the number of contending stations. Two-state information is available to the operating system so this technique does not require any changes to existing Ethernet hardware interfaces.

3. PRACTICAL REALIZATION

We now focus on the transition from the theoretical concept described above to an actual implementation. Several options and limitations imposed by the target system architecture have to be considered.

3.1 Host system

The two distributed computer networks on which GAMMON has been implemented connect SUN 3/280 servers and SUN 3/50 diskless clients by Ethernet. The first system consists of one server and two clients whereas the second one has one server and ten clients. The NFS (Network File System) supports transparent access to remote file systems. This mechanism allows uniform access by servers and clients to secondary storage. All hosts run SUN Unix version 3.4.

3.2 Adaptation of the GAMMON strategy

In the following, we describe the differences between the theoretical concept [3] and the practical implementation. A number of requirements have been relaxed to result in a practical implementation. A number of strategies that allow a practical implementation but do not significantly degrade performance are discussed in this section.

The first strategy is to truncate the search after one iteration to reduce network traffic. This is equivalent to the use of a precomputed static search window instead of a dynamic one as presented in Section 2. This does not severely affect global performance since workload information is heuristic in nature and small errors in identifying processors for load balancing are not critical.

The second strategy to reduce computational overhead is to deliberately not enfore priorities of network usage as outlined in Section 2. A strict enforcement would have required changes to the operating system kernel which would have greatly reduced the portability of our system. Moreover, the effect of interleaving packets containing results and migrating jobs will not have a significant impact on the overall performance.

In the third strategy to reduce overhead, the search for the maximum load is not performed explicitly. It was observed that if a processor's load was above the initial window in the minimum search, its load was the maximum the majority of the time. As a result, processors with loads above the initial window can migrate the current job if the minimum load is current, and an explicit search of the maximally loaded processor is not needed.

3.3. Implementation overview

During the program development, a number of decisions were made concerning the implementation of certain functions. On the highest level, the search for the minimally loaded processor should be running periodically and independently of eventual job migrations. This will minimize job startup times. The underlying search algorithm is the previously described window broadcast strategy.

The search daemon is a background process that periodically participates in a search for the minimum load among all network hosts. A search is initiated by a periodic timeout signal on each host, or in response to an incoming data packet. In any of those events, each host will send out a data packet containing its own load information, provided it is within the search window. Incoming packets are used to update the local information about the minimally loaded host. This algorithm proved to be the most efficient in terms of network load and processor overhead.

The second part of the GAMMON load-balancing software is the remote execution handler, which operates asynchronously from the search. Upon a user request to execute a certain program, the load-balancing software decides if the job will be executed locally or remotely. Job migration is performed by sending the necessary information of a job at its entry point to the remote processor. The input and output channels of the user's shell and the remote execution shell are linked to ensure transparent operation.

4. IMPLEMENTATION DETAILS

4.1. Distributed search

A search is initiated, as mentioned before, by a periodic alarm signal, or by a packet arriving from another search module. When an alarm is received, the processor's current load is compared to the lower bound of the window. If the local workload value is smaller than the currently known lower bound, it is broadcast with a timestamp and processor address. If a packet is received, the load is accepted as the minimum and is stored with the current time. If more than one processor sends a load packet due to receiving alarms simultaneously, the minimum load is accepted. If the loads are identical, the processor address is used as a tie breaker. The alarm timer is reset when packets are received, therefore, the process is loosely synchronized.

The contents of received packets are time-stamped and written to a status log file. When the status information is retrieved, the time stamp is checked and is used to determine if the load value is out of date. In that case, the processor considers itself ineligible for job migration until it receives another status packet. This time stamping allows an unreliable communication mechanism (broadcast datagrams) to be used to communicate load information. This reduces network traffic since no circuit has to be established and no acknowledgements have to be sent. In case that load information is lost by one processor, that processor does not migrate jobs. This has a minimal effect on the overall performance of the scheduling strategy. Moreover, since packet loss is a relatively rare occurrence, acknowledgements are unnecessary.

A simplified block diagram of our method is shown in Fig.2. The search is started at the shell level with the *searchd* daemon. An internet datagram socket is established for sending and receiving packets containing load balancing information.

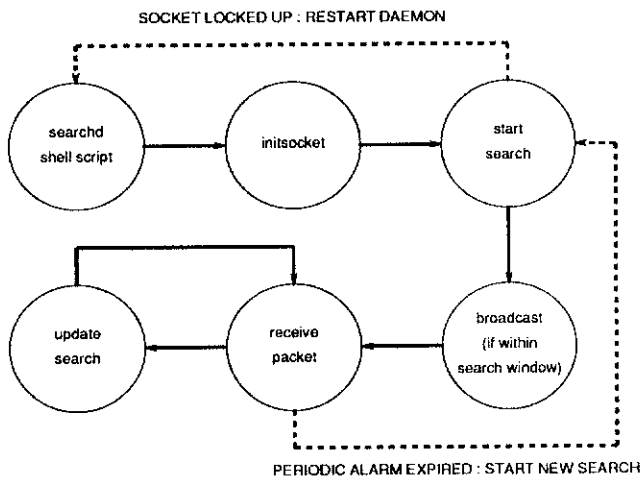

Fig.2 Simplified state diagram of the *searchd* daemon

Next, the search process is started by broadcasting a search packet containing load balancing information if the processor's load is within the required window. Then, the program enters a loop in which it waits for incoming packets and updates its status information. If no packets are received within a specified time slot, a new search is initiated. In rare instances, the broadcast socket can lock up preventing incoming packets from being received. A watchdog procedure restarts the search at the shell level if that event occurs.

The search routine used to find the minimally loaded host can be implemented on single broadcast networks connecting independent workstations or a file server and a number of clients. It can also be run in a hierarchical manner, for example, on a system having a number of the above mentioned networks plus a backbone net connecting the file servers. In this case the search would be run on each subnetwork independently. In addition, there will be a global search among the file servers on the main net; however, this search would use the values found by all the local searches. The remote execution handler could then either use the local result, if the scope of the job migration should be limited to the local network, or the global result, if load balancing among the networks is allowed.

4.2. Remote execution

In an NFS environment, it is not necessary to explicitly send files from the source to the destination because all processors and clients have access to the same secondary storage. Job migration simply requires sending the command from the source to the destination, and returning results and error messages.

The job migration decision is made as follows. First the current load of the processor is compared to the threshold which would be used for a maximum search. If the load is above that value, the global minimum location and its timestamp are read from a local file. If the timestamp indicates that the minimum is current, the job is executed on the processor with the minimum load using a remote shell. This causes any error messages associated with the remote execution to be sent back directly. The remote shell uses a reliable communication mechanism (TCP/IP protocol), therefore execution of the job is guaranteed. The remote execution handler *x* uses the information generated by the search process and the current load on the host it is running on to decide if the job given in the command line argument is to be executed remotely or locally.

Load balancing is only initiated if the current local load is larger than a certain local minimum and the current minimal load on the network. It is initiated by obtaining the current working directory and using the *rsh* system command to establish communication with the remote host. First the working directory on that host is set to the working directory of the local host. Then, the command is executed. The input, output, and error message channels are linked to the remote host such that job execution appears transparent to the user.

A sample *x* command looks like this:

x ditroff -me -Tps filename > ~/printfile

It should be noted that the only difference to the usual command invocation is the leading *x* which indicates load-balanced execution. Even shell metacharacters such as { ~ * > } and the like are allowed. The above command translates into the following line:

rsh minloadhost ' cd workdir;
ditroff -me -Tps filename > homedir/printfile '

5. PERFORMANCE OF GAMMON

The overall performance of the search algorithm was tested on two differently configured systems: one with one server (Aquinas) and two clients, and another with one server (Dwarfs) and ten clients. On Aquinas, a complete search was resolved in 50 to 80 milliseconds, and on Dwarfs, it was resolved in 150 to 180 milliseconds. The performance on Dwarfs can be considered the worst case since the number of clients and the physical net length approach the allowable limits. The results for both Aquinas and Dwarfs were consistent with the simulation results in which the absolute minimum was located 70 percent of the time for all resolved searches.

Another obvious technique for distributing status information would be to broadcast it periodically, as is done with the *rwhod* daemon in Unix. To determine the savings of using the proposed method as opposed to using the technique of the *rwhod* daemon, the resource utilization of both was measured for the period of one hour. The overhead is summarized in Table 1. For comparison, a daemon that only broadcasts the load average value is studied (as opposed to the *rwho* daemon which broadcasts other information as well). Recall that the daemons perform a search every minute. When a search using GAMMON is performed, it is possible for daemons at two computers to start a broadcast simultaneously; hence the total number of broadcast is slightly above 60 (first value in the last column of Table 1). When the broadcast or *rwhod* daemons are used, each computer initiates a broadcast every minute independent of other computers. These results indicate that both *rwhod* and the simple broadcast daemon introduce considerably more communication overhead than the search daemon we proposed.

To compare the overall improvement in performance due to the GAMMON load balancing strategy as compared to that without load balancing, the server Aquinas with its two clients, Calvin and Hobbes were used. First, workload was generated by processes that either initiated a CPU-bound job or slept for the amount of time the job consumed during its last execution with equal probability. As these processes were executed, the load and the amount of time consumed by executing processes was tabulated. Also, a history of the initiation of jobs was created. During the second portion of the test, the global scheduling strategy was enabled, and the jobs were initiated according to the history. Again the load was monitored, and the execution time tabulated.

Table 1. Comparison of cumulative overhead

Deamon	Cumulative system time (seconds)	Cumulative user time (seconds)	Total number of broadcasts
search	0.1094	0.0137	65
broadcast	0.2593	0.0316	181
rwho	0.2633	0.0613	181

Table 2. Performance of the global scheduling strategy. (Aquinas is a Sun 3/260 server, Calvin and Hobbes are Sun 3/50 diskless clients.)

Host	Job Time without global scheduling (seconds)	Job Time with global scheduling (seconds)	Difference (%)
Sun 3/260 server	18963	18540	2.23
Sun 3/50 client	12143	10430	14.10
Sun 3/50 client	7319	5912	19.22
Totals	38425	34882	9.2

The results as summarized in Table 2 show a small improvement for the server, and a much larger improvement for the clients. This is expected as the server is about 2.7 times faster than the clients, and has direct access to the secondary storage through the VME bus rather than via Ethernet. As a result, the clients will benefit more by sending jobs to the server.

6. CONCLUSIONS

In this paper, we have presented an efficient technique for a distributed extremum search and a load balancing protocol using this technique. The search technique can be implemented on *existing* CSMA/CD networks at the applications level. This is important because it is typically not possible to make hardware modifications to existing networks. The maximum or the minimum of a set of numbers can be identified in a small bounded number of broadcasts, on the average. Since the search technique has a constant average behavior, the GAMMON strategy using this efficient search technique is feasible. Performance of GAMMON was found to be favorable at low to moderate traffic intensities, and load balancing intervals (the total time to perform max/min identification, job migration, and result transfer) that are small relative to the average service time.

Implementing GAMMON on a network of Sun workstations reveals a number of disparities between the assumptions made in the theoretical model and the physical limitations of distributed computing systems. By relaxing a number of assumptions without degrading the performance, we have developed a load balancing system that is comparable in performance to the theoretical model and allows improvements in overall as well as in individual processor performance.

The algorithm for distributed extremum search effectively exploits the broadcast ability of multiaccess networks and information readily available on the network to find the target computers for load balancing. Hence, it is not limited to the load balancing strategy we describe in this paper. Our search method can be easily adapted to other systems implementing load balancing strategies on local computer systems [1, 4-6].

REFERENCES*

[1] R. V. Baron et al., in *MACH Kernel Interface Manual*, Pittsburgh, PA, Jan. 1987.

[2] K. M. Baumgartner and B. W. Wah, "Load Balancing Protocols on a Local Computer System with a Multiaccess Bus," in *Proc. Int'l Conf. on Parallel Processing*, University Park, PA, pp. 851-858, Aug. 1987.

[3] K. M. Baumgartner, "Resource Allocation on Distributed Computer Systems," in *Ph.D. Thesis, School of Electrical Engineering, Purdue University*, West Lafayette, IN, May, 1988.

[4] P. Krueger and M. Livny, "Load Balancing, Load Sharing and Performance in Distributed Systems," Computer Science #700, University of Wisconsin, Madison, August, 1987.

[5] M. Litzkow, "Remote UNIX," in *Proc. 1987 Summer Usenix Conferences*, Phoenix, AZ, June 1987.

[6] W. A. Mason, "Distributed Processing: The State of the Art," *Byte*, vol. 12, pp. 291-297, Nov. 1987.

* A more comprehensive list of references on load balancing can be found in our paper published in the Proceedings of the 1987 International Conference on Parallel Processing [2]. They are not included here due to space limitation.