The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme

Thomas Kunz

December 1991

TI-6/91
Institut für Theoretische Informatik
Fachbereich Informatik
Technische Hochschule Darmstadt

The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme*

Thomas Kunz
Technical University Darmstadt
Institut for Theoretical Computer Science
Alexanderstraße 10
6100 Darmstadt

Federal Republic of Germany

December 1991

Abstract

This paper discusses load balancing heuristics in a general-purpose distributed computer system. To minimize the mean response time of a task, every new task is scheduled to be executed either locally or at a remote host, depending upon the current load distribution. We implemented a task scheduler based on the concept of a Stochastic Learning Automaton on a network of Unix workstations. The used heuristic and our implementation are shortly discussed. Creating an artificial, executable workload, a number of experiments were conducted to determine the effect of different workload descriptions. These workload descriptions characterize the load at one host and determine, whether a newly created task is to be executed locally or remotely. Six single workload descriptors have been examined. Also, two more complex workload descriptions combining single workload descriptors were considered. The best results were obtained with a relatively simple workload description, the number of tasks in the run queue per host. Using more complex workload descriptions, in contrast, did not improve the mean response time, as compared to the best single workload descriptor.

Index terms: Distributed systems, heuristics, load balancing, stochastic learning automata, workload descriptions

*Appeared in IEEE Transactions on Software Engineering, July 1991, pages 725–730.

1 Introduction

The advantages of a distributed computer system over a single powerful general-purpose computing facility are multiple, for example: availability, extensibility, and increased overall performance. In order to make use of these advantages, efficient algorithms for the system-wide control of resource sharing are needed. One especially important resource is processor power. In a general–purpose computing facility, each host (processing unit) has to execute a number of tasks. The scheduling problem consists in determining the host at which a specific task is to be executed such that a system–wide function is optimized. Possible global goals are minimizing the mean response time of all tasks, the costs of executing the tasks, etc. This paper concentrates on heuristic load balancing algorithms, because finding the optimal solution has shown to be NP–complete in general, see Price and Krishnaprasad[7] or Stramm and Berman[14]. Efficient optimal algorithms exist only for special cases, like Stone's algorithm for assigning tasks to a system of two or three processors. This algorithm requires that the complete knowledge of the behaviour of a task (e.g. run costs and interprocessor communication costs) is available, see Stone[13].

Global task scheduling heuristics have been proposed in a number of articles. These heuristics differ with regard to the characteristics of the underlying network, the information required to make a scheduling decision, considered constraints, used performance criteria, and the design choices as discussed by Casavant and Kuhl[1]. We impose the following requirements upon a scheduler for a general-purpose distributed computer system:

- no assumptions about the underlying network (e.g. topology, homogeneity, etc.),
- no a-priori knowledge about incoming tasks,
- dynamic, physically distributed and cooperative scheduling,[1] and
- mean response time of a task as performance criteria.

In any specific case, other requirements may prove to be more important. But in general those schedulers with the above mentioned characteristics are best suited for a general-purpose distributed computer system. Among the reviewed heuristics, those proposed in [5, 9, 10, 11] come closest in fulfilling these requirements.

One important characteristic of a load balancing scheme is the used workload description to characterize the load of a single host. Most schedulers reported in the literature use the number of tasks in the run queue as workload descriptor. This descriptor seems to be an important workload characteristic and one obvious advantage of it is the simplicity of its measurement. But an ongoing area of research is the impact of more sophisticated workload descriptions.

Although intuitively it may seem obvious that the use of more sophisticated workload descriptions may only improve the overall performance, this is not necessarily the case. More sophisticated descriptions are typically more difficult to measure and to broadcast, thereby increasing the scheduling overhead. Eager et al.[2] for example noticed in their experiments, using different kinds of scheduling algorithms, that extremely simple load sharing policies using small amounts of information perform nearly as well as more complex policies that utilize more information.

[1] as defined by Casavant and Kuhl[1]

The research reported here examined the effects of experimenting with different ways of characterizing the workload of a single host. The reported results have been collected by running experiments with an implemented scheduler. This scheduler is based on the concept of a Stochastic Learning Automaton, as described in Mirchandaney and Stankovic[5]. The next section describes this heuristic and our implemented scheduler. Section 3 discusses the general setup of our experiments and section 4 reports the results of these experiments. Six single and two more complex workload descriptors, combining multiple single descriptors, have been examined. Our goal was to identify which workload description would result in the lowest mean response time of a task, creating an artificial, executable workload. The paper ends with a summary of our findings and discusses their implication.

2 The load balancing heuristic

2.1 The implementation

A scheduler based on the heuristic described in [5] was implemented in C on five Sun 3/50 workstations connected by ethernet. All Suns are diskless, sharing a common file server. Because all workstations are homogeneous and all files are equally accessible from each host due to the implemented NFS (Network File System), executing a task on a remote host only requires transferring the appropriate command to this host.

A copy of the scheduler runs at each workstation. This copy decides, where locally created tasks are to be executed. It also receives requests from other copies to execute one of their tasks. No tasks are executed at a remote host as long as the local host is underloaded. Is the local host overloaded, newly arriving tasks are executed at a remote host according to the below described heuristic. In the case that all hosts are overloaded, remote task execution is suppressed, because no performance gains are possible. In this case, locally created tasks will be executed locally.

2.2 The heuristic

2.2.1 An extended Stochastic Learning Automaton

The algorithm is based upon the concept of a Stochastic Learning Automaton as described in [5, 6, 12]. Such an automaton serves the purpose of finding optimal actions out of a set of allowable actions. Possible actions for our scheduler are of the form "Execute a task at host x", where host x is a remote host. A stochastic automaton attempts to solve this problem in the following way. A probability for the selection of an action is attached to all possible actions. Originally, all these probabilities are equal, since nothing is known about the optimality of each action. One action is selected at random and the response of the environment to this action is observed. Based on this response the action probabilities are changed. The way in which the action probabilities are updated is determined by the learning scheme used. The next time an action is to be selected, the updated action probabilities are used and the whole procedure is repeated.

This general scheme has been extended in [5], where the notion of automaton states

is introduced. The automaton as described above uses only one probability vector (one probability for each action). An automaton with multiple states uses multiple probability vectors for the selection of actions. When a scheduling decision is to be made, one of these vectors is selected and used as described above. Since every automaton state is uniquely associated with a certain probability vector, we will use the terms "automaton state i" and "probability vector i" as synonyms.

The network-wide state determines, in which automaton state the scheduler will work. This network-wide state is determined by passing status information of the form "Host i is underloaded/overloaded" between all hosts. To determine the appropriate automaton state, a mapping between the status information and the automaton states/probability vectors is defined. In our experiments, a scheduler with 4 states (on a network of five hosts) was used. The mapping between the observed network–wide state and the internal automaton state was defined as follows. Host i (i=0,...,4) successively scans the received status information for hosts i+j+1 mod 5 (j=0,..,3) until an underloaded host is found. If host i+j+1 mod 5 is the first underloaded host, the probability vector j will be used in making a placement decision. According to [12], this mapping leads to a stable behaviour of the scheduler.

2.2.2 The load threshold

Each host periodically broadcasts its status (underloaded or overloaded) to all other hosts. To be able to determine this status, the value of a specified workload descriptor is measured and compared to a given threshold. A host is considered overloaded when the measured workload exceeds the threshold. Determining the optimal threshold value is not trivial and depends on the network–wide load, see [8].

2.2.3 The learning process

The stochastic learning automaton aims to learn the best actions for each automaton state to improve the performance of the scheduler. That is, the action probability for remotely executing a task at an underloaded host is increased and/or the action probability for remotely executing a task at an overloaded host is decreased. The goodness of an action is evaluated at the remote host and transmitted to the local scheduler copy as a binary measure of goodness (selected action was good/bad). The way in which the action probabilities are updated is defined by the learning scheme used, which can be described by the following high-level description. Let $p_j(n)$ denote the probability for choosing action j at time n. Then $p_j(n+1)$ denotes the same probability at time n+1. Assume action i was chosen at time n. The probabilities $p_j(n)$ are updated in the following way:

For Reward update:
$$p_j(n+1) = p_j(n) - f[p_j(n)] \qquad j \neq i$$
$$p_i(n+1) = p_i(n) + \Sigma\, f[p_j(n)]$$

For Penalty update:
$$p_j(n+1) = p_j(n) + g[p_j(n)] \qquad j \neq i$$
$$p_i(n+1) = p_i(n) - \Sigma\, g[p_j(n)]$$

where f and g are reward and penalty functions respectively. The learning scheme reported in [5] and used in our implementation uses the following reward and penalty functions:

$$f[p_j(n)] = A * p_j(n) \qquad 0 < A < 1$$
$$g[p_j(n)] = B/(r\text{-}2) - B * p_j(n) \qquad 0 < B < 1$$

where r is the number of hosts in the network and A and B are constants. The functions f and g should be nonnegative for the whole range from 0 to 1 to fit the intuitive impressions of reward and penalty. But nonnegativity is no necessary requirement, as the used penalty function g demonstrates.

2.2.4 Summarized description of the scheduler

The scheduler can be summarized as follows. Each copy periodically checks the status (overloaded or underloaded) of the host it runs at and broadcasts this status to all other copies. Using these status information, each copy determines the overall network-wide state which it is observing at this time and maps this network-wide state to one of its automaton states. Each time a task is to be executed remotely, a remote host is randomly chosen, using the probability vector currently in effect (i.e. the probability vector associated with the current automaton state). The selected remote copy evaluates the goodness of the selection, depending on the status of its local host. This evaluation (called measure of goodness) is send back to the scheduling copy, which updates the appropriate probability vector. After a while, each copy supposedly learns the best actions for different observed network–wide states.

2.3 Preliminary tuning results

The scheduler, as described above, contains a number of parameters that influence its performance. In a number of tuning runs, values for these parameters have been established and have been used in the subsequently reported experiments. Status information are measured and broadcasted every 8 seconds. The learning scheme uses a reward constant A of 0.25 and a penalty constant B of 0.3. Other experiments not reported here examined the influence of different numbers of automaton states, the behaviour of the scheduler under different network sizes and the use of different learning schemes.

3 The test environment

3.1 Generating a workload

The tasks that are scheduled are invocations of a single generic task. Since this task is written especially for our experiments and does not represent a sample of a production workload, we are using an executable artificial workload approach as defined in [3]. We chose this approach because artificial workloads are easier to reproduce and have a greater flexibility than natural workloads. The main disadvantage of artificial executable workloads

is that they require the computer network to be totally dedicated to each experiment. This is the main reason why we ran our experiments during the nights.

Tasks are created in real-time, that is, at the time they are needed, by a driver. The driver is a program which generates task parameters from given distributions. Then it passes these values to a copy of the task and simulates the task's arrival.

3.2 The artificial workload model

To emphasize the differences between workload descriptors, we developed a workload model which characterizes a workload along the following four dimensions:

- arrival process,
- processing time requirements,
- I/O-volume, and
- memory requirements.

The generic task has the following structure. First, the required memory is allocated. Second, a specified amount of data sentences are read from a file. Third, a processing phase is simulated by repeatedly reading/writing 256 consecutive bytes into the allocated memory space for a specified amount of time. The starting position for each of these byte blocks is selected randomly. Fourth, the data sentences are written back into the file and finally, the memory is deallocated.

A specific probability distribution underlies each of these dimensions. The task arrival process follows a Poisson distribution with arrival rate μ. The processing times are exponentially distributed with processing rate λ. The I/O-volume is determined by multiplying the processing time with a constant, depending on the class of the task. We distinguish three different classes: balanced (b), I/O-bound (i), and CPU-bound (c). Balanced tasks read/write 50 data sentences per processing second, I/O-bound tasks 300 sentences and CPU-bound tasks 5 sentences. Tasks are assigned randomly to these classes, the percentage of tasks in each class varies from host to host. The required memory (m) also depends on the processing time. Tasks with a processing time less than 4 seconds require memory in the range from 0.25 to 50.25 Kbytes. Longer tasks have memory requirements in the range from 0.25 to 200.25 Kbytes. The following table shows the significant values for our new workload model.

host #	μ	λ	%b	%i	%c	ø m (Kbytes)
1	0.102	0.224	0.355	0.350	0.295	55.218
2	0.036	0.196	0.053	0.789	0.158	58.066
3	0.020	0.210	0.048	0.952	0.000	58.631
4	0.130	0.176	0.159	0.062	0.779	61.630
5	0.140	0.209	0.097	0.171	0.732	57.832

Table 1: Workload model data

Creating a good workload is far from simple. Leland and Ott[4] or Svensson[15] discuss some problems with modelling real workloads. Even though we do not claim that our workload is a typical one, we feel confident that our artificial workload allows us to observe some interesting differences with regard to the used workload descriptor.

3.3 Conducting a single experiment

A single experiment runs for 36 minutes, leaving enough learning time for the scheduler. Each task writes his response time (measured as the difference between creation time and end of execution time)[2] in a result file. Mean response times for each host and for the experiment as a whole were calculated from these result files. Each run was repeated twice, varying the random number seed for the selection of a scheduling action. Thus, our results are independent from a specific learning sequence. The results reported in this paper are the mean response times over all three runs.

4 Experimental results

First, we examined whether there is any single workload descriptor which is better suited for the implemented scheduler than the commonly used number of tasks in the run-queue. In a second step, we considered more than one descriptor at the same time, describing the workload of a host by using a combination of multiple descriptors.

4.1 Single workload descriptors

The criteria used to evaluate the scheduler versions is the mean response time of all tasks. To obtain a reference point for the performance of different scheduler versions, we generated our artificial workload without trying to balance the load. The mean response time for the whole system of five hosts is 31.215 seconds. The results for each individual host are presented in table 2:

host #	# tasks	ø response time (sec)
1	220	19.098
2	076	13.980
3	042	12.509
4	276	45.012
5	299	34.402

Table 2: No load balancing

Unix provides a large amount of statistical information that can be used to describe a workload. We successively used the following descriptors:

[2]For tasks that are created at one host but executed at another host, clock differences between the hosts had to be taken into account.

– number of tasks in the run–queue,
– size of the free available memory,
– rate of CPU context switches,
– rate of system calls,
– one minute load average, and
– the amount of free CPU time.

For every workload descriptor used, we expected the mean response time first to decrease and then to increase with increasing threshold value. For lower threshold values, executing tasks remotely is suppressed frequently because the whole network is characterized as being overloaded. For higher threshold values, the loads at different hosts become more and more imbalanced before an attempt is made to balance the load.

The first workload descriptor examined was the number of tasks in the run queue. Table 3 shows the obtained results.

threshold	ø response time (sec)
0	15.217
1	13.576
2	14.602

Table 3: Using the number of tasks in the run queue as workload descriptor

Task scheduling improves the performance of the distributed system by as much as 56.5% (13.576 seconds versus 31.215 seconds) for our workload model. The performance resulting from the use of the other descriptors will be compared to the lowest observed mean response time of 13.576 seconds for a threshold value of 1.

The second workload descriptor we examined is the size of the free memory list in Kbytes. Table 4 shows our experimental results as we varied the threshold value.

threshold	ø response time (sec)
400	18.255
450	16.710
500	17.993

Table 4: Using the size of the free memory as workload descriptor

The results show that this workload descriptor also improves the performance of the distributed system over the "No Load Balancing" case. The lowest mean response time, however, is with 16.710 seconds over 3 seconds or 23% worse than when using the number of tasks in the run queue as workload descriptor.

As third workload descriptor we selected the CPU context switch rate. This rate is provided by Unix as the average context switch rate per second over the last five seconds.

We obtained the results reported in table 5 when using this descriptor to measure a host's load.

threshold	ø response time (sec)
15	25.609
20	17.613
25	16.127
30	14.747
35	15.762
40	17.566

Table 5: Using the CPU context switch rate as workload descriptor

The use of the CPU context switch rate as workload descriptor improves the performance of the distributed system too. The best performance we obtained lies with 14.747 seconds for a threshold value of 30 between the performances we achieved with the first two workload descriptors.

The fourth workload descriptor we examined is the rate of system calls. This rate, again, is provided by Unix as the average system call rate per second over the last five seconds. Table 6 shows our experimental results.

threshold	ø response time (sec)
250	15.242
300	14.702
350	14.423
400	16.307
450	16.514

Table 6: Using the system call rate as workload descriptor

The best obtained performance is 14.423 seconds for a threshold value of 350. This is clearly better than the "No Load Balancing" case. Compared to the first three workload descriptors, the use of the system call rate as workload descriptor resulted in the second best performance so far.

The fifth workload descriptor is the one-minute load average. This descriptor, measuring the average number of tasks in the run queue during the last minute, is similar to our first descriptor, the number of tasks in the run queue. But the new descriptor measures the load of a host over a period of time rather than at a certain point of time. And whereas the number of tasks in the run queue is always an integer value, the new descriptor is a real value, potentially allowing for a finer tuning of our implementation. Varying the threshold value, we obtained the results reported in table 7.

threshold	ø response time (sec)
0.8	22.915
1.2	18.988
1.6	17.932
2.0	19.364
2.4	20.238

Table 7: Using the one-minute load average as workload descriptor

Our results show that the use of this workload descriptor results in the worst performance so far, despite its similarity to the best workload descriptor so far. The load average is calculated over a relatively long time period. The scheduled tasks spend less than 20 seconds on the average in the distributed system. Our workload descriptor averages the number of tasks in the run queue over the last minute, a period more than three times as long. Therefore, the descriptor value is influenced by tasks that are no longer in the system, reflecting the current load in an inaccurate way. Using a load average over a shorter time period may result in a better scheduler performance. But Unix does not provide such a descriptor. Using the one-minute load average as workload descriptor may be more appropriate when the tasks stay longer in the system.

The last workload descriptor we examined is the amount of idle CPU-time between two successive load measurements. We measured this descriptor as the percentage of idle time compared to the total period length. A host is considered overloaded when this percentage is less than or equal to a given threshold. We obtained the performance results reported in table 8.

threshold	ø response time (sec)
0	16.501
2	15.692
5	16.255
10	16.469
15	16.841

Table 8: Using the idle CPU-time as workload descriptor

In contrast to the other workload descriptors, the percentage of idle CPU–time is influenced by the used status update period length. But experiments varying this period length showed no improvements in the resulting mean response time.

Table 9 summarizes our results using a single workload descriptor. For each workload descriptor used, it shows the optimal threshold value and the resulting mean response time of the distributed computer system. All examined workload descriptors lower the mean response time, compared to the "No Load Balancing" case. The best workload descriptor

workload descriptor	optimal threshold	ø response time (sec)
number of tasks in the run queue	1	13.576
system call rate	350	14.423
CPU context switch rate	30	14.747
percentage of idle CPU-time	2	15.692
size of the free memory (Kbytes)	450	16.710
one minute load average	1.6	17.932

Table 9: Using a single workload descriptor

in our experiments is the number of tasks in the run queue. Using this descriptor results in a performance of 13.576 seconds for a threshold value of 1. The worst workload descriptor is the one-minute load average for the reasons discussed above. Its usage resulted in a mean response time of 17.932 seconds for a threshold value of 1.6. This is over 4 seconds or 32% worse than the performance resulting from the use of the best descriptor.

4.2 Combination of Workload Descriptors

So far, we characterized the workload of a host by using a single workload descriptor. We also examined the combination of multiple descriptors to measure a host's load. N workload descriptors d_1, d_2, ...,d_n can be combined in a number of ways to determine the status of a host. One possibility is to define a function $f(d_1,d_2,...,d_n)$ of the form $f(d_1,d_2,...,d_n) = a_1 * d_1 + a_2 * d_2 + ... + a_n * d_n$. A host may then be characterized as being overloaded whenever the value of this function exceeds a given threshold. The coefficients a_i have two different functions. First, they are necessary to equalize the different workload descriptors (the number of tasks in the run queue for example is typically less than 10, the size of the free memory list is somewhere in the range between 200 Kbytes and 1000 Kbytes). And second, they determine the relative impact of different descriptors on the function value.

We pursued two different approaches to combine multiple workload descriptors. In one scheduler version, the status of a host is determined by an "Or-Combination" of multiple descriptors. A host is considered overloaded whenever at least one of the workload descriptors has a value above its respective threshold. In another scheduler version, a host is considered overloaded only when all descriptors have values above their respective threshold (an "And-Combination").

A complete examination of workload descriptor combinations is beyond the scope of this paper. In two experiments, we combined the best two workload descriptors, the number of

tasks in the run queue (descriptor 1) and the system call rate (descriptor 2), and observed the resulting performance. The goal of these experiments was to provide a first idea about the relative merits of combining multiple workload descriptors rather than to determine an optimal workload characterization.

Our experimental results indicate that no performance improvements over the scheduler versions using a single workload descriptor can be obtained. An "Or-Combination" of two workload descriptors shows the following characteristic. The lowest measured mean response time of 14.41 seconds is considerably higher than the one that can be obtained when using descriptor 1 as single workload descriptor. 14.41 seconds is measured when the threshold for descriptor 2 is set to a high value, thereby decreasing its influence on a host's measured status. The second scheduler version, using an "And-Combination" of workload descriptors, showed better performance results. The lowest mean response time of 13.68 seconds was obtained when setting the two threshold values to the optimal values reported in table 9. Still, even this scheduler version is not superior to a scheduler using the best single workload descriptor.

5 Conclusions

We implemented a scheduler based upon the concept of a Stochastic Learning Automaton on a network of Unix workstations. Using this implementation, we conducted a number of experiments to examine the influence of workload descriptions on the performance of the implemented scheduler. First, we examined single workload descriptors. All examined workload descriptors lower the mean response time of tasks, compared to the "No Load Balancing" case. We found that the best single workload descriptor is the number of tasks in the run queue. The use of the worst workload descriptor, the one-minute load average, resulted in an increase of the mean response time of over 32%, compared to the best descriptor. Therefore, while the use of all descriptors results in shorter mean response times, the selection of a workload descriptor is non-trivial when minimal mean response times are desired.

In a second step, we combined the best two workload descriptors, the number of tasks in the run queue and the system call rate, to measure a host's load. Our experimental results indicate that no performance improvements over the scheduler versions using a single workload descriptor can be obtained. Although these two experiments do not cover the area of combined workload descriptors comprehensively, we conclude that major performance improvements can hardly be expected when more complex workload descriptors are used.

6 Acknowledgements

The research reported here was partly done while the author stayed at the University of Illinois at Urbana–Champaign. I would like to thank especially Tony P. Ng, who constantly provided valuable support and advice.

References

[1] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General–Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, pages 141–154, February 1988.

[2] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, pages 662–675, May 1986.

[3] Domenico Ferrari. *Computer Systems Performance Evaluation.* Prentice Hall, Inc., Englewood Cliffs, 1978.

[4] Will E. Leland and Teunis J. Ott. Load–balancing Heuristics and Process Behavior, ACM 0–89791–184–9/86/0500–0054.

[5] Ravi Mirchandaney and John A. Stankovic. Using Stochastic Learning Automata for Job Scheduling in Distributed Processing Systems. *Journal of Parallel and Distributed Computing*, pages 527–551, 1986.

[6] Kumpati S. Narendra and M. A. L. Thathachar. Learning Automata – A Survey. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 323–334, July 1974.

[7] Camille C. Price and S. Krishnaprasad. Software allocation models for distributed systems. In *Proceedings of the 5th International Conference on Distributed Computing*, pages 40–47, 1984.

[8] Spiridon Pulidas, Don Towsley, and Jack Stankovic. Design of Efficient Parameter Estimators for Decentralized Load Balancing Policies. Technical Report 87–79, University of Massachusetts, Amherst, August 1987.

[9] John A. Stankovic. The analysis of a decentralized control algorithm for job scheduling utilizing Bayesian decision theory. In *Proceedings of the 1981 International Conference on Parallel Processing*, pages 333–340, 1981.

[10] John A. Stankovic. Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms. *Computing Networks*, pages 199–217, June 1984.

[11] John A. Stankovic. An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling. *IEEE Transactions on Computers*, pages 117–130, February 1985.

[12] John A. Stankovic. Stability and Distributed Scheduling Algorithms. *IEEE Transactions on Software Engineering*, pages 1141–1152, October 1985.

[13] Harold S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, pages 85–93, January 1977.

[14] Bernd Stramm and Francine Berman. Communication–Sensitive Heuristics and Algorithms for Mapping Compilers. In *Proceedings of the ACM SIGPLAN Conference on Parallel Programming: Experiences with Applications, Languages and Systems*, pages 222–234, July 1988.

[15] Anders Svensson. History, an Intelligent Load Sharing Filter. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 546–553, May 1990.