Quantitative
Modeling
In
Parallel
Systems

Incremental Dynamic Load Balancing in Distributed Systems

Erol Gelenbe and Rakesh Kushwaha
EHEI - Université René Descartes
45 rue des Saints-Pères
75006 Paris

R W3.T2.1.v1

29 September 1993

Distribution Level: Public

Approved by: E. Gelenbe

Abstract We consider distributed system architectures composed of an arbitrary number of processing units, interconnected by a high-speed network. We

The QMIPS Project: The QMIPS project is a collaborative research project supported by the CEC as ESPRIT-BRA project no 7269. It is being carried out by the following organisations: CWI (Amsterdam), EHEI (University of Paris V), Imperial College (London), INRIA (Sophia-Antipolis), University of Erlangen, University of Newcastle, University of Torino and University of Zaragoza.

design and evaluate novel *Adaptive Load Balancing* algorithms for both processes or tasks, and data or files. We introduce a gradient descent paradigm to compute on-line load balancing decisions. The purpose is to substantially improve system performance. We have implemented and tested this paradigm on an nCube target architecture, and compared it with the case where no load balancing is carried out, as well as with "random" load balancing. We demonstrate via extensive measurements that simple adaptive load balancing algorithms can substantially improve distributed system performance, over that of systems with no load balancing. The order of magnitude improvement obtained is a reduction of 50% in the average process response time. We also show that simple algorithms can achieve better performance than more sophisticated algorithms, and that they result in less overhead. Numerous measurement results on an nCube are presented, using performance metrics which have already been discussed in the literature.

Keywords: Distributed systems, load balancing, gradient algorithms, performance measurement.

1 Introduction

Task assignment in a parallel or distributed system uses information available when a job is to be executed, so as to judiciously distribute the tasks among processing units (PUs) in order to maximize performance. Optimal task assignment is well known to be computationally intractable, *i.e.* it is NP-hard, so that it cannot be solved exactly when the number of PUs and the number of tasks is large. Therefore it must be implemented by the use of relatively fast heuristics. It is of greatest use when full information about the execution characteristics of parallel or distributed jobs are available before execution, and can only be carried out using sub-optimal strategies.

On the other hand, **load balancing** is the dynamic allocation of tasks and data (or files) to processing units (PUs) so that work is equally shared and performance improved. Its role is complementary to that of task assignment since it uses on-line information which becomes available at process creation time. It can also be an alternative to task assignment if relevant information about jobs, such as the number of nature of their parallel tasks or processes, only becomes available during execution.

Full load balancing, *i.e.* one which dynamically and optimally reallocates all tasks and files as new jobs or tasks arrive, is also NP-hard. However *incremental load balancing*, which makes decisions concerning newly created tasks, or which makes adjustments to task or file assignments as a function of current information, can be carried out efficiently, with algorithms of low complexity.

This is what we will demonstrate and test in this work. We design simple algorithms for dynamic load-balancing of both tasks and files. The process balancing policies we propose are based on gradient descent of a function which expresses the cost of task execution at different sites. File balancing is based on redistribution of files so as to maintain an equal file load at all nodes.

We compare these algorithms with cases where no load balancing is carried out, and test them in an nCube distributed processing environment.

The results indicate that simple load balancing policies can be very beneficial to system and user performance.

1.1 Previous work

There is an abundant and substantial literature on load sharing and balancing policies for distributed systems, some of which is given in the bibliography. Much of this work uses analytical modeling and simulation to evaluate static and dynamic load balancing policies. Load balancing concerns both the placement of processes for execution, and the placement of data or of files which will be used by processes.

In the earliest formulations [1], load balancing was viewed as a file allocation problem. Models which study the file assignment problem are presented in [2]. Most of this work considers static load balancing algorithms to optimally place files at different sites in a distributed system. Soon after, researchers turned their attention to dynamic and adaptive algorithms for file placement, as in [3] where file storage cost and file availability are optimized with the possibility of duplicating files, for a known maximum number of file copies.

Efficient process migration techniques and remote execution have led to studies of load balancing strategies dealing with task allocation and process migration. As with file placement, both static and adaptive strategies have been discussed. The typical goal of a static task allocation strategy is to deterministically allocate tasks to PUs so that the total time to process all tasks is minimized [4, 5]. Tantawi and Towsley developed a technique for static optimal probabilistic assignment [6]. Silva and Gerla in [7], used queueing network models to evaluate static load balancing policies, where jobs are migrated from heavily loaded sites to lightly loaded sites. In [8] it is said that "in a network of nodes, there is a very high probability that at least one node is idle while jobs are queued at some other nodes," which motivates interest in the design of adaptive strategies for job allocation. Several adaptive load sharing strategies [9, 10, 11, 12, 13] for process migration have been proposed. Comparative and comprehensive studies include [14], and [10], which point to the potential benefits of adaptive load sharing and compare different policies, concluding that very simple adaptive load sharing policies which collect a small amount of system state information and use this information in simple ways, can yield dramatic performance improvement.

In distributed programming environments, where a program can be represented as parallel tasks or modules, the optimum module to processor assignment problem is a version of load balancing [19, 20, 21]. In distributed databases, transaction response time is the measure of interest; it is optimized by efficient data allocation and appropriate transaction routing [22].

Load balancing in general distributed systems has been analyzed by modeling the system as a set of N parallel queues which represent the resources, and a central dispatcher which distributes load among queues [10, 23, 24, 25, 26]. These studies include the analysis of static and dynamic policies. Simulation has also been extensively used by researchers investigating adaptive algorithms [29, 30, 31] because of their flexibility and ability to express fine details.

Load balancing has also been investigated in other distributed resource environments. In computer networks, routing can be examined as an instance of load balancing [15, 16, 17, 18]. The goal is to find the optimum paths for packet flow so that some performance measure (delay or response time) is optimized. In [27] a central controling site or network management center, is used to monitor loads and traffic patterns, to periodically compute optimal load distribution, and provide load balancing information to nodes. The load monitoring and load distribution computations can also be carried out in a distributed fashion [28]. These policies are static in nature and are exploited for dynamic load balancing.

More recently in [32], a general method for quantitative and qualitative analysis of adaptive load sharing algorithms is discussed, suggesting that remote execution should be relatively restricted, and that more than 90% of the decisions made by the adaptive algorithm should be correct. The adaptive algorithms considered in [10] and [32] deal only with task or process movement; they assume that jobs and data are one information unit and can migrate to any site.

In our work the control is distributed, load balancing policies are adaptive and they utilize current system state to make decisions.

Our approach will cover both process and file movement and distribution among processing units. We will see in this context that even simplistic load balancing procedures provide overall performance, than no load balancing at all. We will also see that simple load balancing techniques provide performance which is comparable to much more sophisticated approaches.

2 The distributed system environment

We consider a message passing, distributed memory, distributed system in which both processes and data (files) can move among various sites. The target system we consider is a family of processing units, which we interchangeably call "sites" or "nodes", each of which is able to store files, and execute processes. These sites are completely interconnected using a network, which may either be a local area network, or a fast switch or similar device.

The distributed system considered consists of N interconnected PUs, each composed of a processor and storage.

Any task or process is generated at some PU, and requires data (one or more files) for execution. Each file resides normally in some PUs storage; the node where a file resides is referred to as its "host node". Files can be transferred to other PUs on demand. Any file is either *free* or *busy*. It is busy if a copy has been transferred to some other PU; otherwise it is free. A file-copy at a PU is in one of two states: *active* or *non-active*. It is active if it is currently used by a task executing at the node and is non-active if it still resides in that PUs storage without being used.

When a file is requested by some task at a PU, the request is directed to its host. Then:

- If the file is free, a file-copy is transfered to the requesting node. This action marks the file as "busy".
- If the file is busy but not active, the file-copy is transfered back to the host from the node which currently holds it. The host node updates the file status making it active, then sends it to the node requesting it.
- If the file is busy and the file-copy is active, a "file-busy" message is sent to the host node; the corresponding process or task must then wait for the file to become available.

Hence, for every file, either no file-copy exists, or one exists and the host node has the address of the site where it is. We avoid having more than one extra copy of a file. This is a pragmatic choice which is close to what would be expected in practical systems.

We will call the PU where a task is created the "local node"; all other PUs are "foreign nodes" for the file and the task. When a task is created the following **Four Alternatives** can occur:

1. The local node is the same as the host node; the task is simply executed at the local node. The only extra cost (other than job execution) would be in case the requested file is busy, and the file-copy has to be transfered from a foreign node to the host node. This overhead cost will be denoted *file-busy-overhead*.
2. In this and the other two cases below, the local and host nodes are not the same. The local node requests a file-copy from the host node and executes the job locally. Overhead in this case includes the file-copy transfer cost from the host to the local node plus the file-busy-overhead.
3. The job migrates to the host node for execution. Overhead includes process migration cost and file-busy-overhead.
4. The job migrates to some foreign node and the foreign node requests the file-copy from the host node to compute the job. Extra expense includes process migration cost, file-copy transfer cost and file-busy-overhead.

Case 1 requires the least overhead, while Case 4 is the most costly in terms of overhead. The choice among these alternatives must be made so that the job completion time - including overhead - is minimum and load is equally distributed among all PUs.

2.1 Assumptions concerning system operation

Throughout this presentation we assume the *single request acceptance policy* [31], with no rejection allowed *i.e.* migrated tasks arriving at a node cannot be refused and will be executed there.

During the execution of processes in the system, new files can be created and old ones could be deleted. Initially, a file is always created at the local node and other nodes are informed about its creation. After a fixed time interval t_D, if files are not distributed equitably, a redistribution algorithm is executed which redistributes files so that each node has roughly the same occupied space. Files can also be deleted to make sure that the file-system is not full. File deletion is carried out only when 1) a delete process is invoked at a host node whose file system is full, and 2) the file is free. Both file deletion and redistribution use the send-acknowledge policy [33].

The service discipline at all PUs is FCFS for all messages, files, new job requests etc. except for file requests and migrated jobs. File requests for the same file are served in Time-Stamp order. When a file is busy at a node, the node forwards the file request to some foreign node which holds the file-copy. Any subsequent requests for the same file must wait at the host node until the file copy reaches the requesting node. In the meantime, the node services the other requests in FCFS order.

File placement and job execution are transparent to the users. A user submits a task at the node where it is logged (local node), but the task receives service at some arbitrary node determined by the load balancing algorithm.

In the experiments described here, we use the nCube, a commercial MIMD multiprocessor, to experiment the algorithms and obtain measurements for different strategies, where each PU of the nCube represents a site of the distributed system. We will compare our proposed dynamic gradient descent based policies with the case when no load balancing is carried out, and with random load balancing, using the efficiency and performance measures discussed in [32].

3 Simple and effective load balancing policies

Before we enter into the specific algorithms we will study in the sequel, let us consider four simple strategies to migrate processes and transfer files-copies.

For the ease of comparison, in all four strategies we initially allocate an equal number of files at each node; as files are created and deleted, the redistribution algorithm reallocates files in such a manner that an equal number of files are kept at all nodes. Only one node at a particular time is in-charge of redistributing the files, and this node is selected by a token passing algorithm. When the quantum for file redistribution t_R is reached at a node, it holds the token and redistributes the files if the distribution is unequal. When redistribution is complete or if the node has not reached its time quantum, it passes the token over to the next node.

The four process allocation policies we compare are:

- NO-PROCESS-MIGRATION (NP): In this strategy only file-copies are allowed to move and all processes are executed at the local node. This strategy does not use the adaptive algorithm, and every file request is forwarded to the host node. It incurs no overhead since load status information not collected, nor exchanged between nodes.

- ADAPTIVE-ALL-NODES (AD-ALL): This strategy uses adaptive load sharing to choose between executing the job at the local node or migrating it to *any* other node. A process can execute at the local, host or foreign node. Load indices are maintained and exchanged, resulting in overhead.

- ADAPTIVE-TWO-NODES (AD-TWO): Instead of choosing among all nodes, the algorithm only considers the local and the host node for process execution. This strategy is interesting because if a process is allocated to some foreign node, the host node still has to send (and receive) the file-copy to (and from) the foreign node, resulting in additional delay. Moreover in AD-TWO the adaptive algorithm is limited to a choice between two rather than between N sites, resulting in a much lower computational overhead.

- RANDOM-TWO (RD-TWO): Here a random selection, with equal probability, is made for process execution between the host node and the local node. Just as with the NP policy, here we do not have system overhead related to the collection of a load index and its transmission to (or reception from) other sites.

In order to illustrate the effect of the file redistribution strategy we have described above, we will also test a NO-FILE-PLACEMENT (NO-FI) strategy, which will operate in conjunction with the best of the above four policies, but *without* file redistribution.

4 Adaptive Algorithm Design

Adaptive algorithms for load sharing, comprise two main activities - *information dissemination* and *decision making (control)* [32]. In order to maintain a complete and consistent view of the entire system, load information about all nodes is maintained and periodically updated at each site.

Load indices considered here are based on total load information and will use the number of processes assigned to the node and their average service time. In addition, the load index will also include all other information units such as messages and file-copies. Let L denote this instantaneous load index for some arbitrary node.

Periodically each t_B time units, the node broadcasts its load to all other nodes, and each node maintains a load vector which is updated whenever information arrives from other nodes. It updates the load of other sites when it receives them, which depends on t_B, and refreshes its own when a new customer starts service. Overhead due to broadcast can be controlled by varying the t_B, and a balance should struck between overhead and performance improvement.

The algorithms we consider make all decisions locally, and are activated upon the creation of a tasks. The transfer policy (which job should be migrated) will depend not only on the load vector but also on the task's characteristics and other system parameters. These characteristics include process execution time and requested file size; relevant system parameters are the process migration and file transfer delays. We summarize below the main properties of the algorithms we consider.

Algorithm Characteristics

Decision making invocation	event driven (process creation, file redistribution)
Transfer policy	global load information, process characteristics, system parameters
Location policy	any node which provides minimum cost
Acceptance policy	single request - no rejection allowed
Information policy	periodic load dissemination

4.1 Algorithm design

Consider a fully connected N node system, where each site maintains a load vector as well as a file-table which indicates the host node for each file.

For any process created at a node, let us introduce the following notation:

- t_E is the "pure" execution or run time of the process.
- t_{FT} is the time it takes to transfer a particular file from the host to the local node, including the file-busy overhead. This quantity depends on the size of the file.

- t_M is the time it takes to move a process to a remote site.
- L_l is the current total load at the local node, expressed as the estimated execution time of all waiting processes, messages, etc..
- L_r is the total load at the remote node, expressed as the estimated execution time of all waiting processes, messages, etc..
- π is the probability of deciding that the process will be executed at the local node, where it was created.

Notice that π is the decision variable; it will be computed by the algorithms we design. In practice, if $0.5 \leq \pi$, the decision will be *not* to move the process. However in certain cases a decision threshold larger than 0.5 may be used.

The algorithms we consider will compute π using an update rule of the form:

$$\pi_{k+1} \Leftarrow \pi_k - \eta \frac{\partial K}{\partial \pi}|_k \tag{1}$$

where k is the index of the update step we are considering, and K is an appropriate cost function obtained from the quantities defined above, and evaluated at each update step. Note that this is a gradient descent rule which is guaranteed to reduce the cost at each step. η is the speed of gradient descent, and the algorithm must be stopped whenever two successive values of the cost function are less than some "level of diminishing returns" ϵ.

Notice that we will *not* invoke the algorithm if the local node is same as host node (Alternative 1 of Section 2). In the other cases (Alternatives 2,3 & 4 of Section 2), the cost K of executing the job on every node is computed using a cost function and the decision is taken using the above iterative procedure for computing π_k.

We will now indicate how a meaningful cost K is chosen. The cost of executing the process locally W_l, or of executing it remotely W_r, are computed at the local node from the load information and job characteristics :

$$W_l = t_{FT} + t_E + L_l \tag{2}$$

$$W_r = \begin{cases} t_M + t_E + L_r & \text{if the remote site is a host node} \\ t_M + t_E + L_r + 2 * t_{FT} & \text{if the remote site is a foreign node} \end{cases} \tag{3}$$

The net cost K is then the average value of the total execution time of the process, under the policy π:

$$K = \pi W_l + (1 - \pi) W_r \tag{4}$$

Clearly :

$$\frac{\partial K}{\partial \pi} = W_l - W_r + \pi(\frac{\partial W_l}{\partial \pi}) + (1-\pi)(\frac{\partial W_r}{\partial \pi}) \tag{5}$$

The last two terms in the above equation represent the effect of the decision π on the workload of the nodes. These work increments can be computed from:

$$W_l(before - decision) - W_l(after - decision) = \pi_{stay}(t_{FT} + t_E + \rho_l) \tag{6}$$

This yields:

$$\frac{\partial W_l}{\partial \pi} = (t_{FT} + t_E + L_l). \tag{7}$$

Similarly we have:

$$\frac{\partial W_r}{\partial \pi} = \begin{cases} t_M + t_E + L_r & \text{if n is a host node,} \\ t_M + t_E + L_r + 2 * t_{FT} & \text{if n is a foreign node.} \end{cases} \tag{8}$$

Substituting (7) and (8) in (5) we get :

$$\frac{\partial K}{\partial \pi} = W_l(1+\pi) - W_r(2-\pi). \tag{9}$$

The algorithm can now be summarized as follows:

- Whenever a decision must be taken, i.e. when a new process is created, first set π_0 and K_0 to the most recently stored values of π and of the cost function at the host node.
- The gradient descent is initiated and at any $k - th$ step of the algorithm, π_k is computed, using (1) and (9). Then the cost K_k is computed based on π_k.
- This is repeated until the point of diminishing returns, i.e. until $|K_k - K_{k-1}| \leq \epsilon$.
- The resulting value of π is used for the decision. Clearly if π is less than 0.5 the process is moved. Otherwise the process is executed locally.

Obviously this does not apply to the NP (no process migration), or the RD (random two) policies, which are both static in that they do not use on-line load information to make decisions.

For the AD-TWO policy, where we choose between the local and host nodes, the simple computation of π suffices to determine the choice of a site to execute the process.

However for the AD-ALL policy, where a choice must be made among all nodes, this gradient iteration will compute the alternative between a local node and *every* other node; thus the computation is $N-1$ times more time consuming than the AD-TWO policy. Let π^n denote the probability that the local node is chosen over some other node n. Then the node n^* with the *smallest* value of π^n will be chosen for remote execution *if* the corresponding probability is less than 0.5; otherwise the execution will be local.

5 Performance metrics and experimental comparison of policies

The four policies discussed in Section 3 are compared in this Section with respect to the following set of pragmatic performance criteria [32].

- *Hit ratio:* This is the ratio of correct decisions to the total number of decisions, in the sense that the effective response time of the execution is smallest with the decision taken.
- *Overhead:* This is the fraction of CPU time consumed by the load balancing algorithm's execution time only.
- *Quality:* This metric measures the distance of some particular load balancing policy X from the NP (no-process-migration) strategy; it is defined by

  $$Quality(X) = \frac{W(NP)-W(X)}{W(NP)},$$

  where $W(.)$ is total process response time. A positive $Quality(X)$ indicates that policy X is worth being pursued, and the larger it is the more useful X will be. However if $Quality(X)$ is negative, policy X will be detrimental to system performance and should not be considered.
- *Percentage Remote Execution:* This measures the percentage of processes executed on remote (foreign and host) nodes.

Earlier studies [10, 31, 32] have supported the view that for an adaptive load balancing algorithm to be of practical interest, the percentage of remote execution should be limited, say of the order of 10%, and the hit ratio should be high, say greater than 90%. Since these studies considered systems which allowed only process migration and not data transfer, we can argue for a higher percentage of remote executions in our case.

Other, more conventional, performance measures which can easily be obtained on the nCube environment will also be considered in this research. They include:

- Average node response R_n; this is the response time obtained by any request for service at a node (including processes and messages),
- Average process response time R_p; this is just the response time, including process migration, experienced by processes.

5.1 Experimental results

The nCUBE provides an ideal test environment for simulating such distributed systems. We have therefore tested all these policies on a 5 node nCube with an artificially generated workload. The time measurements were carried out in "ticks" where one tick is 128 micro-secs. The delay for sending a message from one node to another is of less than one tick.

The AD-TWO and AD-ALL results are based on the simplest possible form of the load-balancing algorithm since **only one step of the gradient descent** is used, requiring the execution of 8 instructions only. A very simple decision rule is used, based *only* on deciding to use the remote node if π is reduced by one step of the iteration.

In this section we present a first set of experimental results for the following workload:

- Each task or process's execution time is uniformly distributed between 9,000 and 12,000 ticks. This corresponds to an average execution time of 1.34 seconds of CPU time. Process transfer takes 6,000 ticks. Each message transfer from one node to another will take up 4 ticks (0.512 milliseconds).
- Each node creates one process, and will create a new process as soon as it is informed that the previous one it created has finished execution.
- File sizes are uniformly distributed between 1K and 4K bytes so that file transfer times are uniformly distributed between 1000 and 4000 ticks. This results in an average file transfer time of 0.32 seconds.
- The load indices are computed as indicated in Section 4. They are broadcast every $t_B = 50,000$ ticks or 6.4 secs.
- Initially, 500 files are allocated to each site; then on an average 20 files are created at each node every 2,500,000 ticks and are deleted at the same rate.

More detailed experimental evaluations are presented in the next sections. Table 1 summarizes measurements which were collected for a total of 1500 process creations at each node.

Strategy	% Remote Exec.	Overhead	R_n	Quality	R_p
NP	not relevant	not relevant	3,838	not relevant	19,355
AD-ALL	34%	0.00082	1,977	0.48	18,568
AD-TWO	30%	0.00079	1,923	0.50	18,082
RD	42%	not relevant	3,352	0.13	19,037

Table 1: Comparative efficiency and performance for four load balancing policies. All times are given in ticks. AD-TWO is the best, followed by AD-ALL. RD is the third and NP the worst.

The *Average Node Response Time*, R_n is an average over *all* requests made to a node. *Quality* expresses the improvement over the NP policy. These are the primary measures of interest. We see that the *Average Process Response Time*, R_p is less sensitive to the policy; this is simply because process migration time is an important component in the AD-ALL and AD-TWO policies. Clearly, load balancing is always beneficial. Note that the overhead presented only includes the algorithm's computations, and is therefore very limited.

The ADAPTIVE-TWO-NODES strategy compares favorably with the ADAPTIVE-ALL-NODES strategy, since the results it yields are slightly better but the overhead is also much less. This supports our claim that simple load balancing policies are better than sophisticated ones, and much better than not having load balancing at all.

For the ADAPTIVE-ALL-NODES policy, we noticed that only a very small fraction (1%) of the processes were transferred to foreign nodes. Thus, the effort overhead of the AD-ALL policy only benefits these 1% of the processes.

This observation validates the assertions made earlier [10], and encourage us to consider restricted load balancing policies. However, the results concerning AD-TWO and its comparison with NP also indicate that load balancing can be effectively used in order to improve user and system performance.

The results of Table 2 illustrate the advantage of file redistribution which we used in all policies. In order to do so, we modify the AD-TWO strategy so that the system does *not* redistribute the files; initially all the nodes have an equal number of files and as time proceeds files at all the nodes are created and destroyed such that nodes have different numbers of files, without redistribution. We call this the NO-FILE-REDISTRIBUTION (NO-FI-RE). The measurements shown below concerning this policy are taken at the nodes having the minimum and the maximum number of files, and the two values are shown. We clearly see the superiority of policies which do redistribute files, over one which does not adjust file distribution.

Strategy	% Remote Exec.	Hit Ratio	Overhead	R_n	Quality
AD-TWO	30%	0.97	0.00070	1,923	0.50
NO-FI-RE	40%-29%	1.0 0-0.97	0.00070	2,100	0.33

Table 2: Comparative efficiency and performance with and without file redistribution.

On Figures 1 and 2 we present measures over time (in milliseconds) of the total load at a node. On Figure 1 we compare the system running under AD-TWO with the system running under the NP policy. On Figure 2 we compare the system running with the AD-TWO policy, and the system running with the RD policy. The parameters are the same as those for the experiments described above. Very clearly, the AD-TWO policy is very effective in reducing total node at each node.

The results of Figures 1 and 2 are not altogether intuitive. Indeed, Figure 1 indicates that the AD-TWO policy results in lower load on nodes than no load balancing (NP). Yet one would expect that load balancing would not affect average node, and only reduce differences between heavily loaded and lightly loaded nodes. What is being observed however is an overall improvement of performance, under the effect of load balancing. The same can be said about Figure 2.

On Figure 3 we compare the four policies considered when the average execution time C of a process is varied. Process execution time is uniformly distributed as follows: it is between [C-200, C+200] if $200 \leq C$; for smaller values of C it is uniformly distributed between 0 and 2C.

The workload is being changed by varying the average run time of each process, and each point on the curve corresponds to the average process response time measured for 1500 process executions. The measurements are taken under the following load conditions: each PU generates a new process *as soon as* the previous process it has created has completed execution, and the PU has received information about this. Thus there is exactly one active process running for each PU in the system. We see that when the average process execution time is small, the four policies are relatively equivalent, though AD-TWO still remains the best. However when the average process execution time is large, there is a great benefit in choosing the best load balancing policy.

5.2 Response Time Comparisons with Different Load Values

In the experiments described above, recall that each site is allowed to have a single active process, and a new process is created only when the previous one is completed. This workload does not allow us to vary the load on the system, except by modifying the average execution time of processes.

In this section we present results of experiments which we have conducted by increasing the number of processes generated at each node. In order to do so, each node is allowed to generate synthetic processes, on the average each T milli-seconds or at a rate of $\lambda = 1/\mathrm{T}$ processes per unit time. A process will execute on the average for C time units, and we will also present measurements obtained by varying C. Of course, all these times are specified at the system level in "ticks". All other parameters are the same as those indicated in the previous subsection.

In all cases reported here, for each value of λ, response time and overhead measurements are collected for 1000 process executions.

Measurements are collected for the process response time R_p for all four policies considered: NP, AD-TWO, AD-ALL and RD. Some results are reported on Figure 4. Here the process generation rate is 55 processes/sec, or 7.04 processes per 1000 ticks, at each node. For small values of C under 200 ticks (which is of the order of 25 milli-seconds) AD-ALL, AD-TWO and RD seem to provide equivalent performance. For larger values of C, AD-TWO provides superior performance.

We have measured the overhead for AD-TWO and AD-ALL and report it on Figure 5. This is the fraction of node processing time devoted to the control policy including load data collection and broadcasting, and carrying out the gradient algorithm. We see that overhead remains acceptably low, at less than 5%.

On Figure 6 we report measurements where we keep C constant at 1050 ticks or 0.1344 seconds, with uniform distribution between 900 and 1200 ticks. We vary λ, the process creation rate. The average process response time as a function of average process creation rate is plotted for the four policies. For very low λ, AD-ALL, AD-TWO and RD provide equivalent performance. For medium to high λ AD-TWO is the best, very closely followed by AD-ALL. Both are much better than RD and NP.

Finally on Figure 7 we present measurements of the fraction of node overhead as a function of load, in number of processes created per unit time at the nodes. This comparison is restricted to the two adaptive policies, even though the random RD policy would also generate some overhead. Clearly AD-TWO results in less overhead, but for both AD-TWO and AD-ALL overhead is relatively low (below 5%) even at relatively high loads.

6 Conclusions

In this paper we have presented a dynamic load balancing principle based on on-line gradient descent for distributed system architectures composed of an arbitrary number of processing units, interconnected by a high-speed network. The purpose is to substantially improve system performance using adaptive decisions concerning the choice of processing units where processes are executed.

We have implemented and tested two gradient-based load balancing algorithms on an nCube target architecture, and compared them with the case where no load balancing is carried out, as well as with "random" load balancing. We demonstrate via extensive measurements that simple adaptive load balancing can substantially improve distributed system performance, over that of systems with no load balancing. A 50% reduction in average process response time is observed, compared to no load balancing.

We also indicate experimentally that a simple algorithm can achieve better performance than more sophisticated algorithms, and that it obviously result in less overhead.

References

[1] P. Chen, "Optimal file allocation in Multi-level Storage Systems," *Proc. AFIPS National Computer Conference*, vol. 42, pp. 277–282, 1973.

[2] L. Dowdy and D. Foster, "Comparative Models of the File Assignment Problem," *ACM Computing Serveys*, vol. 14, pp. 287–313, June 1982.

[3] L. Laning and M. Leonard, "File allocation in a Distributed Computer and Communication Network," *IEEE Trans. Computers*, vol. C-32, pp. 232–244, 1983.

[4] A. Thomasian and P. Bay, "Analytical queueing network models for parallel processing of task systems," *IEEE Transactions on Computers*, vol. C-35, pp. 1045–1056, Dec 1986.

[5] D. Menasce and L. Barroso, "A methodology for performance evaluation of parallel applications on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 14, pp. 1–14, 1992.

[6] A. Tantawi and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *Journal ACM*, vol. 32, pp. 445–465, April 1985.

[7] E. Silva and M. Gerla, "Queueing Network Models for Load Balancing in Distributed Systems," *Journal of Parallel and Distributed Computing*, vol. 12, pp. 24–38, 1991.

[8] M. Livny and M. Melman, "Load balancing in homogeneous broadcast distributed systems," in *Proc. ACM Comput. Network Performance Symp.*, pp. 47–55, 1982.

[9] T. Liu, "Dynamic load balancing algorithm in homogeneous distributed systems," in *Proc. of the Sixth Int. Conf. on Distributed Computing Systems*, pp. 216–222, May 1986.

[10] D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transaction on Software Engineering*, vol. 12, pp. 662–676, 1986.

[11] R. Mirchandaney, D. Towsley, and J. Stankovic, "Adaptive Load Sharing in Hetrogeneous Systems," in *Proc. of the Ninth International Conference on Distributed Computing Systems*, (Newport Beach, California), pp. 298–306, June 1989.

[12] H.-C. Lin, G.-M. Chiu, and C. Raghavendra, "Performance Study of Dynamic Load Balancing Policies for Distributed Systems with Service Interruptions," in *IEEE INFOCOM'91 Proceedings*, pp. 797–805, 1991.

[13] L. Ni, C. Xu, and T. Gendreau, "A Distributed drafting algorithm for Load Balancing," *IEEE Transaction Software Engineering*, vol. SE-11, pp. 1153–1161, October 1985.

[14] Y. Wang and R. Morris, "Load sharing in Distributed Systems," *IEEE Transactions on Computers*, vol. C-34, pp. 204–217, March 1985.

[15] A. Agrawala, S. Tripathi, and G. Ricart, "Adaptive routing using a virtual waiting time technique," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 76–81, 1982.

[16] C. Brown and M. Schwartz, "Adaptive routing in central computer communication networks," in *Porc. IEEE Int. Comp. Commum.*, pp. 12–16, June 1979.

[17] Y. Chow and W. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Transaction Computers*, vol. 28, pp. 354–361, 1979.

[18] L. Ni, "A Distributed Load Balancing Algorithm for Point to Point Computer Networks," in *Proc. of IEEE COMPCON*, pp. 116–123, 1982.

[19] T. Chou and J. Abraham, "Load balancing in distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-8, July 1982.

[20] G. Rao, H. Stone, and T. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. Comput.*, vol. C-28, pp. 291–298, April 1979.

[21] H. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 85–94, Jan 1977.

[22] A. Leff and P. Yu, "An adaptive startegy for load sharing in distributed database environments with information lags," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 91–103, 1991.

[23] C. Gao, J. Liu, and M. Railey, "Load balancing algorithms in homogeneous distributed systems," in *Proc. 1984 International Conference on Parallel Processing*, (Silver Spring, MD), pp. 302–306, IEEE Computer Society, 1984.

[24] K. Lee and D. Towsley, "A comparison of priority-based decentralized load balancing policies," in *Proc. Performance '86 and 1986 ACM SIGMETRICS conf*, pp. 70–77, 1986.

[25] T. Yaun and H. Lin, "Adaptive load balancing for parallel queues," in *Proc IEEE International Conf. on Communications*, (Amsterdam), 1984.

[26] L. Ni and K. Hwang, "Optimal load balancing in a multiple processor system with many job classes," *IEEE Transaction on Software Engineering*, vol. 11, pp. 491–496, 1985.

[27] A. Avritzer et al, "The advantage of dynamic tuning in distributed asymmetric systems," in *Proc. Infocom '90*, 1990.

[28] R. Gallager, "A mimimum delay routing algorithm using distribted computation," *IEEE Transaction on Comm*, vol. COM-25, pp. 73–85, 1977.

[29] R. Bryant and R. Finkel, "A stable distributed scheduling algorithm," in *Proc. 2nd Int. Conf. Distributed Computer Systems*, pp. 314–323, 1981.

[30] P. Kruger and R. Finkel, "An adaptive load balancing algorithm for a multicomputer," Tech. Rep. 539, Dept. Computer Science, University of Wisconsin, Madison, April 1984.

[31] S. Zhou, "A Trace-Driven Simulations Study of Dynamic Load Balancing," *IEEE Transactions on Software Engineering*, vol. 14, No. 9, pp. 1327–1341, Sept 1988.

[32] O. Kremin and J. Kramer, "Methodical Analysis of Adaptive Load Sharing Algorithms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 3, pp. 747–760, November 1993.

[33] A. Goscinski, *Distributed Operating Systems, Logical Design*. Addison Wesley, 1991.

Figure 1: **Comparison of the instantaneous load on a node for the AD-TWO and NP policies over a long period of time. The AD-TWO policy provides uniformly better performance.**

Figure 2: **Comparison of the instantaneous load on a node for the AD-TWO and RD policies over a long period of time. The sporadic effect of the RD policy, and the improvement obtained with AD-TWO are clearly apparent.**

Figure 3: **Average process response time as a function of average process execution time C for the four policies. For medium to high average process execution times, AD-TWO provides superior performance.**

Figure 4: **Average process response time as a function of average process execution time C for the four policies. Here the process generation rate is 55 processes/sec, or 7.04 processes per 1000 ticks, at each node. For low C, AD-ALL, AD-TWO and RD seem to provide equivalent performance. But for medium to high C, AD-TWO provides superior performance.**

Figure 5: **Overhead as a function of average process execution time C for two policies AD-TWO and AD-ALL. The process generation rate is 55 processes/sec, or 7.04 processes per 1000 ticks, at each node. Overhead decreases as C increases. As expected AD-ALL has systematically higher overhead than AD-TWO. Overhead is under 5% (0.05) in all cases.**

Figure 6: **Average process response time as a function of average process creation rate for the four policies. The average process execution time C is constant at 1050 ticks or 0.1344 seconds. For varying process creation rates, AD-ALL, AD-TWO and RD provide equivalent performance. For medium to high rates, AD-TWO is the best, and is closely followed by AD-ALL. Both are much better than RD and NP.**

Figure 7: **Overhead as a function of the process creation rate at each node, for the two policies adaptive policies AD-TWO and AD-ALL. The average process execution time C is constant at 1050 ticks or 0.1344 seconds. As expected AD-ALL has systematically higher overhead than AD-TWO. Overhead is under 5% (0.05) in all cases.**