INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Forecasting for Cloud computing on-demand resources based on pattern matching

Eddy Caron, Frédéric Desprez and Adrian Muresan

N° 7217

February 2010

Domaine 3

apport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7217--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Forecasting for Cloud computing on-demand resources based on pattern matching

Eddy Caron, Frédéric Desprez and Adrian Muresan

Domaine : Réseaux, systèmes et services, calcul distribué
Équipes-Projets Avalon

Rapport de recherche n° 7217 — February 2010 — 20 pages

Abstract: The Cloud phenomenon brings along the cost-saving benefit of dynamic scaling. Knowledge in advance is necessary as the virtual resources that Cloud computing uses have a setup time that is not negligible. We propose a new approach to the problem of workload prediction based on identifying similar past occurrences to the current short-term workload history.

We present in detail the auto-scaling algorithm that uses the above approach as well as experimental results by using real-world data and an overall evaluation of this approach, its potential and usefulness.

Key-words: Cloud Computing, auto-scaling, pattern matching

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Prédiction des réservations de ressources de Cloud Computing à la demande par la méthode de reconnaissance de motifs

Résumé : Le Cloud Computing permet de bénéficier de l'extensibilité dynamique. Une connaissance anticipée des évènements est nécessaire afin de prendre en compte le temps non négligeable de mise en place des ressources virtuelles fournies par les plates-formes de Cloud Computing. Nous proposons une nouvelle approche à ce problème de prédiction de charge basée sur la corrélation d'événements passés similaires à l'historique à court terme de la charge observée.

Nous présentons en détail un algorithme de gestion automatique de l'extensibilité qui utilise cette nouvelle approche. Nous proposons également des expérimentations utilisant les traces de plates-formes réelles ainsi qu'une évaluation de cette approche.

Mots-clés : Cloud Computing, extensibilité automatique, reconnaissance de motifs

1 Introduction

The evolution of IT software services in the direction of Cloud Computing took a step forward in the efficient use of hardware resources. The single most important benefit of Cloud Computing is its ability to allow the use of on-demand resources. This leads to dynamically-scalable systems and platforms. To take full advantage of the benefits of dynamic scaling, a Cloud client (user or middleware) needs to be able to make accurate decisions on when to scale up and down.

To achieve good performance, the Cloud client needs to be able to make accurate scaling decisions in advance to compensate for the overhead of using virtual resources, specifically their setup time. Therefore, a prediction method is usually used as it best suits this task. The current article presents a new approach to the auto-scaling problem based on identifying past patterns that are similar to the present use of the system.

2 Existing approaches

There are currently two main approaches to the cloud auto-scaling problem:

1. The first approach treats the past server usage as a predictable sequence and constructs a mathematical model around it. As a result, the next value of the request sequence is obtained by evaluating the obtained model at the next time point.

2. The second approach is a reactive one, based on the current server load and auto-scaling rules that are set up by a human operator (usually a cloud client). This approach has been referred to as the "Elasticity rules" approach or the "SLA" approach.

In [6] a description and comparison of three different auto-scaling algorithms is given: auto-regression of order 1 (AR1), Linear Regression and the Rightscale algorithm.

The auto-regression of order 1 algorithm is from the first category of auto-scaling algorithms. Its approach is to use a finite history window and identify appropriate parameters so that a recurring sequence can be obtained and therefore used to calculate the next values. The obtained parameters are adapted as the window slides along the time axis.

The linear regression algorithm is also from the first category and calculates a polynomial approximation of the history of requests. The predicted value is then obtained by evaluating the polynomial at a higher point along the time axis.

The Rightscale algorithm is from the second category, being a version of threshold-based auto-scaling. Its approach is to use a democratic voting system that is based on the current server load. Each virtual machine owned by the cloud client has a vote based on its current load level and two thresholds: low threshold that corresponds to a "scale down" vote (with a default value of 30% system usage) and a high threshold that corresponds to a "scale up" vote (with a default value of 85% system usage). The votes are collected by a central machine and the majority decides the scaling decision for the whole platform.

The three algorithms have been put side-by-side and compared by a metric proposed in the same article. Their performance is considerably high.

A more complex for of SLA-based dynamic provisioning can be described by using elasticity rules that dictate what part of the cloud client needs to scale, in what direction and by how much. In [4] we find such an example with threshold-based rules.

Scalability rules have the benefits of combining the high performance of threshold-based algorithms such as Rightscale with tune-ability and therefore have been widely used in practice in commercial clouds.

In [1] a Decentralized Online Clustering model is described and proposed for automatic workload provisioning for enterprise grids and clouds and addresses their distributed nature. In this approach a workload prediction algorithm is used and integrated into the system to model the application dynamics. More specifically, a quadratic response surface model is used.

1	A A	B B	A A	B B	A C	B	C
2	A	B A	A B	B A	A B	B C	C
3	A	B	A A	B B	A A	B B	C C

Table 1: KMP example

3 Idea description

A Cloud client is provisioned depending on its use. The use of a Cloud client can sometimes have a repetitive behavior. This can be caused by the similarities between tasks that the Cloud client is running or the repetitive nature of human behavior.

The idea of self-similarity in web traffic is not new [3]. Based on this a new auto-scaling strategy can be elaborated. By identifying usage patterns that have occurred in the past and have a high similarity to the present usage pattern, a decision can be made as to the necessity and/or direction of scaling for the present situation.

The pattern strategy has two inputs: a set of past Cloud client usage traces and the present usage pattern that consists of the last usage measure of the Cloud client. Due to the similarities of usage when talking about Cloud clients working in the same application domain it follows that the best type of historical data to be used for the pattern matching are taken from the same application domain as the application that is trying to predict its own usage. Therefore it would make sense to isolate historical data based on application domains before usage.

The present usage pattern of the Cloud client is used to identify a number of patterns in the historical set that are close to the present pattern itself. Identified patterns should not be dependent on their scale, just on the relation between the elements of the identified pattern and the pattern we are looking for. The resulting closest patterns will be interpolated by using a weighted interpolation (the found pattern that is closest to the present pattern will have a greater weight) and will have as result an approximation of the values that will follow after the present pattern. In essence, the usage of the Cloud client is predicted by finding similar usage patterns in the past or in other usage traces.

4 The string matching problem

The problem of finding a pattern inside an array of data that is very similar to a given pattern is close to the problem of string matching. We shall now describe the problem of string matching and its relation to the problem that the current article addresses.

The string matching problem consists of finding the position of a string (called pattern) inside a larger string. There are several approaches and plausible solutions to this problem. We have chosen the Knuth-Morris-Pratt (abbreviated KMP) as its performance are good as described in [2]. The KMP algorithm consists of a preprocessing step with a running time of $\Theta(m)$ where m is the length of the matching pattern and a matching step with running time of $\Theta(n)$ n where is the length of the string to match against. The algorithm is also trivially parallelizable as it is data independent therefore the input data can easily be divided into independent blocks on which the algorithm can run in parallel.

The efficiency of the KMP algorithm is due to its approach in saving unnecessary comparisons in case of a mismatch between the pattern and the string to match against. It is able to do this by first identifying repetitive prefixes of the input pattern in the preprocessing step.

Consider the following example: input pattern P = “ABABC” and matching string T =“ABABABC”.

There are three possible positions for P to be found in T, by using a sliding window approach, until one of the matches succeeds:

P =	A	B	A	B	C
π =	-	-	0	1	-

Table 2: Calculating the auxiliary array

When step 1 fails in the example in Table 1, the pattern slides to the next possible position in the matching string and a new comparison is made in step 2. After step 2 fails, the pattern slides once again and reaches step 3 which makes a full match.

In Table 1, step 2 can be skipped altogether if we consider the relation that the pattern has with itself, i.e. its repetitive prefix. Once the first 4 characters of P have been matched against the 4 consecutive characters in T (the following 4 characters starting from position 0) we deduce that there is no need to restart the whole matching from position 1 in T because, from analyzing P we know that the match will fail as the 4 characters of T starting from 0 are the same as the first 4 characters of P starting from 0.

To aid the matching process, an auxiliary array is constructed over P (called π) that contains at position i, the ending position of the largest prefix of P that is a suffix of P[0..i]. For the P in our example, we have the results in Table 2.

The entries in π that have a value of "-" represent entries that are not prefixes of P. For example the second "A" in P is both a prefix and also a suffix of P[0..2] = "ABA". The largest prefix that is also a suffix for P[0..2] is "A" and has the ending position at P[0]. This means that once we have matched P[0..2] = "ABA" in T and P[3] does not match in T, we can continue matching in T from the same index of T, and we can start in P knowing that we have already matched the first character in P, as it is a prefix of length 1 of P[0..2]. Therefore we resume matching with the P index of $\pi[2] +1 = 0 +1 = 1$. Now, resuming our matching example, in step 1 we have matched P[0..3] to T[0..3]. We have P[4] != T[4], but we know that P[0..3] = "ABAB" has "AB" as the largest prefix that is also a suffix. So we can resume by matching T[4] to $P[\pi[3]] = P[1]$, skipping P[0].

The preprocessing step The goal of the preprocessing step is to calculate the π array. At each index i, π stores the end position of the longest prefix of P[0..i], that is also a suffix of P[0..i]. The algorithm for this has a runtime of $\Theta(m)$ where m is the length of P (Algorithm 1).

Algorithm 1 Calculate-prefix(P)

```
1: m ← length(P)
2: π[0] ← -1
3: k ← -1
4: for q ← 1 to m - 1 do
5:    while k > -1 and P[k+1] ≠ P[q] do
6:       k ← π[k]
7:    end while
8:    if P[k+1] = P[q] then
9:       k ← k+1
10:   end if
11:   π[q] ← k
12: end for
13: return π
```

The matching step The matching algorithm (Algorithm 2) has a runtime of $\Theta(n)$, where n is the length of T, the string to match against. It is very similar to a naive matching algorithm, but improved to skip redundant comparisons.

Algorithm 2 KMP(T, P)

```
1: n ← length(T)
2: m ← length(P)
3: π ← Calculate-prefix(P)
4: q ← -1
5: for i ← 0 to n - 1 do
6:    while q > -1 and P[q+1] ≠ P[i] do
7:       q ← π[q]
8:    end while
9:    if P[q+1] = T[i] then
10:      q ← q+1
11:      if q = m-1 then
12:         write "Found at position" i-m
13:         q ← π[q]
14:      end if
15:   end if
16: end for
```

5 Algorithm description

The KMP Algorithm (2) is a good solution to the string matching problem. Despite the great similarities, our own pattern matching problem has some particularities of this own:

1. an approximate matching is needed since the odds of finding an identical pattern to the one we are looking for are considerably low;
2. matches which are too dissimilar either on small intervals or as a whole need to be discarded;
3. when comparing the pattern to the matching data, scale also needs to be taken into consideration. To be more exact, the scale of the pattern and the scale of the possible match should not affect the comparison, therefore it needs to be scale-independent.
4. the resulting matches are interpolated having different weights on the final result, based on their similarity to the identified pattern.

In order to do an approximate matching, the original KMP algorithm needs to be changed in the content of both functions, therefore they need to be modified accordingly.

Two types of approximation errors are used for the matching:

1. an instant error which dictates the amount by which the current match is allowed to differ from the pattern by comparing in smallest possible units;
2. a cumulative error that characterizes the amount by which the current match is allowed to differ from the pattern as a whole. This is basically a sum of the instant errors of the whole matching.

Figure 1 illustrates graphically the difference between the two types of acceptable errors (instant and cumulative) when comparing two patterns.

5.1 Scale-independent comparison

The distance between the pattern we are trying to match and a candidate pattern should be calculated in a scale-independent manner by first normalizing the two pattern values to a common scale. To decrease floating point approximation errors, one can opt for a distance calculation that does not use divisions and therefore calculating only on integer values.

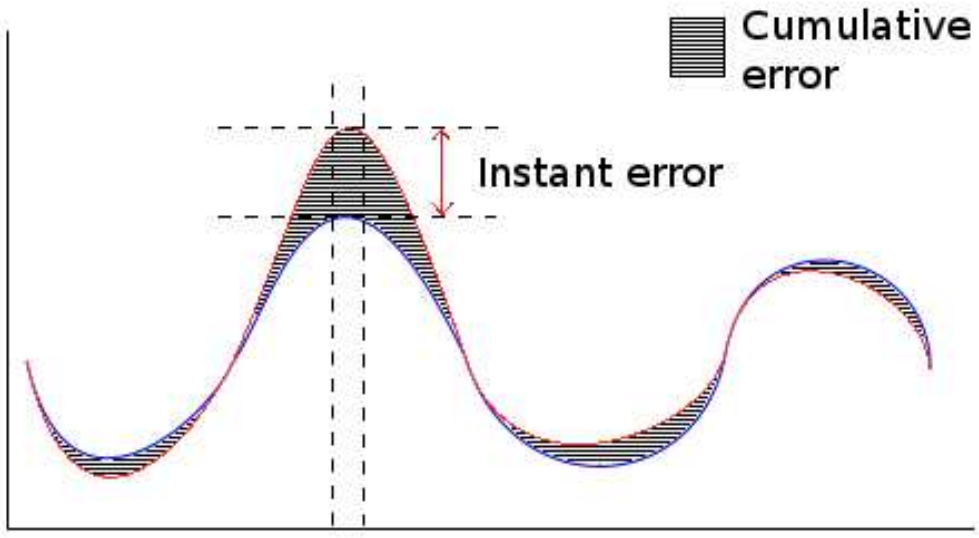

Figure 1: Difference between the two types of acceptable errors.

As an example consider the pattern and the candidate from Figure 2. The pattern is an array containing the values: 20 , 38 , 21 and the candidate match contains the values: 42 , 81 , 39 . In this form we cannot compare the two patterns. A first idea would be to normalize both arrays to a floating point [0..1] interval and then compare. Working with floating point numbers can be avoided by working with big integer numbers. To reach a common scale we simply multiply each array by the scale of the other. For the scale of each array we can simply consider the first element. As a result, the pattern array is multiplied by the scale of the candidate (this is 42) and the candidate is multiplied by the scale of the pattern (which is 20). The result is depicted in Figure 3. In this new situation, comparing two components of each array is done simply by subtraction. The instant error is used here to assure that there are no two components that differ two much (in percentage) from the two arrays.

Once the comparison is done, the identified candidate is stored along with its total distance from the pattern. This facilitates the significance of the result, as the candidate that is closest to the pattern has a higher weight in final result.

The pseudocode for computing the instant error is illustrated Algorithm 5.1 in the Distance function.

Algorithm 5.1 Distance(PatternElement, PatternScale, DataElement, DataScale)

return
PatternElement × DataScale
- DataElement × PatternScale

The cumulative error is obtained by summing up the instant errors from all the elements of the pattern and candidate. This is illustrated in the CumulativeDistance function.

5.2 KMP modification

The prefix calculation function is changed as described in Algorithm 4. The scales of the two components that we are comparing are represented by the first value of each component. This is arguable, but in practice we have achieved good results with this approach. In the function, scaleK represents the scale of the prefix and scaleQ represents the scale of the postfix of the pattern.

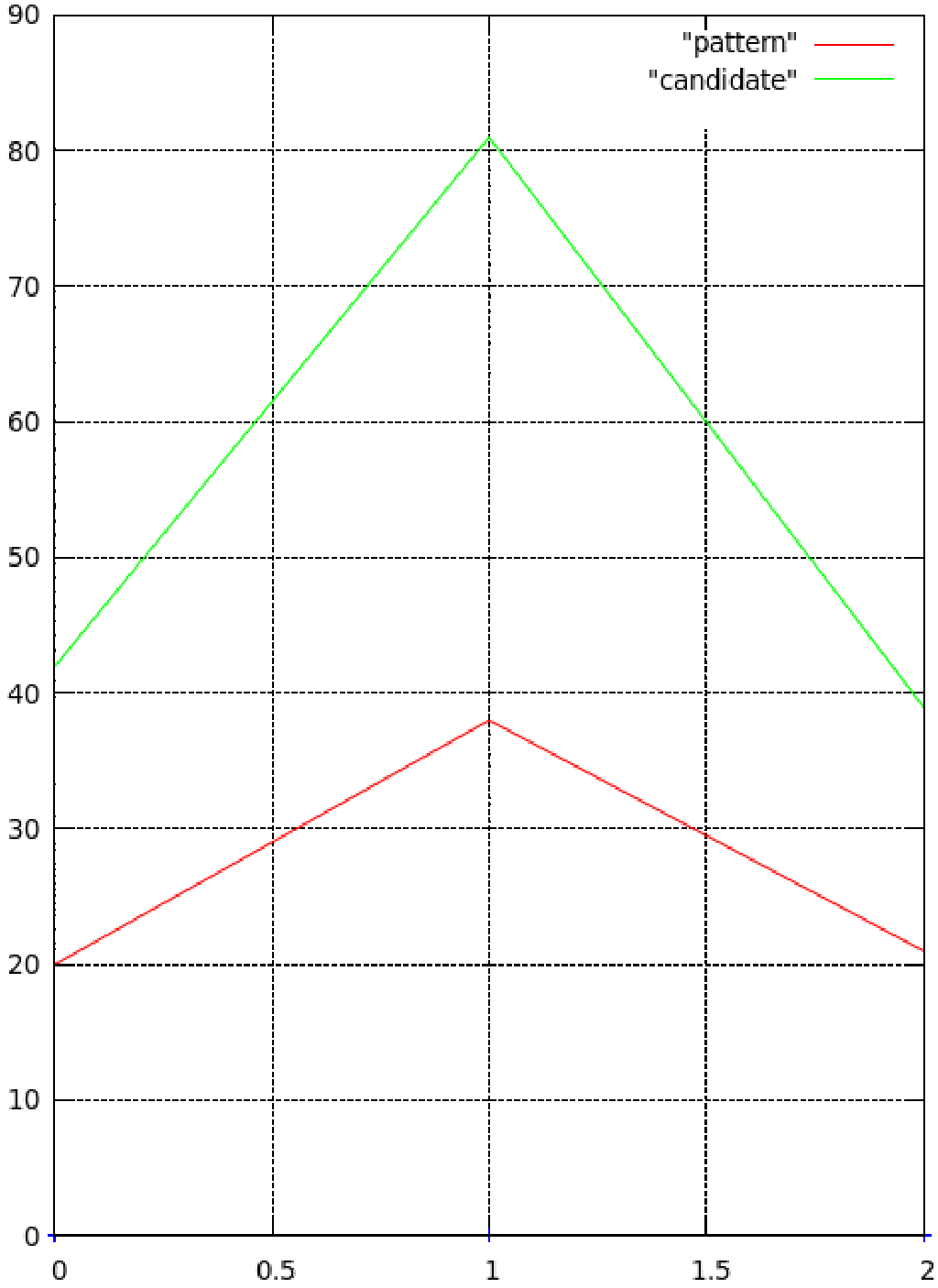

Figure 2: Scale-independent comparison - initial

The Distance function returns an appreciation of the distance between two different pattern instances, each having a different scale which is passed as parameter. The comparisons on lines 9 and 14 assure that the current instant distance does not differ by more then the acceptable error (in percentage) from the actual

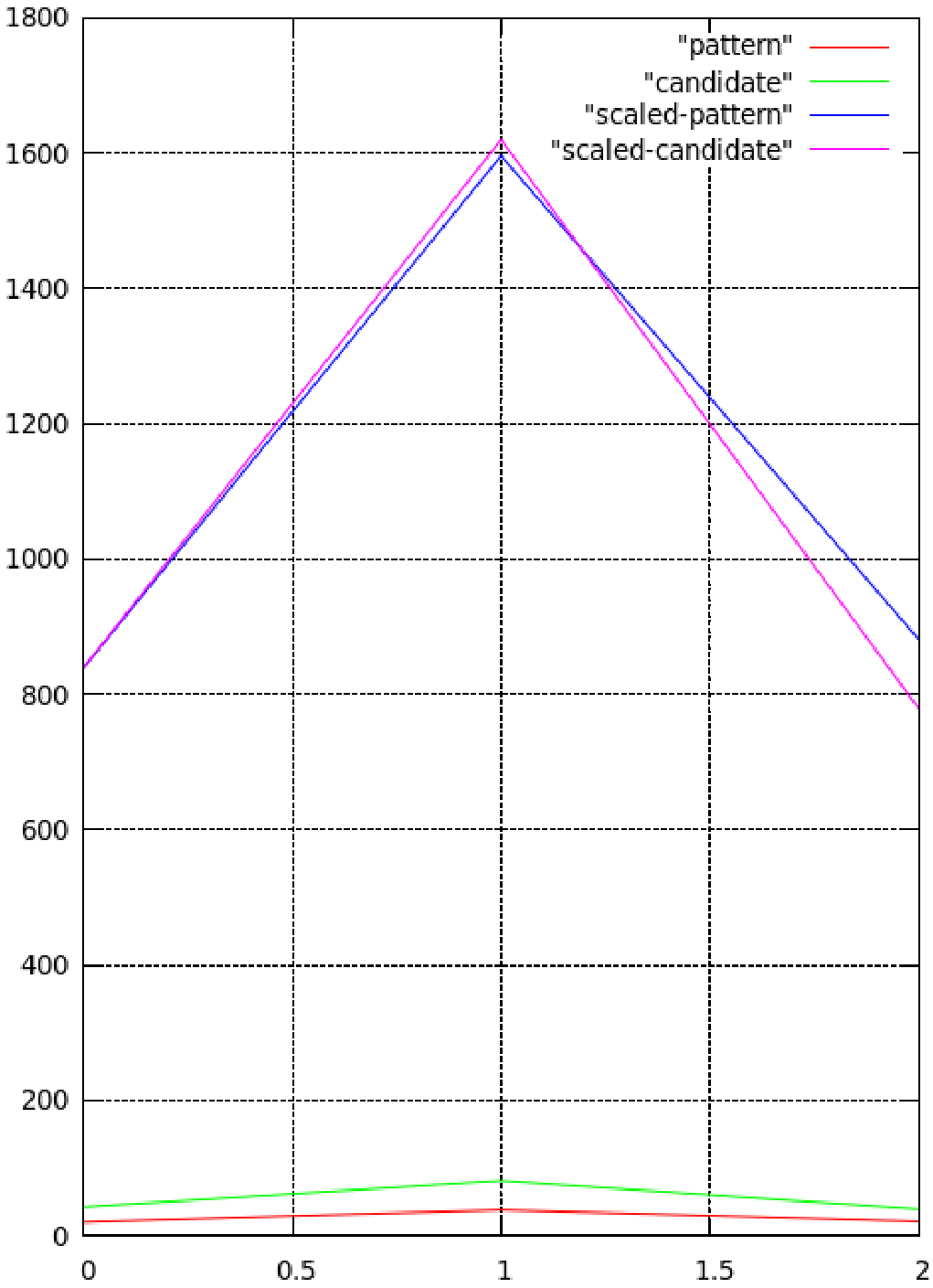

Figure 3: Scale-independent comparison - common scale

pattern that we are matching. The scaleQ term, representing the scale of the data, from the comparison is needed for bringing the current term of the pattern to the same scale as the data.

The matching algorithm is changed as described in Algorithm 5.

Algorithm 3 CumulativeDistance(P, T, DataOffset)

```
1: patternScale ← P[0]
2: dataScale ← T[DataOffset]
3: length ← length(P)
4: distance ← 0
5: for index ← 0 to length do
6:    distance ← distance + | dataScale × P[index] - patternScale × T[index + DataOffset] |
7: end for
8: return distance
```

Algorithm 4 Calculate-prefix-approx(P, ACCEPT_INST_ERR)

```
1: m ← length(P)
2: π[0] ← -1
3: k ← -1
4: scaleK = P[0]
5: scaleQ = P[1]
6: for q ← 1 to m - 1 do
7:    dist ← Distance(P[k+1], scaleK, P[q], scaleQ)
8:    maxDistance ← ACCEPT_INST_ERR × scaleQ × P[k+1]
9:    while k > -1 and dist > maxDistance do
10:      k ← π[k]
11:      dist ← Distance(P[k+1], scaleK, P[q], scaleQ)
12:      scaleQ = P[q - (k+1)]
13:   end while
14:   if dist ≤ ACCEPT_INST_ERR × scaleQ × P[k+1] then
15:      k ← k+1
16:   end if
17:   π[q] ← k
18: end for
19: return π
```

The main difference when compared to the original KMP algorithm is the use of the instant and cumulative distances as a means of filtering out potential matches that are too different either on small time intervals or as a whole.

On lines 10 and 16 we assure that the instant distance between the identified candidate and the pattern is no more than what the acceptable error permits. In order to assure a correct comparison, the pattern term needs to be scaled to the same size as the data, hence the scaleT term is used in the comparison.

Filtering by cumulative distance is done in lines 20 to 24. The CumulativeDistance function returns a sum of instant distances for every instant of the two compared arrays. The running time of this function is $\Theta(m)$ where m is the length of the arrays, which in our case is always equal to the length of P.

Line 22 of the algorithm assures that the cumulative distance of the candidate does not differ more than is accepted by the cumulative error from the pattern itself. The pattern itself is represented by the patternSum term in the comparison. This is a sum of all the terms in the pattern and should be calculated only once, at the beginning of the algorithm. The pattern sum needs to be brought to the same scale as the candidate sequence and therefore the scaleT term is used.

Filtering by an acceptable cumulative error that is smaller or equal to the acceptable instant error is useless. This conclusion is trivial when taking into consideration that the cumulative error is a sum of all the instant errors.

The use of the cumulative error changes the running time of the matching algorithm to $\Theta(n \times m)$ in the worst case, where n is the length of the string to match against and m is the length of the input pattern.

Algorithm 5 KMP-approx(T, P, ACCEPT_INST_ERR, ACCEPT_CUMUL_ERR)

```
1: n ← length(T)
2: m ← length(P)
3: π ← Calculate-prefix(P)
4: q ← -1
5: scaleP = P[0]
6: scaleT = T[0]
7: for i ← 0 to n - 1 do
8:    dist ← Distance(P[q+1], scaleP, T[i], scaleT)
9:    maxDist ← ACCEPT_INST_ERR × scaleT × P[q+1]
10:   while q > -1 and dist > maxDist do
11:      dist ← Distance(P[q+1], scaleP, T[i], scaleT)
12:      q ← π[q]
13:      scaleT = T[i - (q+1)]
14:      maxDist ← ACCEPT_INST_ERR × scaleT × P[q+1]
15:   end while
16:   if dist ≤ maxDist then
17:      q ← q+1
18:   end if
19:   if q = m-1 then
20:      dist ← CumulativeDistance(P, T, i - m + 1)
21:      maxDist ← ACCEPT_CUMUL_ERR × patternSum × scaleT
22:      if dist ≤ maxDist then
23:         StoreSolution(dist / scaleT, i - m + 1)
24:      end if
25:      q ← π[q]
26:      scaleP = P[q+1]
27:      scaleT = T[i - (q+1)]
28:   end if
29: end for
```

5.3 Interpolating the found values

Once approximate matches have been found, the problem of obtaining a relevant result from those matches is raised. Each match should have a contribution to the final result that is proportional to its relative distance to the pattern with respect to the other identified patterns. This corresponds to a weighted sum of the identified matches, where weights are calculated by considering the distance of the current match to the pattern and to the rest of the matches.

Once the weights are calculated, the interpolation is performed between the following L elements after each approximate match. The result is a predicted sequence of length L.

5.4 Algorithm parameters

The algorithm accepts a number of parameters useful for fine-tuning in accordance to the each use-case. These parameters are:

- The maximum number of matches (called closest neighbors) to take into consideration (denoted K).
- The length of the predicted sequence (denoted L).
- The acceptable instant error representing the amount by which the identified sequence is allowed to differ on the smallest possible interval lengths from the pattern we are looking for.
- The acceptable cumulative error which represents the amount by which the identified sequence is allowed to differ as a whole from the pattern we are looking for.

- The input set of data representing the database of past requests.
- The input pattern representing a sequence with the last period of requests received.

The first parameter is not independent of the others. It is actually influenced considerably by the acceptable errors. The correlation is strong and can be expressed very easy: the larger the acceptable error, the more matches the algorithm identifies, but the more irrelevant they will be.

Calculating the acceptable errors

The value of the acceptable errors can be calculated based on the maximum number of neighbors that we wish to find. The approach for this is to use a binary search to zone in on the appropriate values for the acceptable errors.

By using the binary search approach, we have obtained values that have proven to be good in practice. We have used a lower bound of 20% of K for a minimum of identified neighbors and 90% of K as the upper bound for maximum number of identified neighbors.

Calculating the appropriate pattern length

The length of the pattern that represents the last traces of server usage has a great impact on the results of the algorithm. Finding the appropriate length is a problem on its own as we have a trade-off between patterns of big lengths that yield a small number of similar candidates, that might be to small in order to be usable, and patterns of small lengths, that find more candidates but they tend to be more irrelevant to our current situation.

We have taken two approaches to this problem. The first approach is to find the most lengths of the most frequent repetitive patterns and use the same length as input to the prediction algorithm.

We have the following constructive approach to identifying the length of the most frequent repetitive patterns:

1. find all similar patterns of length 2 in the historic data
2. take all similar patterns of length 2 and try to match the next element too. This yields all similar patterns of length 3.

   ...
3. take all patterns of length n and try to match the next element too. This yields all similar patterns of length $n + 1$.

The result is that the number of identified similar patterns decreases as the length of the patterns increases:

$$count[n+1] \leq count[n] \leq ... \leq count[3] \leq count[2]$$

The conclusion is that the most frequent patterns are of the ones with length 2. In practice, using a pattern length of 2 would have the following consequences:

- Good for predicting very short in advance (i.e. 1)
- Loses meaning when trying to predict longer sequences
- The idea of trend is lost as the steps are very small while the trend is a longer sequence

We need to have a better way for choosing the pattern length, that would give more relevant results and avoid pollution as much as possible.

The length of the pattern should be influenced by the time it takes to service a request on the server. We then have the following possibilities:

- Median / average
  - Representative of most of the requests

	LCG	Nordugrid	SHARCNET
Avg	8970	91893	33516
Min	0	0	0
5%Min	69	11	73
10%Min	79	24	152
Median	255	3861	12165
Max	586702	1452763	7449415
Tested	200 2500 5000 10000 50000 100000	10000	10000

Table 3: Request length measurements on different grid traces

- Minimum
  - A large pattern cannot match against a smaller pattern that's half different
  - A small pattern can match against a large pattern that's half different
  - The minimum is very probably close to 0 (grid testing experiments)
  - A close minimum can be selected (ex. 5% - 10% from the bottom)

By using real-world grid traces from the workload archive of TUDelft University [9] we have obtained the results in the table below for the time it takes to service a request. We have tested traces from several research grids [5, 7, 8] to get a real-world appreciation of possible values for the pattern length, by taking different metrics. The results can be viewed in the table below (values represent the time it takes to service a request in seconds, weighted by CPU count per request):

We can also consider plots of sorted request lengths weighted by the number of CPUs assigned to each request.

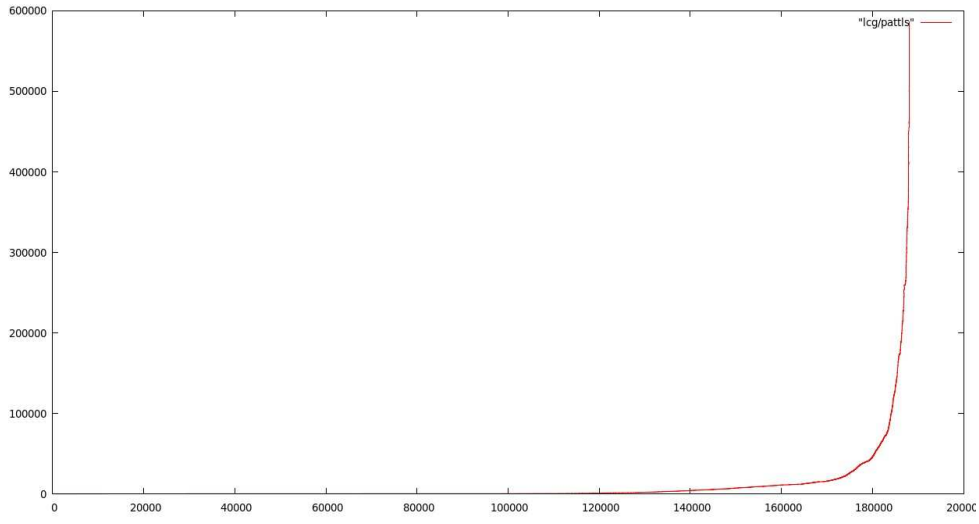

Figure 4: Requests on the LCG platform, sorted by time length and weighted by the CPUs used

The implications of Figures 4, 5 and 6 along with the previous table are that for all practical purposes, a pattern length that is a minimum or even median of the time it takes to service a request is unusable when

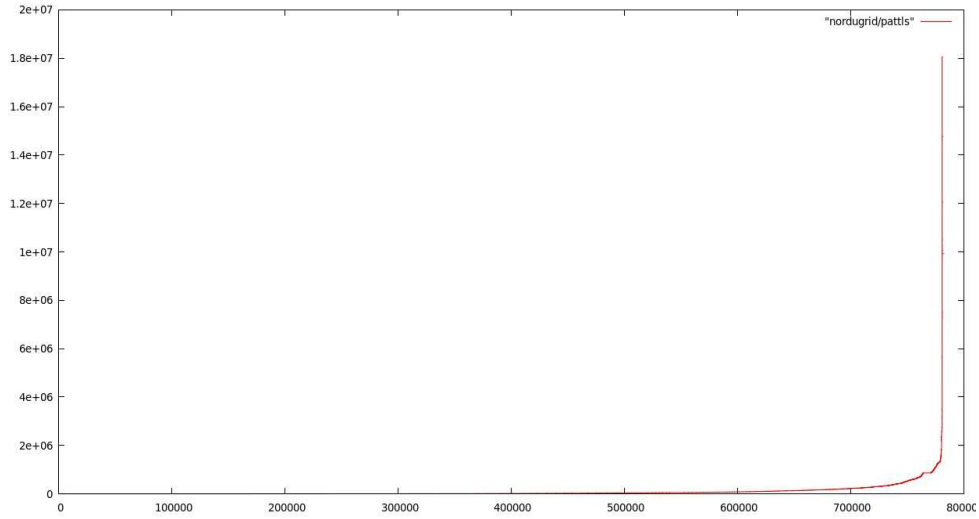

Figure 5: Requests on the NorduGrid platform, sorted by time length and weighted by the CPUs used

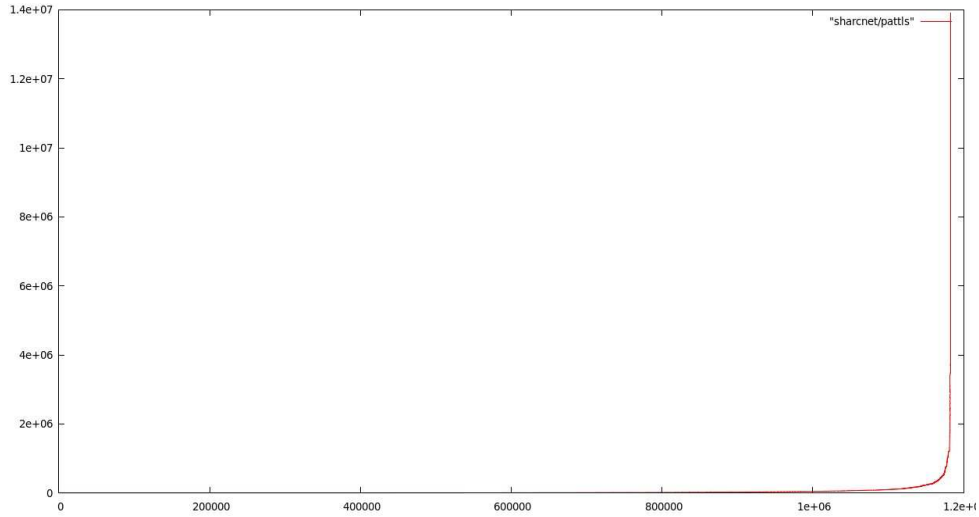

Figure 6: Requests on the SHARCNET platform, sorted by time length and weighted by the CPUs used

dealing with servers that have a similar usage to the research grids described above. In practice we have used the average of the request service time and have obtained good results.

6 Obtained results

6.1 Data sources

We have tested our auto-scaling approach with traces from three different research grids, each having its own particularities of use, with main differences in the frequency and amplitudes of changes in their overall usages.

LCG - Large Hadron Collider Computing Grid [1]

[1] `http://lcg.web.cern.ch/LCG/`

Here we find traces from several node from the computing grid associated to the Large Hadron Collider. Its behavior is mildly oscillatory and a plot of number of different simultaneous processes running on the grid, descretized across 100 seconds can be found in Figure 7.

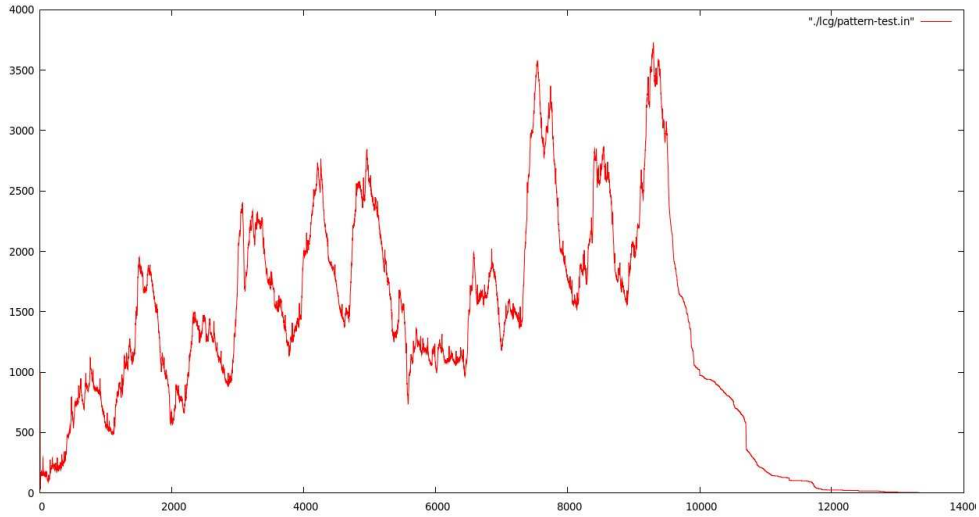

Figure 7: LCG - plot of number of concurrent requests versus time

NorduGrid [2]

Here we find higher amplitudes for oscillations as the gird is more heterogeneous then the previous. A plot of the number of different parallel requests descretized across 100 seconds intervals can be found in Figure 8.

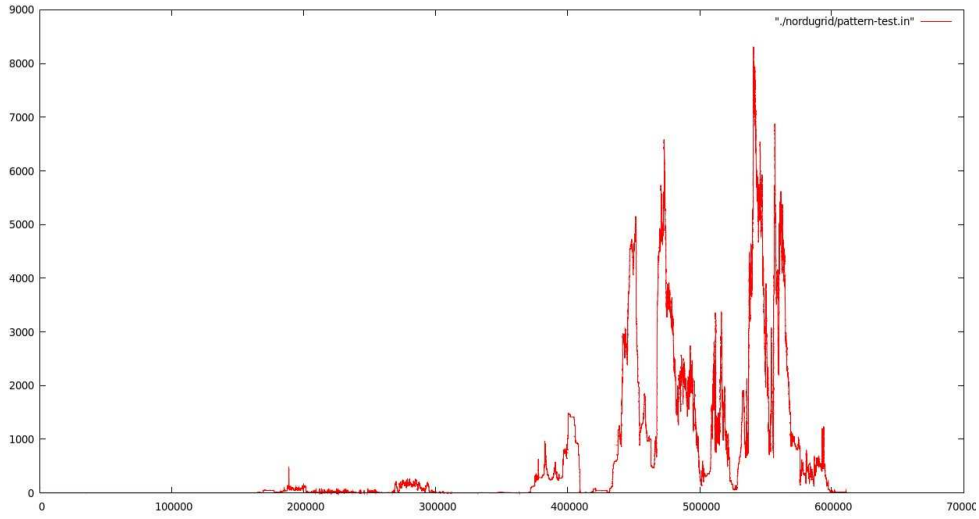

Figure 8: NorduGrid - plot of number of concurrent requests versus time

SHARCNET [3]

SHARCNET has been described as a "cluster of cluster". Its volatility is very high and its amplitudes can reach surprising peaks. A plot of the number of different parallel requests descretized across 100 seconds intervals can be found in Figure 9.

[2] `http://www.nordugrid.org`
[3] `http://www.sharcnet.ca`

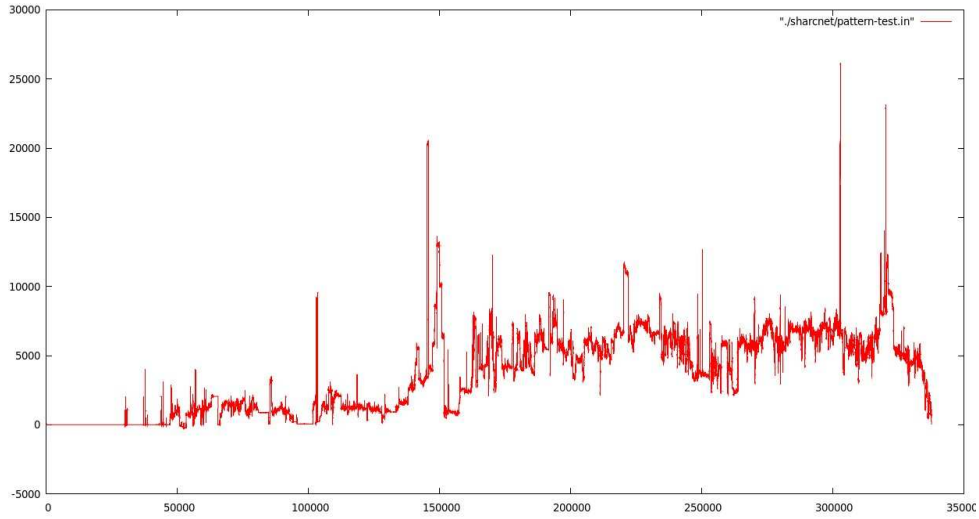

Figure 9: SHARCNET - plot of number of concurrent requests versus time

6.2 Experiment setup

All the experiments use the server traces of the same form of input data as described above with time units of 100 seconds, and request value consisting of the number of different parallel requests across the 100 seconds. A pattern length of 100 time units has been used for all the experiments (this is 100 × 100 seconds - approximately 2.7 hours of server time) and predictions are made for one time unit, this is 100 seconds, which is a little over 1 minute 30 seconds.

The results are displayed under the form of a set of standard metrics that include minimum, maximum, median and average percentage and value difference between the prediction and the actual value.

A second set of metrics has also been used that allows the comparison to other existing auto-scaling algorithms. This metric was proposed and used by UCSB to compare the performance of three existing auto-scaling algorithms [6]: auto-regression of order 1, linear regression and the Rightscale democratic voting algorithm.

We have used two versions of the metric proposed by the UCSB team:

- An instant score where we considered resource cost as being charged per fraction of an hour, although this is not the case in current cloud providers
- A second score where we take the maximum prediction over the course of an hour and use that as static provisioning for the whole hour

6.3 Results

Predicting LCG with LCG as historic data

We have done a self-prediction test by using LCG as historic data with the purpose of predicting LCG itself. When filtering out potential pattern candidates, exact matches have been ignored, since the pattern itself is a piece of the historic data.

The results of this experiment can be found in Table 4.

Predicting NorduGrid with LCG as historic data

We have experimented with using traces from a different grid, but that resembles the the one we are trying to predict. In the current test case we have tried to predict NorduGrid workloads by using LCG as historic data.

The experiment's results can be seen in Table 5.

Predicting LCG with NorduGrid as historic data

Metric	Value
Minimum percentage difference (%)	0.0
Minimum value difference	0.0
Maximum percentage difference (%)	53.4
Maximum value difference	220.97
Median percentage difference (%)	1.0
Median value difference	8.9
Average percentage difference (%)	1.749
Average value difference	15.33
UCSB metric (maximum per 1 hour)	5.44
UCSB metric (instantaneous)	-11.08

Table 4: Results of predicting LCG with LCG as historic data

Metric	Value
Minimum percentage difference (%)	0.0
Minimum value difference	0.0
Maximum percentage difference (%)	1146.00
Maximum value difference	435.2
Median percentage difference (%)	1.74
Median value difference	0.26
Average percentage difference (%)	35.38
Average value difference	5.739
UCSB metric (maximum per 1 hour)	27.68
UCSB metric (instantaneous)	23.06

Table 5: Results of predicting NorduGrid with LCG as historic data

Metric	Value
Minimum percentage difference (%)	0.0
Minimum value difference	0.0
Maximum percentage difference (%)	100.0
Maximum value difference	217.12
Median percentage difference (%)	1.2
Median value difference	13.98
Average percentage difference (%)	7.32
Average value difference	18.46
UCSB metric (maximum per 1 hour)	3.43
UCSB metric (instantaneous)	-10.71

Table 6: Results of predicting LCG with NorduGrid as historic data

Metric	Value
Minimum percentage difference (%)	0.0
Minimum value difference	0.0
Maximum percentage difference (%)	528.03
Maximum value difference	5.64E17
Median percentage difference (%)	0.9
Median value difference	11.26
Average percentage difference (%)	375.65
Average value difference	4.03
UCSB metric (maximum per 1 hour)	-3.23
UCSB metric (instantaneous)	-2.06

Table 7: Results of predicting SHARCNET with NorduGrid as historic data

We have experimented with the symmetric of the previous experiment in trying to predict LCG workloads by using NorduGrid as historic data.

The results are viewable in Table 6

Predicting SHARCNET with NorduGrid as historic data

We have also experimented the behavior of the algorithm when using historic data that does not have a high resemblance to the workload that is being predicted. In our experiment we have used NorduGrid traces as historic data when trying to predict SHARCNET traces. The results of this experiment are available in Table 7.

An analysis of the results reveals that this is a feasible approach to auto-scaling. It is clear that the algorithm yields better results when the set of historic data that is used has a similarity to the signal that is being predicted. This similarity is influenced by several parameters that constitute the domain of the server whose load is being predicted. It follows from the obtained results that data from the same domain can easily be used to predict one-another.

Predicting LCG with LCG as historic data and varying pattern lengths and historic data lengths

Although we cannot show that the algorithm yields the best results, we can show that its results improve as we increase the size of the historic data and as we find the best pattern length to take into consideration when predicting. The tables below illustrate results when varying the pattern length and the length of the historic data used for prediction.

We have varied the historic data from 100% - the full set, to 50%, 25% and 12.5% of the set.

The pattern length has also been varied from 1000 time units to 500, 100, 50 and 25.

Table 9 contains the results of the experiment when calculating the metric proposed in [6] and using instant values for the the number of virtual resources.

Pattern / data length	100.0%	50.0%	25.0%	12.5%
1000	-18.99	-36.37	-57.83	-97.37
500	-9.43	-19.97	-23.47	-43.06
100	5.44	3.32	4.05	4.05
50	9.41	9.6	8.48	8.21
25	10.67	11.11	12.62	11.79

Table 8: UCSB metric (maximum per one hour)

Pattern / data length	100.0%	50.0%	25.0%	12.5%
1000	-38.96	-59.57	-79.25	-103.57
500	-31.36	-38.54	-45.88	-63.18
100	-11.08	-13.81	-16.49	-15.44
50	-4.54	-4.83	-7.22	-8.23
25	-0.14	-0.3	-0.05	-1.36

Table 9: UCSB metric (instant)

Table 8 contains results of applying the previous metric by using the maximum across each hour as reference point for virtual resources and cost.

There is a clear tendency for the score of the prediction to improve as the set of historic data increases in size. There is another tendency for the prediction score to increase as the length of the predicted pattern decreases. This is obvious if we take into consideration that a smaller pattern length corresponds to more identified pattern candidate, yet we do not recommend the usage of considerably small pattern lengths as this makes the result be more and more independent of the server usage trend.

7 Conclusions and future work

One of the most important benefits of Cloud Computing is the ability of a Cloud Client to be dynamically scalable based on its use. This has great implications on cost saving as resources are not paid for when they are not used.

Dynamic scalability is achieved through virtualization. The downside of virtualization is that they have a non-zero setup time that has to be taken into consideration for an efficient use of the platform. It follows that a prediction method would greatly aid a Cloud Client in making its auto-scaling decisions.

In the current work a new resource usage prediction algorithm is presented. It uses a set of historic data to identify similar usage patterns to a current window of records that occurred in the past. The algorithm then predicts the system usage by interpolating what follows after the identified patterns from the historical data.

Experiments have shown that the algorithm has good results when presented with relevant input data and, more importantly, that its results can improve by increasing the historic data size. This makes the evaluation of the algorithm be context dependent.

As future work directions we will be looking into ways that a relevant set of historic data can be composed for a particular application domain.

References

[1] M. Parashar N. Gnanasambandam A. Quiroz, H. Kim and N. Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Proceedings of the 10 th IEEE/ACM International Conference on Grid Computing (Grid 2009)*, pages 50 – 57, 2009.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms, Chapter 32: String Matching*. McGraw-Hill Higher Education, 2001.

[3] Mark Crovella and Azer Bestavros. Explaining world wide web traffic self-similarity. Technical report, Boston, MA, USA, 1995.

[4] Fermín Galán, Americo Sampaio, Luis Rodero-Merino, Irit Loy, Victor Gil, and Luis M. Vaquero. Service specification in cloud environments based on extensions to open standards. In *COMSWARE '09: Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, pages 1–12, New York, NY, USA, 2009. ACM.

[5] Large Hadron Collider Computing Grid. http://lcg.web.cern.ch/lcg/.

[6] Patricio Jara Jeff Browne Jonathan Kupferman, Jeff Silverman. Scaling Into The Cloud. 2009.

[7] NorduGrid. http://www.nordugrid.org.

[8] SHARCNET. http://www.sharcnet.ca.

[9] TUDelft University The Grid Workloads Archive. http://gwa.ewi.tudelft.nl/pmwiki/pmwiki.php.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399