A Method for Estimating the Execution Time of a Parallel Task on a Grid Node

Panu Phinjaroenphan[1], Savitri Bevinakoppa[1], and Panlop Zeephongsekul[2]

[1] School of Computer Science and Information Technology,
[2] School of Mathematical and Geospatial Sciences,
RMIT University, GPO Box 2476V, Melbourne Australia
{pphinjar, savitri}@cs.rmit.edu.au
panlopz@rmit.edu.au

Abstract. The mapping problem has been studied extensively and many algorithms have been proposed. However, unrealistic assumptions have made the practicality of those algorithms doubtful. One of these assumptions is the ability to precisely calculate the execution time of a task to be mapped on a node before the actual execution. Since the theoretical calculation of task execution time is impossible in real environments, an estimation methodology is needed. In this paper, a practical method to estimate the execution time of a parallel task to be mapped on a grid node is proposed. It is not necessary to know the internal design and algorithm of the application in order to apply this method. The estimation is based upon past observations of the task executions. The estimating technique is a k-nearest-neighbours algorithm (knn). A backward predictor elimination, leave-one-out cross validation, and a statistical technique are used to derive the relevant parameters to be used by knn. Experimental results show that on average the proposed method can produce 2.3 times the number of accurate estimated execution times (with errors less than 25%) greater than the existing method.

1 Introduction

Computational grid has been introduced as a new distributed computing paradigm that is able to interconnect heterogeneous networks and a large number of computing nodes regardless of their geographical locations [1]. This new paradigm provides an access to tremendous computational power that can be harnessed for various applications. Parallel applications are developed to solve implementations of computational intensive engineering or scientific problems that require such power.

The main aim of solving such problems with a parallel application is to reduce the execution time. As a computational grid involves a large number of nodes, one of the challenging problems is to decide the destination nodes where the tasks of the application are to be executed. This process is formally known as the *mapping problem* [2]. In this paper, we broadly categorise studies of the mapping problem into two classes: *practical* and *theoretical*.

P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 226–236, 2005.

In practical studies, the focus is on investigating an efficient approach to map a *specific* parallel application on real environments. GrADS [3] and CGRS [4] are examples of practical mapping studies. The internal knowledge of the application, such as design and algorithm need to be known.

In theoretical studies, the problem is usually modelled at an abstract level using graphs, and internal knowledge of the application is assumed to be unknown. The developed mapping approach is therefore *generic*, but cannot be used in reality due to unrealistic assumptions of the graph based model.

The current situation indicates a lack of a *practical* and *generic* mapping approach. A methodology that can be undertaken to develop one such approach is to address the unrealistic assumptions in theoretical studies. One of these is the ability to precisely calculate the execution time of a task to be mapped on a node before the actual execution. In practice, such a calculation is impossible; however, an estimate (i.e. *estimated execution time*) can be made. This problem is formally called the *execution time estimation problem* [6].

In this paper, a method is developed to estimate the execution time of a parallel task, a task with inter-task communication, on a grid node. We only assume that the input problem size (e.g. the sizes of the matrices in a matrix-matrix multiplication application), the number of tasks, and the topology of the application are known. The estimation method is based on past observations of the task executions. A k-nearest-neighbours (*knn*) algorithm is employed as the estimating technique. The relevant parameters to be used by *knn* are dynamically and automatically chosen using the combination of a *backward predictor elimination*, a *statistical technique* and *leave-one-out cross validation* [5].

In the experiments, the proposed method is compared with the existing estimation method presented in [6] by estimating the execution times of the tasks of a matrix-matrix multiplication application developed with Cannon's algorithm (Cartesian topology). Experimental results show that on average the proposed method can produce 2.3 times the number of accurate estimated execution times (with error less than 25%) greater than the existing method.

2 Related Work

The solutions to the execution time estimation problem are categorised into *code analysis*, *analytic benchmarking and code profiling* and *past observations* [6].

The first two classes assume that the internal design and algorithm of the application are known. The user-supplied performance model used in GrADS [3] and CGRS [4] fit into these categories. On the other hand, estimation based upon past observations does not require any knowledge of the internal design and algorithm. However, some previous observations are essential.

An estimation method based upon past observations is proposed in [6]. The employed estimating technique is a k-nearest-neighbours algorithm. Even though their experimental results suggest a promising method, there are some shortcomings. Their method is inflexible since the number of predictors (variables used to make an estimate) is fixed. Another restriction assumption is that the exe-

cution time of a task only depends on the performance of the node the task is mapped on, and the input problem size. However, in practice, the execution time of a task may depend on its communications to other tasks. Another limitation of this method is that the number of neighbours (k) chosen is always equal to $n^{\frac{4}{5}}$, where n is the number of known observations, and no justification has been offered as to the choice of this number.

3 Estimation Based upon Past Observations

The execution time of a task on a given node depends on a vector $\mathbf{x}$ of p predictors, $\mathbf{x}^\top = (x_1\ x_2\ ...\ x_p)$. Given y as the execution time of a task, y is considered to be a function of $\mathbf{x}$.

$$y = f(\mathbf{x}^\top) \tag{1}$$

Some vectors of predictors and their corresponding execution times are known. Let $\mathbf{X}$ and $\mathbf{y}$ represent n of these vectors, respectively.

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \mathbf{x}_2^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & ... & x_{1p} \\ x_{21} & x_{22} & ... & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & ... & x_{np} \end{bmatrix}, \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{2}$$

The goal is to estimate the execution time $\hat{y}$ (*dependant*) from a vector of predictors $\mathbf{x}_q$ (*query point*) using the known observations $\mathbf{X}$ and $\mathbf{y}$ (*dataset*), and the percentage relative residual error ($\%e$) is used to evaluate the accuracy, i.e.

$$\%e = \frac{|\hat{y} - y|}{y} \cdot 100. \tag{3}$$

4 The Proposed Estimation Method

The proposed method consists of two processes: *estimating* and *learning*. The former is to estimate the execution time of the query point. The latter is to derive the relevant parameters to be used in the estimating process.

4.1 Estimating Process

Given $\mathbf{x}_q$ as the query point, with knn, $\hat{y}$ is the average of the other k execution times which are nearest neighbours of $\mathbf{x}_q$, i.e.

$$\hat{y} = \frac{\sum_{j=1}^{k} y_j}{k}. \tag{4}$$

The other k execution times are determined from the Euclidean distance, $d(\cdot)$, of their predictors to the query point $\mathbf{x}_q$, which is given by

$$d(\mathbf{x}, \mathbf{x}_q) = \sqrt{\sum_{i=1}^{p} (w_i \cdot (x_i - x_{q_i}))^2} \tag{5}$$

where x_i and x_{q_i} are the i^{th} predictor in $\mathbf{x}$ and $\mathbf{x}_q$, respectively. A *distance factor* w_i, is used to multiply the i^{th} predictor to represent how important a predictor is (the greater the distance factor, the more the important).

Using distance as a criterion, it is better to give greater weight to observations that are close to $\mathbf{x}_q$ and less weight to those that are remote. To assign the weight to an observation, a weighting (kernel) function is necessary. The Gaussian kernel is used as the weighting function and is given by

$$K(d) = e^{-d^2}. \tag{6}$$

Using this, $\hat{y}$ is now the weighted average of the execution times of k nearest neighbours and is given by

$$\hat{y} = \frac{\sum_{j=1}^{k} y_j K(d(\mathbf{x}_j, \mathbf{x}_q))}{\sum_{j=1}^{k} K(d(\mathbf{x}_j, \mathbf{x}_q))}. \tag{7}$$

4.2 Learning Process

It can be seen that the predictors, the number of neighbours, and the distance factors need to be defined for the *knn* in the estimating process. The learning process explained here is for specifying these parameters. The process consists of two steps: *preprocessing* and *parameter-deriving*.

Preprocessing: Let $\mathbf{p}$ represent a predictor type, and $\mathbf{p}^\top = (x_1\ x_2\ ...\ x_n)$. Hence, $\mathbf{X}$ in (2) can now be rewritten as

$$\mathbf{X} = \left[\, \mathbf{p}_1\ \mathbf{p}_2\ ...\ \mathbf{p}_p \,\right] = \begin{bmatrix} x_{11} & x_{12} & ... & x_{1p} \\ x_{21} & x_{22} & ... & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & ... & x_{np} \end{bmatrix}. \tag{8}$$

The first step is to transform the elements in each $\mathbf{p}$ such that their values range from zero to one using (9).

$$x_i = \frac{x_i - min_x}{max_x - min_x} \tag{9}$$

where x_i, max_x, and min_x are the i^{th}, the maximum, and the minimum elements in $\mathbf{p}$, respectively.

The second step is to remove the predictors that have no influence on the dependants $\mathbf{y}$. $\mathbf{p}$ has no influence on $\mathbf{y}$ if all elements in $\mathbf{p}$ are identical. This situation usually occurs when the number of observations in the dataset is small.

The final step is to remove *multicollinearity*. Multicollinearity refers to the situation that a pair of predictors are highly correlated, in which one of them can be ignored. The linear relationship between predictors $\mathbf{p}_i$ and $\mathbf{p}_j$ can be measured from their *correlation coefficient* (r_{ij}) [7], which is given by

$$r_{ij} = \frac{n \sum x_i x_j - \sum x_i \sum x_j}{\sqrt{[n \sum x_i^2 - (\sum x_i)^2][n \sum x_j^2 - (\sum x_j)^2]}}. \tag{10}$$

00. min_{cv}, k_{opt}, l_{opt} = **LOOCV** (**X**, **y**)
01. **for** ($l = 1$; $l \leq max_l$; $l = l + 1$)
02. calculate w for each **p**;
03. **for** ($i = 1$; $i \leq n$; $i = i + 1$)
04. $\mathbf{x}_q = \mathbf{x}_i$;
05. $y = \mathbf{y}_i$;
06. $\mathbf{X} = \mathbf{X} - \mathbf{x}_i$;
07. $\mathbf{y} = \mathbf{y} - \mathbf{y}_i$;
08. **for** ($j = 1$; $j \leq n - 1$; $j = j + 1$)
09. $D[j][l] = d(\mathbf{x_j}, \mathbf{x_q})$;
10. sort D by distance;
11. **for** ($k = 1$; $k \leq max_k$; $k = k + 1$)
12. $\hat{y} = knn$ from k neighbours based on the distances in D;
13. $E[i][k][l] = \frac{|\hat{y}-y|}{y} \cdot 100$;
14. $\mathbf{X} = \mathbf{X} + \mathbf{x}_q$;
15. $\mathbf{y} = \mathbf{y} + y$;
16. min_{cv} = minimum cv in E;
17. k_{opt}, l_{opt} = k^{th} and l^{th} indices that minimum cv is found;

Fig. 1. Leave-one-out cross validation algorithm

In our study, if $|r_{ij}|$ is greater than 0.90 then a multicollinearity exists between them.

Parameter-deriving: In this step, the predictors are related to the dependants in the dataset as to derive the actual predictors ($\mathbf{X}_{act}$ – the predictors that will be used in the estimating process), and the optimal number of neighbours (k_{opt}), and distance factors ($\mathbf{w}_{opt}$). The main technique to accomplish this is *leave-one-out cross validation (LOOCV)* [5]. The algorithm is shown in Fig.1, which is to leave the i^{th} observation out of the dataset, and use the other observations to estimate the left out observation. The objective is to find **X**, k, and **w** that minimise the cross validation (cv) function

$$cv(\mathbf{X}, k, \mathbf{w}) = \frac{\sum_{i=1}^{n} \%e_i}{n} \tag{11}$$

where $\%e_i$ is the percentage relative residual error of the i^{th} observation when it is left out and estimated by using **X**, k, and **w**.

The distance factor for each predictor in the algorithm is derived from the statistical technique *Spearman's rank correlation coefficient* (r_s) [7], and the *level of importance*.

r_s is used to measure the strength of association between two sets of data, assuming that the underlying relationship is unknown. r_s is given by

$$r_s = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n^3 - n} \tag{12}$$

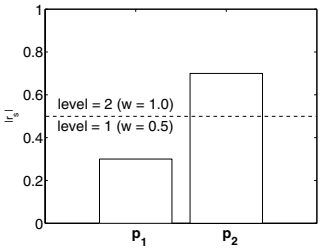

Fig. 2. $|r_s|_1$ and $|r_s|_2$ are 0.3 and 0.7, respectively. Since $l = 2$, $m_1 = 1$ and $m_2 = 2$, and w_1 and w_2 are 0.5 and 1.0, respectively

where d is the difference in *statistical rank* – the ordinal number of a value in a list arranged in increasing order – of corresponding variables. The values of r_s range from -1 to 1, indicating perfect negative and positive association, respectively. Let $|r_s|_i$ represent $|r_s|$ between the i^{th} predictor and the dependants $\mathbf{y}$, this predictor is in level m if and only if

$$\frac{m-1}{l} < |r_s|_i \leq \frac{m}{l} \tag{13}$$

where l is the number of levels and $m = 1, ..., l$. Given m_i as the level that the i^{th} predictor is in, the distance factor for this predictor, w_i, is given by

$$w_i = \frac{m_i}{l}. \tag{14}$$

Fig.2 shows an example of how to calculate the distance factors. In $LOOCV$, max_l is the predefined maximum number of levels while max_k is the predefined maximum number of neighbours. The possible numbers of neighbours range from 1 to $n-1$.

The execution time of each left out observation is estimated over different ks and ls, and the associated error ($\%e$) is stored in an array E. After all the errors have been stored, the algorithm calculates cv (over different ks and ls), and returns the minimum cv and the k^{th} and l^{th} indices (as k_{opt} and l_{opt}) that lead $LOOCV$ to min_{cv}.

Thus far, the predictors given to $LOOCV$ are the ones after the preprocessing step. However, they are not yet the actual predictors. To derive the actual predictors, the simplest approach is to generate all possible combinations of the predictors, process each to $LOOCV$, and pick the one that yields the least cv.

However, a problem arises when p is large since the complexity grows exponentially with the number of predictors, i.e. $O(2^p)$. To address this problem, the *backward predictor elimination* is augmented into the *parameter-deriving algorithm* (as shown in Fig.3) to seek the actual predictors.

The idea is to drop each predictor one by one to make p new sets of predictors. If all cvs from processing these sets to $LOOCV$ are more than the cv from processing all predictors in $\mathbf{X}$ to $LOOCV$, the predictors in $\mathbf{X}$ are the actual

```
00. X_act, k_opt, l_opt = parameter-deriving (X, y)
01.     min_cv, k_opt, l_opt = LOOCV(X, y);
02.     X_act = X;
03.     for (i = 1; i ≤ p; i = i + 1)
04.         tmp_cv, tmp_k, tmp_l = LOOCV(X − p_i, y);
05.         if (min_cv ≤ tmp_cv)
06.             min_cv = tmp_cv;
07.             X_act = X − p_i;
08.             k_opt = tmp_k;
09.             l_opt = tmp_l;
10.     if (X is X_act)
11.         return;
12.     else
13.         X_act, k_opt, l_opt = parameter-deriving(X_act, y);
```

Fig. 3. The parameter-deriving algorithm

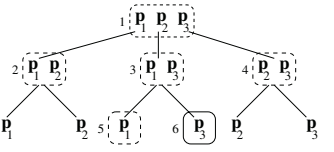

Fig. 4. Only the sets of predictors in the oval are considered in the backward predictor elimination. Here, $\mathbf{X}_{act} = \mathbf{p}_3$ and $cv(\mathbf{X} = [\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3]) \geq cv(\mathbf{X} = [\mathbf{p}_1\mathbf{p}_3]) \geq cv(\mathbf{X} = [\mathbf{p}_3])$

predictors. Otherwise, the *parameter-deriving* is recursively called with the set that yields the least *cv* (see Fig.4 for an example). The returned $\mathbf{X}_{act}$, k_{opt}, and l_{opt} are the actual predictors, the optimal number of neighbours, and levels which is used to derive the optimal distance factors, respectively. The complexity of the *backward predictor elimination* is $O(\frac{p\cdot(p+1)}{2})$ for the worst case.

5 Experiments

In the experiments, three estimation methods, as shown in Table 1, are evaluated. Method-1 is the estimation method proposed in [6]. In method-3, max_k is set to $n-1$, which is the maximum possible number of neighbours, and max_l is set to 10. max_k in method-2 is also set to $n-1$. Notice that method-2 is a specialisation of method-3. The simulator used in the experiments is GMap[1]. All the experiments are conducted on a 2.8 GHz Intel Pentium-4 computer.

[1] GMap is a simulator developed to study the mapping problem, available at http://www.cs.rmit.edu.au/~pphinjar/GMap

Table 1. The experimented estimation methods

method	learning process	estimating process
method-1	fixed $\mathbf{X}$, $k_{opt} = n^{\frac{4}{5}}$, and $l_{opt} = 1$	*knn* explained in [6]
method-2	dynamically chosen $\mathbf{X}$ and k_{opt}, and $l_{opt} = 1$	*knn* explained in Sect.4.1
method-3	dynamically chosen $\mathbf{X}$, k_{opt}, and l_{opt}	*knn* explained in Sect.4.1

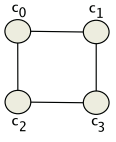

Fig. 5. The topology of the experimented parallel application. Here, c_1 and c_2 are the neighbour tasks of c_0, c_0 and c_3 are the neighbour tasks of c_1, and so on

5.1 Parallel Application

The execution times of the tasks of a four-task square matrix-matrix multiplication are estimated. The topology of the application is Cartesian (as shown in Fig.5). The algorithm is Cannon's algorithm (see [8] for details), in which the tasks perform some different instructions.

As for simplicity, the small number of tasks is experimented on. However, the same method can be applied directly to applications with larger number of tasks and other types of topologies.

5.2 Grid Testbed

The experimental testbed is partially modelled from ThaiGrid testbed [9]. The testbed consists of 7 clusters 96 nodes and 141 processors. It is assumed that the clusters are located in 7 different countries, and all nodes are dedicated. Mapping tasks to unreliable nodes are not considered in the experiments.

The bandwidth among the nodes in the testbed are derived from the network statistics among nodes located in those 7 countries measured during May 2004, available from the PingER project [10].

5.3 Experimental Results

The application is executed on the testbed 100 times by randomly varying the size of the matrices (i.e. 100, 200, ..., 5000). At each run, 10 nodes are first randomly chosen. Then, the nodes to run the tasks are randomly chosen from these 10 nodes. This leads to multiple tasks being executed on the same node.

Assume that task c_0 (in Fig.5) is to be mapped on node v_0 while c_1 and c_2 (which are the neighbour tasks of task c_0) are mapped on nodes v_1 and v_2, respectively. The predictors used to estimate the execution time of task c_0 are the problem size of the application, the performance and *load factor* of nodes v_0, v_1 and v_2, and the bandwidth from node v_0 to nodes v_1 and v_2 and vice versa.

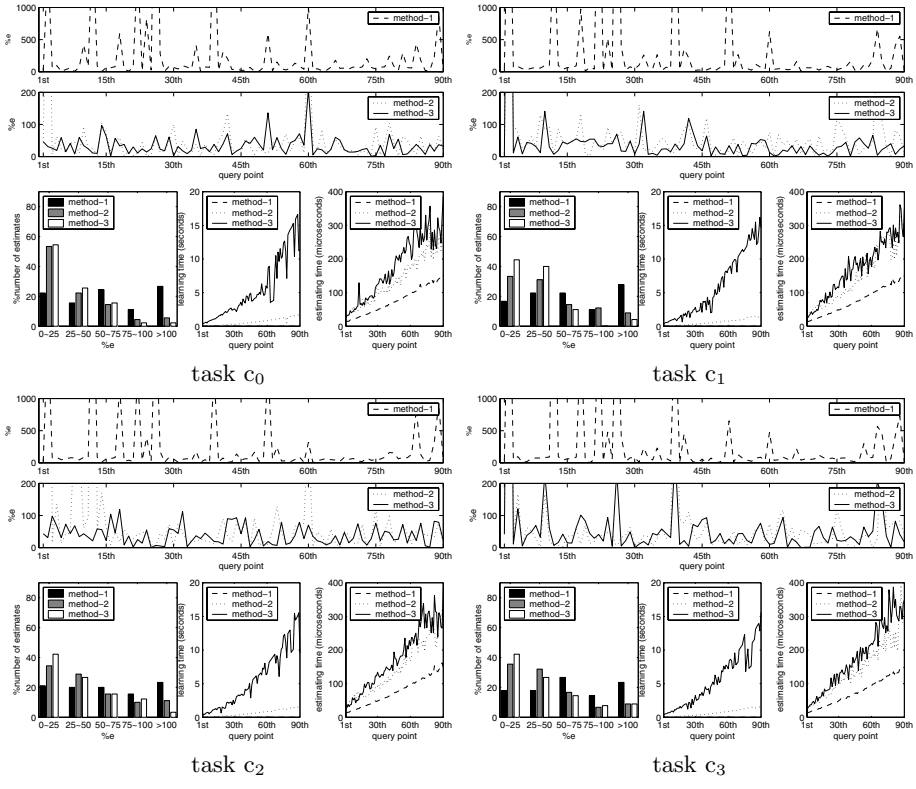

Fig. 6. Estimation evaluation for the application

The load factor of a node is defined as the upper bound of the ratio between the number of tasks to be mapped on that node and the number of processors of the node. For instance, if node v_0 has two processors, its load factor is $\lceil \frac{1}{2} \rceil = 1$.

The estimation starts from 10 observations in the dataset, and the total of 90 query points are estimated. Fig.6 shows the results from estimating the execution time of each task of the application.

Results from the top two sub-figures of each task show that $\%e$ of estimates produced from method-1 are very large when compared to method-2 and method-3. The more the number of observations in the dataset, the lower the $\%e$ of estimates produced from method-2 and method-3.

Results from the bar graphs of each task show that, in terms of estimation accuracy, method-3 outperforms the others. For example, from the results of task c_1, about 43%, 35%, and 18% of the total number of estimates have their $\%e$ less than 25%, which are considered as accurate estimates, when estimated with method-3, method-2, and method-1, respectively. Method-2 performs reasonably well whereas method-1 is the worst. On average, 27% of the total number of estimates produced from method-1 have their $\%e$ greater than 100% whereas around 10% and 6% with method-2 and method-3, respectively.

As expected, the learning time of method-3 is the greatest ($\simeq$ 15 seconds with 99 observations in the dataset) since it needs to perform cross validation to find $\mathbf{X}_{act}$, k_{opt}, and l_{opt}. Method-1 has the fastest learning as only adding the new observation to the dataset needs to be done. However, note that the learning can be done off-line. From the bottom right sub-figure of each task, results show that the estimating times of all methods are performed quite efficiently in the scale of less than 400 microseconds with 99 observations in the dataset.

It can be seen from the results that on average method-3 produces 2.3 times the number of accurate estimates ($\%e < 25\%$) greater than method-1.

6 Conclusions

In this paper, a new method to estimate the execution time of a parallel task on a grid node is proposed. This is to address the problem caused by an unrealistic assumption that the execution time of a task to be mapped on a node can be precisely calculated before the actual execution. The proposed estimation method is based upon past observations of the task executions. The employed estimating technique is a k-nearest-neighbours algorithm (knn). Leave-one-out cross validation technique, a backward predictor elimination, and a statistical technique are used to derive the relevant parameters to be used by knn. In the experiments, the proposed method is compared with the existing method by estimating the execution times of the tasks of a matrix-matrix multiplication developed with Cannon's algorithm (Cartesian topology). Experimental results show that on average the proposed method can produce 2.3 times the number of accurate estimates (with error less than 25%) greater than the existing method.

References

1. Foster, I., Kesselman, C.: The Grid: Blueprint for Future Computing Infrastructure. Morgan Kaufmann (1998)
2. Bokhari, S.: On the mapping problem. IEEE Transaction on Computers **C-30** (1981) 207–214
3. Dail, H., Berman, F., Casanova, H.: A decoupled scheduling approach for grid application development environments. Parallel and Distributed Computing **63** (2003) 505–524
4. Zhang, W., Fang, B., He, H., Zhang, H., Hu, M.: Multisite resource selection and scheduling algorithm on computational grid. In: 18^{th} International Parallel and Distributed Processing Symposium (IPDPS). (2004) 105–115
5. Atkeson, C., Schaal, S., Moore, A.: Locally weighted learning. AI Reviews **11** (1997) 11–73
6. Iverson, M., Ozguner, F., Potter, L.: Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. IEEE Transaction on Computers **48** (1999) 35–44

7. Walpole, R.: Introduction to Statistics. 3^{rd} edn. Collier Macmillan (1982)
8. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. 2^{nd} edn. Pearson Education Limited, Essex, England (2003)
9. Varavithya, V., Uthayopas, P.: ThaiGrid: Architecture and overview. NECTEC Technical Journal **2** (2000)
10. The PingER Project, http://www-iepm.slac.stanford.edu/pinger/ (2004)