AN ADAPTIVE LOAD BALANCING ALGORITHM FOR A MULTICOMPUTER

by

Phillip Krueger
Raphael Finkel

Computer Sciences Technical Report #539

April 1984

An Adaptive Load Balancing Algorithm for a Multicomputer

Phillip Krueger
Raphael Finkel

University of Wisconsin

Abstract

A new multicomputer load-balancing algorithm is proposed which is adaptive, decentralized, preemptive and simple. The algorithm uses broadcast messages to achieve consensus about average load in the network and to match machines that are over- and under-loaded. The communication bandwidth used by the algorithm can be controlled to be as narrow as desired. Results from a simulation are presented, showing that the algorithm is effective. Extensions to the algorithm that may increase its effectiveness without a large increase in complexity are presented.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems - network operating systems; D.4.1 [Operating Systems]: Process Management - scheduling; D.4.7 [Operating systems]: Organization and Design - distributed systems

General Terms: Algorithms, Performance

Additional Key Words and Phrases: distributed operating system, dynamic load-balancing, multicomputer, process migration, processor scheduling, resource allocation

---

This research was supported in part by National Science Foundation grant number MCS-8105904 and by the Defense Advanced Research Projects Agency of the Department of Defense under contract number N00014-82-C-2087.
Authors' address: Computer Sciences Department, University of Wisconsin, 1210 West Dayton Street, Madison, WI 53706

1. Introduction

Recent advances in VLSI and computer communication technology have made the multicomputer* an increasingly attractive design for powerful, robust and cost-effective computer systems. The design of general-purpose operating systems for multicomputers is complicated, however, by several unique requirements that have not been addressed in traditional operating system designs. Most notable is the requirement to efficiently allocate processor resources to programs.

In a general-purpose multicomputer operating system, it is desirable to automate the allocation of processor resources for the same reasons that it is desirable to automate primary memory allocation in a traditional multiprogramming operating system: Automatic allocation applies a consistent allocation policy throughout the multicomputer and provides a level of abstraction that hides particular instances of the physical resource.

Load-balancing is a policy for allocating processor resources. Processes are assigned to processors, or *nodes*, in such a way that each node has approximately the same workload. When there is little variance in loads among the nodes, the multicomputer is said to be *balanced*; otherwise it is *unbalanced*.

Similar to optimal primary memory allocation, optimal load-balancing is computationally prohibitively expensive and requires foreknowledge of the runtime characteristics of all processes [1, 2, 3, 4, 5]. Alternatively, a suboptimal heuristic algorithm can be devised that allocates resources in real time according to the currently observed state. The goal of such an algorithm is to approach optimality at a fraction of the cost and with less knowledge than is required by an optimal algorithm. An example of a heuristic algorithm for resource allocation is the Working Set policy [6]

* A multicomputer is a computer system composed of many fully independent computers, or *nodes*, connected by a communication device. Memory is not shared between nodes; processes residing on different nodes communicate through messages only.

for allocating primary memory.

When a process is created, a *non-preemptive load-balancing algorithm* permanently assigns it to the node that appears at that moment to be best. The process is not moved even if its runtime characteristics, or the runtime characteristics of any other process, change in such a way as to cause the multicomputer to become unbalanced. Load-balancing occurs only when a new process is created. Non-preemptive load-balancing is appropriate if the nodes are constrained to have only one process resident at a time, but its inability to adapt to changes in the runtime characteristics of processes is a serious drawback when nodes are multiprogrammed, which may be necessary to achieve even a moderate level of node utilization. Examples of heuristic non-preemptive load-balancing algorithms can be found in the literature [7, 8].

A *preemptive load-balancing algorithm* allows load-balancing to occur whenever anomalies appear in the workloads of the nodes. If the multicomputer becomes unbalanced, a process in the middle of execution can *migrate* to a better node to continue its execution. Load balancing can occur at any time, rather than being limited to times when new processes are created. The increased adaptability of preemptive load-balancing can make it more effective than nonpreemptive load-balancing in a multicomputer having multiprogrammed nodes. This adaptability allows both an increase in throughput and a decrease in process response time [9].

Several heuristic preemptive load-balancing algorithms have been discussed in the literature [10, 11, 12]. Many have used constrained communication topologies, at least for the purpose of migration, including arrays [13] and trees [14, 15].

We propose a new family of decentralized heuristic preemptive load-balancing algorithms for a multicomputer having the following characteristics:

(1) The node architecture is homogeneous.

(2) The communication device fully connects all nodes and allows broadcast transmission.

(3) New jobs may arrive at any node in the multicomputer. Jobs execute equally well on any node, independent of the node where they arrived.

(4) Each node of the multicomputer supports multiprogramming.

Our algorithms are meant to adapt to changes in the overall load on the multicomputer, the load on the communication device, the runtime characteristics of processes, and the configuration of the multicomputer. Nonetheless, the algorithms are not computationally complex, so gains made by load-balancing are not overshadowed by costs. To attain these goals, our algorithms use broadcast messages to maintain global information and to negotiate exchanges of processes. The algorithms do not require that broadcast messages be reliable, but remain stable even when some broadcast messages are lost.

We present the basic algorithm, called the **Above-Average Algorithm**, and report on results obtained from a detailed simulation. We then suggest several extensions to the basic algorithm.

2. The Above-Average Algorithm

2.1. Overview

The Above-Average algorithm, like any preemptive load-balancing algorithm, must decide:

(1) When a process should migrate.

(2) Between which nodes the process should migrate.

(3) Which process should migrate.

Since the object of a load-balancing algorithm is to reduce the variance in loads among the nodes of the multicomputer, the appropriate time for a process to migrate to another node is when its node's load increases beyond the average for all nodes in the multicomputer, making its node *overloaded*. Similarly, an appropriate node to which the process should migrate is a node whose load is below the overall average, an *underloaded* node. Thus, to negotiate the migration of a process correctly, the average load over all nodes of the multicomputer should be known. Because of the time required to calculate and store the average load value, it must always be an approximation, and the cost of its maintenance can always be reduced with an accuracy penalty.

In the Above-Average algorithm, negotiations of process migrations and the maintenance of the average load value are intimately connected. The responsibility for maintaining the average load value is distributed by allowing any node to broadcast an updated average load value whenever it believes that the current approximation is inaccurate. A node becomes convinced that the average load value is inaccurate whenever its load is distant from the average, yet there is no complimentary node, one whose load is distant from the average in the opposite direction, with which it can exchange processes. For example, if a node's load is above the average, and there are no nodes having loads below the average, it can be inferred that the average load value is too low and must be updated.

An overloaded node locates an underloaded node not by polling, but rather by broadcasting its predicament and waiting for a response from an underloaded node. If a response arrives within a reasonable amount of time, the node responding becomes the destination of a process migrating from the overloaded node. If no response arrives in time, the overloaded node can infer that there are no underloaded nodes and that the average load value is inaccurate. Similarly, when a node becomes under-

loaded, it waits for an overloaded node to broadcast its status. If a node reports that it is overloaded within a reasonable amount of time, the underloaded node can respond and thus commit itself to accept a migrating process from the overloaded node. If no node reports that it is overloaded in time, the underloaded node can infer that there are no overloaded nodes and that the average load value is inaccurate.

If the load-balancing algorithm requires that each node strive to adjust its load to be exactly equal to the average, processor thrashing [12] may result. The migration of a process to an underloaded node may increase its load to the the point of making it overloaded, necessitating the migration of that process to yet another node. Processor thrashing of this type can be avoided by defining an *acceptable load range* that extends on each side of the average load. A node whose load falls within the acceptable range is neither overloaded nor underloaded, so is not a suitable source or destination for a migrating process.

The width of the acceptable range determines the responsiveness as well as the communication cost of the load-balancing algorithm. A wider acceptable range allows more variation in load among the nodes and disallows less fruitful migrations between nodes having similar loads. A narrower range causes the load-balancing algorithm to strive to keep the variance in loads small at the cost of performing less fruitful migrations. The minimum width of the acceptable range should be the maximum load that can be imposed by a single process.

Choosing an appropriate width for the acceptable range allows the load-balancing algorithm to adapt to changes in the multicomputer environment. The acceptable range should be widened whenever a large portion of the bandwidth of the communication medium is occupied, causing long delays for migrating processes and load-balancing messages; it should be narrowed whenever the communication medium is underutilized. While the width of the acceptable range must be globally accessible,

there is no need to distribute it to all nodes if each node is equally able to measure the relevant factors and calculate its value.

2.2. Determining which process to migrate

Once the negotiation of a process migration has been completed, the load-balancing algorithm must choose which process should migrate from among those resident on the overloaded node. Factors that are important in choosing the best process to migrate include the following:

(1) The migration of a process that is not ready to execute will not affect the source node's load, so it is less useful.

(2) Migrating the process that is currently executing incurs an extra context switch.

(3) The process with the most favorable current response ratio can best afford the time penalty imposed by a migration.

(4) A process with a small executable image places the smallest burden on the communication medium and the source and destination nodes.

(5) The process that will require the most CPU time in the future is likely to amass the greatest long-term benefit from migration. The amount of CPU time that a process will use in the future can be approximated by the amount of time it has used in the past [12].

(6) Processes that communicate with peers on the destination node more often than with peers on the source node will reduce the load on the communication medium when they migrate. Again, future performance can be approximated by past performance.

(7) Migrating the process responsible for the largest fraction of the load on the source machine has the largest influence on the remaining load.

Some or all of these attributes can be combined to form a heuristic choice of the best process to migrate. How these factors are combined depends on the aspect of load that the load-balancing algorithm aims to improve. If overall multicomputer utilization is important, then the process' size should be a primary consideration in the decision. However, if the average response ratio over the multicomputer is important, then the process' response ratio is a more important factor. The Above-Average algorithm chooses the process to migrate from among those that are ready to execute, but are not currently executing. From these, the process having the lowest response ratio is chosen. If several processes have approximately the same response ratio as the lowest, then the process having the smallest size is chosen from among these.

2.3. Measuring Processor Load

The mechanism for measuring local processor load is at the heart of a load-balancing algorithm. Depending on what one wishes to improve, it may be a measure of throughput, processor utilization, or process response ratio. As a measure of the instantaneous load on a node, we have chosen to count the number of processes that are currently ready to execute. This value is quickly evaluated, so it is responsive to changes in the local state. The number of processes currently ready to execute has been shown to be significant in the measuring of processor load [8]. This value is correlated with the instantaneous processor utilization. If one or more processes is ready to execute, the utilization is one; otherwise it is zero. This value also correlates with the instantaneous response ratio, and is an exact measure if each process always uses all of its timeslice.

A node must not commit itself to accept so many processes that, when they finally arrive, it becomes overloaded, since processor thrashing might result. This problem can be avoided by augmenting the node's actual load with its *virtual load*, the load expected to be induced by processes that are in transit. A node's virtual load

increases when the node agrees to accept a migrating process, and decreases when the migrating process arrives or the node becomes convinced that the migrating process will never arrive. The instantaneous load on the node is the sum of its actual and virtual loads.

Rapid fluctuations in the instantaneous load on a node may cause many spurious and nonproductive migrations. A node may find that it is underloaded and negotiate the migration of a process only to find that, while the migrating process is in transit, it has become overloaded and must rid itself of a process. To be of greater use to the load-balancing algorithm, the load measured should be an indicator of the load on the node over a period of time at least on the order of the time required for a process to migrate. The appropriate low-bandpass filter for processor load can be constructed by averaging the instantaneous load over a number of measurements.

For other approaches to measuring load, see Bryant and Finkel [12].

2.4. Detailed Description

The types of messages used for load balancing are:

TooHigh	The sender is overloaded. This message is broadcast to invite any node to send an *Accept* message.
Accept	The sender is underloaded and commits itself to accept a migrating process. This message is directed to an overloaded node that previously sent a *TooHigh* message.
ChangeAverage	A node believes that the average load value, which serves as the center of the acceptable range, is inaccurate. This broadcast message contains an updated average load value.

The actions taken by a node in each of the following states are:

A node becomes overloaded:

> Broadcast a *TooHigh* message, start listening for an *Accept* message and set a *TooHigh* timeout alarm.

An *Accept* message is received at a node that is waiting for a response to its *TooHigh* message:

> Cancel the *TooHigh* timeout alarm. If this node is still overloaded, choose the best process to migrate and send it to the source of the *Accept* message.

A *TooHigh* timeout alarm expires, indicating that the node has been overloaded for too long:

> If this node is still overloaded, increase the average load value by broadcasting a *ChangeAverage* message.

A node becomes underloaded:

> Start listening for a *TooHigh* message and set a *TooLow* timeout alarm.

A *TooLow* timeout alarm expires, indicating that the node has been underloaded for too long:

> If this node is still underloaded, decrease the average load value by broadcasting a *ChangeAverage* message.

A *TooHigh* message is received at a node that is underloaded:

> Cancel the *TooLow* timeout alarm, send an *Accept* message to the source of the *TooHigh* message, increase the virtual load for this node and set an *AwaitingProcess* timeout alarm.

An *AwaitingProcess* timeout alarm expires:

> The node loses hope of ever receiving the migrating process for which it sent an *Accept* message. Decrease the virtual load.

A migrating process arrives:

Install the process on this node. Decrease the virtual load.

The utilization of the communication medium is high enough to cause long message delays:

Increase the local copy of the global *width of the acceptable range* value.

The communication medium is underutilized:

Decrease the *width of the acceptable range* value.

3. Computer Simulation

A multicomputer having 40 nodes was simulated.

The simulation assumes a processor-sharing discipline. A Poisson arrival process is assumed at a rate such that the overall multicomputer utilization ρ is 0.8. Process service times have an exponential distribution with a mean of 3 seconds. The mean time interval that a process runs before blocking is 100 milliseconds; the distribution of these intervals is exponential. Block intervals have a normal distribution, with a mean of 40 milliseconds and a variance of 100 milliseconds. Process sizes have an exponential distribution, with a mean of 20K bytes.

A communication device having a bandwidth of 10 million bits per second is assumed. The load-balancing algorithm adjusts the width of the acceptable range of processor loads in an attempt to keep the communication line utilization between 0.05 and 0.5. The maximum size of a packet on the communication device is 4K bytes. The size of each load-balancing control message (TooHigh, Accept or ChangeAverage) is 12 bytes. Receiving a packet from the communication device requires 1 millisecond of processor time; sending a packet requires 3 milliseconds. Messages other than those used by the Above-Average algorithm are not simulated.

The time interval for the *TooHigh* and *TooLow* and *AwaitingProcess* timeout alarms is 30 milliseconds.

The duration of the simulation is 100 seconds.

The effectiveness of the Above-Average algorithm as a load-balancing algorithm is illustrated by figure 1, which shows a reduction in the standard deviation of processor loads.

We define the process response ratio as the total time required by a process to complete, divided by the time that would have been required had processor resources been allocated ideally from the point of view of that process. Using this definition, the time spent by a process waiting for its I/O requests to complete does not affect its response ratio. A process response ratio is degraded (increased) only when a process is ready to execute, but is waiting in the ready list or is in transit to another node. Figure 2 shows that the Above-Average algorithm improves the mean process response

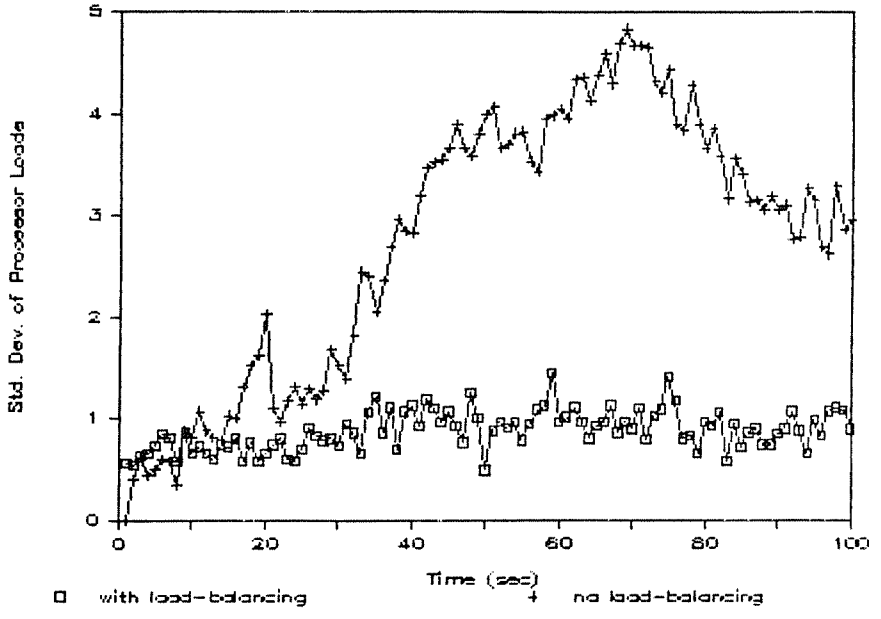

Fig. 1. Standard deviation of processor loads

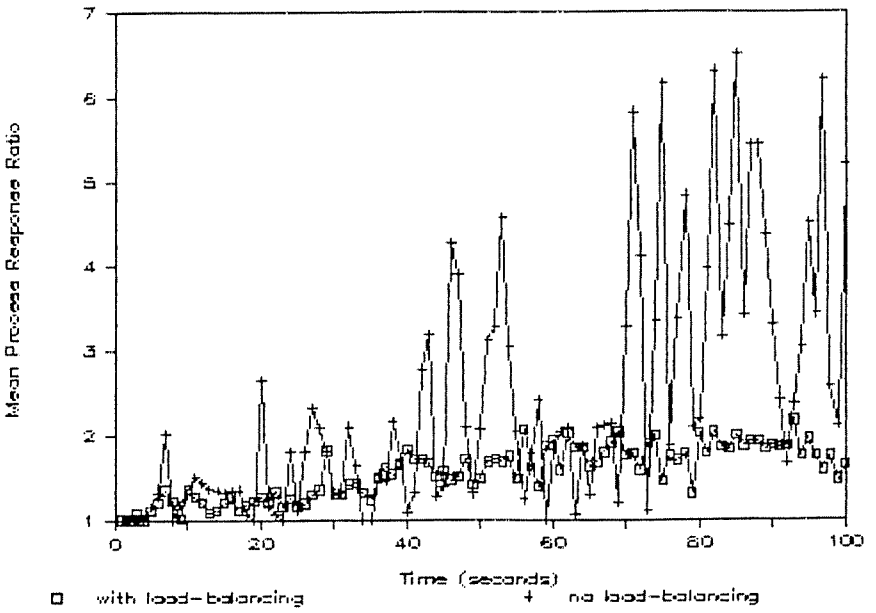

Fig. 2. Mean process response ratio

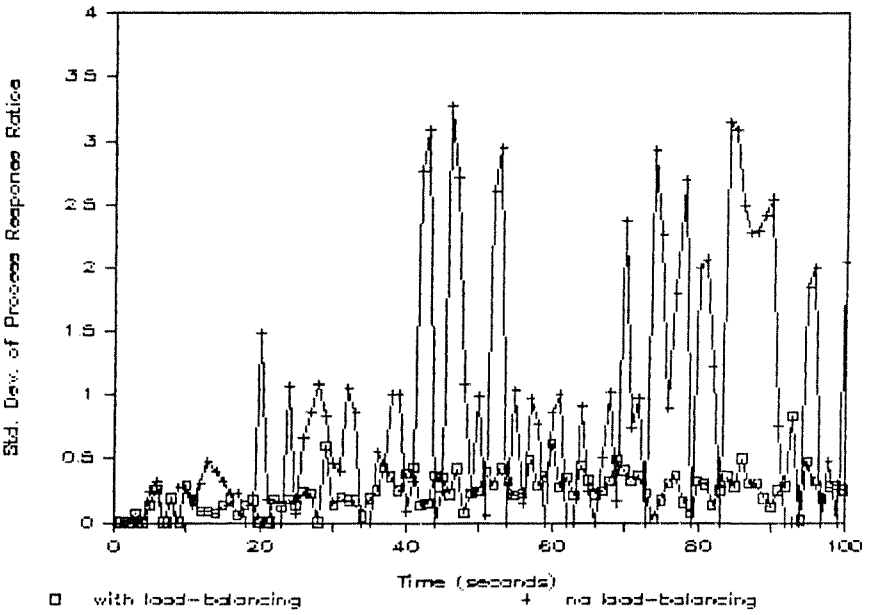

Fig. 3. Standard deviation of process response ratios

ratio. Figure 3 shows that the Above-Average algorithm also improves the fairness of processor resource allocation by reducing the standard deviation of process response

ratios.

Figure 4 shows that the Above-Average algorithm does not impose a heavy load on the communication device. It also shows that the algorithm can adapt to changes in the load on the communication device. After the first ten seconds, the utilization of the communication medium remained within the specified bounds throughout the simulation.

4. Extensions to the Above-Average Algorithm

4.1. Symmetry

As presented, the Above-Average algorithm suffers a lack of symmetry. An overloaded node broadcasts its plight and is noticed by all underloaded nodes. However, a node whose load plummets below the acceptable range after the broadcast has occurred will not realize that an overloaded node is awaiting a response. The underloaded node may then incorrectly infer that the average load value is too high. To

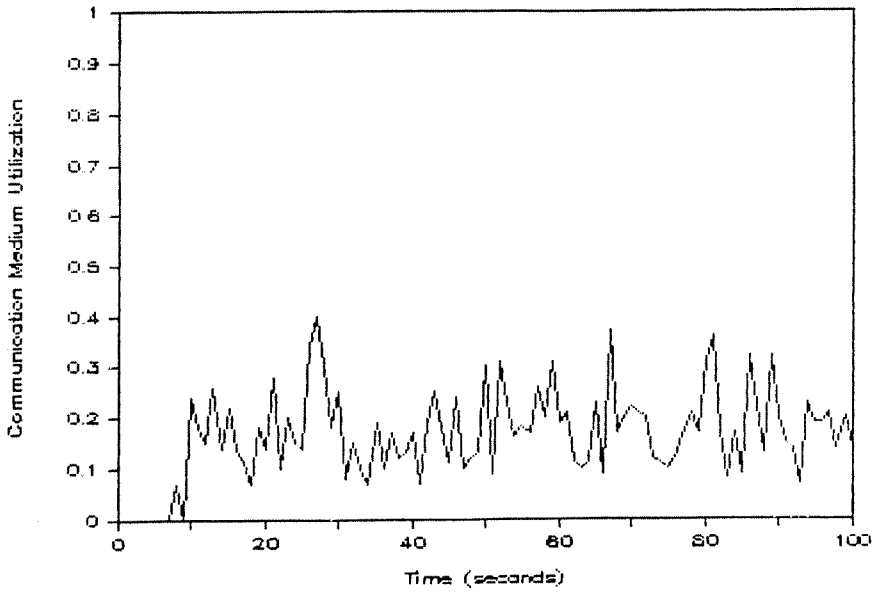

Fig. 4. Utilization of the communication device over 1-second intervals

remove this asymmetry, a node that becomes underloaded broadcasts its *TooLow* status, which serves as an invitation to all nodes that are waiting for a response to their *TooHigh* status messages to send another *TooHigh* message to this node.

4.2. Limiting Acceptances

Two problems arise when an overloaded node sends a *TooHigh* message at a time when there are many underloaded nodes. The first is that handling the profusion of *Accept* messages sent in response to the *TooHigh* message may further degrade the throughput of the node, which already has too heavy a workload. The second is that, since only one process will migrate from the overloaded node, all but one of the nodes that send *Accept* messages must wait until they are certain that the process will not arrive before being able to safely commit to accept another process. An overloaded node may get no response from underloaded nodes, even though some may exist, because they are all waiting to hear if they will be chosen by some other overloaded node.

A solution to both of these problems, which we will refer to as *Single Accept*, is to broadcast the underloaded node's commitment to accept a process, its *Accept* message. No node should commit itself to accept a process that has already been accepted by another node. Assuming that no broadcast messages are lost, no more than one node will wait for a migrating process to arrive from a particular TooHigh node.

A migration between nodes having a large difference in loads is better than a migration between nodes having a smaller difference in loads because it causes a greater reduction in the variance in loads. Another solution, *Multiple Accept*, allows a better match between overloaded and underloaded nodes than does Single Accept. Under Multiple Accept, the underloaded node includes the value of the current load in its broadcast *Accept* message. The overloaded node chooses the most underloaded

node among those that sent *Accept* messages. An underloaded node is allowed to commit itself to accept a process that has already been accepted by another node if its load is lower than that of any other node that has accepted the process. An underloaded node's commitment is released if a more underloaded node later accepts the same process or if the node times out waiting for the process to arrive. Assuming that no broadcast messages are lost and n is the number of underloaded nodes, each with distinct loads, that might respond, the number that will send *Accept* messages in response to a particular *TooHigh* message is H_n, the nth harmonic number. This number grows logarithmically in n [15, p. 74].

Of all the underloaded nodes that send *Accept* messages, only one waits for a migrating process. Multiple Accept, then, allows an overloaded node to choose the best underloaded node as a destination for a migrating process at the cost of only a few more *Accept* messages than the Single Accept version.

4.3. Constraints Imposed by Finite Memory

So far, we have ignored the constraint that each node has a finite amount of memory. To extend Single Accept to satisfy this constraint, the amount of available memory on the underloaded node may be included in its *Accept* message. If the *Accept* message received by the overloaded node indicates that the underloaded node does not have enough memory available to contain any of the overloaded node's processes, then the acceptance is ignored and a new *TooHigh* message is broadcast. An underloaded node is released from its commitment to accept a process if it receives a *TooHigh* message from the node to which it is committed before the initial portion of the migrating process arrives. It does not send an *Accept* message to this overloaded node unless it now has more memory available than it did when its last *Accept* was sent, or if no other node sends an *Accept* message. The mean number of *Accept* messages sent in response to a *TooHigh* message, and the number of nodes

that wait for the arrival of a particular process, is identical to Single Accept without the finite memory constraint.

To extend Multiple Accept to satisfy the finite memory constraint, the *Accept* message is expanded to include the amount of available memory on the underloaded node as well as its current load. An underloaded node is allowed to commit itself to accept a process if no node that has at least as low a load and at least as much available memory has already committed for that same process. A node's commitment to accept a process is released if another node, whose load is at least as low and that has at least as much available memory, later accepts the same process, or if the node times out waiting for the process to arrive. Again, assuming that no broadcast messages are lost, if n is the number of distinct loads on underloaded nodes at the instant a *TooHigh* message is broadcast, then the mean number of nodes that send *Accept* messages in response to a single *TooHigh* message is $\frac{H_n^2}{2}$ plus lower-order terms. In general, if each *Accept* message contains d pieces of information, and a message is sent by an underloaded node unless its message is dominated in all respects by some other *Accept* message that has already been sent in response to the *TooHigh* message, then the mean number of responders out of n underloaded nodes is $\frac{H_n^d}{d!}$ plus lower-order terms. This quantity is roughly $\frac{\log_2^d(n)}{d!}$. A proof of this assertion can be found in the Appendix.

4.4. Detailed Description of the Extended Algorithm

The types of messages used for load-balancing are:

TooHigh — The sender is overloaded. This message may be broadcast or may have a single destination. It serves as an invitation to a node to send an *Accept* message.

TooLow The sender is underloaded. This broadcast message serves as an invitation to any node that is waiting for a response to its *TooHigh* message to send another *TooHigh* message to this node.

Accept The sender is underloaded and commits itself to accept a migrating process from an overloaded node. This is a broadcast message and includes the load of its sender.

ChangeAverage A node believes that the average load value, which serves as the center of the acceptable range, is inaccurate. This broadcast message contains an updated average load value.

The actions taken by a node in each of the following states are:

A node becomes overloaded:

Broadcast a *TooHigh* message, start listening for a *TooLow* or an *Accept* message and set a *TooHigh* timeout alarm.

A *TooLow* message is received at a node that is waiting for a response to its *TooHigh* message:

If this node is still overloaded and no *Accept* messages have been received, send a *TooHigh* message to the source of the *TooLow* message and reset the *TooHigh* timeout alarm.

An *Accept* message is received at a node that is waiting for a response to its *TooHigh* message:

If the load of the source of this *Accept* message is lower than that of any other *Accept* received, remember the source node of this message.

A *TooHigh* timeout alarm expires, indicating that the node has been overloaded for too long:

If this node is still overloaded:

If an *Accept* message has been received, choose the best process to migrate and send it to the source of the *Accept* message. Otherwise, increase the average load by broadcasting the updated value in a *ChangeAverage* message.

A node becomes underloaded:

Broadcast a *TooLow* message, start listening for a *TooHigh* message and set a *TooLow* timeout alarm.

A *TooLow* timeout alarm expires, indicating that the node has been underloaded for too long:

If this node is still underloaded, decrease the average load value by broadcasting a *ChangeAverage* message.

A *TooHigh* message is received at a node that is underloaded.

Cancel the *TooLow* timeout alarm, send an *Accept* message to the source of the *TooHigh* message, increase the virtual load for this node and set an *AwaitingProcess* timeout alarm.

An *Accept* for the same process that this node has Accepted is received from a node whose load is at least as low and has at least as much available memory as this node.

If the *Accept* message has not yet been sent due to contention on the communication medium, abort it. Decrease the virtual load on this machine and cancel the *AwaitingProcess* timeout alarm.

An *AwaitingProcess* timeout alarm expires:

The node loses hope of ever receiving the migrating process for which it sent an *Accept* message. Decrease the virtual load.

A migrating process arrives:

Install the process on this node. Decrease the virtual load.

The utilization of the communication medium is high enough to cause long message delays:

Increase the local copy of the global *width of the acceptable range* value.

The communication medium is underutilized:

Decrease the *width of the acceptable range* value.

5. Conclusions

A new decentralized load-balancing algorithm has been described that uses broadcast messages to maintain global information and to find nodes with which to exchange processes. The Above-Average algorithm is relatively simple and has been shown, by simulation, to require little processor or communication overhead, yet it is quite effective. It improves the mean process response ratio and improves the fairness of processor allocation. It is adaptive to processor and communication device load, so it is able to function well within a very wide range of environments. In addition, several extensions to the basic algorithm have been suggested, which will be examined more closely in a future paper.

6. Acknowledgements

We would like to thank Sam Bent for showing us how to prove the results shown in the appendix. We also thank Ruth Ginzberg, who provided insights in early discussions of the Above-Average algorithm.

7. Appendix

We now prove the formulas presented in the text concerning the number of *Accept* messages that are sent. First, consider the case $d = 1$, in which only one datum is contained in each message. Assume the n nodes sending the message have data $a_1, a_2, \cdots a_n$ respectively. Define D_i to be the number of nodes $j < i$ such that $a_j < a_i$. Clearly, $D_i < i$. If the original values of a_i are a random permutation, then

the D_i are independent, and $\text{Prob}(D_i = j) = \frac{1}{i}$ for all $0 < j < i$ [17, pp. 12-13]. The number of messages sent is the number of D_i that are 0. The expected number of such messages is therefore $\sum_{0<i\leq n} \frac{1}{i} = H_n$.

In the case $d = 2$, the data are represented by sequences a_i and b_i. Once again, we define D_i based on a_i. The ith node will send an *Accept* message if b_i is lower than all b_j for the D_i values of j for which $a_j < a_i$. If $D_i = k$, then the probability of sending is $\frac{1}{k+1}$. The expected number of messages is then

$$\sum_{1\leq i\leq n} \sum_{0\leq k<i} \frac{1}{i} \frac{1}{k+1} = \sum_{1\leq k\leq i\leq n} \frac{1}{ik}$$

Similar arguments show that for arbitrary d, the expected number of messages is

$$\sum_{1\leq i_1\leq \cdots \leq i_d\leq n} \frac{1}{i_1 \cdots i_d}$$

The largest term in this sum is $\frac{H_n^d}{d!}$ [16, p. 91-92].

8. References

[1] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer 13*, 11, pp. 57-69 (November 1980).

[2] S. Krishnaprasad and C. C. Price, "Distributed computer network scheduling algorithms and their complexities," Technical Report No. CSE-8105, Southern Methodist University Department of Computer Science and Engineering (August 1981).

[3] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Transactions on Software Engineering SE-8*, 4, (July 1982).

[4] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Transactions on Software Engineering SE-7*, 6, (November 1981).

[5] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Transactions on Computers C-28*, 4, (April 1979).

[6] P. J. Denning, "Working sets past and present," *IEEE Transactions on Software Engineering*, pp. 64-84 (January 1980).

[7] R. G. Smith, "The contract net protocol: high-level communication and control in a distributed problem solver," *IEEE Transactions on Computers C-29*, 12, (December 1980).

[8] L. M. Ni, "A distributed load balancing algorithm for point-to-point local computer networks," *Proceedings of Compcon, Computer Networks*, pp. 116-123 (September 1982).

[9] Y. C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Transactions on Computers*, pp. 354-361 (May 1979).

[10] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," *Computer Network Performance Symposium*, pp. 47-55 (April 1982).

[11] A. B. Barak and A. Shiloh, A Distributed Load Balancing Policy for a Multicomputer, Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel (1984).

[12] D. A. Reed, "A simulation study of multimicrocomputer networks," *1983 International conference on parallel processing*, pp. 161-163 (August 1983).

[13] R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. Second International Conference on Distributed Computing Systems*, pp. 314-323 (April 1981).

[14] L. D. Wittie and A. Van Tilborg, "Control hierarchies for arbitrarily connected microcomputer networks," Technical Report 126, Department of Computer Science, State University of New York at Buffalo (May 1977).

[15] Y. Eran, M. Livny, and M. Melman, "Modeling and evaluation of a tree structured network: a case study," *Proceedings of MELECON*, (1981).

[16] D. E. Knuth, *The Art of Computer Programming Volume 1–Fundamental Algorithms (second edition)*, Addison-Wesley (1973).

[17] D. E. Knuth, *The Art of Computer Programming Volume 3–Sorting and Searching*, Addison-Wesley (1973).