Effects of Job and Task Placement on Parallel Scientific Applications Performance

Javier Navaridas, Jose A. Pascual, Jose Miguel-Alonso
Department of Computer Architecture and Technology
The University of the Basque Country UPV/EHU
P.O. Box 649, 20080 San Sebastián, SPAIN
{javier.navaridas, ja-pascual, j.miguel}@ehu.es

***Abstract*—this paper studies the influence that task placement may have on the performance of applications, mainly due to the relationship between communication locality and overhead. This impact is studied for torus and fat-tree topologies. A simulation-based performance study is carried out, using traces of applications and application kernels, to measure the time taken to complete one or several concurrent instances of a given workload. As the purpose of the paper is not to offer a miraculous task placement strategy, but to measure the impact that placement have on performance, we selected simple strategies, including random placement. The quantitative results of these experiments show that different workloads present different degrees of responsiveness to placement. Furthermore, both the number of concurrent parallel jobs sharing a machine and the size of its network has a clear impact on the time to complete a given workload. We conclude that the efficient exploitation of a parallel computer requires the utilization of scheduling policies aware of application behavior and network topology.**

Keywords-interconnection networks; parallel job scheduling; performance characterization; resource allocation; trace-driven simulation.

I. INTRODUCTION

Current high-performance computing facilities are composed of thousands of computing nodes executing jobs in parallel. An underlying interconnection network (such as Myrinet [13], Infiniband [6], Quadrics [19], or an *ad-hoc* network) provides a mechanism for tasks to communicate. Most of these facilities belong to national laboratories or supercomputing centers, and are shared by many researchers (see the Top500 list, [5]). However, it is very uncommon to dedicate 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. For example, in [17] authors describe the scheduling mechanisms in use in the supercomputers of the Numerical Aerospace Simulation (NAS) supercomputer facility, located at NASA Ames Research Center.

Supercomputing sites have one or more job queues 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 (see [17] again), typically taking into account some restrictions: different levels of priority, quotas (both in terms of CPU time and number of processors per job), maximum waiting time, 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. We will discuss in the next section how schedulers assign free nodes—*i.e.* 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 [2]. 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 arbitrary, 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 two most commonly used topologies: fat-tree and torus. To do it so, we carried out a simulation-based performance study in which we fed the networks with different application-like workloads: traces, and synthetic traffic patterns that closely emulate the behavior of actual applications. We tested networks of different sizes, and we explored several alternatives of task allocations for a single parallel job, and job and 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.

To our knowledge, there is no previously published work measuring the interactions between parallel job schedulers, application's sensitivity to placement, and network topology. This is a gap we aim to start filling, with an exploration of the impact that task placement have on the execution time of parallel applications running on supercomputers with different interconnection networks (tori and fat-trees). We will show how some applications are insensitive to placement, but many others run very efficiently under certain topology/placement combinations. The natural continuation of this work will be the inclusion of topology-aware policies in parallel job schedulers.

The rest of this paper is organized as follows. In Section II 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 III. In Section IV we show and analyze the results of the experiments. Section V closes this paper with some conclusions and an outlook of our plans for future work.

II. 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, allowing contiguous allocation of jobs in the parallel system, which can be taken advantage by applications.

In [23] we can see how the contiguous allocation of tasks resulted in improved application performance. Authors run eight sets of 16-node FFTs—benchmark FT, part of the well-known NAS Parallel Benchmarks [14]—concurrently on a 128-node mesh, and compared 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 some other papers addressing this issue [3, 9, 10, 12, 23] allocation algorithms were proposed mainly for *k*-ary *n*-cube topologies. Figures of merit usually did not show how placement strategies affect the runtime of an application instance, but just the completion time of a list of jobs. In [16] we paid attention to tree-based topologies and relied 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 [10] authors showed how the requirement of contiguous allocation may cause poor utilization of the system due to external or internal fragmentation. To avoid this effect, they evaluated several non-contiguous, but non-random, allocation schemes that improved 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 take care of the actual placement of tasks. This is true for job queuing systems and scheduling managers such as Sun's Grid Engine [24], IBM's LoadLeveler [7] or PBS Pro [18] (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 [4] and Slurm [8], in use in ASC Purple (IBM 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, i.e. distance between nodes is considered as the difference between node identifiers. The most notable example of current supercomputer that tries to maintain locality when allocating resources is the BlueGene family (3D tori), whose scheduler [3] puts tasks from the same application in one or more midplanes of 8x4x4 nodes.

III. Experimental Set-Up

We used simulation to assess the impact of allocation strategies on application performance. The simulation environment encompasses a network simulator and a workload generator [21]; 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.

A. Workloads

Throughout this work we evaluate networks using realistic workloads, taken from actual or emulated applications. In particular, we used traces taken from the well-know NAS Parallel Benchmarks [14] (NPB), and a set of application kernels described in [15]. In both cases we assumed *infinite-speed* processors, meaning that we only measured 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—we encourage reader to examine [11] for a deeper explanation of our methodology to perform trace-driven simulation. Under these assumptions, reported results only toke into account the communication and synchronization parts of parallel applications; thus, the actual impact on performance of a given scheduling algorithm would depend on the application's computation to communication ratio.

Regarding traces, we used class A of NPB applications Block Tridiagonal (**BT**), Conjugate Gradient (**CG**), Integer Sort (**IS**), Lower-Up diagonal (**LU**), Multi-Grid (**MG**), Scalar Pentadiagonal (**SP**) and Fourier Transform (**FT**). This study did not include Embarrassingly Parallel (**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 reduced the number of iterations in each benchmark, to only 10 to 20 iterations.

Pseudo-synthetic workloads used in this work are binary-tree (**BI**), butterfly (**BU**), distribution in 2D or 3D meshes (**2M**, **3M**) and wave-front in 2D or 3D meshes (**2W**, **3W**), which are detailed and justified in [15]. The message length was 64 Kbytes. Our experimental set-up also included waterfall (**WF**), a pattern observed in the LU NPB application [22]. We modeled this pattern as a burst of 286 wave-fronts

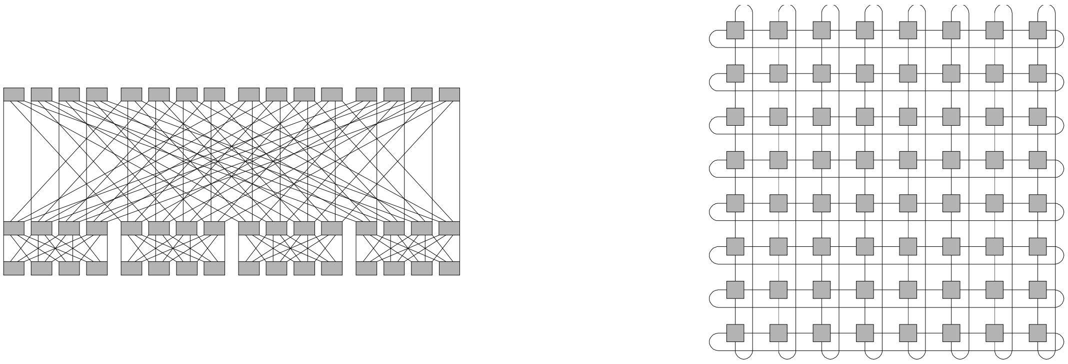

Figure 1. Examples of the topologies used in this study. 4,3-fat-trees (*left*) and 8-ary 2-cube (*right*), both used to interconnect 64 nodes.

(**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 had 64 nodes, so a single application uses the whole computer. In others, the network had 64**N* nodes, so *N* instances of an application shared the computer. Chosen values of *N* were 4 and 16. To simplify the experiments, we never mixed different applications. The figure of merit to measure performance was 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).

B. 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 selected two of the most widely used topologies: 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 [5] to see how most computers in the highest positions of the list fit on one of these categories.

In our experiments we used small to medium-size networks, with a number of nodes ranging from 64 to 1024. Given these sizes, we considered only 2D cubes (3D would be recommended for large-scale networks). In order to allow workloads to fit exactly in the network, we 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 (*i.e.* 8x8 torus) for experiments with a single application instance. Both topologies are depicted in Figure 1.
- 4-ary, 4-tree and 16-ary, 2-cube (*i.e.* 16x16 torus) for experiments with 4 instances of the application.
- 4-ary, 5-tree and 32-ary, 2-cube (*i.e.* 32x32 torus) for experiments with 16 instances of the application.

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. Our focus is on the impact that placement have on the execution time of applications of different sizes, running alone or sharing a parallel computer with other applications.

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*". Switch numbers correspond, left to right, to the lowest level of the tree, the one 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 alternatives only for the experiments with a single application instance. In the other cases we evaluate combinations of task and job placement strategies. Now we describe these strategies. In all cases, we assume that assignment is done first in order using the job identifier and that, for a given job, nodes are assigned to task in order of task identifier. In the case of the 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*.

Allocation for the torus 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.
- 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.

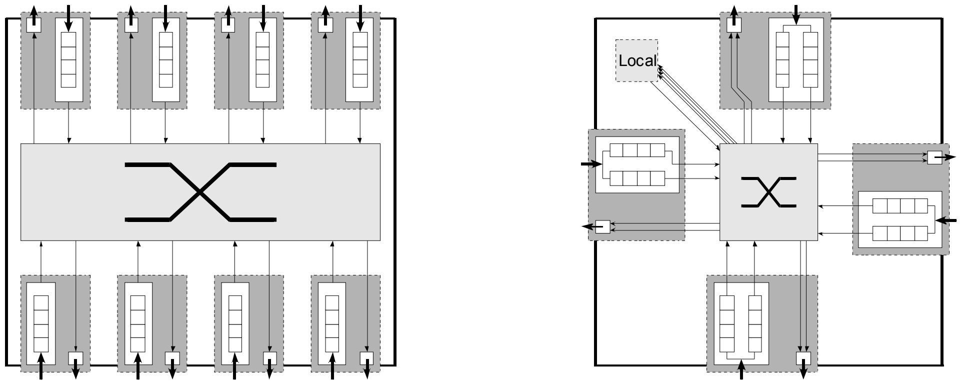

Figure 2. Model of the switches used to build fat-trees (*left*) and 2D tori (*right*). All the ports are depicted showing input queues and output buffers. Note the utilization of two virtual channels sharing a physical link in the torus switch.

Both for 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 generated five random permutations and plotted the average, maximum and minimum values of the measured execution times.

C. *Models of the components*

Nodes were modeled as reactive traffic sources/sinks with an injection queue able to store up to four packets. In order to model causality, the reception of a message may trigger the release of one or several extra messages as defined by the workloads. When 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 were used. Transit queues had room to store up to four packets. The output port arbitration policy was round robin. Switching strategy was virtual cut-through. We depicted the models of switches for the two topologies in Figure 2.

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

Tori were built using radix-5 switches. Four of the ports were regular transit ports, and the fifth one was an interface with the node. We assumed that the consumption interface was wide enough to allow simultaneous consumption of several packets arriving from different ports. The network relied on bubble flow control [20] to avoid deadlock, making use of two virtual channels: one escape channel in which routing is oblivious DOR (Dimension Order Routing), and an adaptive, minimal routing channel.

IV. EXPERIMENTS AND ANALYSIS OF RESULTS

Results of the experiments are depicted in Figure 3. Execution times (actually, communication times) in cycles, as reported by the simulator, were 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 some applications were 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 a single application instance) both in torus and fat-tree, differences between the best and the worst performing placement strategy reached a 250%. This is very significant for such a small network. In general, although there were exceptions, the random placement yielded the worst results, consecutive placement was the best performer for the fat-tree, and both row and column placements performed equally well in the torus topology.

For the medium size configurations (256-node networks and 4 concurrent application instances), the worst-to-best ratio grew up to over 300%, reaching 400% and 450% in the most adverse cases (**LU** in fat-tree and **BT** in torus, respectively). Again, consecutive placement was the best performer in the fat-tree network. In the case of the torus, the best performing strategy was 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 was around 500% and reaches 600% in the most adverse cases. In the case of the torus, these ratios went even higher, being around 700%, and reaching 850% in the worst case. The best placement options were 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, that depends on the 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** were 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 increased even more the communication time. On the other hand, **2W** and **3W** were 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. **IS** also showed not being very sensitive to the placement regardless of the topology.

It is interesting to observe that some workloads were very sensitive to placement when running on one topology, but not that much when the network was different. Extreme examples are **2M** and **BU**. The former adjust perfectly to a mesh topology, and was able to take full advantage of this situation when the placement allowed it. However, **2M** does not map naturally on a fat-tree, so the choice of placement on this network was almost irrelevant. Regarding **BU**, the perfect marriage between this pattern and the fat-tree was exploited only with the consecutive placement, and worked on the torus equally well (or bad) with any placement. For a more detailed explanation of this effect, the interested reader can see [15].

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

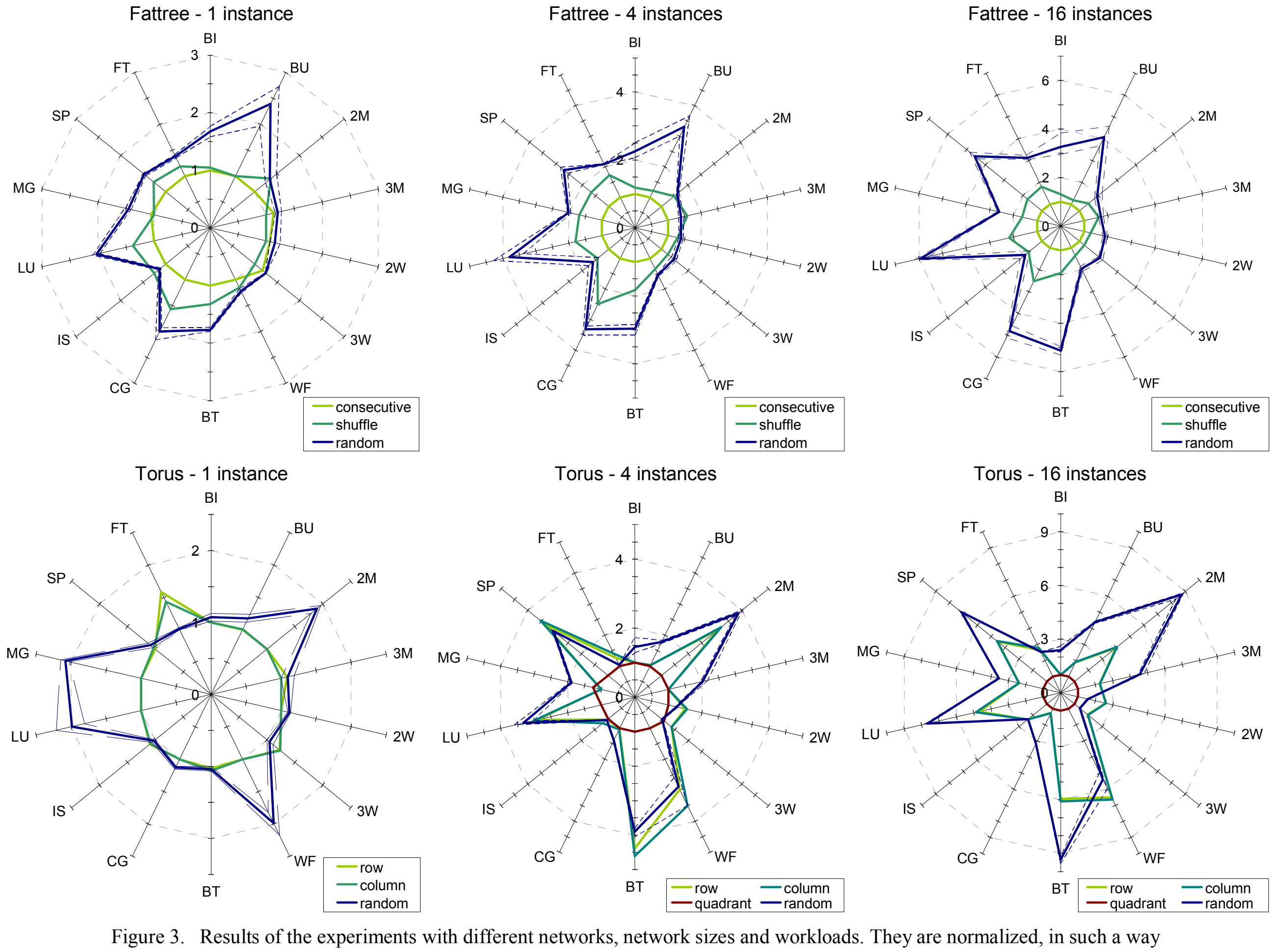

Figure 3. Results of the experiments with different networks, network sizes and workloads. They are normalized, in such a way that 1 represents the execution time for the best placement. Dotted blue lines represent best and worst results for random placement.

To summarize this analysis, we can state that the choice of placement has a very relevant impact on performance on some applications, a fact that should not be taken lightly. Other applications are insensitive to placement, and could be used to fill fragmented gaps of the network, in order to increase system utilization with a minimal impact on the overall performance.

The best performing placements were those that allow for a good matching between the virtual topology (spatial distribution of application's communication) and the physical one, because this way communication locality can be exploited effectively. With very few exceptions, the random placement was the worst performer. The actual benefit of a placement strategy depends heavily on the application and the network topology, but our analysis showed that the flat-network supposition 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 the results presented in this work were obtained under the assumption of infinite-speed processors. Parallel applications pass through computation phases, in addition to communication phases. Our experiments showed how communication can be improved using a good placement; however, computation is not directly affected by placement. Therefore, 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% increase in communication time will increase total execution time over 30%, which in our opinion still makes worthwhile to continue doing research on this topic

V. 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 topology of the physical network matches the virtual one, application performance boosts. 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 a good matching between them will result in the impossibility of efficiently exploiting the potential of the network.

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

Results showed 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 was around 200%. When increasing the number of concurrent application instances and the size of the network, these differences were more noticeable, reaching increases in excess of 300% for 256-node networks, and close to 1000% for 1024-node networks. The obtained improvements are only applicable for communication phases of the applications, being the computation phases unaffected by placement.

In contrast, and depending on the host topology, some applications or kernels were shown to be only slightly sensitive to task placement. In these cases, the effort of looking for a consecutive region of the network will not pay off. The positive side is that, when mixing different applications, the placement-insensitive ones are good candidates to be used to fill gaps that would otherwise remain unused while placement-sensitive applications are waiting for a consecutive portion of the system.

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 different optimization techniques in order to decide the degree of responsiveness to task placement of an application. If this degree is low, we can use the application to fill fragmented regions of the network. Alternatively, if an application is sensitive to placement, we should find appropriate placement for it, even when sharing a parallel computer with other jobs.

Initial results showed that dividing a network in subnetworks 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.

Acknowledgments

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

REFERENCES

[1] R Ansaloni, “The Cray XT4 Programming Environment”. Slides available (November 2008) at: http://www.csc.fi/english/csc/courses/programming_environment

[2] Y. Aoyama and J. Nakano. "RS/6000 SP: Practical MPI Programming". IBM Red Books SG24-5380-00, ISBN 0738413658. August, 1999.

[3] Y Aridor, T Domany, O Goldshmidt, JE Moreira and E Shmueli “Resource allocation and utilization in the Blue Gene/L supercomputer”. IBM J. Res. & Dev. Vol. 49 No. 2/3 March/May 2005. Available (November 2008) at: http://www.research.ibm.com/journal/rd/492/aridor.pdf

[4] Cluster Resources. “Maui Admin Manual”. Available (November 2008) at: http://www.clusterresources.com/products/mwm/moabdocs/MoabAdminGuide52.pdf

[5] JJ Dongarra, HW Meuer and E Strohmaier. “Top500 Supercomputer sites”. Available (November 2008) at: http://www.top500.org/

[6] Infiniband Trade Association. “Infiniband® Trade Asociation”. Available (November 2008) at: http://www.infinibandta.org

[7] S Kannan, M Roberts, P Mayes, D Brelsford and JF Skovira “Workload Management with LoadLeveler”. IBM Red Books SG24-6038-00. ISBN 0738422096. November 2001.

[8] Lawrence Livermore National Laboratory. “Simple Linux Utility for Resource Management”. Available (November 2008) at: https://computing.llnl.gov/linux/slurm/

[9] Y Liu, X Zhang, H Li and D Qian. “Allocating Tasks in Multi-core Processor based Parallel System”. 2007 IFIP International Conference on Network and Parallel Computing Work-shops (NPC 2007), September 2007 pp. 748-753.

[10] V Lo, KJ Windisch, W Liu and 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. DOI: 10.1109/71.598346

[11] J. Miguel-Alonso, J. Navaridas and F.J. Ridruejo. “Interconnection network simulation using traces of MPI applications”. International Journal of Parallel Programming, in press. DOI: 10.1007/s10766-008-0089-y

[12] DH Miriam, T Srinivasan and 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.

[13] Myricom. “Myrinet home page”. Available (November 2008) at: http://www.myri.com/

[14] NASA Advanced Supercomputing (NAS) division. “NAS Parallel Benchmarks” Available (November 2008) at: http://www.nas.nasa.gov/Resources/Software/npb.html

[15] J Navaridas, J Miguel-Alonso and 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.

[16] J Navaridas, J Miguel-Alonso, FJ Ridruejo and 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 Elsevier’s Journal of Parallel Computing

[17] J Patton-Jones and B Nitzberg. “Scheduling for Parallel Supercomputing: A Historical Perspective of Achievable Utilization.” In Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science 1659, pages 1-16. Springer-Verlag, 1999.

[18] PBS GridWorks. “PBS Pro”. Available (November 2008) at: http://www.pbsgridworks.com/

[19] F Petrini, W Feng, A Hoisie, S Coll and E Frachtenberg. “The Quadrics Network: High-Performance Clustering Technology”. IEEE Micro 22, 1 (Jan. 2002), 46-57. DOI: 10.1109/40.988689

[20] V Puente, C Izu, R Beivide, JA Gregorio, F Vallejo and J M. Prellezo “The Adaptive Bubble router”, Journal on Parallel and Distributed Computing, vol 61, Sept. 2001. DOI: 10.1006/jpdc.2001.1746

[21] FJ Ridruejo and J Miguel-Alonso. “INSEE: an Interconnection Network Simulation and Evaluation Environment”. Lecture Notes in Computer Science, Volume 3648 / 2005 (Proc. Euro-Par 2005).

[22] FJ Ridruejo, J Navaridas, J Miguel-Alonso and C Izu “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.

[23] V Subramani, R Kettimuthu, S Srinivasan, J Johnson and, P Sadayappan “Selective Buddy Allocation for Scheduling Parallel Jobs on Clusters”. Fourth IEEE International Conference on Cluster Computing, (CLUSTER'02), September 2002 pp. 107.

[24] Sun Microsystems, Inc. “N1 Grid Engine 6 User’s Guide”. Available (November 2008) at: http://docs.sun.com/app/docs/coll/1017.3.