Effects of Topology-Aware Allocation Policies on Scheduling Performance

Jose Antonio Pascual, Javier Navaridas, and Jose Miguel-Alonso

The University of the Basque Country, San Sebastian 20018, Spain
{joseantonio.pascual,javier.navaridas,j.miguel}@ehu.es

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 a k-ary n-tree network and using well-known scheduling algorithms (FCFS and backfilling). We incorporate an allocation policy that tries to assign to each job a contiguous network partition, in order to improve communication performance. This policy results in severe scheduling inefficiency due to increased system fragmentation. A relaxed version of it, which we call quasi-contiguous allocation, reduces this adverse effect. Experiments show that, in those cases where the exploitation of communication locality results in an effective reduction of application execution time, the achieved gains more than compensate the scheduling inefficiency, therefore resulting in better overall performance.

1 Introduction

Supercomputer centres are usually designed to provide computational resources to multiple users running a wide variety of applications. Users send jobs 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 the use in terms of maximum number of processors or maximum execution time. Other restrictions may be implemented such as user or group priorities, quotas, etc.

Generally, site performance is measured in terms of the utilization of the system and the slowdown suffered by jobs while waiting in the queue until the required resources become available. Consequently, a variety of scheduling policies [1] and allocation algorithms [2] [3] [4] have been developed aiming to minimize both the number of nodes that remain idle and the job waiting times. Scheduling policies are in charge to decide the order in which jobs are launched. 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 arranged 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) jobs in the queue. The main drawback of this policy is that it produces severe system fragmentation because some processors can remain idle during a long period of time due to the sequentially ordered execution of jobs. Idle processors could be used more efficiently 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], backfilling being the most widely 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 queue. If some queued jobs require a smaller 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 waiting times decrease. 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, 4x4 being the smallest size that can be used to run the job. In this case, four processors reserved for the job will never be used. This effect is named internal fragmentation [5]. Some job allocation algorithms try to minimize this effect. However, this work *does not* consider this effect, because each parallel job will be assigned to the exact number of required nodes.

Neither FCFS nor backfilling are allocation algorithms, as they do not take into account the placement of job processes onto network nodes. In a parallel system, application processes (running on network nodes) communicate interchanging messages. 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 randomly placed along the network, communication locality inside each job will not be exploited; and what is more, messages from different applications will compete for network resources, greatly increasing network 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. In order to avoid this effect, we propose the utilization of quasi-contiguous allocation schemes in which some restrictions of the purely-contiguous policy are relaxed, allowing the non-contiguous allocation of part of the required network nodes. This way network occupancy can be increased, at the cost of some penalty in terms of application run times.

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 [6], but the tools and methodology presented here will be extended to other topologies such as meshes or tori. Our final goal is to demonstrate that the introduction of locality-aware policies in the schedulers may provide important performance improvements in systems with multiple users and different applications.

The rest of the paper is organized as follows. In Section 2 we discuss some previous work on scheduling and allocation policies, describing in Section 3 those used in this paper. The simulation environment and the workloads used for the experiments are described in Section 4. Section 5 analyze a few preliminary experiments that provide evidence of the pros and cons of consecutive allocation schemes. These experiments are further elaborated in Section 6, that focuses on the search of a trade-off between application speedup and scheduling slowdown. Section 7 closes the paper with some conclusions and future lines of research.

2 Related Work

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

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

A more recent paper [7] evaluated the positive impact that locality-aware allocations have on applications performance, but focused on three particular applications, running on supercomputers connected by 3-D interconnection networks.

Part of our experiments corroborates the conclusions of the cited papers. However, our work differs from them in several important aspects. Previous research work shows that, depending on the communication pattern of the application, contiguous allocation provides remarkable performance improvements [8]. Therefore, we do not make extensive use of non-contiguity to increase system utilization; instead, we incorporate backfilling scheduling policy into the scheduler. Additionally, we focus on k-ary n-trees, instead of meshes or tori.

A review of schedulers in use in current supercomputers, such as Maui, Sun Grid Engine, and PBS Pro, shows that they do not implement contiguous allocation strategies. Some of them provide methods for the system administrator to develop their own strategies but, in practice, this is rarely done. To our knowledge, the only two current schedulers that maintain the locality are the one used by the BlueGene family supercomputers [9] and SLURM. The BlueGene scheduler puts tasks from the same application in one or more midplanes of 8x8x8 nodes which decreases network contention and allows locality exploitation. SLURM performs always a best-fit algorithm building first a Hilbert curve through the nodes on the Sun Constellation and Cray XT systems in order to keep locality as higher as possible. In contrast, the scheduling strategy used by the default scheduler (PBS Pro) on Cray XT3/XT4 systems (also a custom-made 3D tori) simply gets the first available compute processors [10].

3 Scheduling and Placement Policies

We used simulation to carry out an analysis of the impact that contiguous and quasi-contiguous allocation strategies have on scheduling performance. Our simulator implements two different scheduling policies (FCFS with and without backfilling), as well as three allocation algorithms (non-contiguous, contiguous, and quasi-contiguous) implemented for k-ary n-trees. The workloads used to feed the simulations have been obtained from actual supercomputers and are publicly available at the Parallel Workload Archive [11].

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

1. **First Come First Serve (FCFS):** In this policy, jobs are strictly processed in arrival order and executed as soon as there are enough available resources. The scheduling process is stopped until this condition is reached, even if there are enough free resources that could be allocated to other waiting jobs.
2. **Backfilling (BF):** This strategy permits the advance of jobs, even when they are not at the head of the queue, in such a way that system 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 time 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. This way, the starvation of the first job is avoided. Reservations are computed using a parameter called User Estimated Runtime, which represents a user-provided estimation of the job execution time [12]. In some cases the scheduling system itself may provide this value, based on estimations made over the historical system logs [13].

Other scheduling methods have been proposed in the literature, such as SJF (Shortest Jobs First [1]) which selects the jobs to be executed by their size instead of their arrival time, and several variations of backfilling (see [1]). However, the most commonly used algorithm in production 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 used in this study.

Regarding the allocation algorithms, the following are included in the study:

1. **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 [10]. This scheme provides a flat vision of the network, ignoring its topological characteristics and the virtual topologies of scheduled applications [4]. Note that in the long run it behaves as a random allocation of resources.
2. **Contiguous:** In this scheme job processes are allocated to nodes maintaining them as close as possible. To minimize the distance between processes (nodes) in a k-ary n-tree, 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) [6]. 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 to a collection of nodes attached to the same stage-1 switch, without requiring communication involving switches in upper stages of the tree. These are level-1 jobs. However, jobs larger than k will require the utilization of switches at stages 2, 3, etc. In general, up to k^i nodes can be allocated using stage-i switches.
3. **Quasi-contiguous:** This algorithm is a relaxed version of the previous one. It searches nodes that are contiguously allocated but, if the required number of free nodes is not found at the job level, it searches for the remaining nodes using switches *one* level above; contiguity is partly kept. The threshold of required-but-not-found free nodes that triggers the search on a higher level is a parameter provided to the algorithm, and the value providing best results is highly dependent on the size and type of the jobs that are executed in the systems. This parameter, which we call *qct* (quasi-contiguity threshold) is actually a percentage of the job size representing the number of tasks of that job allowed to be allocated using one extra level of the tree. Using this equation

$$max_{j \in J} = \left\lceil \frac{qct}{100} \times size_j \right\rceil \quad . \tag{1}$$

the algorithm computes $max_{j \in J}$, the maximum number of tasks of the job j allowed to be allocated using switches at the next level.
The utilization of additional stages of the tree may increase network contention, so we try to keep it under control by reducing the number of messages traversing high-level switches. To do so, we maintain the maximum possible number of nodes under switches belonging to the same level; actually, in favorable conditions this algorithm behaves exactly like the purely contiguous one. However, as some tasks can be assigned to non-contiguous portions of the network, external fragmentation is reduced. The *qct* threshold will maintain the number of quasi-contiguously allocated tasks limited, in order to reduce the interference created by the messages of different applications.

The contiguous algorithm starts computing the level to which the job belongs, and the size of this level (*level_ size*, the number of compute nodes below a single switch located at that level, which is the maximum size of a job that can be contiguously allocated below that level). After this preliminary step, the search of free nodes is performed, in groups of *level_ size* nodes following a first fit allocation scheme, because this way all the allocated nodes would be contiguous, that is, connected by the same switch or switches at the required level. If the complete tree is traversed but the necessary number of nodes has not been 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 6-node job (level-2) we need to find 6 free nodes that are connected using only stage-1 and stage-2 switches.

The quasi-contiguous algorithm requires two steps. Firstly, it performs a search for contiguous partitions as we stated before. If not found, because there are not enough free nodes at the job level, and the percentage of non-allocated tasks is below the *qct* threshold, the search continues in the level above. For example, in a 4-ary 3-tree topology, if we need to allocate a 4-node job, we start searching for completely empty stage-1 switches but, if none is available, another search is performed using stage-2 switches.

In Figure 1 we represent some simple allocation examples in a 4-ary 3-tree topology. We can observe how Job 1, of size 4, can be allocated into a single stage-1 switch; this is a contiguous allocation. The level of Job 2, of size 6, is 2; this means that it is allocated to two stage-1 switches that directly connected via switches at stage 2. Therefore, allocation of Job 2 is also contiguous. Job 3 is quasi-contiguously allocated because it should be a level-1 job (size is 4) but it requires the utilization of stage-2 switches.

4 Description of the Workloads

As we stated before, in this work we evaluate the performance of schedulers using logs of workloads extracted from real systems that are available from the PWA (Parallel Workload Archive, [11]). These logs have information about the

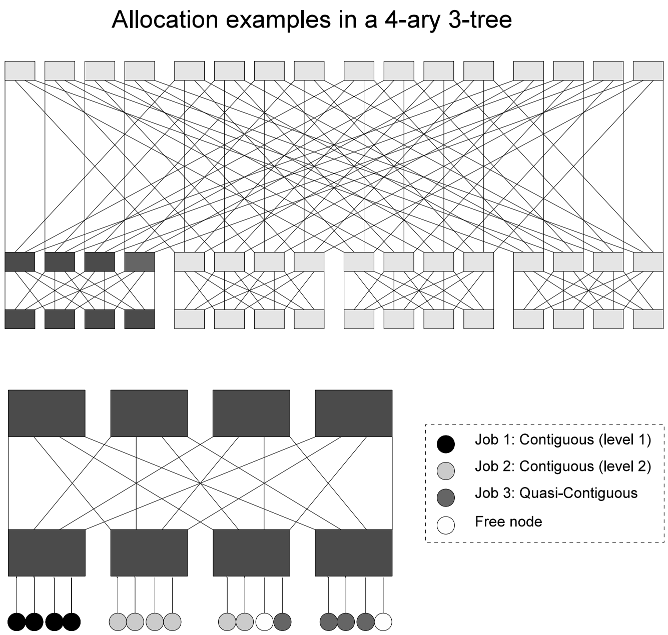

Allocation examples in a 4-ary 3-tree
● Job 1: Contiguous (level 1)
● Job 2: Contiguous (level 2)
● Job 3: Quasi-Contiguous
○ Free node

Fig. 1. Top: a 4-ary 3-tree; compute nodes are not represented for the sake of clarity. Bottom: a section of the network, with some examples of allocated jobs.

system as described in the SWF format (Standard Workload Format) [14]. In this study we used the following fields:

1. **Arrival Time:** The timestamp at which a job arrives to the system queue. Logs are sorted by this field.
2. **Execution Time:** The interval of time that the job was running in the system. In order to simulate the improvement of performance due to the exploitation of communication locality, we scale this field by applying a speed-up factor.
3. **Processors:** Number of processors required by the job.
4. **User Estimated Runtime:** This information is used only by the backfilling scheduling policy and represents a user estimation of the job execution time.
5. **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. Some studies do not include in the simulations those jobs that were not successfully completed (due to failure or cancellation), but we consider important all the jobs because they stayed in the queues, delaying the execution of other jobs.

In our experiments, all times were measured in minutes. We only used workloads that provide User Estimated Runtime information, because of the need of this parameter to perform a backfilling scheduling policy.

In [15], the authors suggested a metric to measure the *load* managed by the scheduler. Selecting workloads with different values of this metric allows us to check our proposals on different scenarios. The *load* is computed as follows:

$$load = \left(\frac{\sum_{j \in J} size_j \times runtime_j}{P \times (T_{end} - T_{start})} \right) \quad . \tag{2}$$

where P is the number of processors, J is the set of jobs between T_{start} and T_{end}, T_{end} is the last termination time and T_{start} is the last arrival time of the first 1% of the jobs. This 1% of firstly arrived jobs and the jobs that terminate after the last arrival are removed, in order to reduce warm up and cool down effects.

From the workloads available at the PWA, we have selected these three:

1. **HPC2N (High Performance Computing Center North).** This is a system located in Sweden, composed by 240 compute nodes and using the Maui scheduler. The workload log contains information of 527,371 jobs. Load: 0.62.
2. **LLNL Thunder (Lawrence Livermore National Laboratory).** 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. The log is composed by 128,662 job records. Load: 0.76.
3. **SDSC BLUE(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. Load: 0.86.

We simulated these workloads in k-ary n-trees adapted to each system sizes. For the first workload 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 workloads, so we have considered that the extra processors are not installed and they are ignored in the simulation.

5 Costs and Benefits of Contiguous Allocation Policies

Parallel applications performance depends on many factors, such as the communication pattern, distance between the application tasks, network contention, etc. The first one is an application-dependent characteristic, but the others are affected by the way the application is allocated.

A contiguous allocation strategy reduces the distance between the application tasks, to accelerate the interchange of messages and to reduce network utilization. An important, additional effect is that interference with other running applications is also reduced. This interference, that causes contention for network resources, may result in severe performance drops. Therefore, the contiguous allocation of a job improves the overall performance of the system, not only of that job.

In [8], the authors evaluate the possible benefits of contiguity for a collection of parallel applications. These benefits are highly dependent on the communication patterns of the applications. However, as we will show, the search of contiguity can be very expensive in terms of scheduling time. The execution of jobs may be delayed for a long time, until the required resources are available, the external fragmentation increases and the overall system utilization suffers. To minimize these negative effects we have introduced the concept of quasi-contiguity, a relaxed version of the contiguous allocation scheme which is expected to be less harmful in terms of scheduling time, while providing the same (or nearly the same) benefits in terms of application acceleration.

In order to validate the benefits of a contiguous and quasi-contiguous allocation policy, we have carried out several simulations using the INSEE simulator [16]. This tool does not simulate a scheduling algorithm, just the execution of a message-passing application on a multicomputer connected via an interconnection network. To feed this simulator we need traces of the messages interchanged by the communicating tasks. We have obtained these traces using a selection of the well-known NAS Parallel Benchmarks (NPB [17]). INSEE performs a detailed simulation of the interchange of the messages through the network, considering network characteristics (topology, routing algorithm) and application behavior (causality among messages). The output is a prediction of the time that would be required to process all the messages in the application, in the right order, and including causal relationships. Therefore, it only measures the communication costs, assuming infinite-speed CPUs. When using actual machines, a good portion of the time (ideally, most of the time) would be devoted to CPU processing, and the impact of accelerated communications in overall execution time would be smaller.

The simulated topology is a 4-ary 4-tree, with 256 nodes. Instead of one application, we simulate the simultaneous execution of sixteen instances (jobs) of the same application (actually, trace), each one using sixteen nodes. The sixteen jobs have been allocated onto the network using three strategies:

1. **Contiguous:** Each job is allocated onto four level-2 switches, so the communications between tasks of the same job never need links or switches at level 3.
2. **Quasi-Contiguous:** In this strategy, we allow a partial non-contiguous allocation of the job tasks. The four experiments performed allow the non contiguous allocation of 1, 2, 3 or 4 tasks of each job, respectively.
3. **Non-Contiguous:** Tasks of each job are distributed along all the switches at level 4 (the maximum level of this tree). This means that intra-job communications do use level-4 switches, and also that messages of different jobs compete for network resources.

Figure 2 shows the execution time of each application using each strategy normalized to the time required by the contiguous placement. The benefits of con-

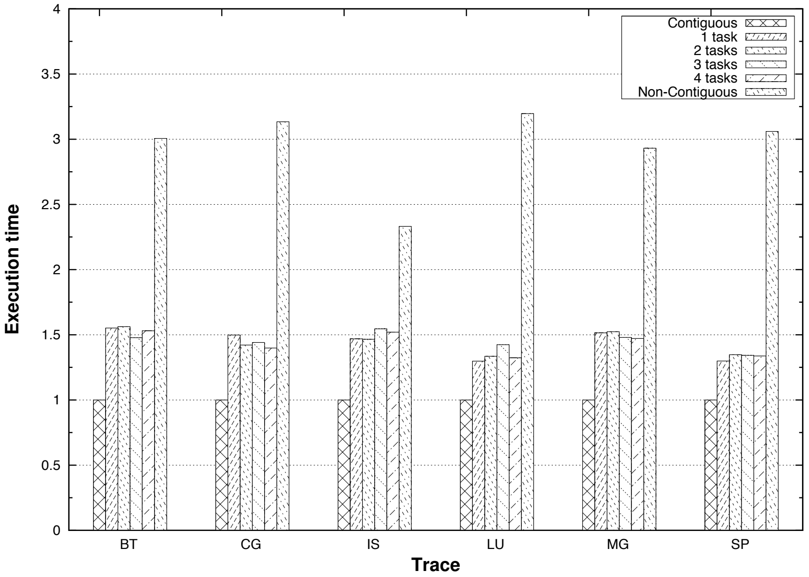


Fig. 2. Execution time for different allocation policies simulating the traces of some NAS Parallel Benchmarks in a 4-ary 4-tree topology. Values are normalized, so that 1 represents the contiguous allocation.

tiguous allocation strategies are clear: non-contiguously allocated applications run between 2 and 3 times slower. Regarding the quasi-contiguous allocation, we can appreciate that performance is always good, being only 30-50% higher to that obtained with purely contiguous allocation. These results confirm our expectations: a good allocation strategy can substantially reduce the execution time of a set of applications sharing a parallel computer as stated in [8].

Now we will asses the real cost of contiguity on scheduling. Using the scheduling simulator with the selected workloads (those from the PWA), we measure application waiting time for FCFS and backfilling scheduling algorithms, for purely contiguous allocation and quasi-contiguous allocation for four values of *qct*: 10, 20, 30 and 40%. Results are plotted in Figure 3. Note that values are relative to those obtained with the same workload and scheduling using non-contiguous allocation. Results are devastating: waiting times can be up to 100 times worse if contiguity is a requirement. Values are better for quasi-contiguity, but still bad. However, note that we *did not* take into consideration the acceleration that jobs experience due to better allocation. We will explore this issue in the next section. It is remarkable the difference between the LLNL workload waiting times and the other workloads waiting times, due to the presence of big

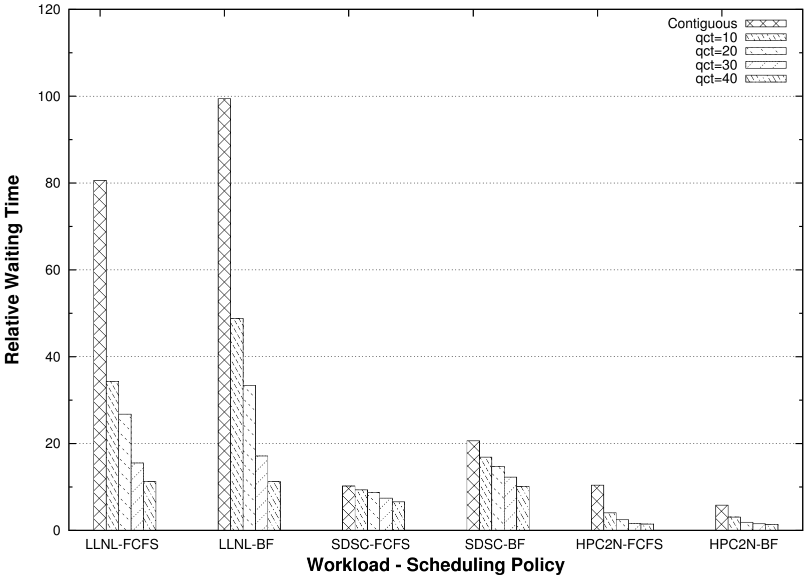


Fig. 3. Cost of contiguous and quasi-contiguous allocation, in terms of waiting time. A value of 1 would represent the average job waiting time for the non-contiguous allocation with the same scheduling policy.

size jobs (some of them of 1024 nodes). Finding contiguous partitions of this size is quite difficult, which results in longer waiting times for them and for the jobs that follow.

6 Tradding Off Costs and Benefits of Contiguous Allocation

In this section we carry out a collection of experiments to thoroughly evaluate the effect that contiguous allocation may have on scheduling performance. In these experiments we consider that contiguous allocation is able to accelerate the execution of parallel jobs. However, the actual values of attainable speed-ups are not available to us – they strongly depend on the communication characteristics of the applications, something that requires an exhaustive knowledge of each and all the applications included in the workload logs. We do not have that knowledge. For this reason, we introduce speed-up as a *parameter* of the simulation. With this setup we are able to know to what extent a certain level of application speed-up compensates the performance drop introduced by a restrictive allocation policy. This parameter is applied only to the parallel applications of the workload remaining the sequential jobs with the same runtime.

We have studied several combinations of scheduling and allocation policies. We evaluate them in terms of these two measurements:

1. **Job waiting time.** The time jobs spent in the queue.
2. **Job total time.** All the time spent in the system, which includes the time waiting at the queue and the execution time.

As stated before, when using contiguous and quasi-contiguous allocation, a speed-up factor has been applied to reduce the execution time. Note again that applying a speed-up factor to a running time improves not only the application finish time, but also reduces the time spent by the jobs using system resources; and therefore, the scheduling performance is increased too. In the simulations we used the workloads from the PWA described in Section 4.

The quasi-contiguous strategy has been evaluated with four values of *qct*. Results are depicted in Figures 4, 5, 6 and 7. Note that, as the range of values is very wide, we used a logarithmic scale in the Y axis of all figures. We represent the averages of total time (waiting plus running) and, in some cases, waiting time alone. In each graph we can see six lines, one per allocation policy. Tested speed-up factors range from 0% to 50%. When this factor is 0% it means that, although the scheduler seeks contiguity, 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 (a value of 10% means that the execution requires 10% less time to be executed with that allocation scheme). Obviously, we cannot assume any acceleration with non-contiguous allocation, and for this reason the corresponding line is flat.

Let us now pay attention to Figure 4, where the LLNL workload is studied in detail. In all scheduling-allocation combinations, results with speed-up=0 are as appalling as described in the previous section. However, when this value increases (that is, when applications really run faster when allocated contiguous resources) the picture changes. At speed-up values between 5% - 30% the contiguous and quasi-contiguous approaches show their potential. It is clear that the quasi-contiguous strategies prove beneficial at lower speed-ups than the purely contiguous. Also, note that if the scheduler uses backfilling, global system efficiency is higher (the workload is processed faster), and the thresholds at which contiguity is advantageous are lower.

Figure 5 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 accordingly reduced, but waiting time is also reduced.

In Figures 6 and 7 we have summarized results for workloads HPC2N and SDSC. To be succinct, and given that the qualitative analysis performed with LLNL is still valid, we only show results of total times for the FCFS and back-

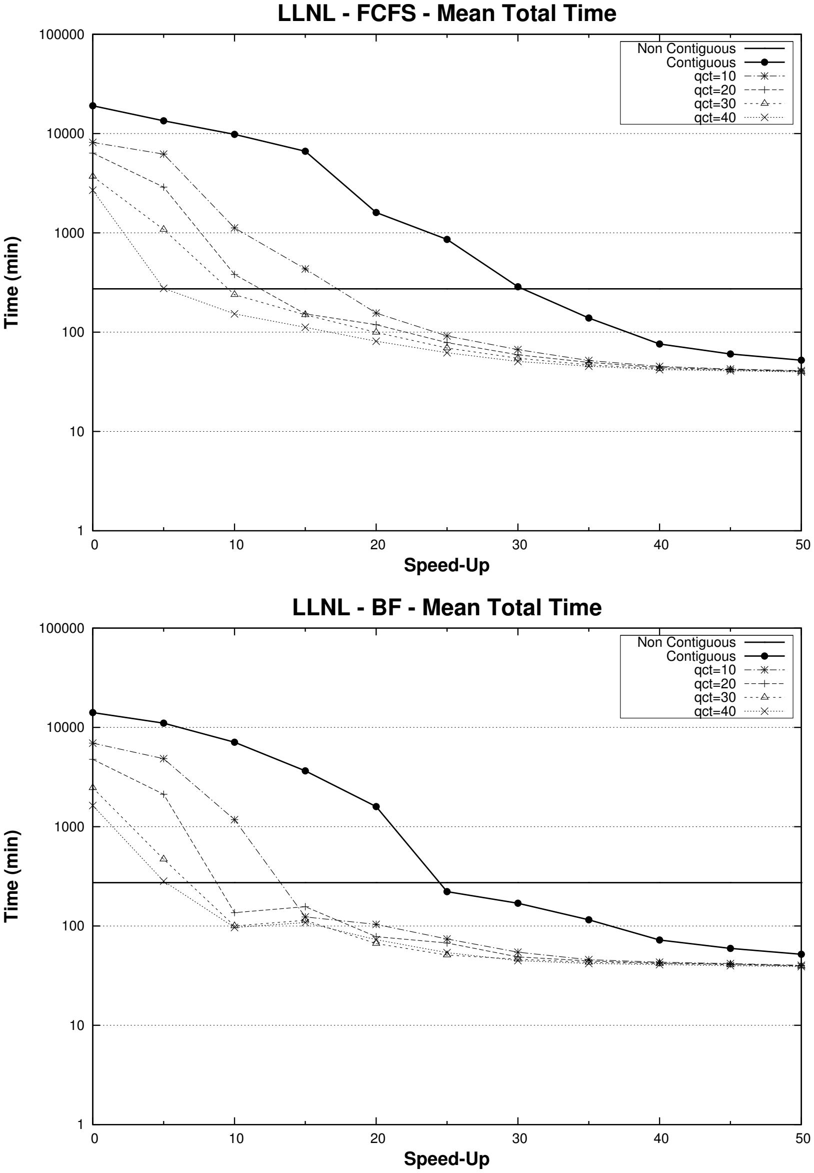


Fig. 4. Results of the experiments with the LLNL workload for FCFS and backfilling scheduling policies for various allocation strategies. Mean Total Time (Wait Time + Execution Time) at different speed-ups. The scale of the Y axis is logarithmic.

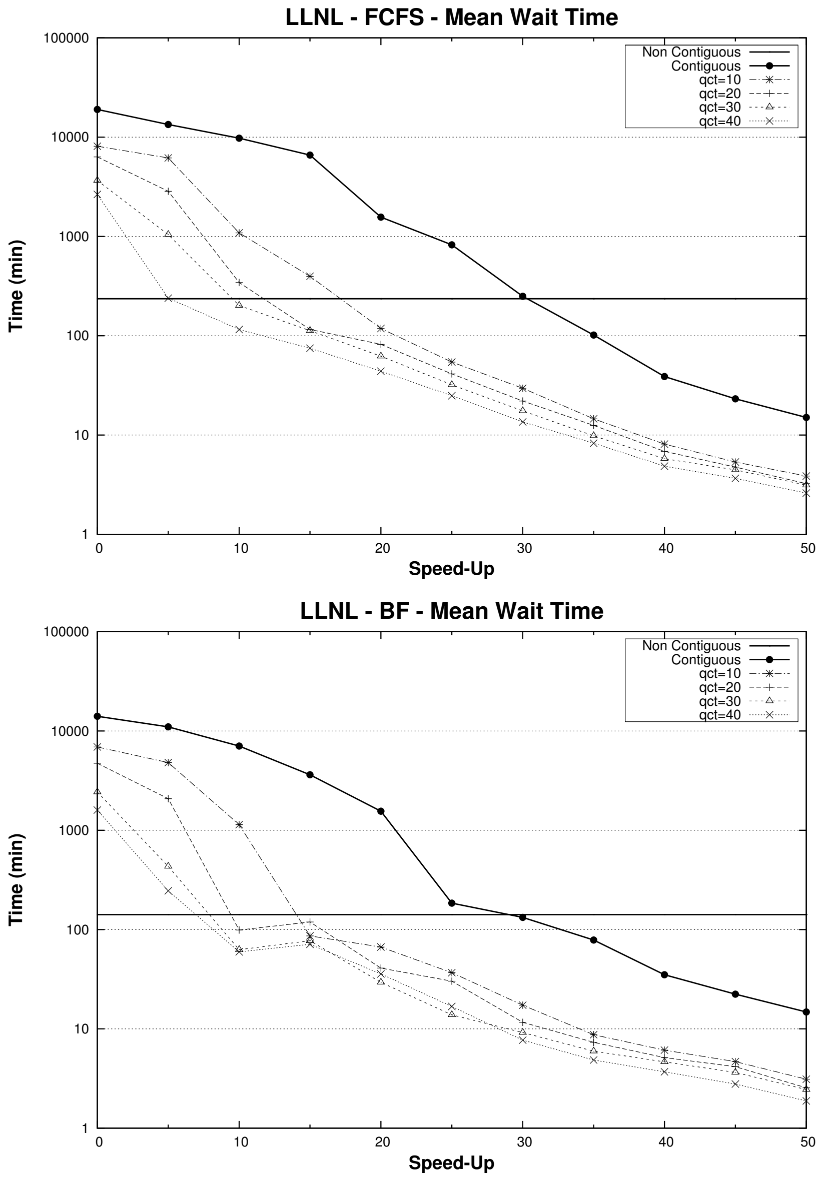


Fig. 5. Results of the experiments with the LLNL workload for FCFS and backfilling scheduling policies for various allocation strategies. Mean Wait Time at different speed-ups. The scale of the Y axis is logarithmic.

filling. For the SDSC workload, the threshold at which contiguous and non-contiguous allocation starts being beneficial falls between 15% and 25% (higher than that of LLNL). Similar, although slightly lower, values required by HPC2N are between 10% and 25%.

In all figures, we can see the benefits of using the quasi-contiguous policy. The scheduler performs better and, as described in the previous section, the expected speed-ups would be only slightly lower that those attainable with contiguous allocation. We have to remark that the implementation of this strategy tries always to find first a contiguous allocation, and only uses non-contiguous nodes as the last alternative. Therefore, if we estimate that we can obtain a certain speed-up when using a given value of *qct*, we will actually obtain better speed-ups, because in some cases the scheduler will obtain a contiguous allocation for the jobs.

Note that the increase of the *qct* parameter results in an equalization of the FCFS and backfilling performance reducing the difference between them. The reason is that the quasi-contiguous allocation strategy has a similar effect to the backfilling policy allowing the schedule of more jobs and thus, reducing the waiting time in the queue.

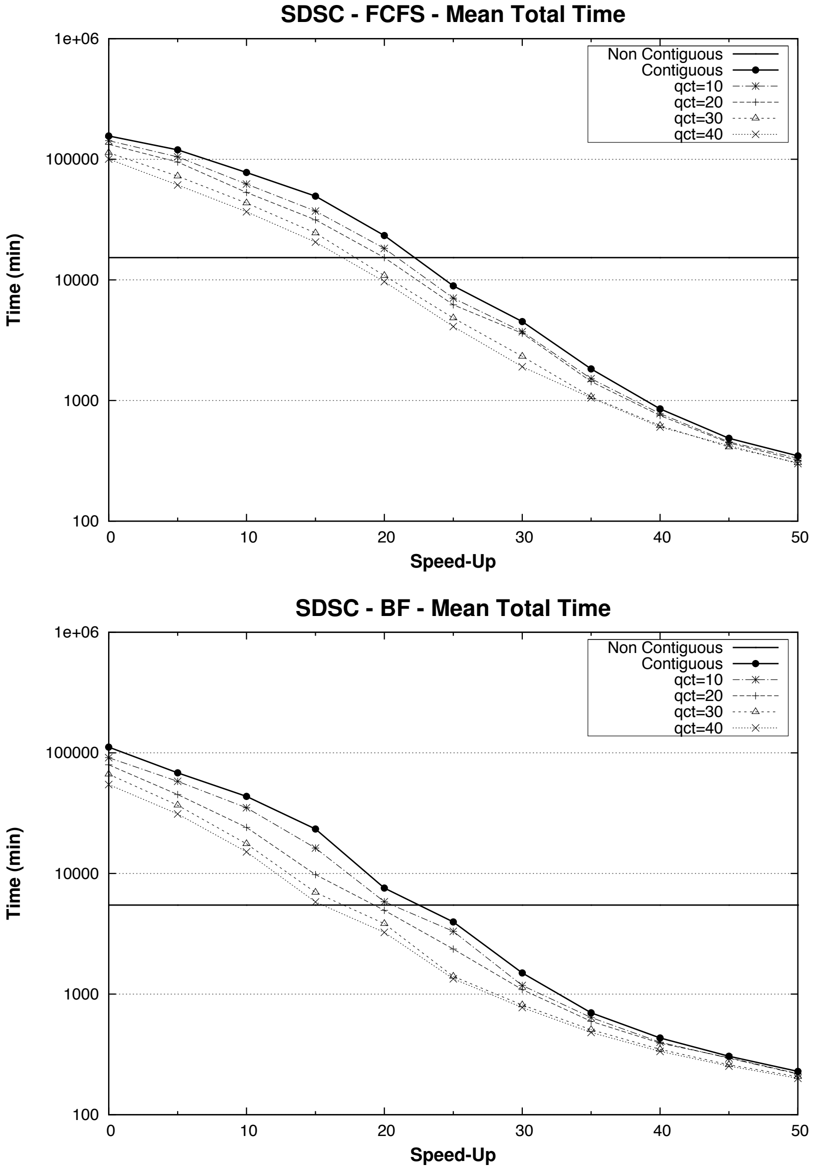
7 Conclusions and Future Work

Most current supercomputing sites are built around parallel systems shared between different users and applications. The optimal use of resources is a complex 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 compared two scheduling techniques combined with three allocation algorithms in a k-ary n-tree network topology. Allocation algorithms that search for contiguous resources have an elevated cost in terms of system fragmentation, but also are able to accelerate the execution of applications. With the quasi-contiguous allocation, this acceleration is slightly penalized but the scheduling performance is significantly improved.

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 20-30%, this cost is compensated. Using relaxed versions of the contiguous allocation strategy (which we have called quasi-contiguous) this threshold lowers significantly, in such a way that in some cases speed-ups around 10% are enough to provide improvements in terms of scheduling efficiency.



SDSC - FCFS - Mean Total Time

SDSC - BF - Mean Total Time

Fig. 6. Results of the experiments with the SDSC workloads for FCFS and backfilling scheduling policies for various allocation strategies. Mean Total Time (Wait Time + Execution Time) at different speed-ups. The scale of the Y axis is logarithmic.

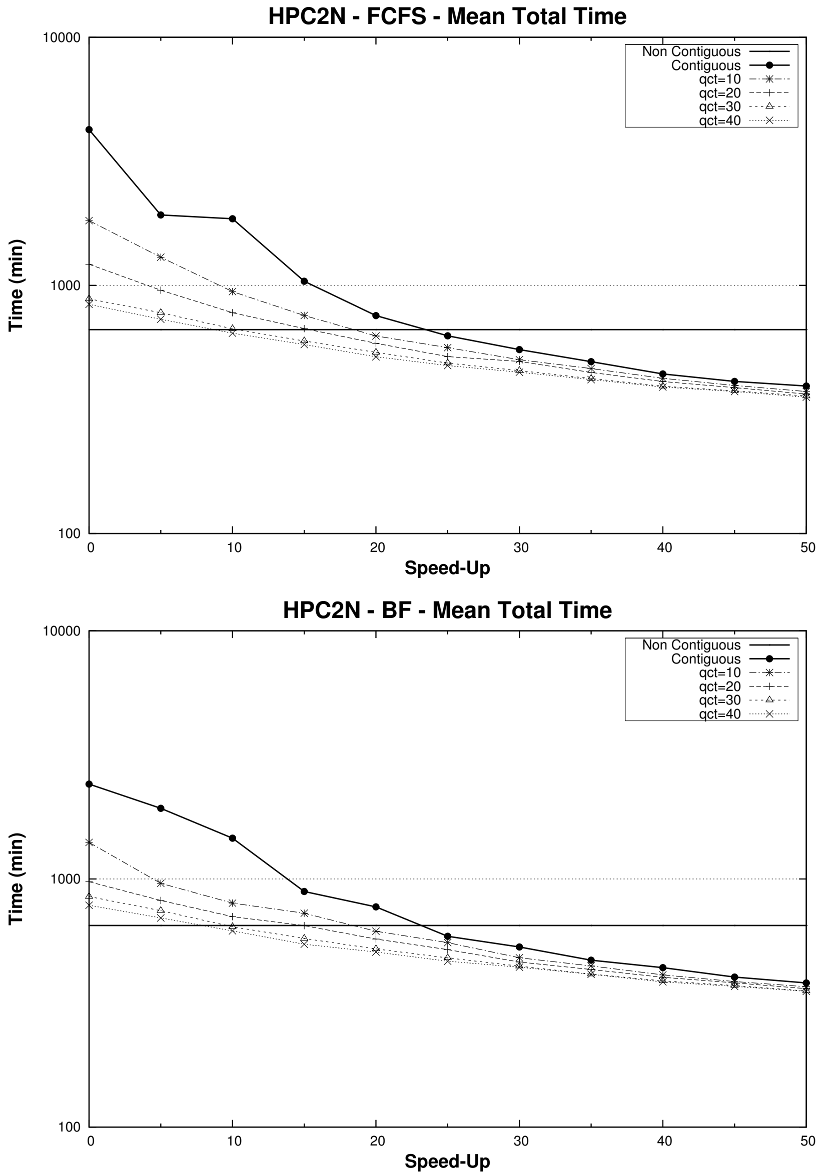


Fig. 7. Results of the experiments with the HPC2N workload for FCFS and backfilling scheduling policies for various allocation strategies. Mean Total Time (Wait Time + Execution Time) at different speed-ups. The scale of the Y axis is logarithmic.

This study has focused only in tree-based networks; the next step will be a performance study for other topologies (in particular, for k-ary n-cubes and k-ary n-tori). Because of the highly dependency of the allocation algorithms on the underlying topology, new quasi-contiguous allocation strategies should be developed for each new studied topology. We have provided application acceleration as a simulation 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 system 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 integrate INSEE [16] into the scheduling simulator.

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

References

1. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling, – a status report. In: Job Scheduling Strategies for Parallel Processing, Springer Verlag (2005) 1–16
2. Gupta, E.K.S., Srimani, P.K.: Subtori Allocation Strategies for Torus Connected Networks. In: Proc. IEEE 3rd Int'l Conf. on Algorithms and Architectures for Parallel Processing. (1997) 287–294
3. Choo, H., Yoo, S.M., Youn, H.Y.: Processor Scheduling and Allocation for 3D Torus Multicomputer Systems. IEEE Transactions on Parallel and Distributed Systems **11**(5) (2000) 475–484
4. Mao, W., Chen, J., Watson, W.I.: Efficient Subtorus Processor Allocation in a Multi-Dimensional Torus. In: HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region, Washington, DC, USA, IEEE Computer Society (2005) 53
5. Lo, V., Windisch, K., Liu, W., Nitzberg, B.: Noncontiguous Processor Allocation Algorithms for Mesh-Connected Multicomputers. IEEE Transactions on Parallel and Distributed Systems **8** (1997) 712–726
6. Petrini, F., Vanneschi, M.: Performance Analysis of Minimal Adaptive Wormhole Routing with Time-Dependent Deadlock Recovery. In: IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing, Washington, DC, USA, IEEE Computer Society (1997) 589
7. Bhatele, A., Kale, L.V.: Application-specific Topology-aware Mapping for Three Dimensional Topologies. In: Proceedings of Workshop on Large-Scale Parallel Processing (held as part of IPDPS '08). (2008)
8. Navaridas, J., Pascual, J.A., Miguel-Alonso, J.: Effects of Job and Task Placement on the Performance of Parallel Scientific Applications. In: Proc 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Weimar, Germany (February 2009)
9. Aridor, Y., Domany, T., Goldshmidt, O., Moreira, J.E., Shmueli, E.: Resource Allocation and Utilization in the Blue Gene/L Supercomputer. IBM Journal of Research and Development **49**(2–3) (2005) 425–436

10. Ansaloni, R.: The Cray XT4 Programming Environment. `http://www.csc.fi/english/csc/courses/programming/`
11. Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/logs.html`
12. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Modeling User Runtime Estimates. In: 11th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP 2005, Springer-Verlag (2005) 1–35
13. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling Using System-Generated Predictions Rather than User Runtime Estimates. IEEE Trans. Parallel Distrib. Syst. **18**(6) (2007) 789–803
14. Chapin, S.J., Cirne, W., Feitelson, D.G., Jones, J.P., Leutenegger, S.T., Schwiegelshohn, U., Smith, W., Talby, D.: Benchmarks and Standards for the Evaluation of Parallel Job Schedulers. In: IPPS/SPDP '99/JSSPP '99: Proceedings of the Job Scheduling Strategies for Parallel Processing, London, UK, Springer-Verlag (1999) 67–90
15. Tsafrir, D.: Modeling, Evaluating, and Improving the Performance of Supercomputer Scheduling. PhD thesis, School of Computer Science and Engineering, the Hebrew University, Jerusalem, Israel (September 2006) Technical Report 2006–78.
16. Ridruejo, F.J., Miguel-Alonso, J.: INSEE: An Interconnection Network Simulation and Evaluation Environment. In: Euro-Par. (2005) 1014–1023
17. NASA Advanced Supercomputer (NAS) division: Nas parallel benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`