Load Balancing: Towards the Infinite Network and Beyond

Javier Bustos-Jiménez[1,2,3], Denis Caromel[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. We present a contribution on dynamic load balancing for distributed and parallel object-oriented applications. We specially target peer-to-peer systems and their capability to distribute parallel computation. Using an algorithm for active-object load balancing, we simulate the balance of a parallel application over a peer-to-peer infrastructure. We tune the algorithm parameters in order to obtain the best performance, concluding that our *IFL* algorithm behaves very well and scales to large peer-to-peer networks (around 8,000 nodes).

1 Introduction

One of the most useful features of current distributed systems in the context of a desktop Grid, is the ability to redistribute tasks among its processors. This requires a redistribution policy to gain in efficiency by dispatching the tasks in such a way that the resources are used efficiently, i.e. minimising the average idle time of the processors and improving application performance. This technique is known as *load balancing* [1]. Moreover, when the redistribution decisions are taken at runtime, it is called *dynamic load balancing*. With the objective of scaling up to very large scale Grid systems, we placed ourselves in the context of using peer-to-peer (P2P) principles and frameworks. In this work we use the definition of *Pure peer-to-peer (P2P)* [2]: each peer can be removed from the network without any loss of network service.

In a previous work [3], we presented a P2P infrastructure developed within ProActive [4]. ProActive is an open-source Java middleware which aims to achieve seamless programming for concurrent, parallel, distributed, and mobile computing, implementing the active-object programming model (see Section 2). In its P2P infrastructure, all peers have to maintain a list of "*known nodes*" (also known as *acquaintances*). Initially, when a fresh peer joins the network, it only knows peers from a list of potential network members. A peer inside the network will receive a fresh-peer request and it has a certain probability of accepting the fresh peer as an acquaintance. If the fresh peer was accepted by the one inside the network, the latter forwards the fresh-peer request to its own acquaintances. We exploited the P2P nature of this network in a randomised

load-balancing algorithm and demonstrate that this approach performs better than a server-oriented scheme in a proprietary network [3].

Dynamic load balancing is a well-studied issue for distributed systems [5]. For instance, well-known load-balancing algorithms have been studied in the heterogeneous network context by Shivaratri, Krueger and Singhal [6] and in the P2P context by Roussopoulos and Baker [7]. However, these studies focus on balancing tasks (units of processing), while load-balancing of active objects is achieved by redistribution of queues.

Randomised load-balancing algorithms were popularised by work-stealing algorithms [8, 9], where idle processors randomly choose another processor from which to "steal" work. A work-stealing algorithm aims to maintain all processors working, but its random nature causes the algorithm to respond slowly to overloading. Therefore, due to the fact that processors connected to a P2P network share their resources not only with the network but also with the processor owner, new constraints like reaction time against overloading and bandwidth usage become relevant [10].

Most of the research in load-balancing for P2P networks is based on a structured approach using a *distributed hash table* (DHT) [11], where each machine can be represented by several keys, and parallel applications are mapped into this DHT. As a consequence, load balancing becomes now a search problem on key/data spaces [12]. Our P2P infrastructure is unstructured and shared resource are computational nodes (JVMs). Therefore it is not necessary to identify resources uniquely as would be the case for P2P data. Another approach for load balancing on P2P environments is the use of *agents* which traverse the network equalising the load among them. The agents follow a model of an ant colony [13, 14], *carrying* load among computers, and eventually making the system stable. Such a scheme focuses on load equalisation instead of the search of an optimal distribution. Our load balancing algorithm follows the same principles than MOSIX Distributed Operating System [15], but oriented to active objects, which are portable by definition and have no access to kernel calls. Moreover, information dissemination procedures are different: while MOSIX uses periodical randomised information sharing, we use on-demand information sharing because in [10] we demonstrate that no periodical information sharing provides scalability and updated load information [16] together.

In this paper we test this algorithm in a new setting: a simulated peer-to-peer network, trying to find its limits and analysing its behaviour. We show that the algorithm behaves very well but that some parameters need to be tuned for this kind of large networks.

This article is organised as follows. Section 2 presents ProActive as an implementation of *active-object programming model*. Section 3 explains the fundamentals of the randomised active-object load-balancing algorithm for P2P networks. Section 4 presents the simulated environment of our tests, the fine tuning of algorithm parameters, and the scalability tests. Finally, conclusions and future work are presented.

2 ProActive

The ProActive middleware is a Java library which aims to achieve seamless programming for concurrent, parallel, distributed and mobile computing. As it is built on top of

the standard Java API, it does not require any modification of the standard Java execution environment, nor does it make use of a special compiler, pre-processor, or modified virtual machine.

The base model is a uniform *active-object* programming model. Each active object has its own control thread and can independently decide in which order to serve incoming method calls. Incoming method calls are automatically stored in a queue of pending requests (called a *service queue*). When the queue is empty, active objects wait for the arrival of a new request; this state is known as *wait-for-request*.

Active objects are accessible remotely via method invocation. Method calls with active objects are asynchronous with automatic synchronisation. This is provided by automatic *future objects* as a result of remote methods calls, and synchronisation is handled by a mechanism known as *wait-by-necessity* [17]. Another communication mechanism is the *group communication* model. Group communication allows triggering method calls on a distributed group of active objects with compatible type, dynamically generating a group of results [18].

ProActive provides a way to move any active object from any Java Virtual Machine (JVM) to another, called a *migration* mechanism [19]. An active object with its pending requests (method calls), futures, and passive (mandatory non-shared) objects may migrate from JVM to JVM through the *migrateTo(...)* primitive. The migration can be initiated from outside the active object through any public method, but it is the responsibility of the active object to execute the migration, this is known as *weak migration*. Automatic and transparent forwarding of requests and replies provide location transparency, as remote references toward active mobile objects remain valid.

3 IFL: a randomised load-balancing of active-objects on P2P networks

Assume $load_A$ is the usage percentage of processor A. Defining two thresholds, OT and UT ($OT > UT$), we say that a processor A is *overloaded* (resp. *underloaded*) if $load_A > OT$ (resp. $load_A < UT$). Additionally, aiming to minimise the number of migrations until a stable state in load-balancing, we use a *rank* value which gives the processing capacity of a node. Ranks and loads are stored locally by each node. The idea of using a *rank* to generate a total order relation among processors was popularised by the Matchmaking scheme [20] of *Condor* [21]. While Condor uses its rank to measure the desirability of a match, we used it to discard slow nodes at runtime.

We exploited the results of Litzkow, Livny and Mutka, who reported that desktop processors are idle 80% of the time [21] (this value is reported up to 90% in 2005 [22, 23]). Also, we followed the recomendations of [10] about minimization of load-balancing messages to make our randomized scheme of load-balancing. In fact, let p be the probability of having a computer in an idle (or underloaded) state, and let n be the size of a subset of acquaintances. Then, for each node, the probability to have k underloaded nodes in its acquaintances subset is

$$\sum_{i=k}^{n} \binom{n}{k} p^i (1-p)^{n-i}$$

3.1 First version of the IFL algorithm

We have developed a new load-balancing algorithm [3], which we will call *IFL*. The *IFL* algorithm works as follows:

1. If a node (also known as *computation entity* or *processor*) is overloaded, it randomly chooses a minimal subset of (three, four or five of) its acquaintances. In Figure 1 (a) and (b), grey nodes represent the subset of acquaintances.
2. Only underloaded nodes who satisfy the rank criteria $requester\_rank < RB * my\_rank$ (where $RB \in [0, 1]$ constant) will be able to reply the request. In Figure 1 (c), two nodes are discarded using this criteria (those marked by X).
3. Nodes that satisfy the criteria reply to the request. Then, the overloaded node will send an active object to the owner of the first received reply (Figure 1 (d)). We use this scheme because we want to maintain the active objects close to each other to avoid communication latency at runtime.

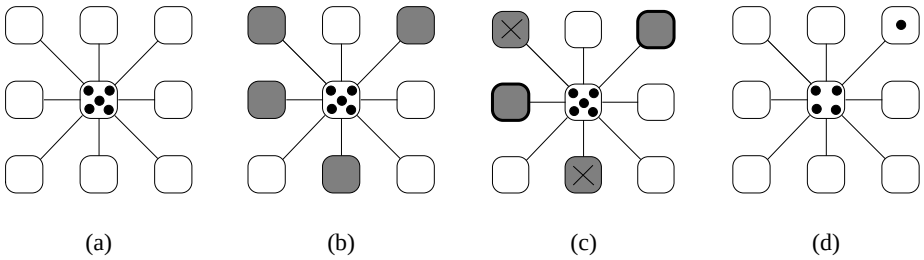


Fig. 1. Load-balancing algorithm for active-objects over Peer-to-Peer networks.

This algorithm performs load balancing until a stable state, reaching a local optimum of balance. If there are no overloaded nodes, no active object has incentive to migrate.

3.2 Second version of the IFL algorithm

Considering that the communication scheme of ProActive is based in RMI that has a high-cost in bandwidth-usage and latency [24, 25], we aimed to optimise the application performance by clustering active objects on the best qualified processors. Therefore, in our second version of load balancing algorithm (the *IFL algorithm*), we add a work-stealing [1] step:

1. If a node is underloaded, it randomly chooses one of its acquaintances to which it sends a work-stealing request.
2. If the receiver satisfies the rank criteria $RS * requester\_rank > my\_rank$, it sends an active object to the caller.

Note that if we consider that each node made its first contact with a "near" peer (usually in the same physical network), it is more probable that stealing occurs between close nodes than remote ones.

3.3 Experimental verification

To verify our theoretical reasoning, we experimented with a small-scale real laboratory environment. We tested the two versions of our algorithm (with and without work-stealing), we used Jacobi matrix calculus to solve a 3,600x3,600 matrix with 36 workers implemented as active objects (implementation details available in [3]). We run the test on a set of 25 of INRIA desktop computers, having 10 Pentium III 0.5 - 1.0 GHz, 9 Pentium IV 3.4GHz and 6 Pentium XEON 2.0GHz, all of them with a Linux operating system and connected by a 100 Mbps Ethernet switched network. Starting from random initial distributions, we measured the execution time of 1000 sequential calculus of Jacobi matrices.

For both versions we used as load index the CPU load and as rank the CPU speed. Also, using our knowledge of the lab networks, we experimentally defined the algorithm parameters as $OT = 0.8$ (to avoid swapping on migration time), and $UT = 0.5$. We experimentally discovered that value of RB between 0.5 and 0.9 has any "*optimal*" behaviour. We explain this by the existence of a correlation between processing capacity and load state: it is highly probable to find a low capacity node overloaded than underloaded. Therefore, we fixed $RB = 0.7$. Also, we experimentally defined $RS = 0.9$.

Figure 2 shows the mean execution time of the Jacobi application and the number of migrations. A low number of migrations corresponds to an initial distribution of active-objects near to an optimal state (local or global), and a high number of migrations corresponds to an initial distribution far from an optimal state. Also, the mean time performed by the Jacobi application without load balancing is represented by the horizontal line marked by (*), this value was obtained using a subset of the 10 best-ranked nodes, having the nodes full availability for Jacobi application. Note that this value is a good approximation of the global optimal distribution.

Figure 2 shows that, for the first version of our algorithm, the presence of a local optimum attempts against a good performance of the application. For the second version, a performance near the global optimal state is reached for all migration counts.

In the next section we experiment *IFL* in the context of Desktop Grids to see if it can reach a near optimal state for a large number of nodes (around 8,000).

4 Simulating the algorithm on a large scale P2P network

In the study of Load-Balancing algorithms, one of the most important characteristics of nodes are their *processing capacity*. A function using this capacity and the amount of work that a node has to perform determine if a node is on an overloaded or underloaded state. To have a reliable model of processing capacity, we made a statistical study of desktop computers registered at the Seti@home project [26]. This project aims to analyse the data obtained from the Arecibo Radio telescope, distributing units of data among

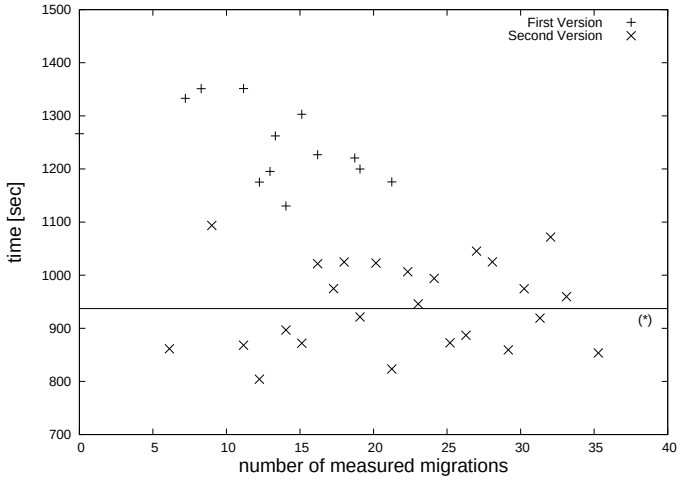


Fig. 2. Mean execution time for different number of migrations of the Jacobi Parallel Application, using the load balancing algorithm with and without work-stealing.

personal computers and exploiting the processing capacity of up to $200,000$ processors distributed around the world to analyse the data. Using the measured *Mflops* by BOINC [27] benchmarks. We consider *Mflops* as a good metric to determine the processing capacity for parallel scientific calculus, because we are interested in processing balance, not data balance.

We grouped all desktop computers *Mflops* (d_r) in 30 clusters (C_t) using the following formula:

$$d_r \in C_t \; iif \; \lfloor \frac{r}{10^6} \rfloor = t$$

The resultant frequency histogram is shown in Figure 3.

Defining a normal distribution $nor(x)$ (equation (1)), we compared the real distribution against our $nor(x)$ model function using Kolmogorov-Smirnov test statistics (KST), giving us a value of $KST = 0.0605$. Therefore, we can deduce that using a level of significance 0.01, the capacity of processors in a Large-Scale network can be modelled by a normal distribution.

$$nor(x) = 16000 \times e^{\frac{(-(x-1300)^2}{2\times 400^2}} \tag{1}$$

We implemented in C a network simulator, using an $n \times n$ matrix for the nodes and an $n^2 \times n^2$ matrix for the edges. We assign the nodes processing capacities (called μ) using a normal distribution $N(1, \frac{1}{9})$. Even thought this simple model seems to be naïve, it permits us to control the topology generated by the P2P Infrastructure of ProActive

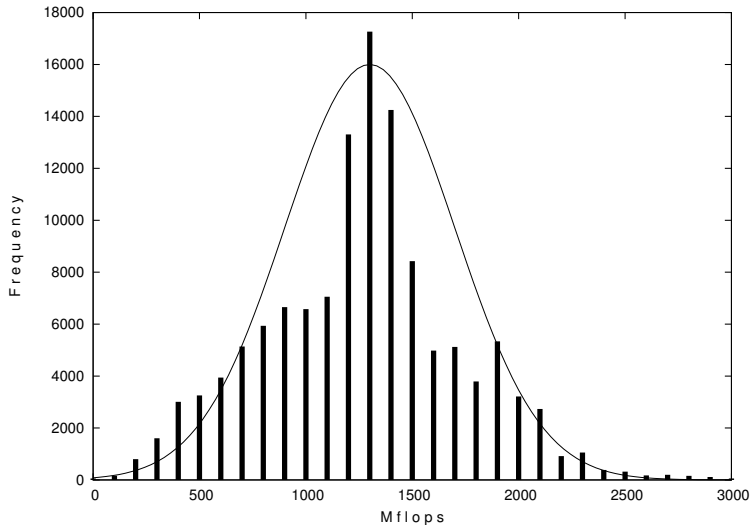


Fig. 3. Frequency distribution of Mflops for 200,000 processors registered at Seti@home and the normal distribution which model it

[3]. Simple models could be very powerfully as Kleinberg shows in his work about Small-world network algorithms [28].

In our simulations, we assume that all active objects are parts of a parallel application; therefore, we assume all service queues to have equal incoming message ratios λ. Clearly, real Grids run different parallel applications from different sources, having different service queue ratios and workloads. Nevertheless, from the point of view of a given parallel application, we consider other applications only as a reduction of processing capacity of network nodes.

Denoting by j the number of active objects in the node i at a given time, we say that the node i is overloaded if $j\lambda \geq \mu_i$ and underloaded if $j\lambda < T\mu_i$, where T is a given threshold between [0.5, 0.9]. The processor capacity μ_i is also used as the node rank. For consistency with the previous section, we use $UT = T \times \mu_i$ and $OT = \mu_i$.

We randomly placed m active objects in $(0 + x, 0 + y)$ (x and y defined on runtime) and tested the load-balancing algorithm, measuring the total number of migrations and the kind of processors used by the algorithm on each time-step. Each experimental sample is the mean number of 100 repetitions, fixing the parameter set $\{n, m, \lambda, T, RB, RS\}$ (see Table 1) and recalculating μ for all nodes in each repetition.

Our goals are to perform a fine-tuning of the constant RS and second to determine whether our algorithm can reach a stable state near to the optimal on large-scale P2P networks using a minimal subset of acquaintances. Even though migration cost seems to be a key issue for load balancing algorithm, it is possible that processors use the blocking or idle time of the parallel application to perform migrations having a low overcost in application total time.

Table 1. Parameters and variables used in the simulation

Simulation parameters	Model parameters	Algorithm parameters
n Number of nodes	μ processor's capacity and ranking	UT threshold to determine an underloaded state
m Number of active objects	λ incoming ratio of an active object service queue	OT threshold to determine an overload state
x, y Initial subset length and hight	T factor used to determine UT	RB Load-balancing similarity factor
		RS Work-stealing similarity factor

4.1 Fine-Tuning

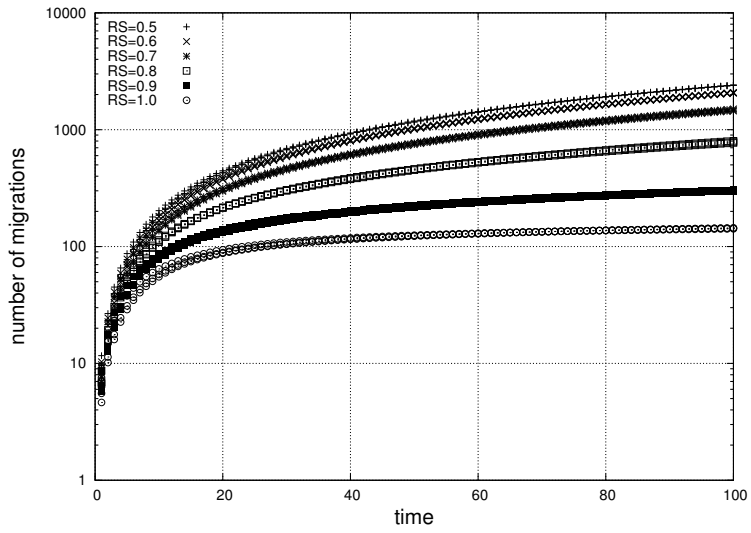
We placed $m = 50$ active-objects in a simulated P2P network of 100 nodes, measuring the total number of migrations performed by the algorithms until a given time-step (Figure 4a) and the number of overloaded nodes per time-step (Figure 4b), because it is imperative for all load-balancing algorithms to avoid increasing the number of overloaded nodes. As we expected, a lower value for RS generates a greater number of migrations. It is easy to see that a low value of this factor will produce bad decisions of balance, migrating active objects to underloaded nodes with low processing capacity. Then, those active objects could cause overload in subsequent nodes, or an infinite migration among underloaded nodes.

Figure 5a presents the mean number of active-objects in nodes with capacity higher than one per total number of active objects during 100 repetitions, and Figure 5b presents the mean number of active objects in nodes with capacity higher than $1\frac{1}{3}$ by total number of active objects during 100 repetitions. Because we are using a normal distribution for the processor capacity μ, 50% of nodes will have $\mu \geq 1$ and 25% of nodes will have $\mu \geq 1\frac{1}{3}$.

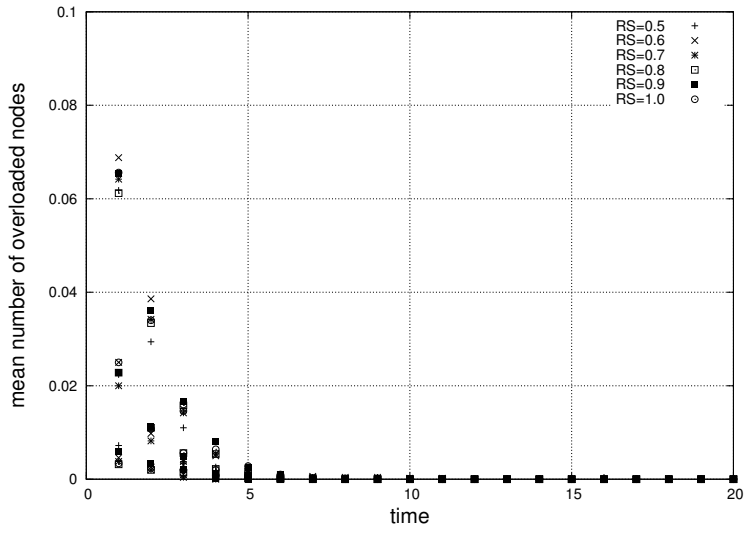
Two behaviours are present in Figure 5 (a) and (b). First, because our algorithm aims to cluster active-objects on the best processors, for high values of RS, the number of active objects int the best quadrant of the processors increase. Second, for low values of RS, some active objects are stolen by worse processors. We can see from the plots that $RS \geq 0.9$ behaves very well, placing all of active objects in nodes with processing capacity greater than one.

4.2 Scaling

As seen in the previous section, we aimed to optimise the application performance clustering active-objects on the best qualified processors. Therefore, using the values of μ, we sorted the nodes from higher to lower processing capacity and we defined the

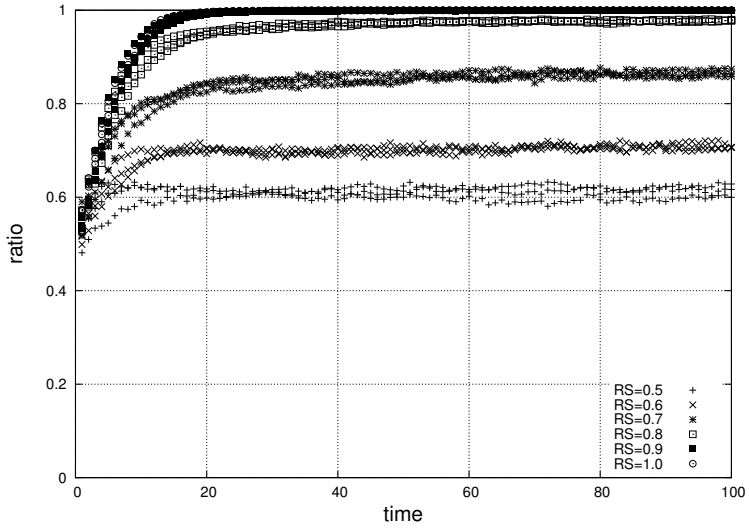


(a)

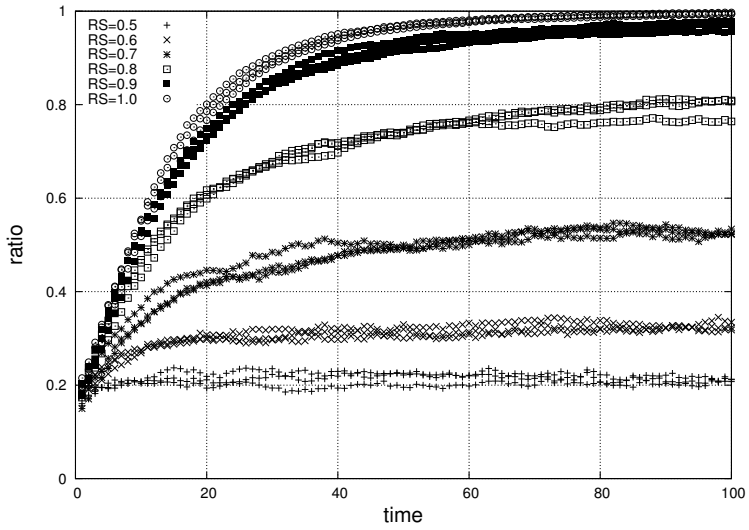


(b)

Fig. 4. Tuning for RS considering: a) mean number of total migrations until each time-step; and b) mean number of overloaded nodes in each time-step. Using $RB = 0.7$, acquaintances subset size = 3, $|x - y| \leq 3$, $\lambda = 0.1, 0.2, 0.3$ and $T = 0.7$



(a)



(b)

Fig. 5. Tuning the value of RS considering: a) mean number of active objects on a node with $\mu \geq 1$ per total number of active objects; and b) mean number of active objects on a node with $\mu > 1 + \frac{1}{3}$ per total number of active objects. Using $RB = 0.7$, acquaintances subset size = 3, $|x - y| \leq 3$, $\lambda = 0.1, 0.2, 0.3$ and $T = 0.7$

optimal subset as the first OPT nodes that satisfy the condition:

$$\sum_{i=1}^{OPT} \mu_i > m \times \lambda$$

Simulating and application of $m = 100$ active objects, we have $OPT_{n=10} = 13$, $OPT_{n=20,30} = 11$, $OPT_{n=40} = 10$ and $OPT_{n\in[50,90]} = 9$.

In order to measure the performance of the *IFL* algorithm for large-scale networks, we define a ratio:

$$ALOP = \frac{Number\ of\ nodes\ used\ by\ IFL}{OPT}$$

And, at the same time, we calculate the mean number of migrations performed by all active objects from time step 0 until time step t. An increase in the acquaintances subset size results in an increase in the probability to find a node to migrate, and hence an increase in the probability to reach the optimal state. Therefore, we only show the results for $subset\ size = 3$.

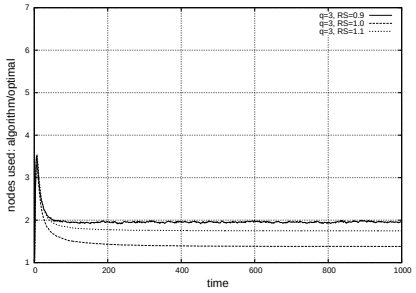
We measured scaling of the *IFL* algorithm in terms of $ALOP$ and the number of migrations, for networks of 100 (Figures 6 (a) and (b)) and 400 nodes(Figures 6 (c) and (d)). Even though in Section 4.1, a value of $RS = 0.9$ was promising, these plots show that the total number of migrations generated by this value makes the algorithm not scalable. Scalability in terms of migrations is presented in Figures 6 (b) and (c) only for values of $RS \geq 1.0$. The optimal scalability, in terms of $ALOP$, is presented in Figures 6 (a) and (c) for a value of $RS = 1.0$.

Taking into account that a $20x20$ network can be still considered as a small network, we test the scalability in terms of $ALOP$ and number of migrations over $n \times n$ P2P networks using $n = [10, 90]$, fixing the parameter RS in 1.0 and RB in 0.7. The results are shown in Figure 7.

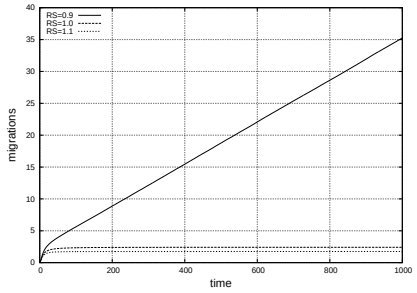
Note that in the beginning the *IFL* algorithm increases the number of nodes used, because active objects are first placed in a small subset of the network generating a high overload in this subset. Then, the algorithm quickly performs migrations to reduce the overload. Then, only the *work-stealing* step of *IFL* algorithm works, clustering active-objects on the best nodes and thus, reducing the number of nodes used by the algorithm. Experiments report no overloaded nodes over 30 time-steps.

Figure 7 (a) presents the $ALOP$ ratio for several $n \times n$ P2P networks. For networks of until 40×40 nodes, the *IFL* algorithm uses less than two times the optimal number of nodes. In other words, *IFL* algorithm uses less than 20 nodes from all the network until 100 time-steps. For networks of 50×50 to 70×70 nodes, the algorithm uses less than three times the number of optimal nodes (i.e: 27). For larger networks, the algorithm uses more than three times the optimal number of nodes at time step 1000, but the curves seem to decrease before that value. Because the distribution of processing capacity μ follows an exponential distribution, the minimal μ in the subset of the "best X nodes" will be higher for larger values of n.

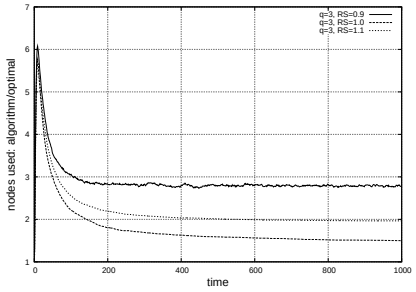
Figure 7 (b) presents the total number of migrations between time step 0 and a given time step t ($t \in [1, 1000]$) for $m = 100$ active objects over P2P networks from 10×10 to 90×90 nodes. The curves remain under 6.5 migrations. Considering only the time



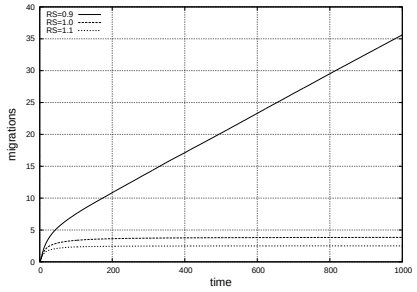
(a) ALOP for a 10x10 network



(b) Migrations for a 10x10 network



(c) ALOP for a 20x20 network



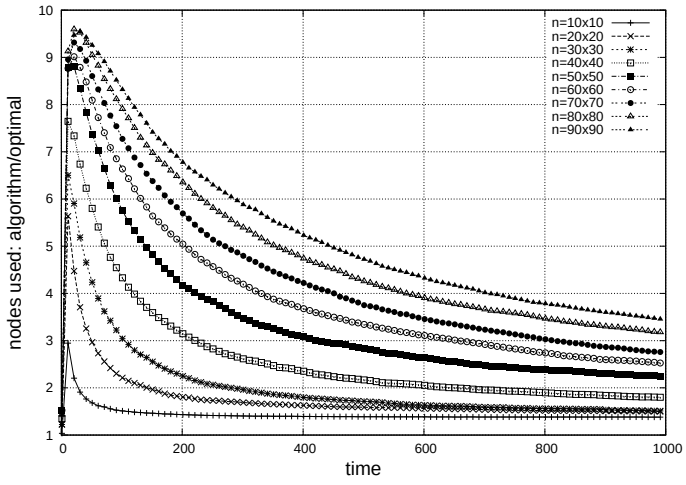
(d) Migrations for a 20x20 network

Fig. 6. Scalability for a network using $RS = 0.9, 1.0, 1.1$, $RB = 0.7$

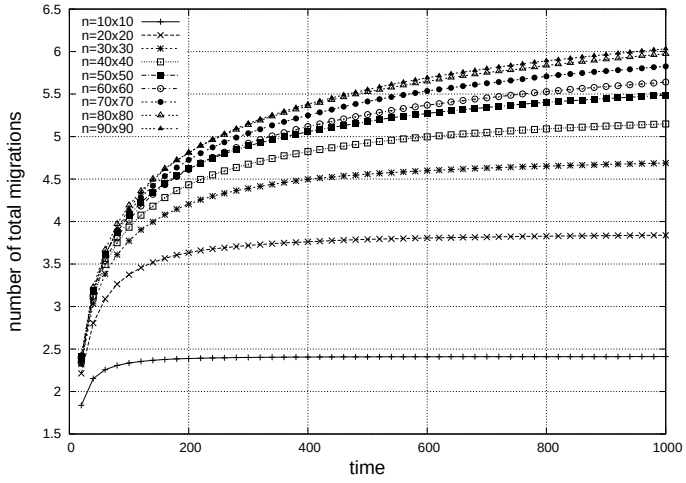
step 1000, we can see that the number of migrations is of order $O(\log(n))$. Both are promising results in terms of scalability of the *IFL* algorithm.

5 Conclusions

We studied the *IFL* load-balancing algorithm on P2P networks, aiming at reaching a near-optimal distribution of active objects using only local information provided by a P2P infrastructure. Using a simulated P2P network, we showed that, using only a low number of fixed links among nodes and a careful tuning of the algorithm parameters, a near-optimal distribution is reachable even for large-scale networks. We suggested to use a value near 1.0 for the stealing factor, which allows using around 1.7 times the optimal number of nodes for networks until 400 nodes, using less than 5.5 migrations



(a) ALOP



(b) Migrations

Fig. 7. Scalability using $RS = 1.0$

per active object. Moreover, the number of migrations appears to be of order $O(\log(n))$ after the first optimal state (without overloaded nodes) is reached.

As seen in Section 4.1, the value of RS is a key factor for a low cost and efficient load balancing and we had many experimental tuning to find "optimal values" of it. RS seems to depend of network topology and we are studying its behaviour to calculate it automatically and dynamically.

As future work, we plan to test the algorithm using a large-scale P2P infrastructure deployed over real desktop computers, balancing a communication-intensive parallel application. It is the continued goal of this work to optimise this algorithm, looking for the best performance in migration decisions and the global distribution using only local information.

Acknowledgements

This work was supported by CoreGrid NoE.

The authors want to thank Satu Elisa Schaeffer for proofreading an earlier version of this paper and the anonymous reviewers for their helpful comments.

References

1. T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.
2. R. Schollmeier, "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications," in *2001 International Conference on Peer-to-Peer Computing (P2P2001)*, (Department of Computer and Information Science Linkopings Universitet, Sweden), IEEE Computer Society, August 2001.
3. J. Bustos-Jiménez, D. Caromel, A. di Costanzo, M. Leyton, and J. M. Piquer, "Balancing active objects on a peer to peer infrastructure," in *Proceedings of the XXV International Conference of the Chilean Computer Science Society (SCCC 2005)*, (Valdivia, Chile), pp. 109–115, IEEE Computer Society, November 2005.
4. Oasis Group at INRIA Sohpia-Antipolis, "Proactive, the java library for parallel, distributed, concurrent computing with security and mobility." http://proactive.objectweb.org, 2002.
5. L. P. P. dos Santos, "Load distribution: a survey." citeseer.ist.psu.edu/santos96load.html.
6. N. G. Shivaratri, P. Krueger, and M. Singhal, "Load distributing for locally distributed systems," *Computer*, vol. 25, no. 12, pp. 33–44, 1992.
7. M. Roussopoulos and M. Baker, "Practical load balancing for content requests in peer-to-peer networks," *The Computing Research Repository*, vol. cs.NI/0209023, 2002.
8. R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.
9. P. Berenbrink, T. Friedetzky, and L. A. Goldberg, "The natural work-stealing algorithm is stable," in *IEEE Symposium on Foundations of Computer Science*, (Washington, DC, USA), pp. 178–187, IEEE Computer Society, 2001.
10. J. Bustos-Jiménez, D. Caromel, M. Leyton, and J. M. Piquer, "Load information sharing policies in communication-intensive parallel applications," in *ISSADS* (F. F. R. Corchado, V. Larios-Rosillo, and H. Unger, eds.), Lecture Notes in Computer Science, Springer, 2006. To appear.

11. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 161–172, ACM Press, 2001.
12. B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured p2p systems," *Lecture Notes in Computer Science*, vol. 2735, pp. 68–79, January 2003.
13. A. Montresor, H. Meling, and Ö. Babaoglu, "Messor: Load-balancing through a swarm of autonomous agents.," in *AP2PC*, vol. 2530 of *Lecture Notes in Computer Science*, pp. 125–137, Springer, 2002.
14. J. Cao, "Self-organizing agents for grid load balancing," in *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, (Washington, DC, USA), pp. 388–395, IEEE Computer Society, 2004.
15. A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1993.
16. M. Mitzenmacher, "How useful is old information?," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 1, pp. 6–20, 2000.
17. D. Caromel, "Toward a method of object-oriented concurrent programming," *Communications of the ACM*, vol. 36, no. 9, pp. 90–102, 1993.
18. L. Baduel, F. Baude, and D. Caromel, "Efficient, flexible, and typed group communications in java," in *Joint ACM Java Grande - ISCOPE 2002 Conference*, (Seattle), pp. 28–36, ACM Press, 2002. ISBN 1-58113-559-8.
19. F. Baude, D. Caromel, F. Huet, and J. Vayssiere, "Communicating mobile active objects in java," in *Proceedings of HPCN Europe 2000*, vol. 1823 of *LNCS*, pp. 633–643, Springer, May 2000.
20. R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed resource management for high throughput computing," in *HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing*, (Washington, DC, USA), p. 140, IEEE Computer Society, 1998.
21. M. L. Michael Litzkow and M. Mutka, "Condor - a hunter of idle workstations," in *Proc. of 8th International Conference on Distribuited Computing Systems*, pp. 104–111, 1998.
22. P. Domingues, P. Marques, and L. M. Silva, "Resource usage of windows computer laboratories.," in *34th International Conference on Parallel Processing Workshops (ICPP 2005 Workshops), 14-17 June 2005, Oslo, Norway*, pp. 469–476, IEEE Computer Society, 2005.
23. D. P. Anderson and G. Fedak, "The computational and storage potential of volunteer computing," in *CCGRID*, IEEE Computer Society, 2006. To appear.
24. Sun Microsystems, *RMI Architecture and Functional Specification*. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.
25. C. Nester, M. Philippsen, and B. Haumacher, "A more efficient RMI for Java," in *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, (New York, NY, USA), pp. 152–159, ACM Press, 1999.
26. P. Paul, "Seti @ home project and its website," *Crossroads*, vol. 8, no. 3, pp. 3–5, 2002.
27. D. P. Anderson, "Boinc: A system for public-resource computing and storage," in *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, (Washington, DC, USA), pp. 4–10, IEEE Computer Society, 2004.
28. J. M. Kleinberg, "The small-world phenomenon: an algorithm perspective," in *Proceedings of the Thirty Second Annual ACM Symposium on Theory of Computing*, (New York, NY, USA), pp. 163–170, ACM Press, 2000.