Information Collection Policies: Towards load balancing of communication-intensive parallel applications

Javier Bustos\textsuperscript{1,2}, Denis Caromel\textsuperscript{2}, and Jose M. Piquer\textsuperscript{1}

\textsuperscript{1} Computer Science Department, University of Chile. Blanco Encalada 2120, Santiago, Chile. \{jbustos,jpiquer\}@dcc.uchile.cl
\textsuperscript{2} INRIA Sophia-Antipolis, CNRS, I3S, UNSA. 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France. First.Last@sophia.inria.fr

Abstract. In communication-intensive applications, the information collection policy influences the load balancing decision process and the communication itself, because the collection reduces the bandwidth used by the application. In this paper, we present a study of collection policies used by well-known load balancing system, now in the context of communication-intensive applications and focusing in response times (time of reaction against instabilities) and bandwidth. We divided the policies in server and broadcast oriented, which can share all of partial load information, implementing them with an asynchronous communicated middleware. Our experimental results show a better performance (in response time and bandwidth) of broadcast oriented policies sharing partial information.

Keywords: Dynamic load balancing, Communication-intensive parallel applications, Collection policies.

1 Introduction

Load balance is the process of distributing tasks of a parallel application on a set of processors, improving the performance and reducing the application response time. For this process, the decisions of when, where and what tasks has to be transferred are critical, and the load information used for that decision has to be accurate and updated [17]. In dynamic load balance, those decisions depend on the load information collected from the system. This information is shared among processors periodically or “on demand”, using centralized or distributed information collectors [20]. On communication-intensive applications (parallel applications which transfer a large amount of data among processors), the information collection policy influences the load balancing decision process and the communication itself, because the collection reduces the bandwidth used by the application.

The performance of load balancing algorithms for non-intensive communication applications has been studied in depth [21, 20, 7, 19], focusing on stability (ability of balancing the work only if that action improves the performance of the system) and response time (ability of reacting against instabilities). Casavant and Kuhl show in [7] that a faster response rate is more important than the stability to improve the performance.
This paper describes experiments which measure the response time and bandwidth of collection policies used by well-known load balancing algorithms. Those policies are studied here within the communication-intensive applications context and they are defined as:

1. **Load Server**: In this policy, the nodes share all of their load information, using a central server to collect and process it. Figure 1.a presents an example using three nodes: nodes A and C sends its load information (L) to the server (B) every certain period of time, this server collects that information and maintains the system balanced (on the figure, ordering A to balance with C). This kind of policy is widely used in practice on systems like Condor [15, 12] and middlewares like Legion [8]. Theoretical and practical studies report this kind of policy as not scalable [20, 7, 1, 14].

2. **Load Change Server**: In this policy, there are partial information sharing among the nodes. Figure 1.b presents an example using three nodes which share information only when they are overloaded. Each node (A) registers on the server (B) when it enters to an “overloaded state” (the load metric is over a given threshold), and node (C) unregisters from the server because it leaves that state. At the same time (C) asks to the server for overloaded nodes, the server choose one node from its registration table and starts the load balancing between them.

3. **Load Broadcast**: This policy is based in the same principle of Load Server, but it use broadcast to share the information. Figure 1.c shows an example using three nodes: Each node (A,B,C) broadcasts its load to the others every certain period of time, those nodes use the information for load balance [18]. Then, (A) and (C) note that they can share with (B) and send the balance message (S). That figure also shows the main problem of this policy: there is no control on the number of balance messages an overloaded node might receive. For our response time measurements, we consider only the first balance message (in the figure: the message from (A)).

4. **Load Change Broadcast**: This policy is based in the same principle of Load Change Server, using broadcast to share their information. Figure 1.d presents an example for the overloaded case: each node (B) broadcasts its load only when its state changes to overloaded, requesting a load balance. Using this information, (A) and (C) reply the request (S) but, like in the previous policy, only the reply of (A) is considered. In practice, this policy is used on the “Robin Hood” algorithm [6] developed for ProActive [3].

We study the given policies on the middleware ProActive [3]. ProActive provides a strong infrastructure for deployment, communication and migration of active objects [10]. Using active objects, *communication-intensive parallel applications can be modeled and developed* [13, 4].

This paper is organized as follows: Section 2 presents the load model and the simulation of the policies using ProActive. Section 3 summarizes the main results of this study. Section 4 shows the conclusions and discusses future work.
2 Model Overview and Definitions

This section provides a brief overview of the model used for the analysis of load balancing algorithms, and its main definitions.

In this paper, each node represents a machine (virtual or real) which participates in the balance. As in [20], we compare centralized and distributed algorithms, also we add partial information algorithms in our experiments.

In ProActive, there is no notion of tasks like parallel batch systems [15, 22], in this paper we use the term task to refer to a service [3], and the term job for a set of services served by one active object.

In the literature, the word load represents an index such as the CPU queue length, the available memory, a linear combination of both, etc. In this paper, load represents the number of tasks on the CPU queue modeled with ProActive (see section 2.2).

In our study, response time is the time between a node enters on the overloaded state (his load index is over a threshold) and the beginning of load balancing.

2.1 Load Model

Following the recommendations of [5, 7], we simulate the load of each node with a birth and death process with incoming rate $\lambda$ and outcome rate $\mu$. The value of $\lambda$ represents the number of jobs which arrive every second to a node. The jobs size (in number of tasks) follows an exponential distribution with mean 1. The value of $\mu$ represents the number of tasks served by a node every second. In our experiments we use $\lambda = 1, 2, \ldots, 10$ and, in order to maintain the system stable, $\mu = 10$.

In our first approaches, we randomly calculated the job size following the exponential distribution, and we obtained similar results on response time and bandwidth among them. So, in order to reduce the randomness in our measurements and because ours experiments have to be comparable for all policies and number of nodes, we calculate the total number of incoming tasks every second (along a period of 60 seconds) for each value of $\lambda$. Those values will not change during the experimentation.

In our experiments, the nodes are numbered 0, ..., $n$ and the value of $\lambda$ assigned to the node $i$ is $\lambda_i = 1 + i \mod 10$. Each node will use the incoming rate calculated
previously for its $\lambda_i$ and after 60 seconds the simulation starts again with the first value of $\lambda_i$.

Several studies have shown that on a set of workstations (without load balancing), more than 80% of them are idle during the day [14, 15, 20]. The concept of not-idle workstations and overloaded nodes are similar: processors which want to share work. So, in our study, if no load balance is made, 20% of the nodes have to reach the overloaded state. To achieve this, using the previously calculated values for $\lambda$, we use the convention:

- Underloaded Node: $\text{load} < 10$.
- Normal Node: $10 \leq \text{load} < 15$.
- Overloaded Node: $\text{load} \geq 15$.

### 2.2 Implementing the Collection Policies

Each node is modeled as an active object with three main methods:

- register: Used by register on the communication channel (server, multicast) This method starts the clock on our experiments.
- loadBalance: This method start the load balancing process, and we use it to stop the clock on our experiments and to calculate the response time.
- addLoad(int): This method simulates the load balance. On local execution updates the local load, on remote execution updates the remote load.

**Load Server:** For this policy there is an active object as a central server who collects the load information and stores that information on one of its tables: underloaded, normal or overloaded. The protocol is:

- Every second, the nodes calls the remote execution of register on the server.
- The load server processes the method call, if the information is from an overloaded node, it randomly chooses the address of an underloaded node (if any) and call the method loadBalance on the overloaded node with this address.
- The overloaded node calls the local execution of addLoad(-myLoad/2) [5] and on the underloaded node calls the remote execution of addLoad(myLoad/2).

**Load Change Server I:** We study this policy looking for a reduction of the information transmitted over the network, also we add an unregister method to the node model. If a node reaches the overloaded state, then it registers on the central server, and if a node leaves that state, it unregisters (remove its reference) from the server.

Every second, if a node is in underloaded state then it asks to the server for overloaded ones, when the server receives that query it randomly chooses the address of an overloaded node (if any) from its tables and start the load balancing: ordering to the overloaded node to balance with the underloaded one who sends the query.
**Load Change Server II:** This policy is similar at the previous one, but in this case the underloaded nodes share their information: the node registers on the server when it reach the underloaded state and unregisters when it leaves that state.

Every second, if a node is in overloaded state it asks to the server for underloaded ones, when the server receives that query it randomly chooses the address of an underloaded node (if any) from its tables ans start the load balancing: ordering to the overloaded node (who sends the query) to balance with the underloaded one.

**Load Broadcast** For this study, we use a multicast channel instead of broadcast, for a better control of the bandwidth measurement and for a good differentiation among the network and policy messages.

The protocol is similar to Load Server, but instead of sending the information to a central server, the nodes broadcast their information. All the nodes are also a server, and all the decisions are local using the information collected from the communication channel.

We expect this policy has similar response time and uses less bandwidth than server oriented policies, because the former uses UDP packets for its coordination and the others use RMI (which uses TCP).

**Load Change Broadcast I** This policy have similar protocol than Load Change Server I, but in this case it uses the multicast channel instead a central server. Like Load Broadcast, every node is also a server and the decisions are local. We expect this policy have similar time delay and uses less bandwidth than the Load Broadcast policy, because the former sends less messages than the later.

**Load Change Broadcast II** This policy is the broadcast version of Load Change Server II, and we expect similar behavior than the Load Change Broadcast I policy.

### 2.3 Hardware and Software

We simulate this model over the Oasis Team Intranet [2]. We tested the policies on a heterogeneous network: 3 Pentium II 0.4 Ghz, 10 Pentium III 0.5 - 1.0 Ghz, 3 Pentium IV 3.4GHz and 4 Pentium XEON 2.0GHz for the nodes and a Pentium IV 3.4GHz for the server. We homogeneously distribute the nodes (active objects) on the processors and for the measurement process we use the system clock (response time) and the software Ethereal [9] (bandwidth).

The nodes and the server were developed using ProActive on Java 2 Platform (Standard Edition) version 1.4.2.
3 Results Analysis

On our preliminary tests, we detect a performance degradation of active objects when we use more than 20 of them at the same machine. To avoid this degradation, we tested the policies on 20, 40, 80, 160, 320 nodes distributed on the 20 machines. For each case we take 1000 samples of response time and the bandwidth report of Ethereal. We present the main results of this study: first we show a response time study, then a bandwidth analysis.

3.1 Response Time

Figure 2 shows response time for all the policies. Following the recommendations of [17], response time has to be less than the update period, and in this study that value is 1000 ms. Using this reference, broadcast oriented policies presents better response time than server oriented policies, because in the former the nodes store the information and in case of load balancing they use it, and in the later the nodes have to ask to the server for that information. Also, policies who send all the information have better performance than policies who share overloaded information (Type I policies), because in the former overloaded nodes generates the request for load balancing and in the later overloaded nodes have to wait until an underloaded node contacts them: underloaded nodes choose a balance candidate randomly. So, the greater number of nodes, the less probability of each one to be selected, then the higher response time.

![Fig. 2. Mean response time for all policies](image-url)
3.2 Bandwidth

In this section we will test the bandwidth used by the information collection policies. While this introduces a comparison between TCP and UDP based communications (resp. centralized and non-centralized policies), our goal is to compare performance between total and partial information policies, developed on centralized and non-centralized load balancing algorithms, it makes this comparison valid.

Simulation results  Figure 3 shows the bandwidth used on the collection phase (only messages to the server):

1. server oriented policies use between 5 (Load Change Server II) and 40 times (Load Server) more than broadcast oriented policies. This phenomenon is produced by the different type of network protocol used by each one, and is well studied in [16].

2. for partial information - server oriented policies: if overloaded nodes shares its information, then less than 20% of the total nodes (see section 2.1 will send register/unregister messages (when they enter/leave the state) and more than 80% of them will send queries for registered nodes (every second). But, if the underloaded nodes share their information, more than 80% of the total nodes will send register/unregister messages and less than 20% of them will send queries for registered nodes. This behavior produces that the former requires more bandwidth than the latter.

If we consider the total bandwidth used by this model, including the loadBalance and addLoad messages, then the figure 3 (right) shows those results:

1. policies which share partial information of underloaded nodes have the best bandwidth utilization for each case (server and broadcast oriented).

2. policies which share partial information of overloaded nodes generate a great increase on the bandwidth utilization, because there are no control of how many underloaded nodes have to send the loadBalance message. In the policy Load Change Server I, this behavior generates a saturation: the server has to process about half the number of messages than Load Server policy, but most of them are balance queries; so, it has to choose an address from its tables and send to that overloaded node the loadBalance message. If the central server is saturated (over 300 nodes on our experiments), then the response time will increase and the bandwidth used will decrease, because the saturation will produce less messages over the network. Using a multithreaded central server will rise the point of saturation, but it is not a permanent solution, because it will generates new constraints like mutual exclusion.

3.3 Testing a real application

We tested the impact of the policies over a real application: the calculus of a Jacobi matrix. This algorithm performs an iterative computation on a square matrix of real numbers. On each iteration, the value of each point is computed using its value and
The value of its neighbors for their last iteration. We divided a 3600x3600 matrix in 25 workers (developed using ProActive) all equivalents, and each worker communicates with its direct neighbors. All the workers are randomly distributed among 13 (of 15) machines, using at most 2 workers by machine. Using this distribution, we measured the execution time of 1000 sequential calculus of Jacobi matrices (see first row on table 1).

To determine the impact of the policies over the Jacobi application, we distributed 30 nodes among the 15 machines and we ran the simulation of the previous section (placing load servers out of the simulation machines), measuring again the execution time of Jacobi. Separately, we measured the CPU cost (in % of busy time) for each policy, over the 15 machines. All those results are on table 1.

Table 1. Information Collection Policies and its effects on execution time of a parallel Jacobi application

<table>
<thead>
<tr>
<th>Policy</th>
<th>Execution Time (sec)</th>
<th>% policy cost (time)</th>
<th>% policy cost (CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>914.361</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Load Server</td>
<td>1014.960</td>
<td>11.00%</td>
<td>1.3%</td>
</tr>
<tr>
<td>Load Change Server I</td>
<td>995.873</td>
<td>8.91%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Load Change Server II</td>
<td>972.621</td>
<td>6.37%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Load Broadcast</td>
<td>1004.800</td>
<td>9.89%</td>
<td>10.7%</td>
</tr>
<tr>
<td>Load Change Broadcast I</td>
<td>925.964</td>
<td>1.26%</td>
<td>4.5%</td>
</tr>
<tr>
<td>Load Change Broadcast II</td>
<td>915.085</td>
<td>0.08%</td>
<td>4.1%</td>
</tr>
</tbody>
</table>

While centralized policies use less CPU on the “client” side, they use more bandwidth that their equivalents distributed policies. In this kind of applications, a great bandwidth introduces greater delays on the parallel application than CPU sharing. A special case is Load Broadcast policy, which use less bandwidth of centralized policies...
but the largest CPU time, and it produces almost 10% of time delay on the application. So, if this policy is used, the load balancing itself would produce overloading.

4 Conclusions and Future Work

In this study we presented a comparison between six communication policies used for load balancing. We focused on two metrics: bandwidth of the communication and response time.

We conclude that broadcast oriented policies have the best performance of information collection using these metrics, and sharing the information of underloaded nodes is the best decision in those kind of policies. We suggest, towards a load balancing architecture for communication-intensive parallel applications developed with asynchronous communicated middlewares, to use this kind of policy: overloaded nodes start the balance with information previously acquired, avoiding the utilization of central servers and hierarchical architectures.

Moreover, if the load index could be updated on larger periods of time (greater than 1 second) with similar accuracy, then the policy will produce less interference over parallel applications.

It is the continued goal of this work to optimize the process of selecting the best candidate to balance and where to do it, looking for the best performance in bandwidth and response time.

References