Load Information Sharing Policies in Communication-Intensive Parallel Applications

Javier Bustos[1,3], Denis Caromel[2], Mario Leyton[2], and José M. Piquer[1]

[1] Departamento de Ciencias de la Computación, Universidad de Chile. Blanco Encalada 2120, Santiago, Chile.
{jbustos,jpiquer}@dcc.uchile.cl
[2] INRIA Sophia-Antipolis, CNRS-I3S, UNSA. 2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.
First.Last@sophia.inria.fr
[3] Escuela de Ingeniería Informática. Universidad Diego Portales. Av. Ejercito 441, Santiago, Chile.

Abstract. In this paper, we present a study of information sharing policies used by well-known load balancing systems. Our approach comes from analyzing the performance scalability of: *response time* (time of reaction against instabilities) and *bandwidth*, from a communication-intensive application context. We divided the policies into: *Centralized* or *Distributed* oriented; and *Eager* or *Lazy* load information sharing. We implement them with an asynchronous communication middleware called ProActive. Our experimental results show that *Eager Distributed* oriented policies have better performance (response time and bandwidth usage).
Keywords: Dynamic load balancing, Communication-intensive parallel applications, Load information sharing policies, Load information collection.

1 Introduction

Load-balancing is the process of distributing parallel application tasks on a set of processors while improving the performance and reducing the application response time. The decisions of *when*, *where* and *which* tasks have to be transferred are critical, and therefore the load information has to be accurate and up to date [18]. In *dynamic load balance*, decisions depend on the information collected from the system. Load information can be shared among processors periodically or "on demand", using *Centralized* or *Distributed* information collectors [21]. When dealing with communication-intensive applications (parallel applications which transfer a large amount of data among processors), the information sharing policy influences not only the load-balancing decisions but also the communication itself. We studied this problem, because our results can be applied in the context of load-balancing on peer-to-peer networks[7].

The load-balancing algorithms performance, for non intensive communication applications, has been studied in depth [22, 21, 8, 20] focusing on *stability* (ability of only balancing the work if that action improves the performance of the system) and *response time* (ability of reacting against instabilities). Casavant and Kuhl [8] show that a faster

response time is more important than stability to improve the performance of load-balancing algorithms. A survey on this topic can be found in [12].

This paper describes experiments which measure the response time and bandwidth usage for different information sharing policies applied by well-known load-balancing algorithms. These policies are studied in a communication-intensive context and are defined as follows:

1. **Centralized Full Information**: Nodes share all their load information with a central server. Figure 1.a presents an example with three nodes: nodes A and C send their load information L to the server B periodically. The server collects that information and keeps the system balanced (in the figure, ordering A to balance with C). This policy is widely used on systems like Condor [16, 13] and middlewares like Legion [9]. Theoretical and practical studies report this policy as non scalable [21, 8, 1, 15].
2. **Centralized Partial Information** There is partial information sharing among the nodes through central server. Figure 1.b presents an example using three nodes which share information only when they are overloaded. A node A registers on the server B when it enters an "overloaded state" (that is, the "load metric" is above a given threshold), and node C unregisters from the server because it exits the "overloaded state". At the same time C asks the server for overloaded nodes, the server chooses one node from its registration table and starts the load-balancing between them.
3. **Distributed Full Information** Nodes share all their information using broadcast. Figure 1.c shows an example using three nodes: Each node broadcasts its load to the others periodically. The nodes use the information for load balancing [19]. Then, A and C realize they can share B's load and send the balance message S. The 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 considered only the first balance message (in the figure: the message from A).
4. **Distributed Partial Information** There is partial information sharing among the nodes using broadcast. Figure 1.d presents an example for the *overloaded* case: a node B broadcasts its load only when changing to the overloaded state, requesting a load balance. Using this information, A and C reply to the request S, but unlike in the previous policy, only the reply from A is considered. In practice, this policy is used in the "Robin Hood" algorithm [6] developed for ProActive [3].

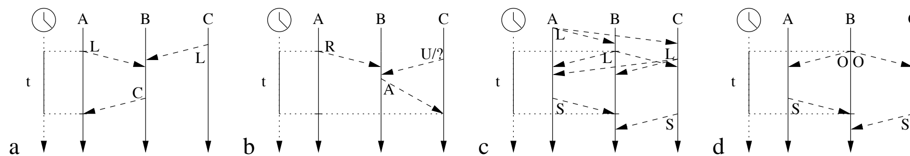

Fig. 1. a) Centralized Full Info. b) Centralized Partial Info. c) Distributed Full Info. d) Distributed Partial Info.

We studied the given policies using the middleware **ProActive** [3]. ProActive provides a strong deployment infrastructure, communication and active object migration [11]. Using active objects, communication-intensive parallel applications can be modelled and developed [14, 4].

This paper is organized as follows: Section 2 presents the load models and the policies simulated with 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 the main definitions and a brief overview of the load-balancing algorithms and information sharing policies used in our analysis.

In this paper, each **node** represents a machine (virtual or real) which participates in the balancing. As in [21], we compare centralized and distributed algorithms, adding also partial-information algorithms in our experiments. In ProActive, there is no notion of **tasks** like in parallel batch systems [16, 23]. 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 an active object. In the literature, the word **load** represents a metric such as the CPU queue length, the available memory, a linear combination of both, etc. In this paper, **load** represents the number of tasks in the CPU queue modelled with ProActive (see section 2.2). In our study, **response time** is the time since a node entering the *overloaded* state and the beginning of the load-balancing.

2.1 Load Model

Following the recommendations of [5, 8], we simulate the load of each node with a discrete-time population process with birth-rate λ and death-rate μ. The value of λ represents the number of jobs which arrive every second to a node. The job size (in terms of number of tasks) follows an exponential distribution with mean 1. The death-rate μ represents the number of tasks served by a single node per second. In our experiments we use $\lambda = 1, 2, ..., 10$, and in order to maintain the system stable: $\mu = 10$. Note that this methodology simulates the load balance process and its communications. Simulation data will conclude whether the policies hinder intensive-communicated parallel applications.

Because our experiments have to be comparable for all policies and number of nodes, we calculated the total number of incoming tasks every second (along a period of 60 seconds) for each value of λ. These precomputed values were used for all the experiments.

In our experiments, the nodes are labelled $0, ..., n$ and the value of λ assigned to the node i is $\lambda_i = 1 + i \mod 10$. Each node used the initial precomputed incoming rate λ_i, and after 60 seconds, the simulation was restarted again with the value of λ_i.

Several studies have shown that on a set of workstations (without load balancing), more than 80% of the workstations are idle during the day [15, 16, 21]. The concept of *occupied* workstations and *overloaded* nodes are similar: processors which want to share work. Therefore, in our study, if no load balance was made, 20% of the nodes had

to reach the overloaded state. To achieve this with the previously calculated values for λ, we used the convention:

– Underloaded Node: load < 10.
– Normal Node: $10 \leq$ load < 15.
– Overloaded Node: load ≥ 15.

2.2 Implementing the Information Sharing Policies

Since the information-sharing policies defined in section 1 can be *full* or *partial*, when unspecified we will be referring to *full* information sharing policies. In *full* information sharing policies, load information from overloaded and underloaded nodes is shared.

On the other hand, we will classify *partial* information policies into two groups: *eager* or *lazy*. *Eager* policies correspond to the ones where an *overloaded* node triggers the load-balancing, and therefore the partially shared information corresponds to the underloaded nodes. *Lazy* policies correspond to the ones where the *underloaded* node triggers the load-balancing, and therefore the partially shared information corresponds to the overloaded nodes.

Each node is modelled as an *active object* with three principal operations:

– `register`: registers on the communication channel (server, broadcast). This method starts the clock in our experiments.
– `loadBalance`: starts the load-balancing process, to stop the clock in our experiments, and to calculate the response time.
– `addLoad(x)`: adds x tasks to the callee.

Centralized For this policy, one active object was chosen as a central server which collected and stored load-balance information of each node as: underloaded, normal or overloaded. The policy works as follows:

– Every second, the nodes call the remote `register` execution on the server.
– The *load server* processes incoming method calls. If the call originates from an overloaded node, the server randomly chooses an address of an underloaded node (if any) and calls the method `loadBalance` on the overloaded node with the chosen address.
– The overloaded node performs locally `addLoad(-myLoad/2)` (according to the recommendations of Berenbrink, Friedetzkyand Goldberg [5]) and the underloaded node (remotely) performs `addLoad(myLoad/2)`.

Lazy Centralized We studied this policy looking for a reduction of the information transmitted over the network. For this, we included an `unregister` method to the node model. This policy is described as follows:

– When a node reaches the overloaded state, it registers on the central server, and
– When a node leaves the overloaded state, it unregisters (removes its reference) from the server.

– Every second, if a node is underloaded it asks the server for overloaded nodes. When the server receives that query, it randomly chooses the address of an overloaded node (if any), and starts the load-balancing: ordering the overloaded node to balance with the node that originated the query.

Eager Centralized This policy is similar to the previous one, but underloaded nodes share their information instead of overloaded ones. The nodes register on the server when they reach the underloaded state and unregister when leaving it:

– When a node is in overloaded state, it asks the server for underloaded nodes once per second.
– Upon receiving the query, the server randomly chooses the address of an underloaded node (if any) and begins the load-balancing by ordering the overloaded node that sent the query to balance with the chosen underloaded node.

Distributed The policy is similar to *Centralized*, but instead of sending the information to a central server, nodes broadcast their information. Therefore, all the nodes are servers, and each node make's its own balance decisions (i.e.: local decisions), using information collected from the communication channel.

Lazy Distributed This policy is similar to *Lazy Centralized*, but in this case the information is shared through the multicast channel instead of a central server. Like *Distributed* policy, every node is also a server and the decisions are local. We expected this policy to have similar time delay but use less bandwidth than the *Distributed* policy due to the reduction in the number of sent messages.

Eager Distributed This policy is the broadcast version of *Eager Centralized*, and we expected a behavior similar to the *Lazy Distributed* policy.

2.3 Hardware and Software

We simulated the models using the Oasis Team Intranet [2]. We tested the policies on a heterogeneous network composed of: 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 uniformly at random distribute the nodes (active objects) on the processors. For *response time* measurements we used the system clock, and for bandwidth measurements we used Ethereal [10] software. The policy methods for nodes and servers were developed using the *ProActive* middleware on Java 2 Platform (Standard Edition) version 1.4.2.

3 Results Analysis

We tested the policies on $20, 40, 80, 160, 320$ nodes distributed on 20 machines. For each case we took 1000 samples of response times and the bandwidth reports from *Ethereal*. In this section we present the main results of this study. We will first discuss the *response time*, and then the *bandwidth* analysis.

3.1 Response Time

Figure 2 shows *response time* for all the policies. Following the recommendations of [18], response time should be less than the periodical update time, and in this study the update time was 1000 ms.

Using this reference, *Distributed* policies presented better response times than *Centralized* policies. Also, policies that sent underloaded information (*Eager* policies) had better performance than policies which shared overloaded information (*Lazy* policies). This happens because in the *Eager* policies, overloaded nodes generate the load balancing request, while in *Lazy* policies overloaded nodes have to wait until an underloaded node contacts them.

Note that for the *Eager Distributed* policy, overloaded nodes obtain the information of underloaded ones before the balance process. Therefore, since the response time is near to zero, we decided not to show this algorithm in the figure. Also note that, the poor scalability of the *Lazy Centralized* policy, can be explained because the server is monothreaded. Using a multithreaded central server can increase the saturation threshold, but it is not scalable solution because new constraints like bandwith usage or mutual exclusion are generated.

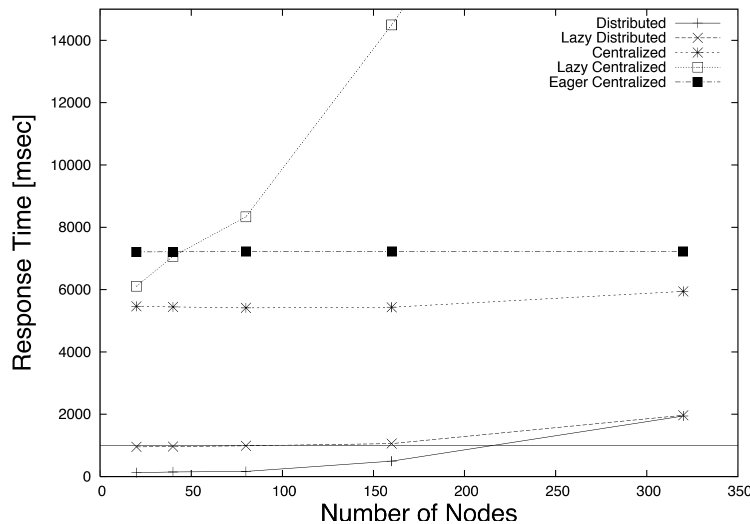

Fig. 2. Mean response time for all policies

3.2 Bandwidth

In this section we tested the policies bandwidth usage. Unfortunately, the underlying implementations introduces an additional difference: TCP or UDP based communications (resp. *Centralized* and *Distributed* policies). To avoid having to interpret such bias,

we compare performance between *full* and *partial* information policies, developed on *centralized* and *distributed* load-balancing algorithms.

Figure 3 shows the bandwidth used during the information sharing phase, counting only messages sent to the server:

1. *Centralized* policies use between 5 (*Eager Centralized*) and 40 times (*Centralized*) more bandwidth than distributed policies. This phenomenon is the result of the different type of network protocols used, and has been well studied in related-work [17].
2. For *partial information* schemes with *centralized* policies: when overloaded nodes share their information, less than 20% of the total nodes (see section 2.1) will send register/unregister messages, and more than 80% of them will send queries for registered nodes (every second).
3. When 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. This behavior causes the former approach to consume more bandwidth than the latter.

Figure 3 (right) shows the total bandwidth used by our load model, including the `loadBalance` and `addLoad` messages:

1. *Eager* policies which share *partial* information of underloaded nodes have the lowest bandwidth usage for each case (*Centralized* and *Distributed*).
2. *Lazy* policies which share *partial* information of overloaded nodes generate a great increase of the bandwidth usage, because there is no control on how many underloaded nodes send `loadBalance` messages. In the *Lazy Centralized* policy, this behavior generates a saturation on the communication channel even though the number of messages is half of that of the *Centralized* policy. This happens because most of the messages are balance queries, and the server has to choose an overloaded node and send the `loadBalance` message to it.
3. When the service queue of a central server becomes saturated (over 300 nodes on our experiments), the response time increases and the bandwidth usage decreases, because the saturation will causes less messages to be sent over the network. As noted for the *response time* analysis (see 3.1), using a multithreaded central server it is not a scalable solution.

3.3 Testing a real application

We tested the impact of the policies with a real application: the calculus of a *Jacobi* matrix. This algorithm performs an iterative computation on a real-valued square matrix. On each iteration, the value of each element is computed using its own value and the value of its neighbors on the previous iteration. We divided a 3600x3600 matrix into 25 disjoint sub-matrices of equal size, each one managed by an active object called "*worker*" (implemented using ProActive). Each worker communicates only with its direct neighbors.

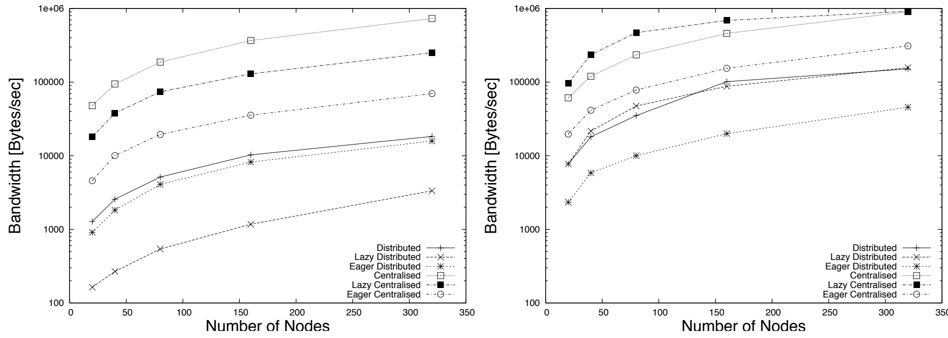

Fig. 3. Bandwidth usage of coordination policies: Information sharing phase (left), Total (right)

As a reference, all the workers are randomly distributed among 15 machines, using at most two workers by machine. Using this distribution, we measured the mean execution time of performing 1000 sequential calculi of Jacobi matrices (first row of Table 1).

To determine the impact of the policies on the *Jacobi* application, we distributed 30 nodes among the 15 machines. We ran the application (placing one load server outside of the simulation machines), and measured the execution time of *Jacobi*. Separately for each policy we measured the CPU cost (in % of busy time) for the 15 machines. The results are in Table 1.

Table 1. Information Sharing Policies and their effects on execution time of a parallel Jacobi application

Policy	Execution Time (sec)	% policy cost (time)	% policy cost (CPU)
None	914.361	—	—
Centralized	1014.960	11.00%	1.3%
Lazy Centralized	995.873	8.91%	1.1%
Eager Centralized	972.621	6.37%	1.1%
Distributed	1004.800	9.89%	10.7%
Lazy Distributed	925.964	1.26%	4.5%
Eager Distributed	915.085	0.08%	4.1%

While *Centralized* policies use less CPU on the “client” side, they use more bandwidth than their distributed equivalents. A special case is the *Distributed* policy, which uses less bandwidth than the *Centralized* policies, but the largest CPU time consumption, and it produces almost 10% of time delay on the application. So, if this policy is used, the load balancing itself will produce overloading.

4 Conclusions and Future Work

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

We conclude that *Distributed* oriented policies have the best performance using these metrics, and sharing underloaded nodes information (*Eager*), is the best decision. In a load-balancing architecture for communication-intensive parallel applications developed with asynchronous communicated middlewares, we suggest using an *Eager Distributed* policy where overloaded nodes trigger the balancing using previously acquired information, thus avoiding the need of *Centralized* servers. Moreover, if the load index could be updated with a lower frequency than one per second and similar accuracy, the policy would use less coordination messages, producing less interference with parallel applications.

Our future goal is to optimize the optimal candidate selection for the balancing process, aiming for the best performance in terms of *bandwidth usage* and *response time*.

Acknowledgments

This work was partially supported by Chile-Korea ITCC and Conicyt Chile.

References

1. Kento Aida, Wataru Natsume, and Yoshiaki Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. In *Proc. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.
2. Oasis Group at INRIA Sohpia-Antipolis. Oasis: Active objects, semantics, internet, and security. http://www-sop.inria.fr/oasis, 1999.
3. Oasis Group at INRIA Sohpia-Antipolis. Proactive, the java library for parallel, distributed, concurrent computing with security and mobility. http://www-sop.inria.fr/oasis/proactive/, 2002.
4. Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. An hybrid message logging-cic protocol for constrained checkpointability. In *Proceedings of Europar 2005*. Springer-Verlag, 2005.
5. Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. In *IEEE Symposium on Foundations of Computer Science*, pages 178–187, 2001.
6. Javier Bustos. Robin hood: An active objects load balancing mechanism, for intranet. In *Proc. of Workshop de Sistemas Distribuidos y Paralelismo, Chile*, 2003.
7. Javier Bustos, Denis Caromel, Alexandre Di Costanzo, Mario Leyton, and José Piquer. Balancing active objects on a peer to peer infrastructure. In *Proceedings of XXV International Conference of SCCC, Valdivia, Chile*. IEEE CS Press, November 2005.
8. T. L. Casavant and J. G. Kuhl. Effects of response and stability on scheduling in distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(11):1578–1588, 1988.

9. Steve J. Chapin, Dimitrios Katramatos, John Karpovich, and Andrew S. Grimshaw. The Legion resource management system. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 162–178. Springer Verlag, 1999.
10. Gerald Combs. Ethereal: The world's most popular network protocol analyzer. http://www.ethereal.com/.
11. Wilfired Klauser Denis Caromel and Julien Vayssiere. Towards seamless computing and metacomputing in java. *Concurrency Practice and Experience*, 1998.
12. Luís Paulo Peixoto dos Santos. Load distribution: a survey. citeseer.ist.psu.edu/santos96load.html.
13. Elisa Heymann, Miquel A. Senar, Emilio Luque, and Miron Livny. Adaptive scheduling for master-worker applications on the computational grid. In *GRID*, pages 214–227, 2000.
14. Fabrice Huet, Denis Caromel, and Henri Bal. A high performance java middleware with a real application. In *Proc. of High Performance Computing, Networking and Storage (SC2004), Pittsburgh, USA*, 2004.
15. M. Lo and S. Dandamudi. Performance of hierarchical load sharing in heterogeneous distributed systems. In *Proc. of International Conference on Parallel and Distributed Computing Systems, Dijon, France*, pages 370–377, 1996.
16. Miron Livny Michael Litzkow and Matt Mutka. Condor - a hunter of idle workstations. In *Proc. of 8th International Conference on Distribuited Computing Systems*, pages 104–111, 1998.
17. Sun microsystems. *RMI Architecture and Functional Specification*. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.
18. M. Mitzenmacher. How useful is old information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–34, 2000.
19. J.L. Bosque Orero, D. Gil Marco, and L. Pastor. Dynamic load balancing in heterogeneous clusters. In *Proc. of IASTED International Conference on Parallel and Distributed Computing and Networks*, 2004.
20. Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.
21. M. M. Theimer and K. A. Lantz. Finding idle machines in a workstation-based distributed system. *IEEE Trans. Softw. Eng.*, 15(11):1444–1458, 1989.
22. M. J. Zaki, Wei Li, and S. Parthasarathy. Customized dynamic load balancing for a network of workstations. In *Proceedings of the High Performance Distributed Computing (HPDC '96)*, page 282. IEEE Computer Society, 1996.
23. W. Zhu, C. Steketee, and B. Muilwijk. Load balancing and workstation autonomy on amoeba. *Australian Computer Science Communications*, 17(1):588–597, 1995.