Effects of Job Placement on Scheduling Performance

Technical Report EHU-KAT-IK-09-08

Jose Antonio Pascual[1], Jose Miguel-Alonso[1]

Abstract—**This paper studies the influence that job placement may have on scheduling performance, in the context of massively parallel computing systems. A simulation-based performance study is carried out, using workloads extracted from real systems logs. The starting point is a parallel system built around an *n*-ary *k*-tree network and using well known scheduling algorithms (FCFS and backfilling). We incorporate allocation policies that try to assign to each job a contiguous network partition, in order to improve communication performance. These policies result in severe scheduling inefficiency due to increased system fragmentation; however, experiments show that, in those cases where the exploitation of communication locality results in an effective reduction of execution time, the achieved gains more than compensates the scheduling inefficiency, thus resulting in better overall performance.**

Keywords—**Scheduler Performance, Locality-Aware Allocation Policies, Scheduling Techniques, Trace-Driven Simulation.**

I. Introduction

SUPERCOMPUTER centres are usually designed to provide computational resources to multiple users and multiple applications. User jobs are sent to a scheduling queue, where they wait until the resources required by the job are available. These jobs may vary from large parallel programs that need many processors, to small sequential programs. The scheduler manages system resources, taking into consideration different policies that may restrict its use in terms of maximum number of processors, maximum execution time, user or group priorities, etc.

Generally, site performance is measured in terms of the utilization of the system and the slowdown that the jobs suffer while waiting in the queue until the required resources become available. This is the reason why a variety of scheduling policies [1] and allocation algorithms [2] [3] [4] have been developed with the aim of minimizing the number of nodes that remains idle, and also the job waiting times. The scheduling policy decides the order in which jobs are allowed to run. Scheduling decisions may be based on different variables, such as job size, user priority or system status. Allocation algorithms map jobs onto available resources (typically, processors). Locality-aware policies select resources taking into account network characteristics, such as its topology or the distance between processors.

The most commonly used scheduling policies are FCFS (First-Come First-Serve) and FCFS+backfilling, sometimes with variations. The FCFS discipline imposes a strict order in the execution of jobs. These are ordered by their arrival time and order violations are not permitted, even when resources to execute the first job are not available but there are enough free resources to execute some other (or others) in the queue. The main inconvenience of this policy is that it produces severe system fragmentation because some processors can remain idle during a long time due to the sequential execution of jobs. This time could be used more effectively running less-demanding jobs, thus achieving a performance improvement.

With the goal of minimizing the effect of this strictly sequential execution order, several strategies have been developed [1] being the backfilling policy the most used due to its easy implementation and proven benefits. This policy is a variant of FCFS, based on the idea of advancing jobs through the head of the queue. If some queued jobs require a lower amount of processors than the one at the head, we can execute them until the resources required by the job at the head become available. This way utilization of resources is improved because both network fragmentation and job slowdown decreases. The reader should note that, throughout this paper, we will often use the word network to refer to the complete parallel system.

Network fragmentation caused by scheduling algorithms is known as *external fragmentation* [5]. But a different kind of fragmentation appears in topologies like meshes or tori when the partitions reserved to jobs are organized as sub-meshes or sub-tori; for example, to allocate a job composed by 4x3 processes, some algorithms search for square sub-meshes, being 4x4 the smallest size that can be used to run our job. In this case, four processors reserved for the job will never used. This effect is named *internal fragmentation* [5]. Some job allocation algorithms try to minimize this effect.

Neither FCFS nor backfilling are allocation algorithms, and they do not take into account the placement of job processes to network nodes. In a parallel system, application processes (running on network nodes) communicate interchanging messages (network packets). Depending on the communication pattern of the application, and the way processes are mapped onto the network, severe delays may appear due to network contention; delays that result in longer execution times. If we have several parallel jobs running in the same network, each of them located randomly, communication locality inside each job will not be exploited; furthermore, messages from different

[1] The University of the Basque Country UPV/EHU, Department of Computer Architecture and Technology,
(contact e-mail: ja-pascual@ehu.es, j.miguel@ehu.es).

applications will compete for network resources, boosting contention. An effective exploitation of locality results in smaller communication overheads, which reflects in lower running times. Note that searching for this locality is expensive in terms of scheduling time, because jobs cannot be scheduled until contiguous resources are available (and allocated), so that network fragmentation increases.

A trade-off has to be found between the gains attainable via exploitation of locality and the negative effects of increasing fragmentation. This is precisely the focus of this paper. We study only the placement in *k*-ary *n*-tree topologies [14], but the tools and methodology presented here could be extended to topologies like rings, meshes or tori. Our final goal is to demonstrate that the introduction in the schedulers of locality-aware policies may provide important performance improvements in systems with multiple users and different applications.

The rest of the paper is organized as follows. In section II we discuss some previous work on scheduling and allocation policies. The simulation environment and the workloads are described in Section III. In Section IV we show and analyze the results of experiments finishing the paper with some conclusions and future lines of research.

II. Related Work

Extensive research has been conducted in the area of parallel job scheduling. Most works are focused on the search of new scheduling policies that minimize job waiting times, and on allocation algorithms that minimize network fragmentation. In [1] authors analyze a large variety of scheduling strategies; however, none of them take into account virtual topologies of applications (the logical way of arranging processes to exploit communication locality) and network topology.

To our knowledge, only [5] describes a performance study of parallel applications taking into account locality-aware allocation schemes. The starting point of this job is the fact that, in schedulers optimized for machines with certain network topologies (they focus on meshes and tori), allocation was always done in terms of sub-meshes (or sub-tori). This policy optimizes communication in terms of locality and non-interference, but causes severe fragmentation, both internal and external. Authors do not use scheduling with backfilling, a technique that would partly reduce this undesirable effect. However, they test a collection of allocation strategies that sacrifice contiguity in order to increase occupancy. They claim that the effect on application performance attributable to the loss of contiguity is low, and more than compensated by the overall improvement in system utilization.

Our work differs from the cited one in several important aspects. Previous research work shows that, depending on the communication pattern of the application, contiguous allocation provides remarkable performance improvements [6]. Therefore, we do not make extensive use of non-contiguity to increase network utilization; instead, we incorporate backfilling into the scheduler. Additionally, we focus on *k*-ary *n*-trees, instead of meshes.

A review of schedulers in use in current supercomputers, such as Maui [7] or SLURM [8], shows that they do not implement contiguous allocation strategies. Some of these provide methods for the system administrator to develop their own strategies but, in practice, this is rarely done. To our knowledge, the only scheduler that tries to maintain locality is the one used by the BlueGene/L supercomputers [9]. This is done in order to reduce network fragmentation and decrease network contention.

III. Experimental Set-Up

We have used simulation to carry out an analysis of the impact that contiguous allocation strategies have on scheduling performance. Our simulator implements two different scheduling policies (FCFS with and without backfilling), as well as two allocation algorithms, one of them designed to search for contiguous resources *k*-ary *n*-trees. The traces (workloads) used to feed the simulations have been obtained from actual supercomputers. They are available at the Parallel Workload Archive [10].

The details of the scheduling algorithms used in the experiments are as follows:

FCFS: In this policy, jobs are processed in strict order of arrival and executed when the resources that they need are available. Until this condition is reached, the scheduling process is stopped, even if there are enough free resources that could be allocated to other waiting jobs.

Backfilling: This strategy permits the advance of jobs, even when they are not at the head of the queue, in such a way that network utilization increases, but without delaying the execution of the jobs that arrived first. The mechanism works as follows. A reservation for the first job in the queue is done, if enough resources are not currently available; the reservation time is computed taking into account the estimated termination times of currently running jobs. Other waiting jobs demanding fewer resources may be allowed to run while the first one is waiting. When the time of the reservation is reached, the waiting job has to run; if at that point resources are not available, some running, advanced jobs must be killed, because otherwise the reservation would be violated, and to avoid the starvation of the first job. Reservations are computed using a parameter called User Estimated Runtime, which represents an estimation of the job execution time ant that is provided by the users [11]. In some cases the scheduling system itself may provide this value, based on estimations made over the historical system logs [12].

Other scheduling methods have been proposed in the literature, such as SJF (Shortest Jobs First) [1] in which the jobs are sorted by their size instead of their arrival time, and several variations of backfilling (see [1]). However, the algorithm most commonly used in productions systems is the EASY backfilling [1], also known as aggressive backfilling. EASY performs

reservations only over the first job in the queue. This is the policy that we use in this study.

Regarding the allocation algorithms, these have been included in the study:

Non-contiguous: This policy performs a search of free nodes making a sequential search over them, ignoring the locality. This is the most used technique in commercial systems, like the Cray XT3/XT4 systems, that simply gets the first available compute processors [13]. This scheme provides a flat vision of the network, ignoring its topological characteristics and the virtual topologies of scheduled applications [4].

Contiguous: In this scheme job processes are allocated to nodes maintaining them as close as possible. To minimize, in a *k*-ary *n*-tree, the distance between processes (nodes), we have defined the concept of level of a job. This level is related to the number of stages in the tree (*n*), and the number of ports per switch (*k* up and *k* down) [14]. Stage 1 corresponds to switches at the bottom of the tree, i.e., those directly connected to compute nodes. Small jobs of less than *k* nodes can be allocated a collection of nodes attached to the same stage-1 switch, without requiring communication using switches upper in the tree. These are level-1 jobs. However, jobs larger than *k* will require the utilization of switches at stages 2, 3, etc.

```
Input:
int k, n; /* Parameters of the k-ary n-tree: ports, stages */
int N; /* Network size (number of compute nodes */
int x; /* Number of required nodes */

Output:
list node_list; /* List of assigned nodes */
                /* empty if contiguous allocation is not found */

Variables:
int level; /* Job level */
int level_size; /* Number of nodes at a given level */
int i; /* Counter to sweep nodes through the network */
int k; /* Counter to sweep nodes under the same level */

Begin
level = 1; level_size = 0;
while ((level ≤ n) && (x > level_size)) level_size = k^level;

empty(node_list);
i = 0; k = 0;
while (i < N) {
   while ((k < level_size) && (length(node_list) < x)) {
      if (available(i)) {
         add(i, node_list);
         if (length(node_list) == x) return(node_list);
      }
      i++; k++;
   }
   k = 0;
   empty(node_list);
}
return(node_list);
```

Figure 1: Contiguous allocation algorithm

Figure 1 shows the algorithm used to search for contiguous nodes in a *k*-ary *n*-tree. The first loop computes the *level* to which the job belongs, and the size *level_size* of this *level* (the number of compute nodes below a single switch located at stage *level*). After this preliminary step, the search of free nodes is performed, in groups of *level_size* nodes, because this way all the allocated nodes would be contiguous, that is, connected by the same switch. If the complete tree is traversed but the necessary number of nodes is not found, the job cannot be allocated. For example, in a 4-ary 3-tree topology, if we need to allocate a 4-node job, we have to find a completely empty stage-1 switch. For a 14-node job (level-2) we need to find 14 free nodes all below the same switch of the second stage of the tree.

As we stated before, throughout this work we evaluate the scheduler performance using logs of workloads extracted from real systems that are available thought the PWA (Parallel Workload Archive). These logs have information about the system as described in the SWF format (Standard Workload Format) [15]. In this study we have used mainly the following fields:

Arrival Time: The time at which a job arrives to the system queue. Logs are sorted by this field.

Execution Time: The time that the job ran in the system. This field is sometimes changed in order to apply a speed-up factor, to simulate the improvement of performance due to the exploitation of communication locality.

Processors: Number of processors required by the job.

User Estimated Runtime: This information is used only by the backfilling scheduling policy and represents the time that the user estimates that the job will need to finish.

Status: This field represents the status of a job. Jobs can fail, or be cancelled by the user or by the system, before or after they started the execution. In some studies, jobs that are not completed successfully are not included in the simulations, but we consider all the jobs important because of the time they stay in the system, delaying the execution of other jobs.

In our experiments, all times are measured in minutes. We only use traces that provide User Estimate Runtime information, because of the need of this parameter to perform a backfilling scheduling policy. From those available at the PWA, we have selected these three:

HPC2N (High Performance Computing Center). This is a system located in Sweden, with 240 compute nodes. It uses the Maui [7] scheduler. The workload log contains information of 527,371 jobs.

LLNL (Lawrence Livermore National Laboratory) Thunder. This is a Linux cluster composed by 4008 processors in which the nodes are connected by a Quadrics network. The scheduler used in this system is Slurm [8]. The log is composed by 128,662 job records.

SDSC (San Diego Supercomputer Center). This system is an IBM SP located in San Diego, with 1152 processors. The scheduler in use is Catalina, developed at SDSC, and performs backfilling. The log contains information of 243,314 jobs.

We have simulated these workloads in *k*-ary *n*-trees adapted to the system size. For the first trace we have simulated a 4-ary 4-tree with 256 nodes. For the other two we have used a 4-ary 6-tree with 4096 nodes. The number of nodes of the topologies does not match with the nodes of the traces, so we have considered that the extra processors are not installed and they are ignored in the simulation.

IV. Experiments And Analysis of Results

In the experiments, we evaluate the performance of scheduling and allocation algorithms in terms of these two measurements:

Job waiting time. The time jobs spent in the queue.

Job total time. All the time spent in the system, which includes the time waiting at the queue and the execution time.

We have studied the four possible combinations of scheduling and allocation policies. When using contiguous allocation, a speed-up factor can be applied to reduce the execution time. This is a parameter of the simulation. For a given speed-up *s*, when a record in the input workload states that a job ran for *t* minutes, in the simulation we reduce this value to $t \cdot (1-s)$.

Note that adding a speed-up factor to a running time reduces not only the application finish time, but also the time that the jobs uses network resources; because of this, the scheduling performance is increased too. We try to find the point at which performance loss derived from a restrictive allocation policy is compensated by the gain derived from communication locality.

Results are depicted in Figures 3, 4 and 5. We represent the averages of total time (waiting plus running) and, in some cases, waiting time. In each graph we can see two lines, one per allocation policy. For the contiguous one we have tested several speed-up factors. When this factor is 0 it means that, although we are seeking locality, we are considering that using it does not accelerate program execution. In all other cases we accelerate the execution times reported in the logs using the indicated speed-up factors. Obviously, we cannot use speed-up factors with the non-contiguous allocation policy, and for this reason the corresponding line is flat.

Before analyzing the results, let us pay attention to Figure 2. It shows how the use of the contiguous allocation policy is not cheap, and only can be compensated if the execution speed-up due to the improved communication locality is high. In the figure we represent the waiting times for contiguous allocation, always with null speed-up. Values are normalized, so that a 1 represents the average job waiting time for the non-contiguous policy. Results are clear: at zero speed-up, for any scheduling policy, waiting times are *much* worse when using contiguous allocation – from 7 to 100 times worse.

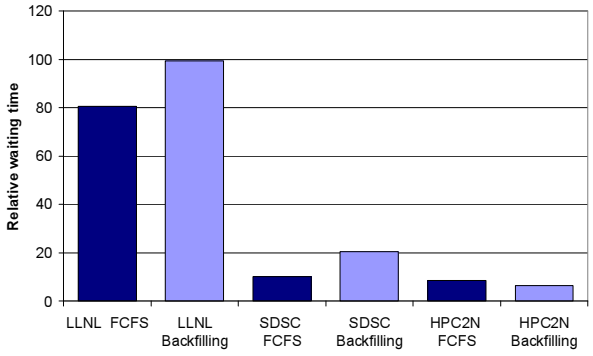

Figure 2. Measured job waiting times for the contiguous allocation algorithm. Values are normalized, so that 1 would be the time with non-contiguous allocation.

The picture presented by Figure 2 is partial, and too negative. Let us now pay attention to Figure 3, where the HPC2N workload is studied in detail.

In all scheduling-allocation combinations, results with speed-up=0 are as appalling as just described. However, when this value increases (that is, when applications really run faster when allocated consecutive resources) the picture changes. However, at speed-up values around 0.25 the picture changes, and the contiguous approach shows its potential. Also, note that if the scheduler uses backfilling, global system efficiency is higher (the workload is processed faster), but the thresholds at which contiguity is advantageous are about the same.

Figure 4 shows the results of the same experiments, but from a different perspective. Only waiting times are shown. A direct comparison with the previous figure help us to determine which part of the total time is spent in the queue, and which part is running time. For the cases with small speed-ups, most of the time is waiting time. When applying a speed-up factor running time is reduced accordingly, but waiting time is also reduced.

In Figure 5 we have summarized results for workloads LLNL and SDSC. To be succinct, and because the qualitative analysis performed with HPC2N is still valid, we only show results of total times for the backfilling scheduling algorithm. Note that, as the range of values is very wide, we have used a logarithmic scale in the X axis. For the SDSC workload, the threshold at which contiguous allocation starts being beneficial falls between 0.25 and 0.30 (higher than that of HPC2N). Similar, although slightly higher, values are required for LLNL.

In [6] authors stated that significant speed-ups can be obtained using the right application-to-network mappings. In this work we have shown that searching for these mapping incurs in severe penalties in terms of waiting times at the scheduling queue. But we have also shown that a speed-up threshold can be found at which the beneficial effects of contiguous allocation schemes surpass their cost. This threshold is different for each machine-workload, falling around 0.25 – 0.3 for the ones included in our study.

V. Conclusions And Future Work

Most current supercomputing sites are built around parallel systems shared between different users and applications. The optimal use of resources is a complex

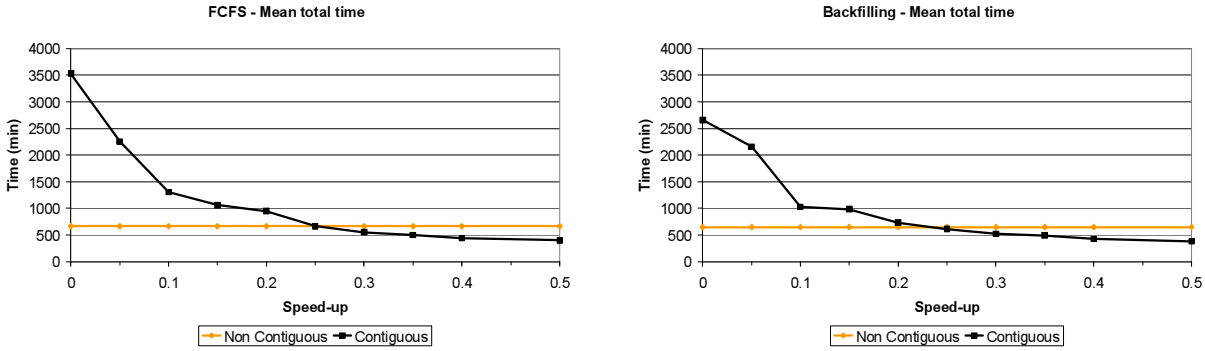

Figure 3. Results of the simulation of the HPC2N workload. Mean Total Time (Wait Time + Execution Time) for FCFS and backfilling scheduling policies at different execution speed-ups.

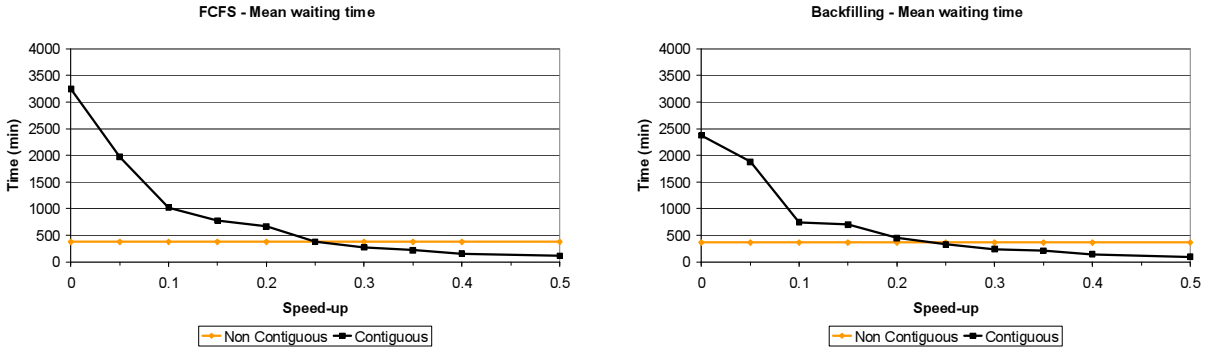

Figure 4. Results of the simulation of the HPC2N workload. Mean Waiting Time for FCFS and backfilling scheduling policies at different execution speed-ups.

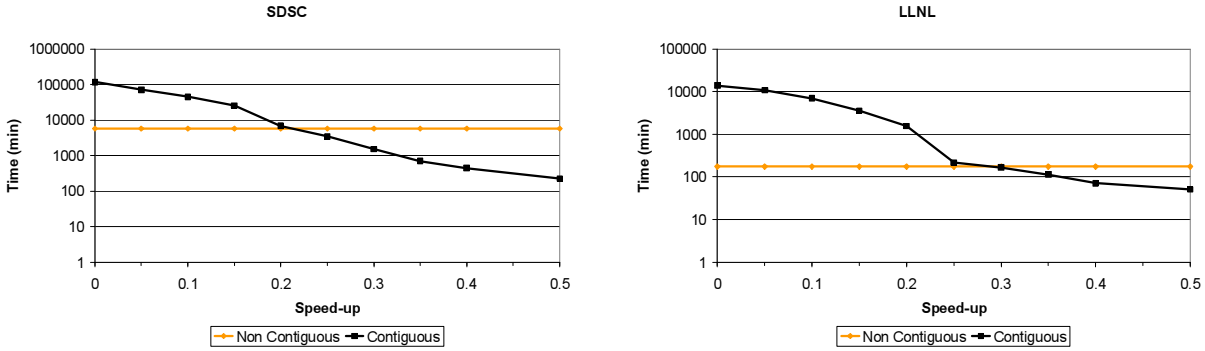

Figure 5. Results of the experiments with the SDSC and LLNL workloads for a backfilling scheduling policy. Mean Total Time (Wait Time + Execution Time) at different speed-ups. The scale of the X axis is logarithmic.

task, due to the heterogeneity in user and application demands: some users run short sequential applications, while others launch applications that use many nodes and need weeks to be completed.

Supercomputers are expensive to build and maintain, so that conscious administrators try to keep utilization as high as possible. However, the efficient use of a parallel computer cannot be measured only by the lack of unused nodes. Other utilization characteristics, although not that evident, may improve the general system performance.

In this paper we have studied the impact on performance of allocation and scheduling policies. We have compared two scheduling techniques in a *k*-ary *n*-tree network topology, combined with two allocation algorithms. Allocation algorithms that search for contiguous resources have an important cost in terms of system fragmentation, but also are able to accelerate the execution of applications.

Experiments with actual workloads demonstrate that the cost of contiguous allocation is very high, but when the improvement of run time experienced by jobs is around 25%-30%, this cost is compensated. Additional accelerations would result in a more efficient scheduling.

This study has focused only in tree-based networks. The next step will be a performance study for other topologies (*k*-ary *n*-cubes). In this work we have taken application acceleration as a simulator parameter, although we know that the real acceleration depends heavily on the communication pattern of the applications, and on the way processes are mapped onto network nodes. For this reason, we plan to perform more

complex simulations, in which the actual interchanges of messages are considered; to that end, we plan to use INSEE [16].

We also plan to introduce relaxed versions of the contiguous allocation algorithm, customized for *k*-ary *n*-trees. Under some conditions, the allocator would search for free nodes not only at the job level, but also one level above – just one, in order to relax contiguity requirements, but not too much.

Finally, we plan to implement our allocation techniques into a real (commercial or free) scheduler in order to make real measurements in production environments.

VI. Acknowledgements

Work supported by the Ministry of Education and Science of Spain (grant TIN2007-68023-C02-02) and by the Basque Government (grant IT-242-07).

VII. References

[1] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, "Parallel job scheduling - a status report," Lecture Notes in Computer Science Vol. 3277.

[2] Sandeep K.S. Gupta and Pradiip K. Srimani, "Subtorii Allocation Strategies for Torus Connected Networks". Algorithms and Architectures for Parallel Processing, 1997. ICAPP 97., 1997 3rd International Conference on Volume , Issue , 10-12 Dec 1997 Page(s):287 - 294

[3] Hyunseung Choo, Seong-Moo Yoo and Hee Yong Youn, "Processor Scheduling and Allocation for 3D Torus Multicomputer Systems". IEEE Transactions on Parallel and Distributed Systems, Vol. 11, No. 5, May 2000.

[4] Weizhen Mao, Jie Chen and William Watson III, "Efficient Subtorus Processor Allocation in a Multi-Dimensional Torus" High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on Volume , Issue , 30 Nov.-3 Dec. 2005 Page(s): 8 pp.

[5] Lo, V.;Windisch, K.J.; Wanqian Liu; Nitzberg, B., "Noncontiguous processor allocation algorithms for mesh-connected multicomputers," Parallel and Distributed Systems, IEEE Transactions on Volume 8, Issue 7, Jul 1997 Page(s):712 - 726.

[6] Javier Navaridas, Jose Antonio Pascual, Jose Miguel-Alonso, "Effects of Job and Task Placement on the Performance of Parallel Scientific Applications", Technical Report EHU-KAT-IK-04-08. Department of Computer Architecture and Technology, The University of the Basque Country.

[7] Cluster Resources, "Maui Admin Manual". Available at URL http://www.clusterresources.com/products/maui/docs/5.2nodeallocation.shtm

[8] Lawrence Livermore National Laboratory, "Simple Linux Utility for Resource Management". Available at https://computing.llnl.gov/linux/slurm/

[9] Y. Aoyama et al., "Resource allocation and utilization in the Blue Gene/L supercomputer", IBM J. Res. & Dev. Vol. 49 No. 2/3 March.

[10] Parallel Workloads Archive, Available at http://www.cs.huji.ac.il/labs/parallel/workload/

[11] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, "Modeling user runtime estimates", Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) June 2005, Cambridge, Massachusetts, Lecture Notes in Computer Science, volume 3834.

[12] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates", IEEE Transactions on Parallel and Distributed Systems (TPDS), June 2007.

[13] R Ansaloni, "The Cray XT4 Programming Environment", www.csc.fi/english/csc/courses/programming/environment.

[14] Fabrizio Petrini and Marco Vanneschi. "Performance Analysis of Minimal Adaptive Wormhole Routing with Time-Dependent Deadlock Recovery". In Proceedings of the 11th International Parallel Processing Symposium, IPPS'97, pages 589-595, Geneva, Switzerland, April 1997.

[15] Steve J. Chapin, et al., "Benchmarks and Standards for the Evaluation of Parallel Job Scheduler", Lect. Notes Comput. Sci. Vol. 1659, pp. 66-89.

[16] 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.