High throughput computing over peer-to-peer networks

Carlos Pérez-Miguel *, Jose Miguel-Alonso, Alexander Mendiburu

Intelligent Systems Group, Department of Computer Architecture and Technology, The University of the Basque Country, UPV/EHU, Paseo Manuel de Lardizabal, 1, 20018, San Sebastian-Donostia, Spain

A R T I C L E   I N F O

Article history:
Received 1 November 2010
Received in revised form 20 July 2011
Accepted 5 August 2011
Available online xxxx

Keywords:
Peer-to-peer networks
High throughput computing
Distributed hash tables

A B S T R A C T

In this work, we present a proposal to build a high throughput computing system totally based upon the Peer-to-Peer (P2P) paradigm. We discuss the general characteristics of P2P systems, with focus on P2P storage, and the expected characteristics of the HTC system: totally decentralized, not requiring permanent connection, and able to implement scheduling policies such as running jobs in a (non-strict) FCFS order. We have selected Cassandra as the supporting P2P storage system for our purposes. We discuss the basic aspects of the system implementation, and carry out some experiments designed to verify that it works as expected.

1. Introduction

Peer-to-Peer (P2P) systems, initially designed for file sharing (Napster [1], Gnutella [2]) are gaining popularity in industry and academia as a powerful mechanism to support other types of applications. Some proposals (or, in some cases, working realities) are massive file storage systems [3], name indexing [4] and voice calls over IP [5]. The computational science & engineering community is paying attention to P2P systems as the foundation of a large computational resource [6].

In this work, we focus on one particular paradigm of massive computation: High Throughput Computing (HTC). An HTC system can be defined as a platform able to execute a large number of jobs per unit of time. These jobs are independent, meaning that they can be submitted by different users, and that jobs submitted to the system can be executed in any order—although some sort of first-in-first-out order is usually expected.

Existing HTC systems, like Condor [7] or Boinc [8], have one important characteristic that make them potentially weak: they require a central administration point. This central point could impose limitations in system scalability and also in fault tolerance. In this work, we propose a design of an HTC system based on P2P protocols in order to overcome these limitations. In the pure peer-to-peer philosophy, all the members of the proposed system should be capable to carry out administrative tasks to maintain the system operational, in addition to executing jobs.

Ideally, this P2P–HTC system should perform like a non-P2P in the absence of failures, and should scale to large networks. A prototype implementation of our proposal (which does not incorporate the full range of planned features) has been used to carry out a collection of experiments. These allow us to check that the system is operational and to assess its performance. The code of the system is available to the research community by direct request to the corresponding author.

The rest of the paper is organized as follows. Section 2 describes a few basic concepts about peer-to-peer networks. Section 3 presents some ideas about P2P-based intensive computing systems, and the minimum set of characteristics that should be included to provide this kind of service; we also discuss some works carried out in this field. In Section 4, we focus on distributed, P2P storage systems, paying special attention to a particular class (distributed hash tables) and a particular implementation, Cassandra, that have been chosen as the foundation of our HTC proposal. The proposal itself is described in Section 5. A prototype implementation of our HTC system is tested in Section 6. Finally Section 7 summarizes the main conclusions of this paper, and proposes future lines of work and research.

2. Peer-to-peer networks

Peer-to-peer systems are distributed systems in which there is neither a central control point, nor a hierarchical structure among its members. In a P2P system, all nodes in the system have the same role, and are interconnected using some kind of network (usually, Internet), defining an application-level virtual network, also called overlay. Nodes communicate using this overlay, in order to find information, share resources or communicate human users.
In theory, DHT systems guarantee that any object can be reached in a number of jumps in order $O(\log N)$, being $N$ the number of network nodes. The weakest point in DHT systems is their behavior in the case of fast changes in system configuration, term also known as churn. Informally, this refers to nodes that join or leave the overlay, forcing a network reorganization and a modification in the mappings. Latency can increase in this case, and several proposals exist that try to reduce this problem. In [13], an algorithm is proposed that tries to reach the optimum latency in potential-law graphs, such as P2P networks, without losing the scalability of DHTs; also, Godfrey et al. propose in [14] an algorithm to maintain load balance in adverse conditions.

3. P2P computing

A distributed computing system can be defined as a collection of computers interconnected by a communication network. These computers try to join their resources in order to collectively do computational tasks. Each computer in the system has its own, independent resources; however, from the user’s point of view, the system should be seen as a single resource pool. An interface is given to the users in order to access the system without taking care of its complexity.

The are many systems, both free and commercial, that accomplish this objective, from Clusters to Grid Computing systems, and including Desktop Computing systems. Some well-known products or middleware sets are Globus [15] for grid systems, Condor [7] for cluster systems (with extensions for grids), or Boinc [8] for desktop computing. All of these systems, however, have something in common: they all revolve around central points of administration. This central resource, in case of failure, can make the whole system unusable.

A true peer-to-peer computing system overcomes this disadvantage by distributing management capabilities among all the system nodes. In the literature, we can find several proposals of P2P computing systems. An extensive survey can be found in [10]. Next we briefly analyze some P2P computing systems paying attention to the following desirable properties:

1. Fully distributed, without centralized administration point or points.
2. Users should be able to submit jobs from their machines and then disconnect from the system.
3. Global scheduling policies should be implemented. In particular, FCFS execution order of jobs is expected, although it needs not be strict.

Some proposals discussed in the literature fail to meet the first property: they use some centralized mechanism for job scheduling. CompuP2P [16] is based on a DHT which divides the nodes set between resource sellers and buyers. A leader node is chosen, which takes care of the resource market. In [17] Chmaj et al. present a system based on a structured overlay in which one of the nodes has the master role, the tracker. This tracker is needed in order to schedule computing tasks and distribute those files the nodes may require.

A common model used for P2P computing systems is the super-peer model in which a group of nodes, the super-peers, form a P2P overlay. Workers are connected to super-peers, which are in charge of scheduling tasks. When a user wants to run any task, he will have to ask nodes in the super-peer overlay to search for idle workers. Examples of this model are JACEP2P-V2 [18], Mining@home [19] and DIETj [20]. A similar idea is implemented in CoDiP2P [21], which is based on a tree-structured P2P network. It is built with the JXTA Java library, used to form non-structured P2P networks. In the tree structure, nodes are separated into masters and slaves, and grouped in tree regions. In each region, there is a master
node in charge of the scheduling procedure. The middleware is written in Java, and it can only execute Java programs linked to a library developed by the CoDiP2P team. All these P2P systems have several manager nodes, which could pose advantages in terms of scalability. Still, they are not fully distributed: failures in one master would negatively affect, if not the whole system, parts of it.

In [22], authors propose a totally distributed scheduler for workflows in a P2P Desktop Grid. A tree-based overlay network is defined to summarize and distribute resource status information. Nodes act simultaneously as job submitters, routers (for the overlay network) and execution nodes. The submitter node of a workflow is in charge of monitoring and controlling its progress. Therefore, it fails to meet the second condition: jobs cannot run unattended.

DHTs have become a common mechanism to safely store and share information in P2P systems. They can be used to provide support to fully decentralized HTC systems. In some cases, the key space encodes the characteristics of the worker nodes in such a way that, given the characteristics of a job, the node more able to run it is easily located (matchmaking). In others, the DHT is simply a storage system used to store/retrieve information about resources and jobs.

In the first group [23–29], DHTs are used to search specific resources in the system, considering this search as a routing problem over overlay networks with different structures (rings, cubes). For example, CrossTree [30] considers a DHT where each node is identified by an ID formed by its resources. All the nodes with the same capabilities will be arranged in a tree, but share the same position in the DHT. WaveGrid [31] uses a similar approach, but nodes are divided by time-zones, so that at a particular time only the nodes in the night zone are used.

In the second group, we can find several proposals that use a DHT-based system as a blackboard to store information about resources, which is used by broker nodes in order to schedule pending tasks [32–35]. Some other proposals use DHTs to store different kinds of information required by the distributed scheduling system. For example, [36] stores session time data, which is used to predict the expected time an execution node will be available, while in [37] a DHT is used to keep node credits in order to implement a distributed credit system.

While the systems mentioned in the previous paragraphs can be considered as fully distributed P2P computing systems, they fail to meet the third criterion: they lack mechanisms to ensure that tasks are being executed in a particular order, or even that they will be eventually executed.

The lack of effective mechanisms to implement scheduling policies is a common weakness of the existing P2P computing systems. They do not provide guarantees that a job submitted to the system will be eventually executed, because no mechanisms are in place to ensure that it will survive to events such as falling nodes or changes in the administrative infrastructure. Additionally, no order is imposed in the execution of jobs. This may look appropriate for a totally distributed P2P system, but at least some effort must be put in place to ensure some fairness and to make old jobs execute before new jobs. In conventional, non-P2P systems, a centralized job queue takes care of this issue.

The implementation of a P2P computing system implementing queue-based scheduling would allow the execution of jobs in an FCFS order, as well as the implementation of other scheduling techniques (for example, priorities). We consider that these are desirable characteristics for HTC systems, as they are present in conventional, centralized HTC systems. Works described in [33,38,39] implement distributed queues based on DHTs that allow for the implementation of FCFS scheduling mechanisms; however, those DHTs are not failure-resistant, and data (job) loss may happen.

In order to achieve the expected goals, additional features are necessary for the DHT, including persistence, resiliency and consistency. In the literature we can find diverse proposals [34,40–44] to build a distributed and highly scalable storage system from a P2P overlay using replication and coherence algorithms.

Our proposal is to build, on top of a reliable DHT-based storage system, a totally distributed queue object so all nodes in the system would know which jobs are waiting and in which order they have to be run. Idle nodes will check the queue object to find the next awaiting job, and execute it. Besides, the DHT storage system can be used as a blackboard to store all the data needed by the jobs (programs, parameters, input data, output data). This would allow nodes to launch jobs and then disconnect from the system, if necessary. Whenever they re-join it, if the job has finished, results will be readily available in the system. All idle nodes can use this information to coordinate themselves and execute tasks.

In the next section, we will review a collection of DHT systems that could be valid for our purposes, and justify the choice of a particular one: Cassandra.

4. P2P storage systems

We have proposed to use a DHT system to store the information about the jobs in our HTC system. But what are the characteristics required from a DHT in order to consider it adequate to our purpose? In this section, we will review a few DHT-based storage systems, with emphasis on those features that may make them good candidates.

We have stated that in order to build a distributed queue for our jobs we have to include in the design of the DHT some fault tolerance features, which are usually achieved via replication techniques. Consequently, we also need coherence protocols to maintain the replicas synchronized and failure detectors to detect which replicas are off-line. The first production system integrating all of these techniques was Amazon’s Dynamo [45]. Dynamo is a key–value storage system based on consistent hashing (DHT) which ensures lookups in constant time and, due to its replication and quorum techniques, provides high levels of scalability and availability, with eventual consistency [46]. The CAP theorem [47] states that, in distributed systems, consistency, availability and partition-tolerance cannot be achieved at the same time. So Dynamo designers decided to limit the consistency of the system providing eventual consistency and gaining in system stability. Note that the “eventual consistency” term means that all updates in the system reach all replicas after some (undefined) time.

Dynamo has been the origin of a collection of systems based on it. Some of them are key–value stores, like Riak [48] or Scalaris [49]. Others are document oriented, like CouchDB [50], and others are wide columns stores like Hadoop [51] or Cassandra [52]. In all of these systems it is possible to implement our target distributed queue. We have chosen Cassandra to build our prototype for several reasons. According to different benchmarks from Yahoo [53], Cassandra has a good behavior compared to other storage systems designed for cloud computing, in terms of read/write latency. We can also highlight its tunable consistency level and the richness of its data model, which enables several implementations of a queue. Other systems provide strong consistency, but at the cost of serializing writes, or under reliable environments. With Cassandra we will have to deal with eventual consistency, but we will have a robust, scalable storage system. Let us now explore the characteristics of this storage system.

Cassandra was developed in 2008 by Facebook to improve its Inbox Search tool. It was released to the Apache Foundation in 2009. It provides consistent hashing/order preserving partitioning, a gossip based algorithm to control membership, replication, anti-entropy algorithms and failure detection. These features give us
a totally distributed, fault tolerant and symmetric storage system with eventual consistency.

In Cassandra we do not store key–value pairs; instead of that, we store several values, columns, for each key. This columns can be grouped under different sets called ColumnFamilies or Super-ColumnFamilies, like in Google’s BigTable [54]. A ColumnFamily is a group of several columns, each one containing a value and a name. A SuperColumnFamily is a set of several columns, each one containing several sub-columns, with each sub-column containing only one named value. The number of columns, or sub-columns, in a ColumnFamily or SuperColumnFamily has no limits. By default, groups of columns are ordered in an ascending order using as key the name of each column, with an ordering function appropriate for the stored data type (numbers, strings, etc.). This allows us to use rows of a Column or SuperColumnFamily as ordered lists of items and make range scans over them.

Different ColumnFamilies with the same key are located in different nodes in Cassandra. Inside each ColumnFamily, the location of different rows are distributed in two different ways:

- Randomly is the default configuration of Cassandra, permitting the system to be load-balanced.
- Preserving a lexicographic order, in which rows are located in sequential positions of the Cassandra system. This allows us to order rows in Cassandra according to its key but it can unbalance the system.

5. A prototype of P2P–HTC system

The structure of Cassandra gives us different options to build our queue system. The first step will be to define the way jobs will be managed in the system. Each job will be identified by a unique identifier in Cassandra. A timestamp alone is not a feasible choice of identifier, because different nodes could submit jobs at the same time. Therefore we also add a random sequence. The Universally Unique Identifier [55], or UUID, combines these mechanisms, providing an identification scheme that serves for our purpose.

There are several ways to use Cassandra’s data model to store our job queue. We propose two approaches. In the first one, we can use a ColumnFamily as a queue. Each row will have the information needed by a job to be scheduled: name, owner, program to be executed, parameters and so on. Table 1 illustrates this design. Rows could be ordered or placed in random positions if we did not care about an ordered structure—which is not our case. Therefore, we use ordered storage, even if there are some penalties in latencies and load-balance.

The second option is to use rows of a SuperColumnFamily as queues. Each SuperColumn will store a job, gathering in different subcolumns the information about the job. SuperColumns in a row can be timestamp-ordered, using the UUID. Regarding write/read latencies, this solution is better than the previous one, because reading a queue row is faster than reading a queue distributed over the entire Cassandra cluster. This has the drawback of storing the entire queue over the $N$ nodes containing the $N$ replicas of the object. There are solutions to this disadvantage like partitioning the queue between several ColumnFamilies, but this is applicable only if the amount of information in each queue is huge. In Table 2, we show this design.

We have chosen the latter option to implement our queue system because we use Cassandra to store many different objects: in addition to the queue, it stores executable programs, configuration files, user’s information, statistical information, etc. The implementation using the first method with row ordering would cause a notable unbalance and could harm the storage of other types of objects.

Once this decision was made, we have developed a prototype of our system where users can:

- Send simple jobs to the system. The submission includes the needed executables and configuration files.
- Manage the state of their jobs.
- Recollect job’s results.
- Retrieve statistical information about the system.

And nodes of the system will automatically:

- Get, when they are idle, waiting jobs from the queue, and execute them.
- Store job’s results.
- Maintain statistical information.

The operation of our system will be as follows: when a node wants to submit a new job, it will insert the job’s relevant data into one ColumnFamily, called Works, with state “Waiting” and identified by an UUID. It will upload all additional files into another ColumnFamily, FileStorage, and then enqueue the job into the waiting queue of the Queues SuperColumnFamily. After that, it is free to disconnect from the system: no additional information will be required from the submitting node. Results of job’s execution, when the job is completed, will be stored in Cassandra, ready for download when the node decides to re-join the HTC system.

At each node there is a scheduling process in charge of searching jobs to run. When a node is idle and wants to execute something from the queue, the scheduling process will look up the Queue SuperColumnFamily in time order until it finds an appropriate job, changing its state to running and erasing it from the waiting queue. After that, it will run the job locally and, when done, will store into Cassandra all the results obtained from the execution. Finally, the job’s state will be changed to “Finished”.

This proposal has been implemented using Cassandra version 0.6, with the Cassandra Ruby client library version 0.8 [56]. Next, we describe the different objects that have been defined:

- Works: in this ColumnFamily, each row stores information needed by a pending job in order to be successfully run, such as the executable UUID, its state or the job’s owner identifier. The row key will be an UUID identifying the job. The configuration in Cassandra looks like this:
  
  ```
  <ColumnFamily Name="Works" CompareWith="BytesType"/>
  ```

  Here we can see how the parameter CompareWith tells to Cassandra that columns inside each row are (by default) ordered in a lexicographic order. As we do not need columns to be ordered we use this default value.

- Queues: this SuperColumnFamily implements queues using rows to store them. Each SuperColumn will contain information needed by a job to be scheduled. Its configuration in Cassandra looks like this:
In this case, SuperColumns are sorted by timestamp, parameter
\[ \text{CompareWith="TimeUUIDType"} \], because we want jobs to be
ordered according to their arriving time. All idle nodes will
read this structure to find suitable works to take care of. The
SuperColumnFamily key will be the queue name. Also, for each
queue in the system – in this proposal we use a general one, but
different queues could be defined, for example, for implement
priorities – there will be related queues in order to store
executing and finished jobs. The name of these related queues
will be QueueName="_Running" and QueueName="_Finished".

- **UserWorks**: this SuperColumnFamily links users with their jobs
allowing each user to administrate its own jobs. Each row contains
information about one user and SuperColumns contain
information about each user’s jobs ordered by time.

- **File storage area**: these ColumnFamilies provide a file storage
area where nodes can store executables, configuration files and
results files needed by the jobs. Each file is identified by a
different UUID, divided into chunks of variable length.
  - **FileMetadata**: this ColumnFamily stores meta-data about
each stored file. Each row represents one file, being the row’s
key an UUID, different for each file.

- **FileStorage**: this ColumnFamily stores an index of all the
different chunks that compose a file. This ColumnFamily
could be joined with FileMetadata but, for the sake of clarity,
we have divided them into two different ColumnFamilies.
Each row represents one file with its index and each column
contains the identifier, usually a SHA1 based UUID, of each
chunk.

- **FileChunks**: each row represents one file chunk in the system,
containing the Base64 encoded content of the chunk and a
SHA1 checksum.

- **StagingArea**: this SuperColumnFamily implements an area
where nodes can apply for getting a job of the queue. The
configuration is:

- **JobWorkers**: used for detecting collisions, it stores information
about the workers that execute each job.

### 6. Experimentation

In our experiments, we have used a workload generator [59]
that can be used to produce synthetic workloads for clusters,
based on probability distributions. Briefly, this generator produces
a workload of jobs with different execution and arriving times.
In order to control the maximum execution time of a job, an
exponential value $e^x$ is used as an upper bound (in seconds).
The arrival time of jobs is adjusted to provide a desired number of
jobs/second per node. Varying these two aspects, job’s duration
and frequency, we can test our system under different utilization
scenarios.

During the experiments we have measured these performance
gauges:

- **Bounded slowdown**: this is the waiting time of a job plus
the running time, normalized by the latter, and eliminating
the emphasis in very short jobs by means of putting in the
denominator a commonly used threshold of 10 s. In Eq. (1)
we can see the definition of bounded slowdown, being $w$
the waiting time of a job and $r$ its running time. Regarding
this value, note that in persistent saturation scenarios (those in
which the needs of the arriving jobs exceeds the system’s
capacity), it grows limitless.

$$\text{bounded slowdown} = \max \left( 1, \frac{w + r}{\max(10, r)} \right). \quad (1)$$

- **Total usage of the system**: the amount of time devoted by nodes
to execute HTC jobs. Expressed as a value between 0 and 1, or
as a percentage.

- **Scheduling time**: the time needed by an idle node to schedule a
waiting job.
• Collisions detected: the amount of jobs in each workload that have been executed by more than one worker.

We have tested our HTC–P2P system in two scenarios: the first one using a cluster with 20 nodes, running on each of them Cassandra version 0.6, Cassandra Ruby 0.8 and Ubuntu Server 8.10. Each node features an Intel Pentium 4 3.20 GHz with 1.5 GB of RAM and 80 GB of hard drive. The workload generator has been adjusted to generate 20 jobs per node on average. Experiments last the time required to consume all the generated jobs, and are designed mainly to assess the behavior of our system under different workloads. Each test has been repeated 10 times.

The second scenario consists of experiments carried out with different clusters of sizes 10, 20, 40 and 80 nodes. These clusters were built using Amazon’s Elastic Cloud Computing platform [60]. Each node has one core, 1.7 GB of RAM and 160 GB of hard drive, running the same software described for the first scenario. These tests have been designed to measure the scalability of our proposal. In this case each test has been executed a minimum of 2 and a maximum of 5 times due to cost restrictions and after confirming that measured figures from different runs show low variability.

Figs. 2–5 summarize results of the first scenario. In these experiments the maximum job size goes from 2 to 150 s, and load varies from 0.2 to 2.0 jobs/s per node. Fig. 2 shows the measured system’s bounded slowdown for these load scenarios. Fig. 3 shows system’s usage. In order to analyze system performance both figures should be studied together. The bounded slowdown is very small in almost all the situations. In the top-right corner it becomes higher; this area corresponds to a heavily loaded system, in which arriving jobs have to wait for nodes to become available after processing those already in the system. The higher the load, the longer is the waiting time. The increase in the bounded slowdown is, therefore, a consequence of the system load, but not an effect of a bad scheduling.

The previous idea can be confirmed looking at Fig. 4. We have measured the scheduling time for each idle node when there are pending tasks in the waiting queue. In all cases, it is very short, ranging from 0.03 to 0.05 s. The scheduling time is longer in lightly loaded systems, because idle nodes (most of them, in those scenarios) must compete harder to take control of pending jobs.

Fig. 5 illustrates the collision problem explained in Section 5. We have measured how often a job is selected to be run by two or more workers. For loads higher than 1.0 jobs/s per node this chance turns to be lower than 1%, independently of the maximum execution time per task. This indicates that our proposal to tackle the eventual consistency problem works and scales well.

Scalability tests have been carried out using the following parameters of the workload generator. The maximum job size was fixed to 150 s. Load varies from 0.2 to 2.0 jobs/s per node. Each node inserts on average 20 jobs into the system. Experiments have been made for cluster sizes 10, 20, 40 and 80.

In Fig. 6, we can see the occupancy of the system for the four different sizes. See how at load 1.0 jobs/second per node, occupancy is near 90%, for all sizes. In Table 3, we can see a summary of system occupancy together with the (per-node) deviation of this data. At low loads deviation is higher than 7% (meaning that it is highly probable that some nodes are idle while some others are busy), but when load increases, deviation decreases to less than 4% for any system size. Therefore, for all network sizes, the system is able to accept a high load level and to balance it evenly among participating nodes.

In Figs. 7 and 8, we can see scheduling times and percentages of collisions for different system sizes. If we focus on the scheduling
time, we can see again how it keeps below 0.04 s for small systems (10, 20 nodes), and below 0.06 for larger sizes (40, 80 nodes)—but in this case only for highly loaded systems. This can be explained again considering the contention at low loads. The percentage of collisions shows a similar trend: it is higher at low loads, and decreases substantially at higher loads, stabilizing around 0.5%. We can state, therefore, that our way of dealing with the eventual consistency problem scales well with system size as well as load.

7. Conclusions

In this paper, we have proposed a novel HTC system based on P2P networks. Our goal is to build a totally decentralized HTC system based on highly scalable and reliable P2P storage, maintaining these target characteristics: (1) lack of central management points, (2) disconnected operation and (3) FCFS execution order of jobs. For this purpose, we have implemented a distributed queue system that manages the execution of jobs over a group of nodes in a totally decentralized way. We have explained the main characteristics of our design, and have implemented a prototype based on the Cassandra P2P storage system. We have carried out an extensive experiment set to assess system behavior, confirming that it works as expected. Some issues remain related to the eventual consistency provided by the DHT storage. Still, a proposal to reduce it to manageable levels has been proposed and tested: the chance of collision is around 0.5%. We plan to further investigate this issue with the objective of eliminating it.

We have not included in the tests carried out for this paper experiments to assess system’s behavior under failure scenarios. We are currently modeling these scenarios in order to obtain meaningful measurements.

We plan to enhance the system by adding tools to simplify its utilization, allowing users to submit not only individual jobs but also more sophisticated workflows. We will also work in implementing different scheduling policies (in addition to FCFS), and have plans to include mechanisms to allow job checkpointing in order to allow higher degrees of fault-tolerance and load balance via job migration.

Acknowledgments

This work has been supported by programs Saiotek and Research Groups 2007–2012 (IT-242-07) from the Basque Government, projects TIN2008-06815-C02-01, TIN2010-14931 and Consolider Ingenio 2010 - CSD2007-00018 from the Spanish Ministry of Science and Innovation, and by the COMBIOMED network in computational biomedicine (Carlos III Health Institute). Mr Pérez-Miguel is supported by a doctoral grant from the Basque Government.

References


Jose Miguel-Alonso received the Ph.D. degree in Computer Science from the University of the Basque Country UPV/EHU, Gipuzkoa, Spain, in 1996. He is a full professor in the Department of Computer Architecture and Technology at the University of the Basque Country UPV/EHU. His research interests include interconnection networks for parallel systems, network (cluster, grid) computing, performance evaluation of parallel and distributed systems, and scheduling for parallel processing.

Alexander Mendiburu received the Ph.D. degree in Computer Science in 2006 from the University of the Basque Country UPV/EHU, Gipuzkoa, Spain. He is an associate professor in the Department of Computer Architecture and Technology at the University of the Basque Country UPV/EHU. His research interests include evolutionary algorithms, estimation of distribution algorithms, optimization techniques, and parallel and distributed computing.