Effects of Job and Task Placement on the Performance of Parallel Scientific Applications

Technical Report EHU-KAT-IK-04-08

Javier Navaridas, Jose Antonio Pascual, Jose Miguel-Alonso

The University of the Basque Country UPV/EHU
Department of Computer Architecture and Technology
P.O. Box 649, 20080 San Sebastian, SPAIN
{javier.navaridas, ja-pascual, j.miguel}@ehu.es

Abstract. This paper studies the influence that job and task placement may have on application performance, mainly due to the relationship between communication locality and overhead. A simulation-based performance study is carried out, using traces of applications as well as application-inspired, synthetic traffic patterns, to measure the time taken to complete one or several instances of a given workload, spread through the whole network. The interconnection networks used in this study have torus and fat-tree topologies. Task placement strategies are simple ones, including random placement. Results of experiments show that both the number of concurrent parallel jobs sharing a machine and the size of its network have a clear influence on the time to complete a given workload. We conclude that the efficient exploitation of a parallel computer requires the utilization of locality-aware scheduling policies.

Keywords: Interconnection Networks, Trace Driven Simulation, Parallel Job Scheduling.

1 Introduction

Current high-performance computing facilities are composed of thousands of computing nodes executing jobs in parallel. An underlying interconnection network (such as Myrinet, Infiniband, Quadrics, or an *ad-hoc* network) provides a mechanism for tasks to communicate. It is very uncommon to devote all the nodes of a site to run a single application. In most cases, nodes are time and/or space shared among users and applications. Supercomputing sites have a job queue to which users send their parallel jobs, where they wait until a scheduler allocates some resources to it. A large variety of scheduling policies have been proposed and are in use in order to manage the queues, many of them based on the First-Come-First-Served discipline and (typically) taking

This work has been supported by the Spanish Ministry of Education and Science, grant TIN2007-68023-C02-02, and by the Basque Government, grant IT-242-07. Mr. Javier Navaridas is supported by a doctoral grant of the UPV/EHU.

into account some restrictions: different levels of priority, quotas (both in terms of time and number of processors), job size, etc.

It may be shocking to know that many scheduling policies disregard any knowledge about the topological characteristics of the underlying system; they see the system as an unstructured pool of computing resources. Schedulers assign free nodes (resources) to jobs, independently of the location of those nodes in the network, and return them to the free pool when jobs finish or are cancelled. After a certain warm-up time, which depends on the number and variety of executed jobs, physical selection of allocated resources is close to random: nodes assigned to a given job may be located anywhere in the network. In other words, resources are fragmented.

The reader should note that programmers of parallel applications usually arrange tasks in some form of virtual topology. This is a natural way of programming applications in which large datasets (matrices) are partitioned among tasks. Programmers favor communication between neighboring tasks, under the assumption that this strategy should result in improved performance. If the assigned execution nodes are not in close proximity, programmers' efforts are totally useless. Furthermore, if job placement is random, the messages interchanged by a job may interfere with those interchanged by other, concurrent ones, in such a way that contention for network resources may be exacerbated. Thus, topological information should be taken into account in the scheduler's decision process to effectively exploit locality and to avoid undesired interactions between jobs.

As we stated before, most job placement policies are not locality-aware. In this work, we want to show how the inclusion of topological knowledge in schedulers can improve the performance of parallel computers. We will explore previous work on this issue, as well as the current state of the art about scheduling tools, focusing on the knowledge of the system they manage. Furthermore, we will discuss the impact that job and task placement has on performance when the network of the parallel computer is based on any of the topologies more widely used nowadays: fat-tree and torus. To do it so, we carry out a simulation-based performance study in which we feed the networks with different application-like workloads: traces, and synthetic patterns that closely emulate the behavior of actual applications. We test networks of different sizes, in which we explore alternatives of task allocations for a single parallel job, and job/task allocations for concurrent, parallel jobs. Results support what we stated before about the effects of topology-unaware placement: it results in unnecessarily long execution times.

The rest of this paper is organized as follows. In Section 2 we discuss some work on job placement and also explore some schedulers and their job placement policies. The experimental environment – studied networks, workloads and placement strategies – is described in Section 3. In Section 4 we show and analyze the results of the experiments. Section 5 closes this paper with some conclusions and an outlook of our plans for future work.

2 Related Work

In the literature we can find a variety of strategies for resource allocation and scheduling. These two problems are strongly interconnected. The use of a good allocation algorithm and a good scheduling policy decreases network fragmentation, allocating jobs contiguously in the parallel system, and applications can take advantage of this.

In [13] we can see how the contiguous allocation of tasks results in improved application performance. Authors run 8 sets of 16-node FFTs (benchmark FT, part of the NPB FT [7]) concurrently on a 128-node mesh, and compare contiguous vs. random node allocation. They observed a 40% improvement in runtime when using contiguous allocation. The obvious way to go is to introduce contiguous allocation strategies in schedulers for parallel machines. In other papers addressing this issue ([3][5][6]) allocation algorithms have been proposed mainly for k-ary n-cube topologies. Figures of merit usually do not show how placement strategies affect to the runtime of an application instance, but the completion time of a list of jobs. In [9] we pay attention to tree-based topologies and rely on contiguous allocation of tasks to, by means of an efficient exploitation of communication locality, dilute or even invert the potentially negative effects of reducing the bisection bandwidth of the network. Interestingly, in [5] authors show how the requirement of contiguous allocation may cause poor utilization of the system due to external or internal fragmentation. To avoid this effect, they evaluate several noncontiguous (but not random) allocation schemes that improve overall system utilization.

A review of commercial and free schedulers shows that, by default, they are not topology aware – in other words, they do not care the actual placement of tasks. This is true for job queuing systems and scheduling managers such as Sun Grid Engine, LoadLeveler or PBS Pro (the latter used in Cray Supercomputers [1]). Although some of them provide mechanisms for the system administrators to implement their own scheduling/allocation policies, in practice, this is not done. For example, the scheduling strategy used on Cray XT3/XT4 systems (custom-made 3D tori) simply gets the first available compute processors [1]. Maui and Slurm, in use in ASC Purple (Federation network) and BSC's MareNostrum (multistage Myrinet), have an option to take into account application placement, but they ignore the underlying topology, considering a flat network. The most notable example of current supercomputer that tries to maintain locality when allocating resources is the BlueGene family (3D tori), whose scheduler [2] puts tasks from the same application in one or more midplanes of 8x4x4 nodes.

3 Experimental Set-Up

We have used simulation to assess the impact of allocation strategies on application performance. The simulation environment encompasses a network simulator and a workload generator [11]; we describe them in this section. It is important to remark that our simulator measures time in terms of *cycles*; a cycle is the time required by a *phit* (physical transfer unit – fixed to 4 bytes) to traverse one network switch.

3.1 Workloads

Throughout this work we evaluate networks using realistic workloads, taken from actual or emulated applications. In particular, we use traces taken from the well-know NAS Parallel Benchmarks [7] (NPB), and a set of application kernels described in [8]. In both cases we assume *infinite-speed* processors, meaning that we only measure the time used by the network to deliver the messages, but not the time used at compute nodes to generate, receive and process them. Note that message causality is preserved, so when the trace states that a node must perform a receive operation, the (simulated) node stalls until the expected message arrives. Under this assumption, reported results only take into account the communication and synchronization part of parallel applications; thus, the actual impact on performance of a given scheduling algorithm would depend on the application's computation/communication ratio.

Regarding traces, we use class A of NPB applications **BT**, **CG**, **IS**, **LU**, **MG**, **SP** and **FT**. This study does not include **EP** because it does not make intensive use of the network. In order to reduce required computing resources, and given that they are iterative applications, we drastically reduce the number of iterations in each benchmark to be in the range 10-20.

Pseudo-synthetic workloads used in this work are binary-tree (**BI**), butterfly (**BU**), distribution in 2D or 3D mesh (**2M**, **3M**) and wavefront in 2D or 3D mesh (**2W**, **3W**), see [8] for details. The selected message length is 64 Kbytes. We also use waterfall (**WF**), a pattern observed in the **LU** NPB application [12]. This pattern is actually a burst of 286 wave-fronts (**2W**) starting at once, each of them composed by small messages (256 bytes, or 4 packets).

All of the workloads used in the experiment were captured (or generated) for exactly 64 tasks. In some experiments the network has 64 nodes, so a single application uses the whole computer. In others, the network has 64*N nodes, so N instances of an application share the computer. Chosen values of N are 4 and 16. To simplify the experiments, we never mix different applications. The figure of merit to measure performance is the time required to consume all the messages in the workload. When using multiple, simultaneous application instances to feed a network, reported time is the one required to complete all the instances (the time taken by the slowest one).

3.2 Networks and placement

Not all the interconnection networks have the same properties, including the topological ones, and the effect of the placement strategies may vary depending on the network. For this reason, we have selected two of the topologies most widely used: fat-trees (typically used to build large-size clusters) and cubes (typically used to build massively parallel computers). The reader can check the Top500 list [3] to see how most computers in the topmost positions of the list fit on one of these categories.

In our experiments we use small to medium-size networks, with a number of nodes ranging from 64 to 1024 nodes. Given these sizes, we consider only 2D cubes (3D would be recommended for large-size networks). In order to allow workloads to fit exactly in the network, we have used fat-trees built with 8-radix switches. Note how fat-trees raise one level from configuration to configuration. Considering all these restrictions, the networks used in our study are:

- 4-ary,3-tree and 8-ary,2-cube for experiments with a single application instance.
- 4-ary,4-tree and 16-ary,2-cube for experiments with 4 application instances.
- 4-ary,5-tree and 32-ary,2-cube for experiments with 16 application instances.

Note that the aim of this paper is *not* to compare the torus against the fat-tree. The evaluation of alternative network topologies goes beyond the scope of this paper.

We assume that parallel jobs are composed of 64 tasks, numbered from 0 to 63. Network nodes are also numbered. In the case of fat-trees, numeration of nodes is: (0,0), (0,1), (0,2), (0,3), (1,0), (1,2), etc., where (*s*,*p*) should be read as "switch number *s*, port number *p*". The switch numbers correspond, left to right, to the tree's lowest level, that to which compute nodes are attached. In the case of 2-cubes, numeration is done using the Cartesian coordinates of the nodes: (0,0), (0,1), (0,2), (0,3), (1,0), etc.

Regarding placement, we consider *task placement* (allocation of the tasks of a single job) and *job placement* (allocation of several jobs that will run concurrently). Actually, we consider task allocation only for the experiments with a single application instance. In the other cases we evaluate a combination of task and job placement strategies. Now we describe these. In all cases, we assume that assignment is done first in order of the job identifier and, for a given job, nodes are assigned to task in order of task identifier. For fat-tree, allocation can be:

- *Consecutive*. Switch/port assignment is done selecting, in order, node (*s*,*p*), increasing first *p* and then *s*.
- In *shuffle* order. Switch/port assignment is done selecting, in order, node (*s*,*p*), increasing first *s* and then *p*.

For torus, allocation can be:

- In *row* order, assignment is done selecting, in order, node (*x*,*y*), increasing first *x* and then *y*. This can be seen as partitioning the network in rectangular sub-networks, wider than tall.
- In *column* order. Assignment is done selecting, in order, node (*x*,*y*), increasing first *y* and then *x*. This can be seen as partitioning the network in rectangular sub-networks, taller than wide.

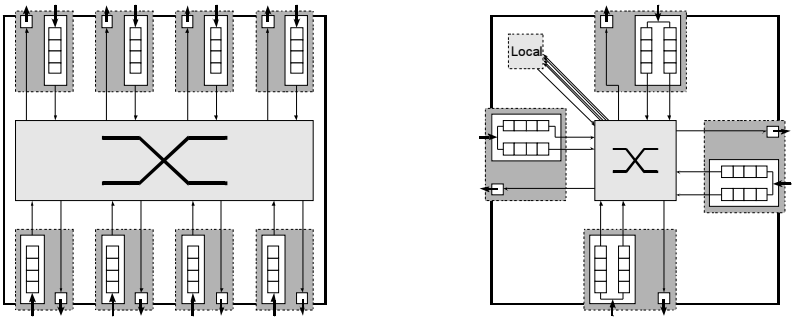

Fig. 1 Model of the switches used to build fat-trees (*left*) and 2D tori (*right*).

- When using several application instances, we can partition the network in perfect squares (this is possible because our choice of network and application sizes). We use a *quadrant* scheme in which the network is partitioned this way. Allocation inside each partition is done in row order.

For both torus and fat-tree, allocation of the tasks of N 64-task jobs to an N*64-node machine can be done *randomly*. When running experiments with this placement, we generate five random permutations and plot the average, maximum and minimum of the measured execution times.

3.3 Models of the components

Nodes are modeled as reactive traffic sources/sinks with an injection queue that can store up to four packets. In order to model causality, the reception of a message may trigger the release of a new one. If necessary, messages are segmented into fixed-size packets (16 phits). One phit is the smallest transmission unit, fixed to 32 bits. If a message does not fit exactly in an integral number of packets, the last packet contains unused phits.

Simple input-buffered switches are used. Transit queues are able to store up to four packets. The output port arbitration policy is round robin. Switching strategy is virtual cut-through. Models of switches for the two topologies are depicted in Fig. 1.

In the case of the fat-trees switches are radix-8. Routing is, when possible, adaptive using shortest paths. A credit-based flow-control mechanism is used, so that when several output ports are viable options to reach the destination, the port with more available credits is selected. Credits are communicated out-of-band, so they do not interfere with normal traffic.

Tori are built using radix-5 switches. Four ports are regular transit ports, and the fifth is an interface with the node. We assume that the consumption interface is wide enough to allow for the simultaneous consumption of several packets arriving from different ports. The network relies on bubble flow control [10] to avoid deadlock, making use of two virtual channels: one escape channel in which routing is oblivious, and an adaptive, minimal routing channel.

4 Experiments and Analysis of Results

Results of the experiments are depicted in Fig 2. Execution times (actually, communication times) in cycles, as reported by the simulator, are normalized to the best performing task placement, in order to highlight the differences between the different placement strategies. We want to remark that we are not betting for a single, *miraculous* task placement which performs best for all possible applications. In fact, we will see that there are applications that are not responsive to task placement, or even to the underlying topology. Plots do not allow for a direct comparison of topologies (because values are not absolute) but, as we stated before, this is not the focus of this paper.

For the smallest networks (64-node networks and single application instance) both in torus and fat-tree, differences between the best and the worst performing placement strategy can reach a 250%. This is very significant for such a small network. In general, the random placement yields the worst results (there are exceptions), consecutive placement is the best performer for the fat-tree, and row and column placements perform equally well for the torus.

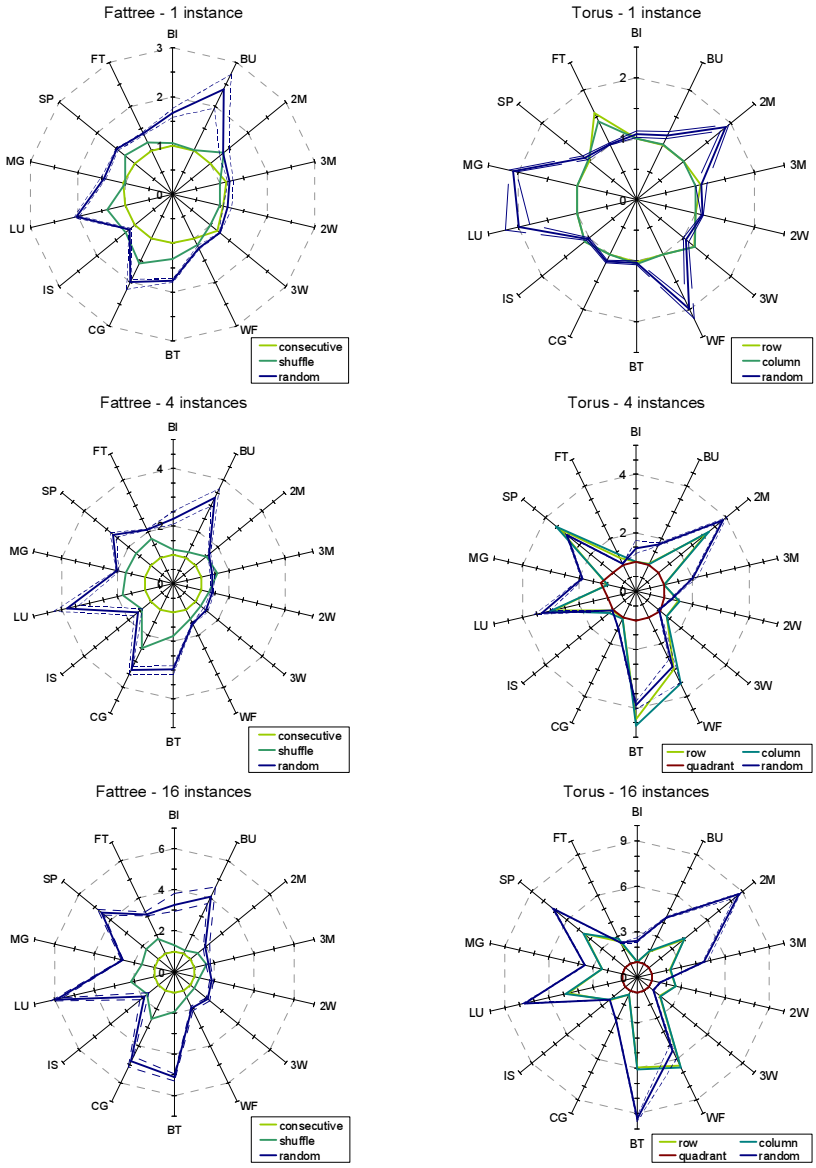

Fig. 2. Results of the experiments with different networks, network sizes and workloads. They are normalized, in such a way that a 1 represent the execution time for the best placement. Dotted blue lines represent best and worst results for random placement.

For the medium size configuration (256-node networks and 4 concurrent application instances), the worst-to-best ratio grows up to over 300%, reaching 400% and 450% for the most adverse cases (**LU** in fat-tree and **BT** in torus, respectively). Again, consecutive placement is the best one for the fat-tree network. For the torus,

the best strategy is quadrant placement, with the single exception of **MG**, for which row and column placements work equally well.

Finally, for the largest systems in our evaluation (1024-node networks and 16 concurrent application instances) in the fat-tree the ratio for some of the patterns is around 500% and reaches 600% for the most unfavorable cases. In the case of the torus, the ratios go even higher, being around 700%, and reaching 850% in the worst case. The best placement options are those described for the medium size case. In general, the negative impact of a bad placement depends heavily on the network size – more exactly, on network distance, which depends on height (number of levels) of the fat-tree and on the length of the rings of the torus.

If we focus on applications, we can see how **LU**, **BT** and **SP** are very sensitive to task placement, regardless of the topology. This is because their communication patterns cause a significant degree of contention for resources. The interferences between communications from different instances worsen this contention, which in turns increase even more the communication time. On the contrary, **2W** and **3W** are the workloads less responsive to task placement or topology. This is because the high degree of causality intrinsic to their traffic patterns does not saturate the network; thus, in the absence of contention for resources, message delay depends slightly on distance, so the short differences.

It is interesting to observe how some workloads are very sensitive to placement when running on one topology, but not that much when the network is different. Extreme examples are **2M** and **BU**. The former adjust perfectly to a mesh topology, and can take full advantage of this situation if the placement allows it. However, **2M** does not map naturally on a fat-tree, so the choice of placement on this network is irrelevant. Regarding **BU**, the perfect marriage between this pattern and the fat-tree can be exploited only with the consecutive placement, and works on the torus equally well (or bad) with any placement. For more detailed explanation of this effect, see [8].

Moreover, if we focus on the plots for single instance experiments, we can see how consecutive allocation strategies are not always the best performers. Let us pay attention to results of **FT** in the torus. Row and column placements perform worse than random placement. It happens that the allocation strategies we tested are not optimal for this pattern, because its regularity leads to the occurrence of highly congested *hot paths*. Random allocation scatters these contention spots around the network, thus its performance is better. For the multi-instance experiments with **FT**, the quadrant allocation of jobs avoids harmful interferences between instances, an effect that overshadows the bad task allocation. We would expect better results if we performed the same quadrant job allocation, but with a better task allocation inside each quadrant.

To summarize this analysis, we can state that the choice of placement has a very relevant impact on performance. The best performing placements are those that allow for a good matching between the virtual topology (that of the application) and the physical one, because this way communication locality can be exploited effectively. With very few exceptions, the random placement is the worst performer. The actual benefit of a placement strategy depends heavily on the application and the network topology, but our analysis shows that the assumption of a flat network embedded in many schedulers is too simplistic and must be reconsidered in order to accelerate the execution of applications.

We want to remark again that results presented in this work have been obtained under the assumption of infinite-speed processors. Parallel applications pass through computation phases, in addition to communication phases. Our experiments show how communication can be improved using a good placement, but computation is not directly affected by placement. So, the actual impact of placement on execution speed would depend on the communication/computation ratio of the application. In other words, the benefits we announce for good placement strategies will be diluted when running actual applications on actual machines. For example, for a 10:1 computation-communication ratio, a 450% deceleration in communication time will increase total execution time over 30%.

5 Conclusions and Future Work

Most parallel applications rely on different virtual topologies to arrange their tasks, and communication is usually performed between neighboring tasks. When the actual network has a topology that matches the virtual one, application performance is boosted. Otherwise the interchange of messages is not done in an optimal way, and performance suffers. Even when virtual and actual topologies are similar, a task allocation mechanism that does not allow for a good matching results in the impossibility of efficiently exploiting network characteristics.

In this paper we have studied the impact of job and task placement strategies on the time parallel applications spend interchanging messages. To do so, we have carried out a simulation-based study with two kinds of workloads: traces from applications and application-inspired synthetic traffic. We have focused our study on two different network topologies widely used in current supercomputers: tori and fat-trees. We have used some simple placement strategies, as well as random placement.

Results show that for a small 64-node network in which we run just one application, for almost half of the workloads the difference in speed between the worst and the best placement is around 200%. When increasing the number of concurrent application instances and the size of the network, these differences are more noticeable, reaching improvements in excess of 300% for 256-node networks, and close to 1000% for 1024-node networks. These improvements are only for the communication phases of the applications, being the computation phases unaffected by placement.

We conclude that the inclusion of locality-aware placement policies within scheduling tools could boost parallel application performance. The way to carry out this inclusion is still a line of research. We plan to apply machine-learning approaches to this issue in order to decide the degree of responsiveness to task placement of an application and, if necessary, find appropriate placements for them even when sharing a parallel computer with other jobs. Initial results show that dividing a network in sub-networks with the same topology result in excellent performance, especially when these networks match the virtual topologies used within applications. Still, both the pros and cons of this approach have to be considered, because the effort required to allocate an optimal sub-network may surpass the possible performance drop derived from a simple, random allocation.

The work described in this paper is focused on parallel applications running on high performing computing systems, and on the kind of interconnection networks used in them. However, the effects of efficiently exploiting locality could be even more noticeable when using a hierarchy of networks. Let us consider a cluster of multiprocessors. In this machine, the communication time within an on-chip network is smaller than that of the external node-to-node network, so if communicating tasks are located in the same node, the execution time should be improved. Furthermore, if the computing resource is a grid of clusters, the cluster-to-cluster communication links are orders of magnitude slower than the other networks, so the allocation of processors for the tasks of a job must avoid the utilization of these links.

References

1. R Ansaloni, "The Cray XT4 Programming Environment". Available at: www.csc.fi/english/csc/courses/programming_environment
2. Y Aridor et al. "Resource allocation and utilization in the Blue Gene/L supercomputer". IBM J. Res. & Dev. Vol. 49 No. 2/3 March/May 2005. Available at: http://www.research.ibm.com/journal/rd/492/aridor.pdf
3. JJ Dongarra, HW Meuer, E Strohmaier. "Top500 Supercomputer sites". November of 2007 edition. Available at: http://www.top500.org/
4. Y Liu, X Zhang, H Li, D Qian. "Allocating Tasks in Multi-core Processor based Parallel System". 2007 IFIP International Conference on Network and Parallel Computing Workshops (NPC 2007), September 2007 pp. 748-753.
5. V Lo, KJ Windisch, W Liu, B Nitzberg "Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers", IEEE Transactions, on Parallel and Distributed Systems, July 1997 (Vol. 8, No. 7) pp. 712-726.
6. DH Miriam, T Srinivasan, R Deepa. "An Efficient SRA Based Isomorphic Task Allocation Scheme for k-ary n-cube Massively Parallel Processors". International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06), September 2006 pp. 37-42.
7. NASA Advanced Supercomputing (NAS) division. "NAS Parallel Benchmarks" Available at: http://www.nas.nasa.gov/Resources/Software/npb.html
8. J Navaridas, J Miguel-Alonso, FJ Ridruejo. "On synthesizing workloads emulating MPI applications". The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08). April 14-18, 2008, Miami, Florida, USA.
9. J Navaridas, J Miguel-Alonso, FJ Ridruejo, W Denzel "Reducing Complexity in Tree-like Computer Interconnection Networks". Technical report EHU-KAT-IK-06-07. Department of Computer Architecture and Technology, The University of the Basque Country. Submitted to IEEE Transactions on Parallel and Distributed Systems.
10. V Puente, et al, "The Adaptive Bubble router", Journal on Parallel and Distributed Computing, vol 61, Sept. 2001.
11. FJ Ridruejo, J Miguel-Alonso. "INSEE: an Interconnection Network Simulation and Evaluation Environment". Lecture Notes in Computer Science, Volume 3648 / 2005 (Proc. Euro-Par 2005).
12. F.J. Ridruejo, et al. "Realistic Evaluation of Interconnection Network Performance at High Loads". The International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), Adelaide, December 3-6, 2007.
13. V Subramaniy, et al. "Selective Buddy Allocation for Scheduling Parallel Jobs on Clusters". Fourth IEEE International Conference on Cluster Computing, (CLUSTER'02), September 2002 pp. 107.