Scalable Job Scheduling on a Distributed Environment

Marco Fiscato, 1637487

Supervisors: G.Pierre, P.Costa

Vrije Universiteit Amsterdam
Master in Parallel and Distributed Computer Systems

October 13, 2008

Contents

List of Figures

Chapter 1

Introduction

Grids are currently composed of a limited number of resources, administered under few administrative domains. Even though large Grid deployments exist, they are rarely composed of more than some thousand computing nodes distributed over a modest number of sites. Even in such medium-scale environments it is no simple task to identify N machines simultaneously available to run a job for a given duration, especially when N is greater than the size of the largest cluster on the Grid. This problem, named scheduling, is often addressed relying on local schedulers or meta-schedulers when dealing with multiple computing clusters. This second class may also try to co-allocate nodes across sites, even though this is not generally the case.

Moreover, Grids are changing. Computing power and bandwidth today are getting cheaper and cheaper, even commodity hardware can now achieve performances that before were only available in more expensive dedicated hardware and we can surely expect this trend to steadily continue in the future. We also observe an ever-increasing interest gained by voluntary computing projects like Boinc[2] and Folding@Home[1]. Here individual users or institutions donate some of their CPU cycles and bandwidth to a scientific project, allowing researchers to process more data than ever. So far, however, these projects have been limited to operate in the scope of the task-farm model, not being offered a real

distributed computing facility.

In general we observe a trend towards a more dynamic environment, with many more administrative domains and a much greater number of computing nodes. It is our opinion that the Grids of the future are not going to look as the ones we are using today. They will instead be characterized by a very high number of heterogeneous resources communicating over non dedicated channels and with a much higher degree of unreliability.

In this scenario it is clear that the nature and the size of these systems make them incompatible with traditional solutions. There is the need for a different and more sophisticated approach to resource discovery and scheduling. It is unfeasible to centralize the management of such a system. The number of resources involved would require a great effort just to be indexed, their fluctuant availability would also require a large amount of monitoring and we can argue that a real global view of the system could not be achieved. There are also some *political* issues related to such a centralized environment. It is important to note that these systems usually span across countries and are used by a number of independent (and often competing) research groups and companies. In such a scenario it is cumbersome to agree on a centralized authority. A distributed approach would definitely result in a simpler management of the infrastructure than meta-schedulers.

In such an Internet-scale kind of system failure will be the common case: due to its size, a fraction of Internet paths and hosts will be unavailable at any time. Another important challenge is represented by the number and heterogeneity of administrative domains involved, ranging from big data centers to single desktop machines. It is difficult in such an environment to keep track of and organize all the administrative domains and it would be useful to find a way to abstract from this.

Finally we have to consider the wide range of applications that could be run on such a system, all of which will have different demands and requirements to be satisfied. Computationally intensive tasks, like scientific simulations, might not be interested in the location of the computing nodes as long as the waiting time is minimized, while other more network demanding applications might want nodes to be as close as possible

to each other in order to reduce communication overhead. In this perspective we cannot employ traditional scheduling techniques based on heuristics, due to the variety of possible scheduling queries. It would be better, instead, to offer a generic optimization approach able to achieve good results for any given request.

A previous work in our group was focused on the solution of the same problem[11], but it presented a number of limitations due to the use of a single randomized overlay and a single "as soon as possible" query policy.

In this thesis we address all these issues by employing a novel approach. We start by building a pure P2P system in which every resource is autonomous and not subject to a higher authority. It means that every computing node is considered as an independent resource, not as a member of a cluster. This allows to abstract from the administrative domain during the process of scheduling. What is really important is the node future availability, defined by the list of future jobs already accepted by the node, and its position, which is defined by its network coordinates.

In order to be able to spread this information quickly and reliably we make use of epidemic protocols. Every node is participating in a number of overlays and spreads information about itself by gossiping with other nodes. This approach has proved to be effective both in regard of node discovery as well as for failure detection, thus making it very suitable for this kind of environment.

In order to deal with the problem of query optimization, our solution employs a novel approach in which the user specifies a policy describing the wanted characteristics of the computing nodes in the form of a group scoring function, which is then optimized by the system using a genetic computing approach. This allows maximum flexibility in terms of possible queries that a user can submit, while still achieving good performance at all times. It is important to note that the user is provided with constant feedback on how well the query is performing, by being able to monitor in real time the improvements obtained by the algorithm.

We propose a novel organizational model for Grids based on P2P approach, in which users have the possibility of requesting computing nodes specifying a policy describing the group characteristics of these nodes according to the nature of the submitted job.

This thesis is structured as follows: chapter 2 discusses some other schedulers which have already been deployed, chapter 3 describes in details the working of the algorithm and chapter 4 provides present the results of the experiments run to evaluate the performance of the algorithm.

Chapter 2

Related Works

Many solutions have been developed to solve the scheduling problem. This section surveys existing systems and evaluates their characteristics. Local schedulers, like Condor, were developed to schedule jobs in a single compute farm. Meta-schedulers were then introduced to allow multiple sites to cooperate. An example of such a system is KOALA. The difficulty of allocating jobs larger than the maximum size of the biggest cluster and the need for a more efficient organization of the Grid led to the creation of other solutions such as the Organic Grid, Zorilla and SAGA.

2.1 Condor

Condor is a specialized workload management system for compute-intensive jobs. It allows the submissions of jobs like a regular batch scheduler. Jobs are placed in a queue and subsequently scheduled on the available resources. The choice of when and where to place a job is defined in a policy expressing the requirements and preferences of the job. The framework also offers the possibility of monitoring jobs during execution and to be notified upon completion. One of the important features offered by Condor is the possibility of exploiting CPU cycles in idle machines. Once a machine inserted in a computing pool is detected as idle it becomes eligible for job execution. In many circumstances Condor is able to

transparently produce a checkpoint and migrate a job to a different machine which would otherwise be idle. An extension of the system, called Condor-G has been developed to allow the use of Condor in Grid-like environments. It leverages the resource management facilities offered by the Globus toolkit, and integrates as the scheduling component for the Grid. This allows to span jobs not only across the local cluster, but also to exploit remote resources, thus characterizing it as a meta-scheduler. Condor-G combines the inter-domain resource management protocols of the Globus Toolkit and the intra-domain resource and job management methods of Condor to allow the user to harness multi-domain resources as if they all belonged to one personal domain. Another important feature of Condor is stability. Its first implementation is more than 15 years old and the community revolving around the project kept it continuously growing and improving. Even though Condor, especially when used with Globus, is a viable solution for medium sized computing pools, it still does not provide the scalability needed to move to the next generation of Grids. In a scenario characterized by a high number of independent administrative domains setting up and maintaining the Globus infrastructure becomes over complicated because of the difficulties in the management of permissions. The issues related to resource discovery and allocation are somehow addressed, but it still proposes a centralized solution, which, as explained before, is not compatible with the large-scale scenario that we will be dealing with.

2.2 Koala

Koala[4][5] is a centralized grid scheduler featuring co-allocation and fault tolerance. In this model a job is submitted to the scheduler, which will try to schedule it, possibly on different clusters according to the local availability. Two main job-placement policies can be specified: Close-To-File and Incremental Claiming Policy. In the Close-To-File policy the job is placed in nodes that are close to the input data, in order to minimize the transfer overhead. The Incremental Claiming Policy, instead, gives the possibility for high priority jobs to preempt low priority ones before

they start executing. Note that this is far from what a real preemptive scheduler, which can interrupt jobs to give precedence to others with more priority. This is more of an specific feature developed to fit the KOALA model. One of the key features of KOALA is its meta-scheduling model. In this model a meta-scheduler monitors the state of several clusters and relies on the local schedulers for job submission. This hierarchical model uses a strict notion of cluster, as an independent site, with a specific scheduler. Whenever useful it tries to co-allocate the job, thus splitting it over a number of clusters. Even though, in principle, this behavior is desirable, in reality it is difficult to obtain true co-allocation, because of concurrent requests, especially when the number of nodes required is bigger than the biggest cluster.

Even though KOALA is a fairly advanced solution for Grid scheduling it still presents a number of issues. First of all, it only offers two policies, which may not be adequate in many scenarios. Because of the adopted model, only a limited number of new policies could be added, still not allowing full control by the user. Another aspect to consider then, is the scalability issues inherited from the centralized model. Monitoring is going to be cumbersome: as the number of resource grows it becomes virtually impossible for a single entity to keep track of all the available resources at all times. Also when the number of scheduling requests grows, a centralized solution would represent the bottleneck of the system.

2.3 Organic Grid

A more decentralized approach to job scheduling is proposed by the Organic Grid[6]. This infrastructure is biologically inspired and exploits autonomous scheduling. This solution shares many assumptions with ours. First of all it assumes a fully decentralized organization of the Grid, completely abandoning the meta-scheduler logic and relies instead on a P2P network. It also realistically drops the assumption of being able to obtain a global view of the system. In such a large and unreliable network the maintenance of this global knowledge would become prohibitively expensive and might even not be possible. A lot of stress

is put into the self-organization of peers. The Organic Grid encapsulates computation into mobile agents that deliver it to the available machines. Once an application is started at a node, it is spread to others, which are called in to contribute resources. New mobile agents are created that, under their autonomous control, readily colonize the available resources and start computing. The authors show that this approach can be employed in many different algorithmic scenarios. In order to demonstrate the capabilities of their system they implemented two very different ones: the NCBI BLAST code for sequence alignment and the popular parallel matrix multiplication. As explained, when a job is initiated at a node, it quickly tries to spread the load to other nodes. In this regard we could say that scheduling is taken care of by the infrastructure, since as new computation is needed, more helper nodes will be contacted. This definition of resource scheduling however, is different than the one we employ. Here only a currently unloaded node will accept computation, thus resembling a resource allocator more than a resource scheduler. At the beginning of a job in the Organic Grid, helper nodes are chosen in a way that is completely oblivious of distance. During the course of the computation agents behavior encourages the propagation of computation among well-connected nodes, while discouraging the inclusion of distant (i.e. less responsive) agents. This selection is based on a particular version of the emergence principle called Local Activation, Long-range Inhibition. Instead of using distance as a metric, performance is used instead. Agents are free to flow from one node to the other, carrying computation around and trying to optimize the overall performances of the application. They are linked together in a tree fashion, in which the root is represented by the initiator node. This tree overlay is incrementally restructured while the computation is in progress, by pushing fast nodes up towards the root of the tree. This system shares many assumptions and techniques with ours. It employs a P2P network and self-organization among its peers. Scheduling a job can be done at any node, but while in our system this is not influencing the choice of the executors nodes, in the Organic Grid we would expect the contrary. Submitting a job at a particular node will change the location of the helper nodes, at least in the initial phases. It is important to point out that a significant difference also exists in the

definition of job scheduling. In the Organic Grid computation starts as soon as it is submitted, grabbing available resources as it goes. We propose instead a solution which is more similar to a classical grid scheduler, in which a job is delayed until a good set of resources becomes available. In our definition of scheduling a computation does not need to be started immediately. It can also reserve resources in the future if the global quality of the reserved resources is better.

Even though the Organic Grid represents an interesting approach to the problem of scheduling on very large grids, it does not represent yet an acceptable solution. It does not allow real job scheduling and it does not provide enough control to the user on where jobs should be allocated.

2.4 SAGA: Scheduling Applications using Genetic Algorithm

Another solution to the scheduling of in an heterogeneous grid is SAGA (Scheduling Application using Genetic Algorithms)[1][9]. They start from the consideration that the problem of scheduling heterogeneous tasks on heterogeneous resources is intractable, thus making room for a good heuristic solution. The scenario in which the scheduler operates is the same as the one we consider in our study, and probably represents the most difficult scenario to be addressed efficiently. It assumes that jobs are submitted to the scheduler at random intervals, also specifying different sets of resources to be allocated for a variable amount of time. Despite the difficulty of addressing the scheduling problem in such a complex system, a solution would be extremely valuable in a real world deployment, giving maximum flexibility and usability to the user. The first problem to address when dealing with large-scale heterogeneous grids is the monitoring of the resources. As we already explain it is very unlikely to be able to keep track of such a large set of resources in a centralized way, and we might argue that a global view is probably impossible to achieve. To deal with this problem SAGA relies on a distributed grid

[1]Note that this work is unrelated to SAGA (the Simple API for Grid Applications).

monitoring tool called MonALISA. This is a monitoring platform specifically designed to obtain real-time information in a heterogeneous and dynamic environment such as a Grid. By the mean of this service a node is able to obtain a global view of the system, in a very scalable way. It is difficult to state how scalable this solution is and, even though it proved to be valuable in many real world deployments[10]. The second main idea behind this solution is the use of a genetic algorithm to quickly approximate the schedule for a job. We also employ a genetic algorithm to solve this optimization problem. It is known that the scheduling of jobs as we intend it is a NP-complete problem, this means that there is no real solution to it in reasonable time and that some kind of optimization needs to be used. Evolutionary computing offers a valid approach to the solution of this problem, offering to calculate near-optimal solutions in very short time. In this scenario the time constraint becomes crucial. When a possible schedule is selected for the job it needs to be evaluated and improved quickly. This is necessary in order to be able to evaluate as many solutions as possible, as well as for avoiding the schedule to expire. The SAGA system also assigns roles to the nodes and distinguishes between schedulers and executors. It means that a certain number of nodes are selected to cooperate in the scheduling action and, once the schedule has been calculated, the Executors will be reserved to the job. This division of duties based on roles is also similar to the one employed by our system. When a job is submitted the schedulers calculate which resources to allocate based on the information retrieved from the MonALISA monitoring system. They can employ different strategies even though experiments show that cooperation and decentralization lead to the best results.
SAGA probably represent the most similar system to the one we are proposing in this thesis, sharing many assumptions and solutions. Despite the similarities, though, many differences remain. The main concerns we have are due to the assumption that a global view can be achieved at all times. We strongly disagree with this assumption and thus our system addresses the same problem, but by only relying on the local knowledge collected by the peers. Among all the considered systems, this probably represents the closest solution to our own and the

most promising of the alternatives.

2.5 Zorilla

Zorilla[7] is a P2P middleware developed within the IBIS project[2]. Zorilla is a replacement for other middlewares like Globus, to be used together with IBIS. The main difference between Zorilla and other systems is its P2P nature, thus breaking down the classical notion of cluster. It takes care of all the organization and maintenance of the overlay by making use of gossiping. It is possible to schedule jobs on Zorilla, which will take care of locating the necessary resources and deploying the job on them. As soon as a job is submitted the middleware starts the scheduling algorithm. This begins exploring the network starting from the node where the job was submitted. It contacts neighbour nodes trying to find available ones, iteratively expanding the set with other nodes close to the ones already contacted. As soon as it has found a sufficient number of idle nodes, the job can start.
The main advantage of Zorilla over other systems is its P2P nature. Not relying on a single entity for organization, but solely on inter-peer communications, it can theoretically scale to a very large number of computing nodes. Another strength of the system is the possibility of submitting a job at any location, thus eliminating the bottleneck represented by a centralized point of submission. Moreover, even though a scheduling action can be started at any node, the choice of this initial nodes does affect the choice of peers for computation. In general we could argue that submitting at different nodes will result in different schedules. The main drawback of Zorilla lies in its scheduling algorithm[8]. It uses, in fact, a bounded flooding mechanism for resource discovery. Every node keeps a cache of its nearby nodes. Once a scheduling request is submitted this is forwarded to all the neighbours, and so on, until N suitable nodes are discovered. It is obvious how this approach presents some serious scalability issues. In general, on a very loaded Grid, all nodes of the system

[2]IBIS is a more high-level environment whose goal is to create an efficient Java-based platform for grid computing.

could be contacted. Also, the algorithm cannot be characterized as a real scheduler, since it only grabs resources as they are found available, but does not take into account future availability. The model is also limited to an "as soon as possible (and close)" policy, which may not suit the needs of the user.

2.6 Gossip Based Grid Scheduling

The work that more than any other else is related to the one here proposed is the "decentralized grid scheduler", proposed by Caglar Oner in his master thesis[11]. Our work is directly influenced by the findings of this previous effort and tries to address some of its shortcomings, as well as to extend its model. The two works try to address the same problem in the same scenario: to allocate N nodes simultaneously available in the future, assuming exclusive use of them. We assume there is no possibility of achieving a true global knowledge of the system. Instead every node maintains some local information which is exploited during the scheduling action. In the case of Oner's work every node is connected to some other nodes, in a randomized overlay. This means that every node keeps a cache of K nodes (with K<<TotalNodes), chosen arbitrarily by the underlying protocol. The maintenance of this overlay is realized with the Cyclon gossip protocol[12]. This is an epidemic algorithm, which has proven to be very robust even in presence of massive failure of nodes in the network, preserving the connectivity of the overlay.
The system does not employ any specific scheduler. Instead any node can become the entry point for the submission of a job. This peer is the only one responsible for the computation of the final schedule, even though it exploits the information stored at other peers in order to calculate the final result. It starts by constructing the first tentative schedule grabbing the first P nodes at random, fetching them from its local random cache. Then at every gossip iteration with another peer, it tries to merge the new known peers one by one into the current tentative schedule. When, by replacing one of the peers with a new one, the obtained schedule improves this becomes the new tentative one. After a certain number of

iterations the schedule is finalized. This happens because the calculated tentative schedule is about to expire, meaning that its starting time is approaching the current wall-clock time. When selecting the executors nodes, the system is only concerned with the starting time, thus only employing an "As Soon As Possible" policy. However in such a large system, where resources are also largely distributed in space it seems naive to optimize only on starting time. Many parallel algorithms make use of massive inter-node communication. In this case reducing the overall latency among executors nodes becomes crucial for the performance of the algorithm.
Building on this previous work we try to extend the criteria used for executors selection, for instance taking latency also into account. Furthermore our solution extends the concept of scheduling policy, by letting the user specify exactly how the executors should be chosen, and by providing some internal data structures that can be exploited in the optimization process. We also argue that the mechanism for improving the tentative schedules is not efficient enough. The main drawback associated with this approach is its difficulty to evolve. Once a tentative schedule has been calculated it tends to remain stable. It is unlikely in fact that a single new peer is able to improve the overall quality of the schedule. On the other end, it would probably be a better idea, especially in the scope of multiple optimizations criteria, to have a larger cluster to merge and calculate from.
This system represents the base for the work of this thesis. We started by analyzing the strength and the weaknesses of the system and building from it. We extended the policy model in order to give maximum flexibility to the user when specifying the characteristics of the requested executors, for instance by adding the opportunity of taking latency into account on the side of starting time. We also extended the overlay model, by adding to the simple randomized one, some others in which information is selective (as the overlay of close by nodes). Also we proposed a different approach to the computation of the schedule, which does not rely only on gossip, but proactively constructs it by gathering information from other nodes. In the end, a genetic algorithm is employed in the computation of the schedule in order to achieve near-optimal results in

reasonable time.

Chapter 3

Decentralized Scheduling Algorithm

3.1 Introduction

The system described in this thesis allows the scheduling and deployment of parallel applications on a distributed environment. A scheduling query requests N machines for a certain time T, on which to have exclusive access for job execution. It also gives the possibility of specifying a policy describing the features that this group of nodes should have as a whole. Throughout this document the entities taking part in the process will be referred to as *submitter*, the node where a job request is issued, and *executors*, the nodes actually carrying out the computation for the job. The scheduling action can be performed by any node in the system. Typically the work is carried out locally by the submitter, but can also be split among several nodes in a cooperative fashion. After the scheduling phase is concluded a set of N machines will be allocated for the job for the requested amount of time T_S, starting at T_0. Then the input files and the actual program are copied to the executors, which at time T_0 will start the computation. Upon completion the submitter receives the generated output files and the job ends. It is important to note that the system enforces mutual exclusion of jobs, meaning that an executor will always be running at most one job at any time.

3.2 System Model

The system described is very different than regular Grids. These are usually composed by few local clusters linked together in a bigger entity. Examples of this kind of systems are Grid-5000 and the DAS-3. In these systems resources are organized by a meta-scheduler at global level, which relies on the local schedulers of the clusters to obtain information about load and to reserve resources to jobs. In this kind of environments there is a big difference between nodes within a cluster and nodes in different ones. In the first case communication is supposed to be very quick due to the very small latency and the presence of dedicated links between them, while, when dealing with nodes in separate cluster, latency is usually larger and the channel public. In the new Grid scenario that we delineated the distribution of resources is much more diverse and there is basically no assumptions we can make about the geometry of the system. In general we can consider resources to be equally scattered around the globe and communication to be happening over the Internet. Because of this peculiarity it becomes useful to extend the query model in order to take latency into consideration. Many distributed applications are characterized by heavy communication patterns, in this cases minimizing the inter-node latency is crucial for the performance of the algorithm. When scheduling this kind of jobs the optimization should not only be on starting time, but it should also aim at minimizing inter-node distances. In general the criteria should minimize the global duration of jobs, from submission to completion.

3.3 Implementation

This system is based on the p2p library ELEOC(Epidemic Library for Easy Overlay Construction) that allows the easy creation of p2p networks in which peers are linked by one or more overlays maintained using a gossip algorithm. At the moment the system is developed in Java. This allows to exploit the "write once, run everywhere" characteristic of this language, making it possible to execute the system on a wide range of architectures and operating systems. In practice it is possible to schedule

for execution applications written in any language, even though at the present time only Java applications are supported. It is easy to safely deploy these applications because of the security mechanisms offered by Java, like sandboxing, while for applications written in other languages a different confinement approach should be used, like virtual machines.

3.4 Overview

The system is designed as a pure p2p system in which all nodes are equally able to schedule as well as to run jobs. Because of the nature and the size of the resource pool it is unfeasible to rely on a global view of the system. It would be impossible to track all resources at all times, first of all because of their number, which is supposed to be very large, also because of their fluctuant availability and their geographical distribution. We also assume a part of the machines will be unavailable at any time, because of the instability of Internet links. The system, instead, takes a different approach and assumes there is *no global view* of the system. The only information available is the one contained in the overlay caches of the nodes, meaning their direct neighbors in every overlay. Every node participates in a number of overlays, each one connecting it to other nodes with certain characteristics. By linking nodes in specific overlays we can maintain useful information on the system (or on a part of it), that can be used to schedule jobs. For instance, nodes can be linked in a *proximity* overlay, in which every node keeps track of its closest nodes according to some distance metric[1], for example latency. In this way every node can be constantly aware of its K "closest" nodes.

3.5 Maintained Overlays

Whenever a scheduling query is submitted a node becomes the *submitter* for that job. This means that all the computation needed to calculate

[1]The system employs *Network Coordinates*[14] to position nodes in an euclidean space in which the distance between two nodes is an approximation of the real-world latency between them.

the schedule is performed locally by this node[2]. We assume there is no possibility of obtaining a reliable global knowledge of the system. The algorithm relies only on the local knowledge maintained by every node. Every peer, in fact, participates in a number of overlays, each containing a cache of other nodes chosen according to some metric. The algorithm exploits this local knowledge in order to calculate a tentative schedule. Instead of relying on a global view of the system it repeatedly use a partial view of the system constructed from these local caches. A node is always participating at least in one randomized overlay, kept up to date using the Cyclon gossip protocol. Furthermore a node can maintain other caches, containing nodes with similar characteristics chosen according to some criteria. Here a slightly different gossip protocol is used, called Vicinity[13]. For instance, it can be useful to keep a cache of the K closest nodes, meaning those node with the lowest latency from the peer. This sort of neighborhood is very useful when the scheduling query aims at minimizing the latency among the *executors*, like in the case of network intensive applications. Another optimization criterion which is very popular is the *As Soon As Possible*. This is usually the case for CPU intensive applications, where the inter-node communication is close to zero, such as parametric simulation. In this scenario it doesn't really matter where the code is executed, while it is more interesting to minimize the waiting time for the user. In this situation it is useful to maintain a cache of nodes ready to execute in the near future. Since the system doesn't restrict the execution time slot to any length, it is necessary to keep more than one overlay for this purpose. For example we can link nodes with an available time slot of at least one hour starting soon, then another cache for slots of at least six and twelve hours. Maintaining slots of one hour it is possible to schedule jobs up to an hour. It is intuitive that it is easier to schedule shorter jobs, while it is becomes much harder when the length of the job gets closer to the length of the time slot for that cache. In general when the number of required executors is high and the required execution time is close to the length of the cache time slot, it becomes more difficult to find a schedule. It is then preferable to use

[2]As we will see this process can be parallelized since there are no dependencies between iterations of the algorithm.

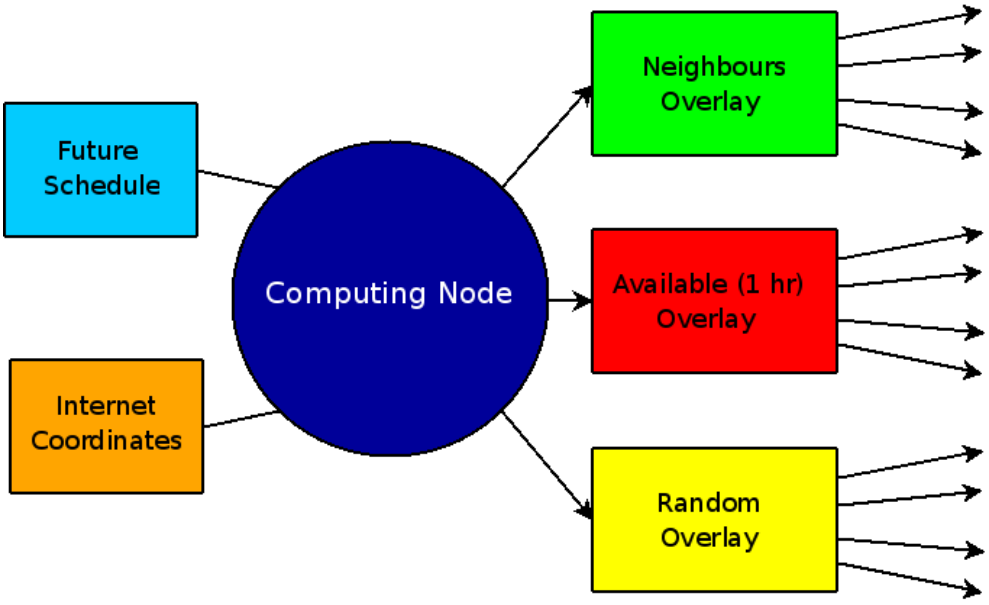


Figure 3.1: A peer structure: on the left the gossiped information are shown, while on the right there are the maintained overlays.

another cache with a greater time slot length. Differently than in the case of proximity, where every overlay is different, these time slot caches will all be the same at every nodes, or at least similar, in case the number of available nodes is much bigger than the cache size. This is particularly useful on very loaded grids, since every node will be aware of the few unloaded ones.

These are just some of the possibilities when choosing the overlays to be maintained locally by the nodes. In this work we focus on optimization criteria for network intensive and for CPU bound applications. In the same way it is possible to maintain other overlays, customized for specific applications or scenarios. It is important to note, though, that in order to work these overlays have to be deployed on a sufficiently large number of nodes. It would be not enough to start an overlay locally because, for how the Vicinity algorithm works, it is necessary to have a big enough number of nodes participating in an overlay in order for it to converge. This

consideration is particularly useful in the context of *voluntary computing*. In this case a certain number of nodes, potentially very big, is interested in running the same code, an example could be the SETI@Home project. These nodes are all participating in a collaborative project and thus share the same goals. Deploying a customized overlay in this situation would result in a great improvement in the schedules, because all the nodes participating in the project could be grouped according to the best metric for the specific application.

In general, when it is not feasible to deploy a custom overlay and the optimization criterion is not among the default ones, it is always possible to rely on the randomized overlay. We will show that the performance, even if lower, is still acceptable.

3.6 Policies

When selecting a tentative schedule the system needs to evaluate its goodness. Different applications have different needs and a good schedule for one may not be as good for another. In order to define a criterion for the goodness of a tentative schedule the system employs a mechanism based on *policies*. These are user-defined criteria that can evaluate a group of nodes and assign them a score. At the core of a policy is the *group scoring function*, which receives in input a set of nodes and returns a number expressing their goodness. In our system a low value is considered better than a high one. Many requirements could be useful when dealing with different applications. Because of this the query model of the system has been extended in order to try to fit all user requirements. As a matter of fact, the chosen approach resembles a black-box model, by letting the user specify how she wants the executors to be, in the form of a *group scoring function*, to be minimized by the system. In practice the user queries the system by specifying the number of requested nodes, the amount of time needed for the job, and a policy. In the simple case of an *As Soon As Possible* policy, for instance, the group scoring function could be expressed as: *the waiting time* to start the job[3]. In the context

[3]All nodes obviously start at the same time.

of network intensive applications instead, the policy becomes: *As Close As Possible* and the group scoring function could be expressed as: *the sum of the distances to the center of the group*. In real world applications the group scoring function would probably be a combination of the two described, in which the starting time and the latency are both taken into account, in some proportions.

Low latency and short waiting time are probably going to be the most common optimizations. For many applications though, these criteria could be not sufficient. A user might want, for instance, to schedule a job that needs to be replicated globally, thus needing peers *as far as possible* to each other. In this case it is easy to understand that the wanted behavior can be achieved by simply reversing the formula above. Another possible constraint could be put on which nodes to avoid for computation, for example a company might not want to run code on a competitor's machines. This can be achieved with a simple change in the policy, which would invalidate a group in the presence of a blacklisted member. Also, a user might want to take hardware characteristics into account, when choosing executors. In this case the group scoring function is going to award points according to the hardware configuration of the members of a group.

From this description it should be evident that almost any kind of grouping policy can be specified when querying the system. The user has simply to submit a group scoring function describing the wanted group scoring function and the system will try to minimize it. This means that of all considered groups, the one returned as the executor pool, is going to be the one for which the policy function scores the minimum.

A policy, besides the *group scoring function*, also includes a set of *significant overlays*. These are the overlays that are thought to be more likely to contain nodes with the right characteristics, and thus, with the higher chances of producing a good schedule. An "As Close As Possible" policy, for instance, is more likely to produce good results when it works on nodes coming from the *proximity overlay*, meaning those nodes that are already known to be close to each other, than with nodes coming from the *random overlay*. Sometimes there is no specific overlay that fits perfectly the requirements of the policy. In this cases it is possible to

approximate, by specifying a combination of overlays, or in the worst of cases, just including the randomized one.

We will see in the next section, how this information is exploited by the system during the scheduling action.

3.7 Scheduling Algorithm

When a scheduling request is received at a node, this node becomes the *submitter* for that job, the node that will take care of calculating the schedule. The sequence of steps required to calculate a tentative schedule is iterated several times. At every iteration the system tries to improve the tentative schedule. It is important to note that these iterations are all independent from each other and, thus, could be carried out also by different cooperating peers.

Looking at every iteration we can divide the algorithm in the following steps: *Fetch/Merge*, where a group of nodes with similar characteristics is gathered, and *Calculate*, where the best schedule is found among that group. The algorithm starts by selecting a random node from its Cyclon cache, we call this the *pivot* for this iteration. This node is contacted in order to obtain the dump of one or more of its caches, according to what specified in the given policy. For instance when using a "As Close As Possible" policy, we are interested in the nodes contained in its proximity cache. Since these overlay caches have a static size (usually 20), it is likely that more nodes will be needed before it is possible to calculate the schedule. This initial group, out of which a tentative schedule will be calculated, needs necessarily to be be larger than the size of the final schedule. In general we can say that, if looking for N machines, we need to gather at least $\alpha \times N$ nodes, where α is a system parameter greater than 1[4]. In order to reach the threshold of $\alpha \times N$, other nodes needs to be contacted to obtain a dump of their caches. These secondary nodes to be contacted are not chosen randomly though, but instead from the partial group, since we assume these to have already some characteristics in common. Then, when the other caches are fetched, they are merged

[4]Experiments show a good trade off is given by $\alpha = 2$.

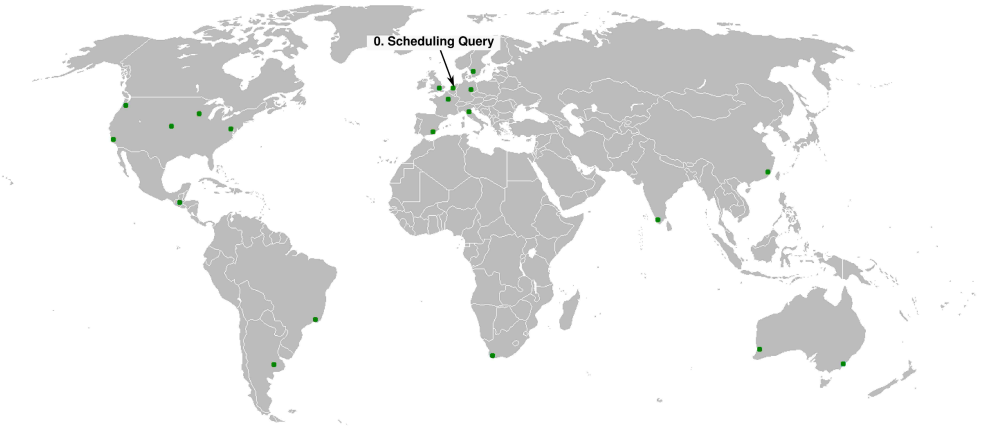


Figure 3.2: 1. First a scheduling query asking for N executors is received by a node

with the partial group and the duplicates are removed. This process continues until $\alpha \times N$ different nodes are collected.

At this point we need to select the best set of N nodes among the gathered ones, the ones representing a tentative schedule. This process consists of selecting the N best nodes out of K($\alpha \times N$), according to the given policy. The space of possible solutions here is very large and impossible to explore exhaustively. In fact we need to consider all the possible combinations of N nodes out of K, which is $N!/(N-k)!k!$. This formula contains factorials and its result becomes very large quickly as the value of N increases. Since it is unfeasible to calculate the best solution in reasonable time, we need to employ some kind of approximation. In order to do so, the system uses a *genetic algorithm*, which is able to obtain a near-optimal solution in polynomial time. Note that we have no guarantee that the found solution is the best one, but we know it is among the best ones. The choice of employing this kind of technique is due to the need of calculating solutions fast, a typical trade-off where the quality constraint is loosened in order to gain speed for the algorithm. Once a solution is calculated it is then compared against the current best one calculated by a previous iteration of the scheduling algorithm. If

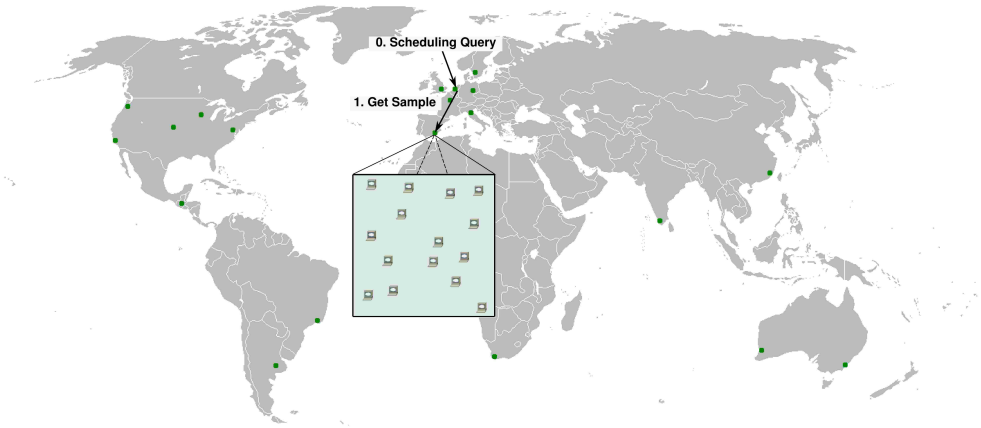


Figure 3.3: 2. This then selects a pivot and gathers nodes

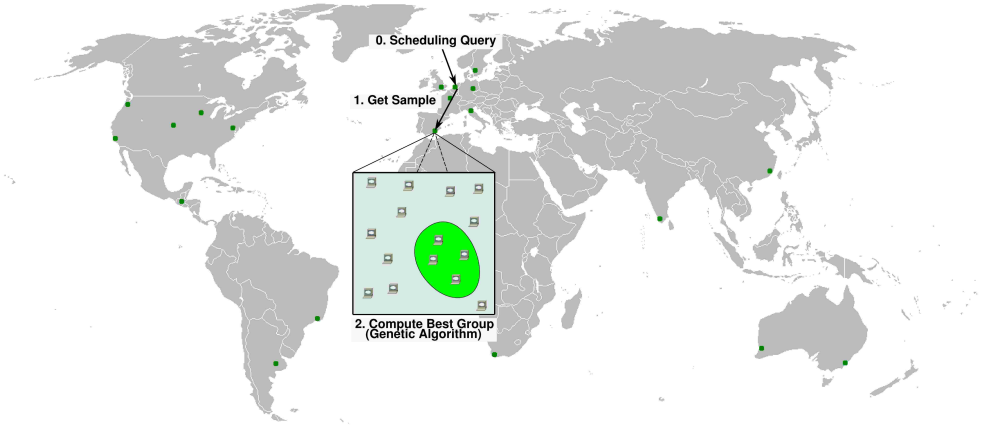


Figure 3.4: 3. A tentative schedule is calculated using the genetic algorithm

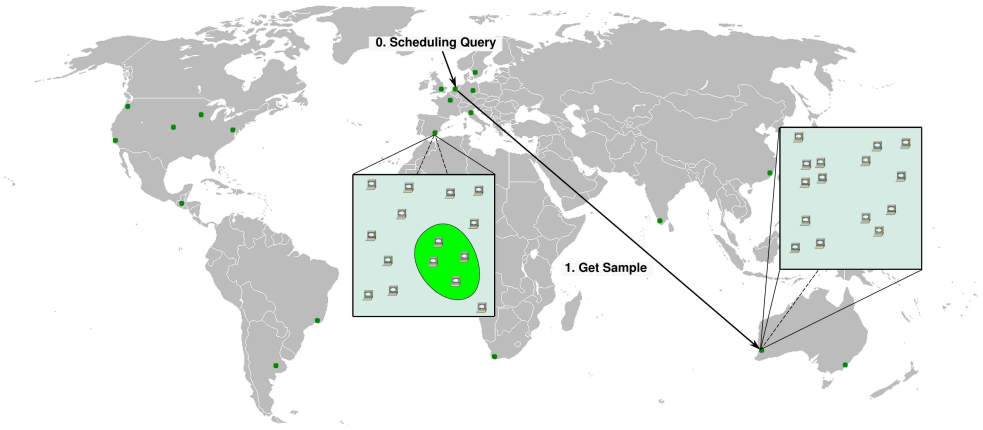


Figure 3.5: 4. A new pivot is selected

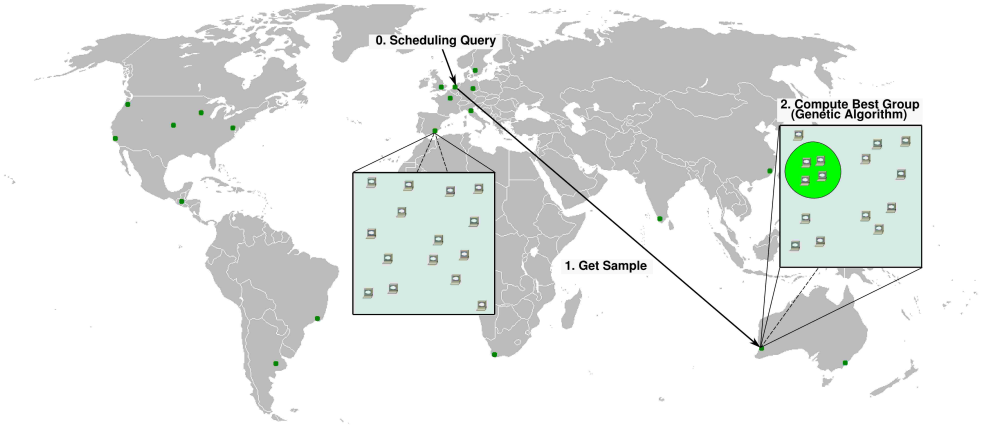


Figure 3.6: 5. A new better tentative schedule is found, the old one is discarded and the resources freed.

it is found to be better than the old one, it becomes the global best. At every iteration an attempt is made to improve the tentative schedule. All the iterations are independent and can be carried out locally by threads or remotely by other helper nodes. When the starting time for the best schedule is approaching the wall-clock time the whole procedure terminates and the tentative schedule becomes definitive.

3.8 Resources Reservation

In the described system it is possible that one resource is included in more than one schedule, when several scheduling queries are performed at the same time on the Grid. The whole calculation, in fact, is performed by the submitter and the executor is largely left out of the procedure. It might happen, for instance, that two or more submitters decide to include the same executor in their schedule, but when finalizing, only one will be able to execute in that timeslot at that executors. This clearly leads to inconsistency of schedules and thus to an overall decrease in the reliability of the system. We need to have a mean of making an executor aware of its role, so that it can update its future schedule accordingly, and at the same time, we need a way to avoid the phenomenon of *overbooking*. The employed strategy is fairly simple and only requires a small booking procedure. As we have seen, many tentative schedules are generated when executing a scheduling query, and every time one is calculated this is checked against the temporary best one, replacing it in case of improvement. It is easy to picture and practice also shows, that this replacement only happens frequently at the beginning, while becoming less and less probable as the algorithm proceeds. Exploiting this consideration we decided to always reserve the *best temporary schedule*, as soon as it is found, without waiting for the end of the scheduling calculation. It means that a submitter reserves the executors as soon as it finds a better schedule, to replace the old best. Obviously as it reserves the new one, it also needs to release the old one. By doing so, a node is booked in a certain time slot as soon as it is included in a schedule, which is considered the best so far, and released as soon as a better one is found.

Because of the possibility that some resource will become unavailable during the reservation process, a two-phase commit transaction is used. Even though this system tends to over reserve resources, this is only for a limited time, but it has the benefit of preventing overbooking.

Chapter 4

Algorithm Evaluation

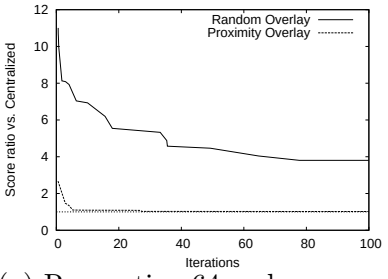
In order to evaluate the performance of the system we would like to compare its results against the theoretical optimal ones. It is known that the scheduling of jobs is a NP-complete problem. This means that for large size of the Grid it becomes unfeasible to calculate the best possible scheduling for a job. Every centralized scheduler needs to make use of some heuristics, we chose to rely on a genetic algorithm. To be able to evaluate the performance of the decentralized algorithm we compare it with its centralized version. This will give us results which are not influenced by the quality of the chosen genetic algorithm. One of the assumption we made throughout this work is that a global view of the system cannot be achieved. For testing purposes though, we can work around this limitation since some attributes of the nodes are known and static. For instance we can use position, expressed as the Internet Coordinates of the nodes. Using these coordinates we evaluate the quality of queries on the system based on the As Close As Possible policy. Instead of choosing random values we opted for a more realistic solution by exploiting a dataset based on a previous work[14]. We used coordinates of real nodes, in order to achieve more realistic results. A scheduling query of this kind tries to obtain N nodes that have the minimum latency among each other. This is particularly useful when dealing with network intensive applications characterized by a high degree of inter-node

communication. Since the position of nodes is known by construction we can run a centralized version of the algorithm and compare it with its decentralized counterpart. This means that the genetic algorithm is run on the whole set of nodes. The best result out of several runs then represents our baseline or optimal value. In principle we would like the decentralized version of the algorithm to perform as close as possible to the centralized one. Another important characteristic to evaluate is the speed of convergence. The decentralized algorithm should find close to optimal schedules in a small number of iterations. Furthermore the algorithm should be stable for different size of the network. This means that its performance should not change significantly as the network size grows. For our experiments we emulated two networks of different scale: one of 800 and the other of 10000 nodes. We run the experiments on the DAS-3 cluster. We submitted queries for 64 and 128 nodes, since this is usually a big enough request to require co-allocation on many traditional clusters. We also distinguish two cases: one only using the randomized overlay and one exploiting the custom proximity overlay. We calculate a ratio between the goodness value found by the decentralized algorithm and the optimal result calculated by its centralized counterpart. This gives an idea of how much these results differs, with one being the optimal value.

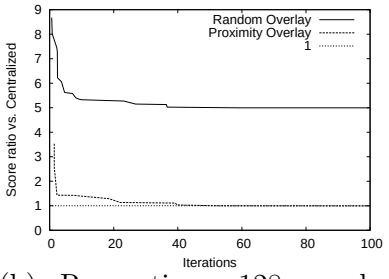
We can see from the graphs that the system behaves really well for both sizes of the network, especially when the custom overlay is employed. The decentralized algorithm performs as good as the centralized one, always finding almost the same results. Moreover we can see that a very small number of iteration is necessary to find this optimal result, showing very good convergence properties. By looking at the different sizes of the network then, we can see that this number of iterations doesn't really change. This allows us to conclude that the quality of the resulting schedule and the convergence speed are not influenced by the number of resources in the system.
We also tested the same algorithm, but just using the information of the randomized overlay. This gives us an idea of how a generic query would behave in absence of a suitable custom overlay. It also gives us an idea of how much a query is improved by the presence of a specific overlay. In

this case the quality of the schedule is not close to optimal and its quality degrades when the size of the network grows. The subset of nodes present in the randomized cached in general will never be the same as the one in a custom overlay and thus will always lead to lower quality of the results. In this case it is obvious that nodes chosen randomly have a very low probability of also being close to each other. Since we are trying to select nodes as close as possible, it is understandable that the quality of the calculated schedule is also lower. One good property that is maintained is the speed of convergence. Even if the network grows we can still expect the system to stabilized in a reasonable number of iterations.

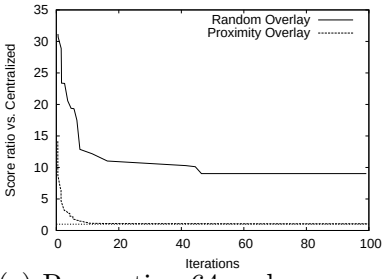
We have seen how the system behaves for a specific policy with and without specific information available. We determined that the decentralized version of the algorithm performs nearly as good as the centralized



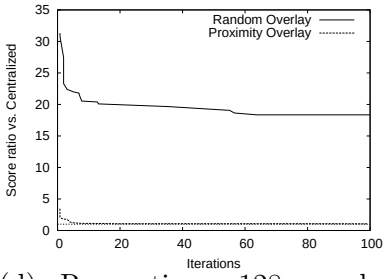
(a) Requesting 64 nodes among 800



(b) Requesting 128 nodes among 800



(c) Requesting 64 nodes among 10,000



(d) Requesting 128 nodes among 10,000

Figure 4.1: Quality of the schedule found against the number of iterations

one. The system also presents a very good convergence speed. We can reasonably assume that it would behave in the same way when other policies are used. One of the key factors to achieve good results is the presence of additional information contained in the custom overlays. Some basic ones are already provided in order to address the more common types of queries, like the *As Close As Possible* and the *As Soon As Possible*. In real systems where the type of queries can be determined, like in cooperative projects, new custom overlays can be deployed in order to achieve close to optimal results.

Chapter 5

Conclusions

This thesis presented an approach to scalable distributed scheduling. It addressed the problem of finding N machines concurrently available in the future on which to schedule the execution of jobs. The main issues that were tackled are scalability and decentralization. When the number of resources is very large it becomes unfeasible to employ a centralized solution based on the global view of the system. Moreover the presence of many administrative domains makes it difficult to organize and manage the access and the fair use the system. We also went beyond the logic of meta-schedulers, that consider the Grid as the sum of its clusters, by considering every single computing node independently. We proposed a pure P2P system where any node can accept the submission of jobs and become the scheduler. An extension on previous models is represented by the possibility of specifying a policy at submission time. This encapsulates the criteria on which the available nodes are chosen in order to fit requirements of the applications. A prototype of this system has been built which allows the scheduling of jobs according to two criteria common to many applications: As Close As Possible and As Soon As Possible. Two of the main strength of this system are the use of gossip algorithms to maintain some local knowledge at every node and the use of a genetic algorithm to exploit this distributed information in order to calculate the schedule for a job. Another important peculiarity of the

system is its use of network coordinates to keep track of the position of nodes and to evaluate the latencies among them. The system has been evaluated emulating two network of different sizes: one of 800 and the other of 10.000 nodes. Experiments were conducted to evaluate the performance of the decentralized algorithm comparing it to its centralized version. The chosen policy to be evaluated in the experiments has been the As Close As Possible. We assumed the position of a node to be static and this allowed us to compare the run a centralized version of the genetic algorithm on the whole set of nodes. Unfortunately it was not possible to evaluate the As Soon As Possible policy due to some limitations in the implementation of the system, but we expect it to behave similarly.
Experiments showed that the decentralized algorithm scales well as the number of nodes in the system grows. In both scenarios it was able to find results very similar to its centralized counterpart. It has been also shown that then number of iterations necessary for the algorithm to converge is very small and not really affected by the size of the system. Further studies on the algorithm will be focused on the evaluation of its performance in regard to different policies and also to a combination of them.

Bibliography

[1] Webpage: *http://folding.stanford.edu*

[2] Webpage: *http://boinc.berkeley.edu*

[3] Webpage: *http://www.cs.wisc.edu/condor*

[4] Webpage: *http://www.st.ewi.tudelft.nl/koala*

[5] H.H. Mohamed, D.H.J. Epema
The Design and Implementation of the KOALA Co-Allocating Grid Scheduler
European Grid Conference, Amsterdam, 2005

[6] A.J.Chakravarti, G.Baumgartner, M.Lauria
Self-Organizing Scheduling on the Organic Grid
IEEE Transactions on Systems, Man, and Cybernetics vol. 35, 2005

[7] Webpage: *http://projects.gforge.cs.vu.nl/ibis/zorilla.html*

[8] N.Drost, R. van Nieuwpoort, H.Bal
Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing
GP2P: Sixth International Workshop on Global and Peer-2-Peer Computing, Singapore, 2006

[9] G.V.Iordache, M.S.Boboila, F.Pop, C.Stratan, V.Cristea
A Decentralized Strategy for Genetic Scheduling in Heterogeneous Environments
Grid Computing, High Performance and Distributed Applications (GADA) 2006 International Conference, 2006

[10] I.C.Legrand, H.B.Newman, R.Voicu, C.Cirstoiu, C.Grigoras, M.Toarta, C.Dobre
MonALISA: An Agent based, Dynamic Service System to Monitor, Control and Optimize Grid based Applications GMW '07: Proceedings of the 2007 workshop on Grid monitoring, Monterey, California, USA, 2004

[11] C.Oner
Decentralized Grid Scheduler
Master Thesis, Vrije Universiteit Amsterdam, 2007

[12] S.Voulgaris, D.Gavidia, M.van Steen
CYCLON: Inexpensive Membership Managment for Unstructured P2P Overlays
Journal of Network and Systems Management, 2005

[13] S.Voulgaris
Epidemic-Based Self-Organization in Peer-to-Peer Systems
PhD Thesis, Vrije Universiteit Amsterdam, 2006

[14] M.Szymaniak, D.Presotto, G.Pierre, M.van Steen
Practical Large-scale Latency Estimation
Computer Networks: The International Journal of Computer and Telecommunications Networking, 2008