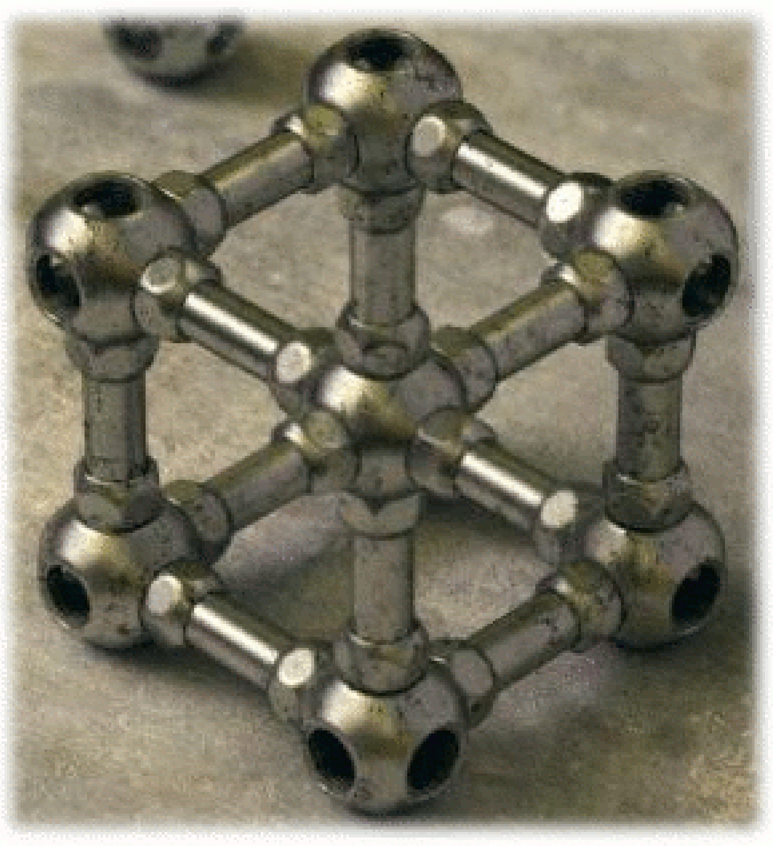

Resource use pattern analysis for predicting resource availability in opportunistic grids [1]

Authors:
Marcelo Finger
Germano C. Bezerra
Danilo M. R. Conde

[1]This work was supported by Fapesp Project LogProb, grant 2008/03995-5, São Paulo, Brazil.

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **00**:1–20 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Resource use pattern analysis for predicting resource availability in opportunistic grids

Marcelo Finger[1,†,*], Germano C. Bezerra[1] and Danilo R. Conde[1]

[1] *Dept. of Computer Science, University of São Paulo, Brazil*

.

SUMMARY

This work presents a method for predicting resource availability in opportunistic grids by means of Use Pattern Analysis (UPA), a technique based on non-supervised learning methods. This prediction method is based on the assumption of the existence of several classes of computational resource use patterns, which can be used to predict resource availability. Trace-driven simulations validate this basic assumptions, which also provide the parameter settings for the accurate learning of resource use patterns.

Experiments made with an implementation of the UPA method show the feasibility of its use in the scheduling of grid tasks with very little overhead. The experiments also demonstrate the method's superiority over other predictive and non-predictive methods.

An adaptative prediction method is suggested to deal with lack of training data at initialisation. Further adaptative behaviour is motivated by experiments which show that, in some special environments, reliable resource use patterns may not always be detected.

KEY WORDS: Use Pattern Analysis, Scheduling, Opportunistic Grids, Grid Computing

1. Introduction

Computationally intensive applications were initially confined to very expensive supercomputers, but can now run over distributed grid computing environments. In a dedicated grid, all

*Correspondence to: Dept. of Computer Science, University of São Paulo, Brazil. E-mail: mfinger@ime.usp.br
†Partly supported by CNPq/Brazil, grant PQ 304607/2007-0, and by Fapesp/Brazil, project 2008/03995-5.
Contract/grant sponsor: Brazilian Research Council (CNPq) project InteGrade; contract/grant number: 550895/2007-8

computing resources are devoted only to performing grid tasks. Opportunistic grids differ from dedicated grids in that the machines may not be always available to run grid tasks. This form of grid has the advantage of being put together using already existing computational resources, permitting a very low cost solution to perform computationally intensive applications.

In *opportunistic grid computing*, grid applications use the idle time of desktop machines to perform high-performance computation, which may last for hours. If a grid application is running when the machine is claimed back by its owner, the grid job is either paused, migrated or simply aborted.

It is a great priority in opportunistic grid computing to ensure that computer resource owners will allow those resources, when idle, to be used for grid computing. For that, the Quality of Service for private computations must not be perceptibly affected by grid applications.

Research groups working on grid systems, such as Globus [7], Condor [15], BOINC [1], OurGrid [4], and our own work on InteGrade [10] have investigated opportunistic grid computing to perform high-performance computation. However, the support for effectively using these shared resources without compromising the Quality of Service perceived by the resource owners is still very limited.

Ideally, one would like to schedule grid applications on machines that are fully available for the duration of the application. To address the problem of effective opportunistic computation, it is very useful to be able to predict when a given computational resource will be idle, becoming available for grid applications. However, predicting may be expensive or ineffective, requiring at least in theory the collection of large amounts of data that may need to cross the system, potentially placing a big load on the system and compromising the very Quality of Service that it is trying to maintain. In this respect, some opportunistic grid environments have chosen not to perform any kind of prediction, preferring instead to concentrate on checkpointing and restarting tasks; see discussion on Condor in Section 5.2. Note that this solution does not rule out the feasibility of predictions.

If the prediction of resource availability can be done with some degree of accuracy and at low cost, the grid scheduler may be more effective in its task of assigning jobs to machines.

This work starts from the observation that many computing resources have a clear pattern of use and thus, also a clear pattern of availability. The *Use Pattern Analysis* (UPA) consists of the task of detecting the local use pattern of each resource in each machine.

We develop a method that performs resource use pattern analysis for machines belonging to opportunistic grids. This method is based on unsupervised machine learning [17, 2], which performs a *clustering analysis* on past records of resource use to discover prototypical patterns of use [11]. As usual in machine learning settings, the learning activity is performed off-line. The use patterns learned are used at run-time to predict the availability of computational resources. Ongoing work aims at employing this run-time prediction as part of grid task scheduling.

The development of the method was done in two steps: simulation and implementation. A simulation phase was needed due to the existence of a large number of parameters to be set in clustering-based unsupervised machine learning methods, and was performed using real data collected from machines belonging to the InteGrade opportunistic grid [10]. Our group developed the *Local Use Pattern Analyser* (LUPA) module for the InteGrade grid, and validated the results of the simulation. The implementation was used to compare the proposed method against other methods for availability prediction.

The rest of the paper develops as follows. The UPA method is described in Section 2. Its basic assumption is validated in the simulation of Section 3 which also determines several parameters that improve the accuracy of the method. Implementation experiments are described in Section 4. Related work is analysed and compared in Section 5. Conclusions and future work are presented in Section 6.

2. The UPA Method

The resource Use Pattern Analysis (UPA) method is based on the assumption that there is a small set of prototypical daily behaviours that models resource availability at each machine. The choice of *daily* patterns may seem arbitrary, and was based on the fact that the method was designed to help scheduling tasks that last for a few hours. Tasks that last a few seconds may run on almost any idle machine, and those that may span over weeks or even months are expected to be executed on non-dedicated machines without being paused or migrated. However, the method described here may be useful in reducing the number of execution interruptions for these very long tasks.

In the case of hour long tasks, the existence of prototypical patterns of behaviour is already a somewhat bold assumption, so prior to implementation a simulation phase was needed to both validate the method and study some of its properties. As the UPA method obtains the prototypical behaviour via unsupervised learning of resource use histories, the validation step was also providential in fine-tuning the method.

Use Pattern Analysis deals with computational resource use objects. Each object is a vector of values representing the time series of a machine resource use, as illustrated in Figure 1. In fact, as we are interested in resource *availability*, the higher the use of a resource, the lower the expectation it will be made available for grid tasks. Resource down-time is represented by a truncated time series, a fact that may affect the computation of prototypical resource patterns; its treatment will be discussed when we describe the training phase.

Machine resource use is sampled at a fixed rate (currently, once every 5 minutes) and grouped in objects covering 48 hours; the measurement of CPU use is a 5-minute average. The sampling rate was chosen to both allow the prediction of hour-long availability and not to overburden the system with too much data. An object starts at midnight, so there is a 24-hour overlap between consecutive objects. The span of 48 hours, instead of 24 hours, is needed for the prediction phase.

The method is capable of monitoring several computational resources, such as CPU use, available RAM, disk space, swap space, network and disk I/O; in this work, only the first two will be discussed, and these are the most relevant for machine allocation decisions. Use Pattern Analysis performs unsupervised machine learning [17, 2] via a data clustering process [11] to obtain a fixed number of *use classes*, where each class is represented by its *prototypical* object. The idea is that each class represents a frequent use pattern, such as a busy work day, a light work day or a holiday.

The method involves two phases: a training/learning phase and an execution/prediction phase.

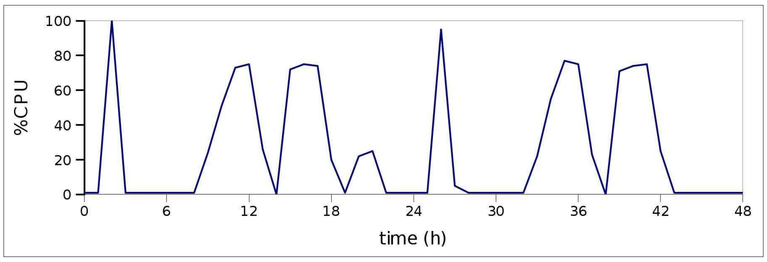

Figure 1. An object representing a machine's CPU use over a 48h period

2.1. Learning

Learning is an off-line activity that inputs a large amount of objects collected during the machine regular operation. A clustering algorithm is applied to the training data [11, 6], which groups the training objects into a fixed number k of clusters. At each cluster, a prototypical element is computed that represents the whole use class. The output of the method are k prototypical vectors to be used by the runtime predictor.

It is common to have incomplete data for training, due mainly to machine down-time. We considered three possible treatments of resource down-time:

- Consider it as 100% use; this makes sense, as we really want to predict resource availability, and down-time implies no-availability, as does 100% use.
- Perform a missing data completion; one way to deal with this problem in the literature is to apply an Expectation Maximisation completion process [5].
- Discard objects containing down-time; those object will simply not be used in the computation of the prototypical objects.

The current approach opted for the simplest solution, and discarded objects with incomplete data. Provided there is sufficient data, this solution does not affect the result. This simplification was made due to the fact that all resources monitored were supposed to be available at all times. If there is a resource that is turned on and off every day, such as most home computers, then it would be reasonable to consider down-time as no-availability, and include that fact in the time-series representation.

The filtering of incomplete data for the learning phase is only a reliable method when there is a considerable mass of data, which in this case consisted of at least 60 objects, or two months of data. At the implementation phase, other methods were tested that need smaller amount of testing data, and their results were compared against the UPA method.

Data clustering analysis can be parameterised in several ways. For the UPA development, the following parameters were considered:

- *Number of clusters.* We considered a fixed number of clusters, either 5 or 10. The idea was to keep the number of prototypical objects small enough to allow for fast prediction.

A small number of clusters also demands less training data, otherwise one may get some clusters with very small number of objects, rendering it a highly unreliable pattern.

- *Data normalisation.* Two possible data normalisation schemas were considered for vector: *no normalisation*; and *variational* normalisation, in which the average value the sequence is computed and subtracted from each element in the sequence, resulting in a sequence with average 0. The difference between these two methods is illustrated in Figure 2. The variational normalisation treats as equal the use at full capacity and full availability, transforming both as a constant 0 sequence, while the non-normalised one does not. As a result, it can group more significant clusters; for example, it groups use at full capacity and full availability without losing the capacity to distinguish between them at runtime. When using the variational method at runtime, the series is reconstructed by adding the average of the recent past to every predicted time point. In this way, use at full capacity is restored at runtime to average 100%.
- *Computation of prototypical element.* Typically this is a *centroid*, that is, the average of all cluster elements, or the *centre*, that is, an actual element of the cluster that is closer to the centroid. The centroid method is to be avoided in non-metric spaces, where a prototypical element cannot be generated and has to be found among the given points. As this was not the case here, only the centroid method was considered.
- *Clustering algorithms.* There are many algorithms for clustering, namely hierarchical, sequential, k-means, etc. No substantial difference in prediction power was noted among different algorithms. To concentrate on distance measurements, only hierarchical algorithms are considered, which constructs clusters on a bottom-up fashion, each step uniting the two closer (i.e., more similar, less distant) clusters. As this methods computes the distance between clusters, several methods for computing such distances have to be considered.
- *Similarity measurement.* There are several ways to compute the distance between two clusters (similarity is the inverse of distance), assuming that the distance between any pair of points a and b is given by the Euclidean distance: $d_{ab} = \sqrt{\sum(a_i - b_i)^2}$. The following distances were considered:

  - *single linkage*: the distance between two clusters is the *smallest* distance between any two points in these clusters;
  - *complete linkage*: the distance between two clusters is the *largest* distance between any two points in these clusters;
  - *centroid method*: the distance between two clusters is the distance between their centroids.
  - *Ward's method*: the distance between two clusters is the variance of the union of the clusters. That is, one takes the two clusters as a single one, finds its new centroid, computes the distance between every point in the union to the new centroid, and takes the variance of this set of pointwise distances as the distance between the two clusters.

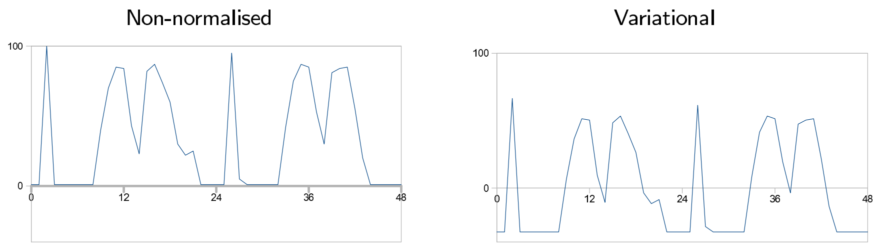

Figure 2. Non-normalised *versus* Variational time series

2.2. Runtime Prediction

During runtime, a request is sent to each machine predictor specifying the amount of resources (CPU, disk space, RAM, etc) and the expected duration needed by an application to be executed at that machine. This expected duration may be provided by users, which may not be a reliable estimate [14]; research in progress within the InteGrade grid manager framework investigates the automated learning of the duration and the computation resource needed by applications, and existing models for task duration prediction will also be considered [19]. The UPA predictor has to decide if the resources will be available for the expected duration provided as input.

This decision is reached according to the following method. A record is kept about the recent use of each resource; usually the last 24 hours, as illustrated in Figure 3. The predictor computes the *distance* between the vector representing recent history and each of the prototypical element of the use classes learned during the training phase. The recent use object has span of 24 hours, but the prototypical elements have span of 48h. To compute the distance, the recent record of resource use is compared with the corresponding times in the use classes; so if a request for a resource is made at 6 pm, the last 24 hours of that resource use is compared to the interval 18–42 in each prototypical element. The class with the smallest distance is the *current use class*.

The interval between the current time and the end of the 48h in the current use class, after some possible processing, is used as the prediction of the near future; so if a request is made at 6pm, this method can predict the next 6 hours.

When variational normalisation is used, the following extra steps have to be observed:

- The average a of resource use for the last 24-hour series is computed.
- This value is subtracted from each element of the series, generating a new series.
- The prototypical element that is closest to this new series is chosen.
- The average a is added to each element of the chosen prototypical element, so that the prediction is made with respect to this assembled series.

If a prediction for a longer period is needed, there are two possibilities:

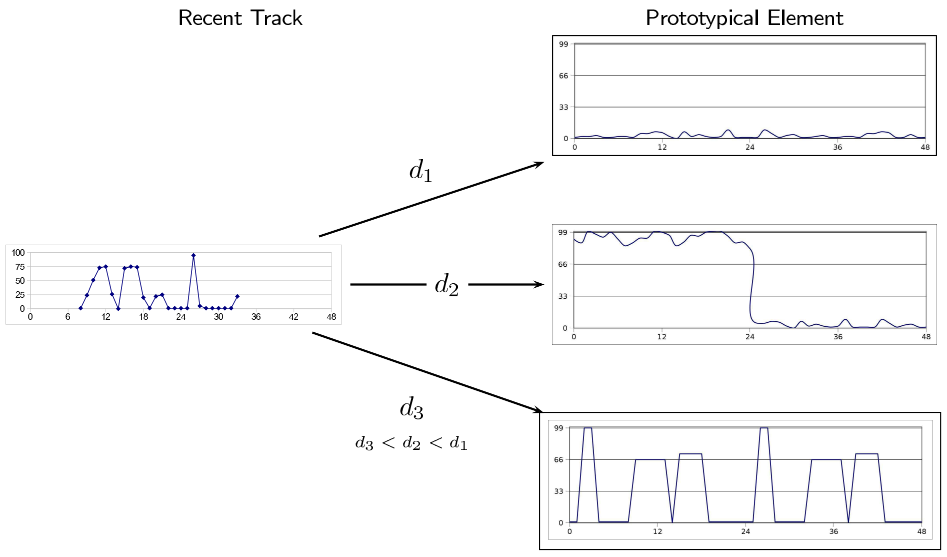

Figure 3. Prediction of class pertinence

(a) To use less that 24 hours of recent record to compare with the start of each prototypical element, which allows for predictions over 24 hours. The reliability of predictions decreases with the span of the recent record used.

(b) To chain predictions, by using the last 24 hours of record or prediction as a basis for the next prediction. This allows for unbounded predictions, but the longer the chain, the less reliable the prediction. This is not implemented yet and remains for future work.

3. Simulation

There are many parameters involved in data clustering analysis. Simulation was used to choose those parameters. In the end, with a set of chosen parameters, the basic assumption of the UPA method could be evaluated, namely that prototypical daily behaviours are good models of resource availability and are worthy of being implemented.

The data for simulation was collected, for CPU and RAM use, during a period of 120 days from four Linux machines with very different types of users:

(a) a general purpose machine with more than 30 users;

(b) a single user machine;

(c) a general purpose machine with 6 users;

(d) a multi-user machine employed for testing heavy computational linguistics programs.

For each machine, the learning process generated both a 5-cluster and a 10-cluster output, called according to the presentation above a5, a10, b5, b10, c5, c10, d5, d10; furthermore, clustering was performed both for pure and for normalised data.

A single test is defined by a resource r, an instant in time t, an availability level α and an interval η; the test output is *yes* if it is predicted that the availability level of r will remain above α for duration η starting at t, and *no* otherwise. The output is *correct* if the decision coincides with that obtained from simulation data.

An *availability matrix* $A = \{a_{ij}\}_{N \times M}$ was constructed for each machine and each resource, with availability α in the interval 10%–90% of total capacity with 10% steps, and for intervals η of $10, 20, 30, 60, 90, 120, 240$ minutes. Each a_{ij} contains the percentage of correct predictions in 100 tests for a given pair $\langle \alpha, \eta \rangle$, with random starting point t. For each matrix A, the average prediction success was computed as $\mu_A = \frac{1}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} a_{ij}$.

One simulation experiment consists of the construction of an availability matrix A containing 63 cells, each cell containing the number of correct prediction in 100 tests, generating a single value for the average prediction success, μ_A. The simulation experiments were made in batches of 8 experiments, one for each machine with 5 and 10 clusters. For each resource, 4 batches of experiments were made, each using a different distance measurement. The 64 experiments were repeated for normalised and non-normalised clusters. Overall, 128 experiments are reported.

3.1. Simulation Results

We present now the results that allowed us to validate the method and fix the learning parameters discussed in Section 2.1.

3.1.1. Validation

Figure 4 shows the results of all experiments, separating the results of predictions based on pure and normalised (variational) data. The values of μ_A were surprisingly high. All values for average prediction success above 75%, and mostly above 80%.

The quality of the results allows for the validation of the basic assumption of the UPA method, namely that computing learning the prototypical elements and computing the distance from the recent past to the closest prototypical element is a good form of predicting future availability of resources.

3.1.2. Pure $\times$ Normalised Data

Besides validating the UPA assumption, Figure 4 compares the prediction results obtained from clustering process with pure data and from normalised clustering with 0-mean, variational data. Predictions with normalised data yield a better result in all experiments, with average prediction success were above 90% in all cases.

For this reason, the following results are presented only for normalised variational clustering.

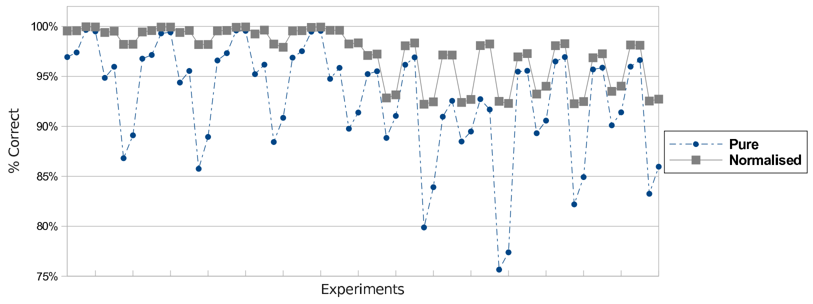

Figure 4. Success rates of all experiments, pure × normalised data

3.1.3. Distance Measurements

Experiments were made for hierarchical clustering using four different types of distance measurements, as described in Section 2.1. Figure 5 presents the results for RAM availability prediction and Figure 6 presents the results for CPU availability prediction.

No distance measurement dominates all the others all the time. However, it seems that computing distances by Ward's method has a slightly superior overall performance. Similarly, no measurement is clearly the worst. As the number of machines analysed is still small, no categorical preference can be established. The results in Figures 5 and 6 may even suggest that each domain of machines may require an initial analysis to determine the best measurement for each case.

It is worth noting that predicting RAM availability is easier than predicting CPU use. It calls the attention that machine (b) yields the best results for RAM prediction and the worst ones for CPU prediction. This reinforces the need for an initial *per machine* analysis prior to the choice of measurement to be implemented.

In case no method is demonstrably superior to others, we propose a different criterion to select the distance measure, namely the *least complex* method with respect to the number of elements in the clusters; the justification for such a criterion is that the chosen method will decrease system overheads both in learning and in runtime phases. If we compute the distance between two clusters with n_1 and n_2 elements, the lowest complexity method is the centroid method, which computes the distance in constant time. For Ward's method, the distance computation is $O(n_1 + n_2)$; for single and complete linkage, the distance computation is $O(n_1 * n_2)$. Thus, the complexity criterion selects the centroid method.

3.1.4. Number of Clusters

Figures 5 and 6 show a consistent prevalence of 10-cluster predictions over 5-cluster prediction for all distance measurements. However, in all cases, this prevalence is in fact very small. Furthermore, by investigating the clusters learned with 10-cluster outputs, one verifies that a considerable number of clusters has a very small number of elements, normally less than 5. On

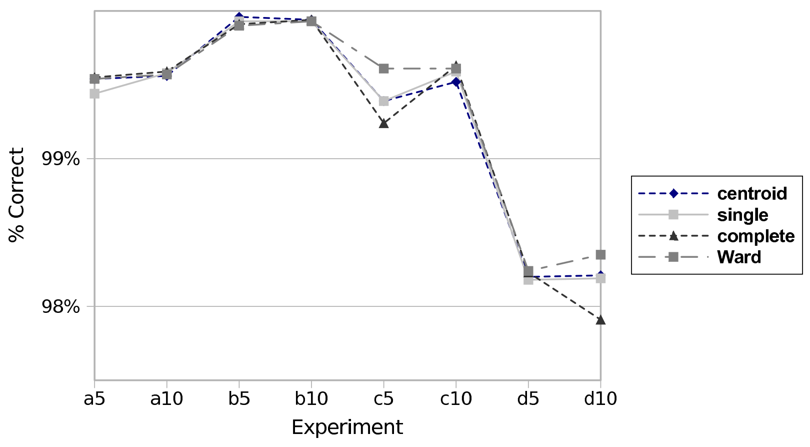

Figure 5. RAM Availability Prediction

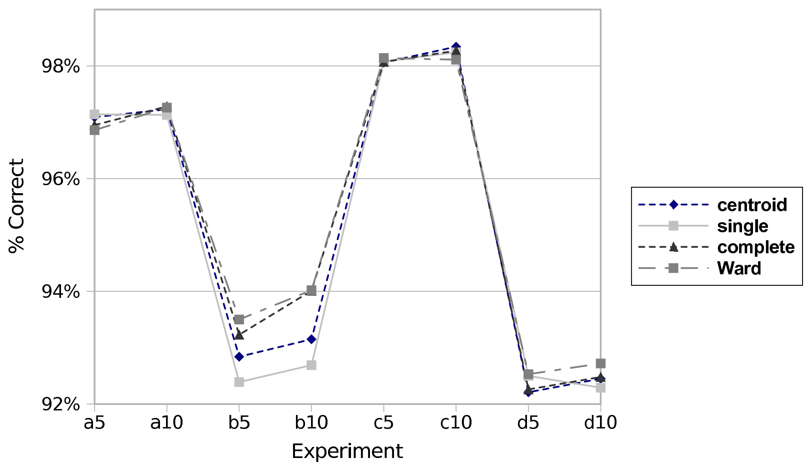

Figure 6. CPU Availability Prediction

the other hand, with 5-cluster outputs the representativity of the clusters learned is always higher.

So no categorical conclusion can be reached as to the fact on the number of clusters, other than that no big difference is achieved by choosing a larger number of clusters.

3.2. Simulation Conclusions

Simulation definitely validated the UPA assumption. Data normalisation is to be preferred, and the number of clusters can be kept as low as 5, without seriously affecting prediction performance. Distance measurements have to be analysed on a case-by-case basis prior to implementation, but no method among the analysed ones is either the best or the worst; in this case, the least complex method was proposed as a prevailing decision criterion, which selected the centroid method.

4. Implementation

The development of an implementation of a resource Use Pattern Analyser had three main goals:

- To explore the scheduling of grid applications using predictions made by the UPA method.
- To compare the UPA scheduling with other scheduling methods.
- As the UPA method needs a few months of resource use data collection to be reliable, identify initialisation strategies to be used during the first period of activation.

Note that the results described here concern only the prediction capabilities of the implemented system modules with regard to resource availability, not their actual use for scheduling.

4.1. Design Decisions

Resource use data information has to be collected locally at each machine taking part in an opportunistic grid. This data can be analysed locally or can be sent to a centralised server for analysis. The latter possibility may make the task of machine allocation easier, but it can be seen as a breach on the privacy by users of single-user machines, which may not allow those machines to take part on the grid for fear that their pattern of work is being monitored. On the other hand, local analysis hides the computed resource use patterns from the rest of the system. Furthermore, the local analysis of data also has the advantage of preventing the flow of use data on the network, decreasing the overhead placed on the system.

So the first design decision selected a local analysis of resource use patterns. As a consequence, a Local resource Use Pattern Analyser (LUPA) module is installed in all machines that allow their resource to be opportunistically used by grid applications. The LUPA module was implemented in C++ on several Linux platforms, and is now an integral part of the InteGrade distribution [†].

The LUPA architecture is shown if Figure 7, and consists of three subsystems:

- **Data Collection**. Performs resource use sampling, recording CPU and RAM use every 5 minutes. This is the same rate used for simulations.
  Data is recorded with no normalisation or other form of preprocessing, so that it can later be used by different clustering methods.
- **Pattern Analyser**. Performs off-line clustering of resource use objects, as described in Section 2, generating a fixed number k of prototypical use classes; for the results presented below, $k = 5$. For time-efficiency reasons, centroid distance is used.
  In the current implementation, patterns are recomputed once a day but that, as will be shown, does not place a heavy burden on the system.
- **Predictor**. Performs runtime predictions based on the recent resource use history. The interface to access the predictor contains the following elements:

  `double[] getPrediction(resource r, int hours)` returns a vector of values representing the `r`-use prediction for the next `hours`, in 5-minute intervals.

  `boolean canRunGridApplication(freeCpu, freeRam, hours)` returns a yes/no answer if the prediction allows for `freeCpu` and `freeRam` for the next `hours`.

  `int howLongCanRunGridApplication(freeCpu, freeRam)` returns the number of hours for which there is a prediction of availability of `freeCpu` and `freeRam`. To avoid running forever in case of very low input data, the method is at the moment limited at 72 hours.

It is important to note that at this stage of development, the LUPA module is not yet integrated with the grid scheduler, for there is no implemented automated mechanism to evaluate an application's resource requirements yet.

In the experiments for evaluating the implementation, scheduling is restricted to selecting the machine with higher CPU availability.

4.2. Experiments

The experiments simulate the scheduling of a grid application. Experiments were performed using 15 machines, with data collection logs varying from 41 to 120 days, none of which were used for the simulation phase, so as to avoid biased results.

Three different scheduling methods were compared:

- **RR: round robin.** Machines are randomly placed on a circular list; when n machines are requested, the initial n machines are returned and the list starting pointer is advanced n positions. No prediction is made.

[†]Freely available at `http://www.integrade.org.br`.

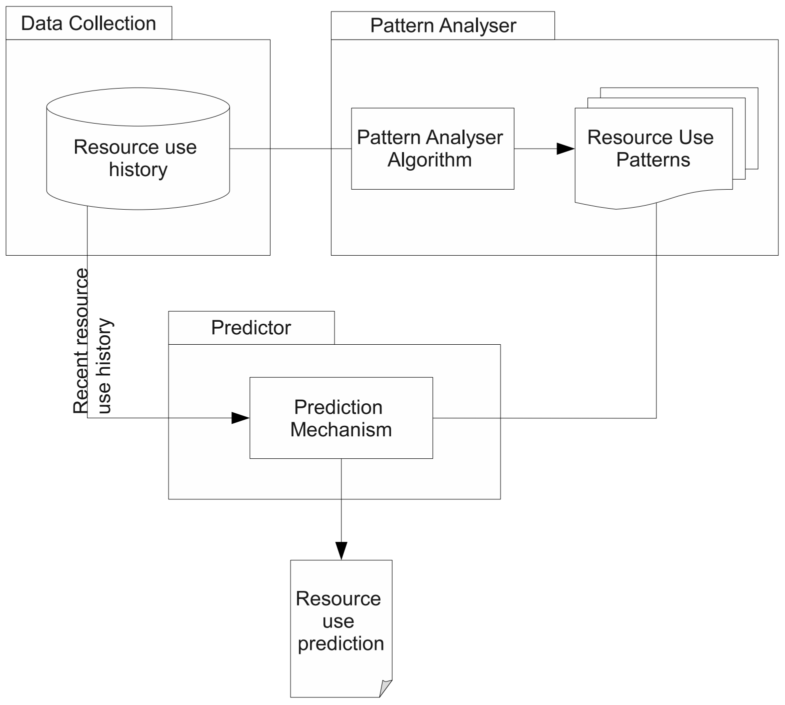

Figure 7. The LUPA Architecture

- **last4.** Resource prediction for any amount of time is considered the average of the last four hours. The method chooses n machines with highest prediction of availability. Preliminary tests were also made for the last 8, 12 and 24 hours, and their performance were considered basically equivalent to the **last4** method.
- **UPA.** Prediction using the UPA method. The method chooses n machines with smaller prediction of CPU use for the next h hours.

An experiment consists of a sequence of tests for a fixed set of test parameters. Initially, the valid days in the data collection logs are selected, in which there are at least m valid machines, where m is a test parameter.

For each selected day, pattern analysis is run and then 24 tests are executed, each for an hour of the day and for each of the three scheduling algorithms above. These are the *instant tests*. Therefore, the number of instant tests in an experiment is initially a multiple of 24, but some instant tests may be discarded if minimum availability is not met, as described below.

The performance metric is the average free CPU percentage in the chosen machines in the interval $[t, t + h]$, given by

$$performance_s(t) = 1 - \frac{\sum_{i=1}^{n} use(m_i, t, h)}{n}$$

where s is the scheduling algorithm, n is the number of chosen machines, m_i is one of the machines chosen to run an application process for h hours starting at time t, and $use(m_i, t, h)$ is the average CPU use at machine m_i for that period according to the log.

Experiments can constrain the set of chosen machines by requiring a minimum CPU availability a at chosen machines; if this requirement is not met, the instant test is not computed. As a consequence, an experiment can have any number of instant tests.

An experiment set of parameters is therefore given by:

- n: number of machines to be chosen to run the application, $1 \leq n \leq 6$;
- h: application duration, in hours, $h \in \{2, 4, 6, 12, 24\}$;
- m: minimum number of machines available to be chosen at a given day, $m \geq n$ and $1 \leq m \leq 15$;
- a: minimum CPU availability at chosen machines, $a \in \{0.6, 0.7, 0.8, 0.9, 0.98\}$.

An experiment is performed for a set of parameters only if the logs allowed for at least 100 instant tests for that set. A total of 45 experiments were made, averaging 527 tests each. The output of an experiment is the performance of each of the three scheduling methods for each test. The performance of scheduling methods s and r were compared on a test by test basis. Methods s_1 and s_2 were considered *equivalent* for a test if the performance difference was less than 2%, that is, $|performance_{s_1}(t) - performance_{s_2}(t)| < 0.02$. Method s_1 performs better than method s_2 if $performance_{s_1}(t) - performance_{s_2}(t) > 0.02$.

Results.

The performance of **RR** was consistently below that of both **last4** and **UPA** scheduling methods, beating **last4** on less than 1% of the tests and **UPA** even less.

Figure 8 displays a coarse-grained comparison between the **UPA** and **last4** scheduling methods showing the total percentage of instant tests that each method outperformed the other, and their equivalence.

It calls the attention that for an average of 77.6% of all tests, the two methods are equivalent. When equivalent results are discarded, the **UPA** performs better than **last4** in 75.6% of the experiments.

Due to the surprise on the results, we performed a detailed analysis of the cases when the **UPA** was not the best of the two, namely, when the two methods were basically equivalent or when **last4** had better results. We arrived at the following conclusions:

- When **UPA** was not the best, the machines have been idle for at least an hour, and remain in an idle state for quite sometime. In fact, this is consistent with the intuition that when machines are idle, no effort is needed for predicting the near future.
  It turns out that the way the tests were constructed permitted a high number of instant tests to occur when the system was idle. By selecting valid days and then performing 24 instant tests, one at each hour, the idle hours of the day have always received a lot of attention. Furthermore, several instant tests were discarded when minimal performance was not met, but this only occurs at busy times, which increase the proportion of instant tests performed at idle times.

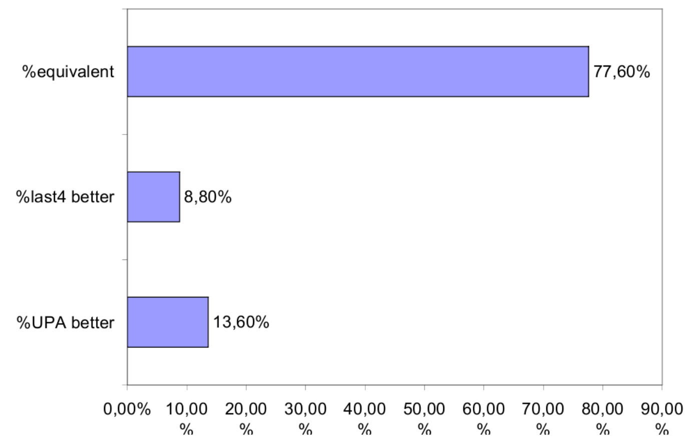

Figure 8. Coarse comparison between UPA *versus* last4

- When **last4** was the best, it chose machines that, besides being idle, were associated to very irregular prototypical behaviours. This amounts to say that these machined did not have clear patterns of use. In general, those machines were at the disposal of a large number of users of a community under low demand. So pattern analysis does not perform very well when no pattern exists.

Those observations were followed by new experiments in which a set of 15 machines belonging to a student lab were selected, and tests were performed only during the period of 10pm–6am. The same jobs were submitted to the grid with the UPA method and without it (namely, just by supposing that the current CPU use will persist into the future). Classes of jobs with different duration were submitted to a scheduler that implements the **UPA** method via `canRunGridApplication` interface.

When jobs taking between 30 minutes and one hour were submitted, predictions without UPA achieved a better performance in 1.3% of the cases. This small disadvantage of the UPA method was amplified with longer jobs. When jobs taking between one and two hours were submitted, the no UPA method achieved a better performance in 12.5% of the cases.

Although detecting irregular patterns may not be very easy to automate, detecting idleness (that is, detecting availability above some high threshold, say 99%) is quite simple, which suggests the following adaptative behaviour for the predictor:

	UPA	**last4**
Performance	Similar to **last4**, but better	Similar to **UPA**
Impact of $\uparrow m/n$	$\uparrow$ performance	no effect
Impact of $\uparrow d$	$\uparrow$ performance w.r.t **last4**	$\downarrow$ performance w.r.t. **UPA**
Impact of $\uparrow h$	no effect	no effect

Table I. Prediction Methods Comparison Summary

```
if    the machine has been idle for two hours
and   UPA predicts idle predominance for the expected duration of the job
then  prefer the last4 method
else  use the UPA method.
```

Table I summarises the effect of the test parameters on the performance of the methods. Scheduling using **UPA** has its relative performance increased when the ratio m/n (available/required machines) increases, as well as when d (required free CPU) increases. Overall, the **UPA** method displays better results. It was also clear that the method performs well with less than 60-day training data.

We conclude that **last4** is quite a good method, and a candidate to be run during LUPA initialisation. Final LUPA implementation has the following adaptative behaviour:

- Use **UPA** method if more than 21 days of data collection is available; else
- Use **last4** method if more than 4 hours of data collection is available; else
- Predict that resource use at request time persists.

With regard to system overhead, the running time for pattern analysis was always below 1s, and the running time for prediction calls was always below 3ms. Measurements were made on a notebook with an AMD Turion 64 1.8GHz CPU, 1GB RAM running Kubuntu 7.10 (32 bits) Linux.

The conclusion is that the load placed by the **last4** method is quite minimal, in favour of its use by a grid scheduler.

5. Related work

Several lines of research in grid computing may be considered as related to the present work, all of which have in common the investigation of methods that may be employed to improve grid application performance; mainly, those techniques aim at reducing application execution time [16, 21].

Each of main grid research project — such as BOINC, Condor and Globus — has a way to deal with the problem of predicting resource availability on the grid, even if this means a strategy to totally avoid the problem. In the following we review the work developed in several grid projects.

Also several non-grid related work on resource prediction is found in the literature, and a few of them are reviewed here.

5.1. Correlated availability

A recent work developed within the framework of the BOINC system [1] is perhaps the work in the literature that most align to ours. That work takes place in the scope of *volunteer computing*, which is an extension of the paradigm initiated by the SETI@home project [22]. One of the possible extensions intends to run communicating applications in parallel, so it becomes very important to be able to predict machines that are simultaneously available on the internet.

Derrick *et al* [13] thus investigates large portions of the internet in the search of *correlated availability in internet-distributed systems*. It computes patterns of simultaneous availability of individual machines which, as in our case, is obtained by a process of data clustering.

To achieve its goal, it reportedly monitors 112,268 hosts over the whole internet, thus employing a global monitoring service that generates a large amount of data that is submitted to the process of pattern discovery. Even if the pattern discovery is divided over time zones, the amount of data to be analysed is still very large.

The system was not originally intended to be used as a tool for scheduling applications, but was used to discover interesting clusters of machines and many facts about classes of users, such as home and office users. Those finds have the potential of enabling several applications.

This method contrasts with the UPA method by taking a *global* approach to resource monitoring, and only idle time is detected. On the other hand, the UPA method performs only *local* monitoring, with considerably smaller, distributed overhead.

5.2. Preemptive resume scheduling

The Condor grid system [24, 15] also deals with opportunistic and non-opportunistic (i.e. dedicated) grids. In its scheduling mechanism it employs a technique called *preemptive resume scheduling* [20], which combines both dedicated and opportunistic scheduling. For opportunistic grids, in which grid jobs may be interrupted when the owner of a computational resource claims it back from the grid, it applies a checkpoint technique.

Condor will checkpoint and preempt jobs when an owner needs it computing resource. When another computer is available to run the job, Condor will resume the job on that computer. In most cases, this does not require the modification of jobs, only their relinking with libraries provided by Condor, which makes the whole process very attractive.

However, the basic point of this method is that it performs *no prediction*, which distinguished it from our approach. One the one hand, this releases the scheduler from the task of predicting job duration; on the other hand, this forces the scheduler to take decisions based only on the current state of resources, which could be improved in the presence of previsions.

5.3. Conservative scheduling

The development of the Globus project [7] has lead to a predictive form of scheduling called *conservative scheduling* [26]. It is a statistical predictive method that aggregates several levels of predictions to obtain a machine's CPU load prediction.

On the lowest level, it applies *one-step-ahead* CPU load prediction [25], a tendency-based time series predictor based on history CPU load information which predicts the next value of CPU load according to the tendency of the time series change.

This initial prediction is aggregated, generating an interval load prediction, averaging over aggregated CPU load time series. This aggregation is performed because the intention is not to predict one step ahead, but to be able to predict the CPU load over the time interval during which an application will run.

Besides the average of an interval load prediction, the method also computes the variation of CPU load, computing the standard deviation of the original CPU load time series.

The conservative scheduling method combines both interval and variance load prediction. It is similar to our case in that it employs histories of CPU load time series, but the sort of treatment to which these data is submitted is a combination of statistical methods, which differs from our cluster-based pattern analysis method.

Unfortunately, we have not performed any experiments that compare the conservative scheduling method and the UPA method for predicting CPU availability, so at this point in time we do not have any knowledge on their relative performance. This remains a topic for future work.

5.4. Parallel computing environments

Outside the world of grid computing, parallel computing has also dealt with the problem of allocating tasks to processes and threads.

Work on parallelisation of commonly used algorithms has led to the concept of *work stealing*, that is, whenever a processor runs out of work, it steals work from a randomly chosen processor. A well known implementation of work stealing is that of the Cilk-5 multi-threaded language [8]; other forms of implementation are [9, 3].

Stealing work is a similar process to computing in opportunistic grids, but while in an opportunistic grid we have an application that searches for available resources, in the work stealing framework it is the idle resource that searches for work.

A recent implementation of work steal [23] employs a data structure that resembles a learning process, namely a list of "successful steals". However, it does not perform a traditional learning task, for there is no distinction between a learning phase and a runtime phase. On the contrary, that list is used as heuristics for the choice of work stealing to be performed. There is both theoretical and experimental evidence pointing that such heuristics outperforms single-threaded executions of the same algorithms [23].

On a different direction, parallel and distributed computer systems are using performance evaluation to manage workload. The work of [12] develops a two model strategy to evaluate performance. On the one hand, there is a model of the application. On the other hand, there is a model of resource use, namely CPU profiling and cache models. For the latter, a toolset for performance prediction is employed [18]. The authors of [12] claim that the difficulty in deriving a priori performance data makes the use of data obtained through monitoring a preferred approach. This kind of approach may be very useful for grid computing, specially in what concerns the prediction of application resource needs, and deserves further investigation in the future.

6. Conclusions and Further Work

The experiments have shown that some form of prediction always perform better than no prediction, and the UPA-scheduling method was favourably compared with respect to other methods, with small overhead. This confirms that the method can be used in practical scheduling of grid application tasks. It also became clear that if a resource is idle for a long period, it is better to adaptatively turn the UPA method off for the expected duration of the idleness.

Future work includes integrating the LUPA module with a task scheduler, so that experiments can be made using real grid applications and several scheduling variants applying Use Pattern Analysis can be tested. Furthermore, the capacity for longer term predictions remains to be explored; there are several scheduling possibilities this predictive capacity allows for, which deserves a detailed analysis, such as preemptive task migration and the automated "booking" of machine resources for future executions. Also, the existence of long histories for training has to be studied, to determine the ideal weight to be given to recent and distant past information.

REFERENCES

1. David P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID '04: 5th IEEE/ACM Int. Workshop on Grid Computing*, pages 4–10, 2004.
2. Horace B. Barlow. Unsupervised learning. In G. Hinton and T. J. Sejnowski, editors, *Unsupervised Learning: Foundations of Neural Computation*, pages 1–17. MIT Press, 1999.
3. Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.
4. Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, September 2006.
5. Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.
6. Brian Everitt, Sabine Landau, and Morven Leese. *Cluster Analysis*. Hodder Arnold, 4th edition edition, 2001.
7. Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997.

8. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *In Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, pages 212–223, 1998.
9. T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23, 2007.
10. Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.
11. Anil K. Jain, M. Narasimha Murty, and Patrick J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.
12. Stephen A. Jarvis, Daniel P. Spooner, Helene N. Lim Choi Keung, Junwei Cao, Subhash Saini, and Graham R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Comp. Syst.*, 22(7):745–754, 2006.
13. Derrick Kondo, Artur Andrzejak, and David P. Anderson. On correlated availability in internet-distributed systems. In *9th IEEE/ACM International Conference on Grid Computing (Grid 2008)*, 2008.
14. Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In *JSSPP*, volume 3277 of *Lecture Notes in Computer Science*, pages 253–263, 2004.
15. Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A hunter of idle workstations. In *ICDCS '88: Proceedings of the 8th Int. Conference of Distributed Computing Systems*, pages 104–111, June 1988.
16. Nguyen The Loc, Said Elnaffar, Takuya Katayama, and Ho Tu Bao. Grid scheduling using 2-phase prediction (2pp) of cpu power. *Innovations in Information Technology*, 150, issue 2:1–5, 2006.
17. Tom M. Mitchell. *Machine Learning.* Computer Science Series. McGraw-Hill, 1997.
18. G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace–a toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comput. Appl.*, 14(3):228–251, 2000.
19. Vahe Poladian, Jo ao Sousa, Frank Padberg, and Mary Shaw. Anticipatory configuration of resource-aware applications. *SIGSOFT Softw. Eng. Notes*, 30(4):1–4, 2005.
20. Alain Roy and Miron Livny. Condor and preemptive resume scheduling. In Jarek Nabrzyski, Jennifer M. Schopf, and Jan Weglarz, editors, *Grid resource management: state of the art and future trends*, pages 135–144. Kluwer Academic Publishers, Norwell, MA, USA, 2004.
21. Mark Silberstein, Dan Geiger, and Assaf Schuster. Scheduling mixed workloads in multi-grids: the grid execution hierarchy. In *Proceedings of the 15th International Symposium on High Performance Distributed Computing (HPDC06)*, pages 291–302, 2006.
22. Seti@home website. `http://setiathome.ssl.berkeley.edu` [27 February 2009].
23. Daouda Traore, Jean-Louis Roch, Nicolas Maiilard, Thierry Gautier, and Julien Bernard. Deque-free work-optimal parallel STL algorithms. In Springer-Verlag, editor, *EUROPAR 2008*, Las Palmas, Spain, August 2008.
24. Derek Wright. Cheap cycles from the desktop to the dedicated cluster: Combining opportunistic and dedicated scheduling with Condor. In *Proceedings of Linux Clusters: The HPC Revolution*, 2001.
25. Lingyun Yang, Ian Foster, and Jennifer M. Schopf. Homeostatic and tendency-based cpu load predictions. In *In Proceedings of IPDPS 2003.* IEEE Computer Society, 2003.
26. Lingyun Yang, Jennifer M. Schopf, and Ian T. Foster. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *Proceedings of the ACM/IEEE Super Computing Conference*, page 31, 2003.