Statistical Prediction of Task Execution Times Through Analytic Benchmarking for Scheduling in a Heterogeneous Environment

Michael A. Iverson, Füsun Özgüner, and Lee C. Potter
Department of Electrical Engineering
The Ohio State University
2015 Neil Avenue, Columbus, OH 43210
{iversonm,ozguner,potter}@ee.eng.ohio-state.edu

Abstract

In this paper, a method for estimating task execution times is presented, in order to facilitate dynamic scheduling in a heterogeneous metacomputing environment. Execution time is treated as a random variable and is statistically estimated from past observations. This method predicts the execution time as a function of several parameters of the input data, and does not require any direct information about the algorithms used by the tasks or the architecture of the machines. Techniques based upon the concept of analytic benchmarking/code profiling [7] are used to accurately determine the performance differences between machines, allowing observations to be shared between machines. Experimental results using real data are presented.

Keywords: *Heterogeneous Distributed Computing, Execution Time Estimation, Nonparametric Regression, Analytic Benchmarking, Distance Matrices.*

1 Introduction

Heterogeneous metacomputing is a type of parallel computing, where a large, distributed network of heterogeneous machines is used as a single computational entity. Applications executing in this environment consist of a set of coarse-grained, precedence-constrained tasks, where the precedence structure can be represented using a directed acyclic graph (DAG). The performance of an application in this environment is largely determined by the manner in which these tasks are assigned to the machines; the construction of such and assignment is called the matching and scheduling problem. Matching and scheduling algorithms need to know the execution time of each task on each machine to perform well, and most matching and scheduling algorithms for DAGs in the literature assume that the execution time of a given task is a known quantity. However, the execution time of a task on a given machine depends upon many factors, including the problem size and the input data, and is not trivial to determine *a priori*. In a heterogeneous environment, the wide variety of machine architectures further complicates the process of determining the execution time, since the execution time is also machine dependent. Methods are clearly needed which can accurately predict the execution time of a task on a variety of machines as a function of the features of the data set. This problem is called the execution time estimation problem.

In the literature, there are three major classes of solutions to the execution time estimation problem: code analysis [18], analytic benchmarking/code profiling [7, 11, 12, 16, 20, 22, 23] and statistical prediction [3, 10, 13]. In code analysis, an execution time estimate is found through analysis of the source code of the task. A given code analysis technique is typically limited to a specific code type or a limited class of architectures. Thus, these methods are not very applicable to a broad definition of heterogeneous computing, and will not be examined here.

A class of methods which are more useful in a heterogeneous metacomputing environment is analytic benchmarking/code profiling. Analytic benchmarking/code profiling was first presented by Freund [7], and has been extended by Pease et al. [16], Yang et al. [22, 23], Khokhar et al. [11, 12], and Siegel [20]. Analytic benchmarking defines a number of primitive code types. On each machine, benchmarks are obtained which determine the performance of the machine for each code type. Code profiling attempts to determine the composition of a task, in terms of the same code types. The analytic benchmarking data and the code profiling data are then combined to produce an execution time estimate. Analytic benchmarking/code profiling has two disadvantages. First, it lacks a proven mechanism for producing an execution time estimate from the benchmarking and profiling data over a wide range of algorithms and architectures. Second, it cannot easily compensate for variations in the input data set. However, analytic benchmarking is a powerful comparative tool in that it can determine the relative performance differences between machines.

The third class of execution time estimation algorithms, statistical prediction algorithms, make predictions using past observations. A set of past observations is kept for each machine, which are used to make new execution time predictions. The matching and scheduling algorithm uses these predictions (and other information) to choose a machine to execute the task. While the task executes on the chosen machine, the execution time is measured, and this measurement is subsequently added to the set of previous observations. Thus, as the number of observations increases, the estimates produced by a statistical algorithm will improve. Statistical prediction algorithms have been presented by Iverson et al. [10], Kidd et al. [13], and Devarakonda and Iyer [3]. Statistical methods have the advantages that they are able to compensate for parameters of the input data (such as the problem size) and do not need any direct knowledge of the internal design of the algorithm or the machine. However, statistical techniques lack an intrinsic method of sharing observations between machines. By allowing observations to be shared between machines, the execution time estimate on a machine with few observations can be improved by using observations from machines with similar performance characteristics.

Given the advantages and disadvantages of both analytic benchmarking/code profiling and statistical methods, this paper presents a hybrid method, which uses analytic benchmarking techniques to create a unified set of observations describing both the input data features and the machine capabilities. This unified observation space is then used by a statistical method to produce execution time estimates. In this paper, as in much of the DAG scheduling literature, each task is assumed to have exclusive use of the machine on which it executes. Thus, the execution time of a task is not a function of the other tasks in the system, and is only a function of the machine capabilities and input data. This method models the execution time of a task as a random variable, allowing the matching and scheduling algorithm to consider the uncertainty present in the execution time estimate. (Several papers have discussed the idea of scheduling with random quantities, including King [14], Tan and Siegel [21], Armstrong [2], Li et al. [15] and Hou and Shin [9].)

The remainder of this paper is organized as follows. First, the stochastic model of the execution time of a task is presented in Section 2. Section 3 presents the prediction algorithm which uses this model. Section 4 presents the results of simulations using real data, and conclusions are offered in Section 5.

2 Modeling the Execution Time as a Random Variable

The execution time of a task on a given machine largely depends on the size and properties of the input data set. For example, the execution time of many matrix algorithms depends upon the size of the matrix. Furthermore, if the matrix algorithm was iterative in nature, the execution time may also depend upon the condition of the matrix and the desired precision of the results. In principle, it is possible to quantify these properties of the input data set as a vector of numeric parameters $X = [x^1 x^2 \cdots x^q]$. Thus, the execution time of the task can be modeled as a function $t = m(X)$ of this parameter vector. However, in many instances, it is not computationally practical to determine all q parameters. Therefore, it is assumed that only a limited number $p \leq q$ of these parameters will be explicitly modeled. Thus, the parameter vector $X = [x^1 x^2 \cdots x^p]$ will be used to model the execution time of the task. However, the presence of unmodeled factors will cause a certain amount of error to be present in an estimate of the execution time. To compensate for this error, the execution time of a task is modeled as a random variable t. This random quantity can be represented as:

$$t = m(X) + \epsilon, \qquad (1)$$

where $m(X)$ is deterministic, and ϵ is purely stochastic. In this equation, ϵ represents the unmodeled factors affecting the execution time, while $m(X)$ represents the modeled factors, and therefore depends upon a p-dimensional vector of parameters $X = [x^1 x^2 \cdots x^p]$. In essence, $m(X)$ represents the mean of t given X, while ϵ represents the zero-mean random error present in the estimate.

While the unmodeled factors which affect ϵ are unknown, it is possible to determine their effect upon the execution time indirectly by estimating properties of the random variable ϵ. Additionally, while ϵ does not directly depend upon the parameter vector $X = [x^1 x^2 \cdots x^p]$, in practice, ϵ does display some dependence on the modeled parameters, due to the fact that the modeled and unmodeled parameters may not be statistically independent. Thus, some degree of correlation may exist between these sets.

Given this model, the goal of the execution time estimation problem is to obtain estimates of $m(X)$ and ϵ

for some given parameter vector X. Before presenting the specific details of how these values are estimated, examples of how two real algorithms behave under this model will be presented to illustrate the concepts presented here. The first example shows an algorithm where the set of unmodeled parameters has a very small effect upon the execution time (i.e. ϵ is small). Figure 1 shows the execution time of the Cholesky matrix decomposition algorithm for various problem sizes on a single machine. The relative continuity of this curve shows that ϵ has a very small impact on the execution time.

The opposite case is illustrated in Figure 2. This algorithm attempts to determine if a given number is prime through the process of trial division. The execution time of this algorithm, for a given number n, is essentially a function of the smallest prime factor of the number n. However, it is not practical to compute the smallest prime factor of the number n, since the computational cost of this problem is equivalent to determining if the number is prime. However, it is possible to use the magnitude of the number as a parameter, since this value bounds the magnitude of the smallest prime factor. Thus there is a loose correlation between the magnitude of the number and the execution time, which can be seen in the figure. Thus, even in this extreme example, it is still possible to obtain some information about the execution time of the task which can be used by the matching and scheduling algorithm. In the next section, the techniques used to estimate the values of $m(X)$ and ϵ will be presented.

3 Execution Time Estimation Algorithm

The algorithm developed in this paper is presented in two sections. Section 3.1 poses the execution time estimation problem as a regression problem, and presents a *k-Nearest Neighbor (k-NN)* regression algorithm to compute estimates from a set of previous observations. For clarity of presentation, the regression algorithm is described using only the vector of parameters describing the task input data. While this algorithm can compute the execution time of a task on a single machine as a function the input data set, it lacks an intrinsic ability to share observations between dissimilar machines. To eliminate this restriction, the regression vector is augmented in Section 3.2 to include a parameterization of different machines. Thus, the execution time may be estimated using previous observations as a function of both machine type and task characteristics.

3.1 Nonparametric Regression

Given that the execution time of a task is modeled as a random variable (as in equation 1), the goal of this paper is to present methods to obtain estimates $\hat{m}(X)$ and $\hat{\epsilon}$ of $m(X)$ and ϵ for a given a parameter vector X which characterizes the input data set. This will be accomplished through the use of a set of n previous observations of the execution time $\{(t_i, X_i)\}_{i=1}^{n}$, where t_i is an observed execution time for the parameter vector X_i. The parameter vectors X_i of the n previous observations are samples in the parameter space $\mathbb{R}_p$ (p-dimensional real vectors). In statistics, this problem is called a *regression problem*. Note that, as presented in this section, each machine requires a separate set of observations. This restriction will be relaxed in Section 3.2.

There are a variety of different techniques to solve regression problems, which can be divided into two classes: *parametric* techniques and *nonparametric* techniques. In general, parametric techniques require knowledge of the functional form of $m(X)$ and ϵ. Since, in this paper, it is difficult to make any assumptions about the functional form of the model without specific knowledge of the task and the machine in question, parametric techniques are not well suited to this problem [10]. Nonparametric regression techniques (also called nonparametric estimators or smoothing techniques) are considered to be *data driven*, since the estimate depends only upon the set of previous observations, and not on any assumptions about $m(X)$ or ϵ. Therefore, nonparametric techniques will be used in this paper.

All nonparametric regression techniques compute $\hat{m}(X)$ using a variation of the equation

$$\hat{m}(X) = \frac{1}{n}\sum_{i=1}^{n} W_i(X) t_i \tag{2}$$

where $W_i(X)$ is a weighting function, or kernel [8]. Observe that, for any given vector X, $\hat{m}(X)$ is a weighted average of the execution time values, t_i, of the n previous observations. The weight function $W_i(X)$ typically assigns higher weights to observations close to the parameter X, and lower weights to observations farther away from X. This is illustrated in Figure 3 for a scalar parameter $x = A$. In practice, many nonparametric regression techniques only include in the average points within some neighborhood of the parameter X, making the estimate $\hat{m}(X)$ a local average of the observations near the parameter vector X.

A similar technique can be used to determine the properties of ϵ. As mentioned above, ϵ is a zero-mean random variable, which can have an arbitrary probability density function (pdf). This arbitrary nature of ϵ

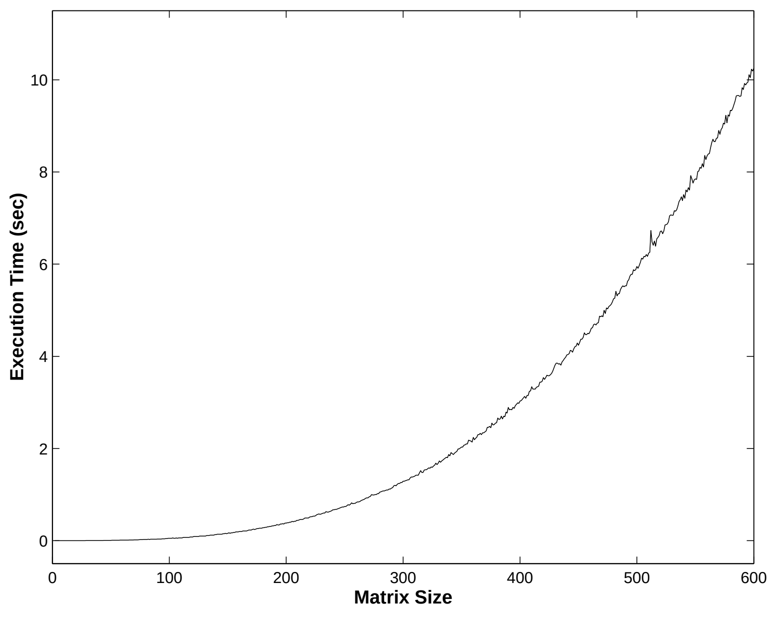

Figure 1. Execution time of the Cholesky Decomposition Algorithm.

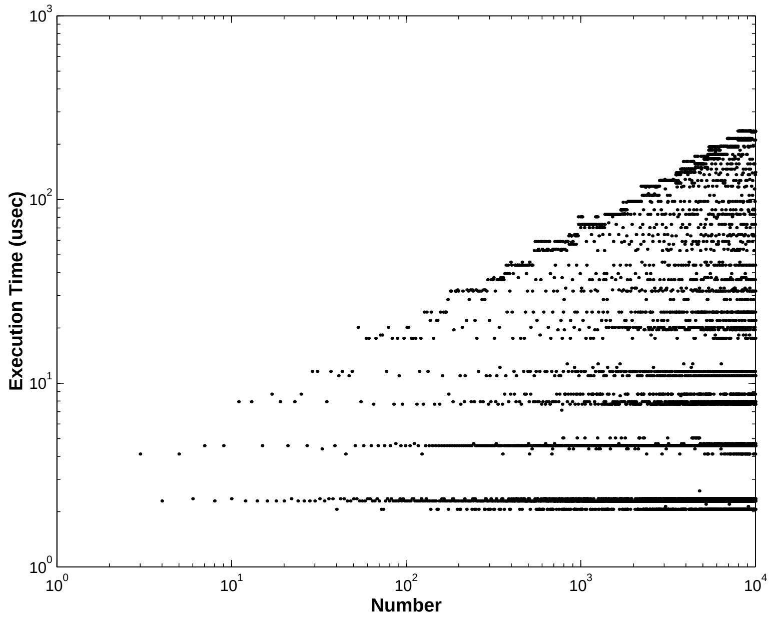

Figure 2. Execution time of the Trial Division Algorithm.

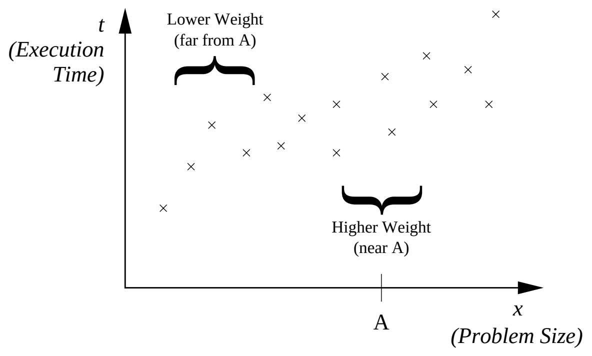

Figure 3. Assigning Weights to Observations.

makes the estimation process difficult. To maintain simplicity, only an estimate of the variance σ^2 of ϵ will be computed in this paper. Given an estimate $\hat{m}(X)$, $\hat{\sigma}^2$ can be computed to be

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} W_i(X)(t_i - \hat{m}(X))^2. \tag{3}$$

In this paper, the *k-Nearest Neighbor (k-NN)* algorithm is used (although, in principle, any nonparametric technique could be adapted). In *k-NN* smoothing, the estimate $\hat{m}(X)$ is constructed from the k observations with parameter vectors closest to the parameter vector X. With regard to the execution time estimation problem, there are two primary advantages of *k-NN* smoothing. First, since the estimate is always constructed from an average of k points, the method can easily adapt to sparse or dense regions in the observations. Second, the method can be implemented in a computationally efficient manner. The computational complexity of the method is $O(n(p + \log n))$, where p is the dimensionality of the parameter vector, and n is the number of past observations.

While conceptually simple, there are many factors to consider when using the *k-NN* algorithm. For example, an important issue in *k-NN* smoothing is the choice of the value k. If too many observations are included in the average, the bias $E\{\hat{m}(X) - m(X)\}$ will be too large, and the details of $\hat{m}(X)$ will be lost. On the other hand, if too few observations are averaged, the variance $E\{(\hat{m}(X) - m(X))^2\}$ of $\hat{m}(X)$ will be too large, resulting in a curve which is "noisy." Choosing k to obtain a balance between these two extremes is known as the *bias-variance tradeoff*, and is present in all nonparametric regression techniques. In general, k should grow in proportion to n such that $k \to \infty$ and $(k/n) \to 0$ as $n \to \infty$ [4].

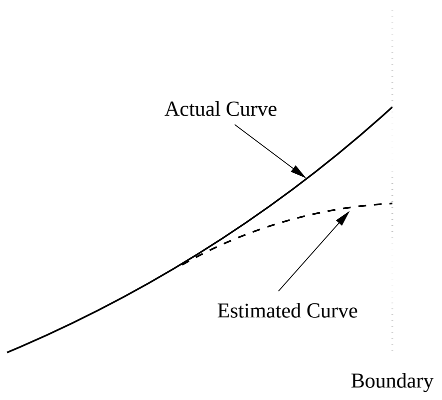

Figure 4. Effects of estimates at the boundary.

Another factor in the design of a nonparametric regression algorithm is the ability to tolerate erroneous data points in the set of observations. (This type of estimator is said to be a robust estimator.) These points, called outliers, do not conform to the model described in equation 1. The technique used to make the estimator robust is called *L-Smoothing* [8], where a fixed percentage of the observations with the largest and smallest values of t_i are eliminated from the local average.

A final issue encountered when using nonparametric regression techniques is the behavior of the estimates near the boundaries of the set of observations (i.e., no observations lie beyond the boundary). As the parameter value X approaches a boundary, the local average becomes biased, since more observation points will be on one side of point X than the other. The one-dimensional case is illustrated in Figure 4, where the estimated function $\hat{m}(x)$ will become biased near the boundary [6, 8]. To ensure accurate estimates near the boundary, a nonparametric regression technique needs to be able to compensate for this effect. A formal definition of the *k-NN* algorithm, including solutions to these issues, is presented in Appendix A.

3.2 Parameterizing Machine Performance

If the *k-NN* algorithm is used as presented above, a separate set of observations must be maintained for each machine. This condition is caused by the lack of a mechanism to translate performance differences of the machines into numeric parameters which can be incorporated into the execution time model presented in equation 1. There are two compelling reasons why it is desirable to eliminate this restriction and to form a unified set of observations. First, separate sets makes the process of adding new machines (or applications) to the network difficult, since a few initial observations are required in each set for the algorithm to function effectively. Thus, widespread benchmarking is required to obtain such an initial set for each machine. Second, a starvation problem can exist, where a machine with few observations will tend to produce poor execution time estimates. If these poor estimates are larger than the actual execution time, it is unlikely that the scheduling algorithm will choose to execute the task on that machine. Thus, the machine will not get any new observations from which estimates could be improved.

To jointly utilize observed execution times across all machines, a method is needed to characterize the available machines using numeric parameters which can then be included in the parameter vector. This needs to be accomplished such that the distance between any two machines, in terms of their parameterization, is a rough indication of the similarity of the performance of those machines. This process can be accomplished through the use of analytic benchmarking [7].

Analytic benchmarking characterizes the performance of a machine using a series of benchmarks. In theory, each of these benchmarks should correspond to a primitive code type; code types form a basis which can exactly characterize the performance of a machine for any task. Because the construction of an ideal set of benchmarks is difficult (if not impossible), a rigorous definition of primitive code types is avoided, and instead it is assumed that a reasonable set of r benchmarks is available to approximate the performance differences between machines. These benchmarks can be used to span $\mathbb{R}_r$ where each axis corresponds to the results of one of the benchmarks, either in terms of the time required to execute the benchmark or the rate at which the machine performs iterations of the benchmark. This space will be called the *machine space*. Thus, a machine i can be represented by a point $B_i = [b_i^1 b_i^2 \cdots b_i^r]$ in the machine space, where b_i^j is the result for benchmark j on machine i. B_i will be called the *benchmark vector* for machine i.

The points in this space can then be used to construct an augmented parameter vector, which can be used with the method presented in Section 3.1. Given the r-dimensional machine space defined above, and a task with an execution time that is a function of a vector of parameters $X = [x^1 x^2 \cdots x^p]$, an augmented parameter vector in the unified parameter space can be constructed by concatenating the benchmark vector B_i and the parameter vector X, creating an $(r+p)$-dimensional parameter vector. Thus, the execution time observation of the task on machine i is associated with the augmented parameter vector $Y = [b_i^1 b_i^2 \cdots b_i^r x^1 x^2 \cdots x^p]$.

While it is desirable to use a large number of benchmarks to accurately characterize machine performance, the dimensionality of the resulting parameter vector is large. An increase in the dimensionality of the parameter vector increases the computational cost of the *k-NN* smoothing algorithm, and decreases the rate at which $\hat{m}(X)$ converges towards the true curve $m(X)$ (due to the requirement of maintaining the bias-variance tradeoff). Therefore, it is desirable to minimize the dimensionality of the benchmark vector B_i, while maximizing the amount of information it contains. Since the distance relationship between the points contains the information on the relative performance differences between the machines, this goal can be accomplished by reducing the number of dimensions in the machine space, while attempting to preserve the distance relationship between the points. Potter and Chiang [17] present an algorithm that can be used to embed the benchmark vectors B_i in an s-dimensional subspace, where $s < r$. The details of this algorithm are given in Appendix B.

This embedding creates a new machine parameter space of smaller dimension, called the *reduced machine space*. This space is combined with the parameters characterizing the input data to yield a unified parameter space, as outlined above.

3.3 Summary of the Complete Algorithm

In this section, the *k-NN* regression technique of Section 3.1 is applied to the augmented parameter vector method described in Section 3.2, to create the completed execution time estimation algorithm. The algorithm begins with a set of n previous observations of the execution time $\{(t_i, Y_i)\}_{i=1}^n$, where $Y_i = [b_{j_i}^1 b_{j_i}^2 \cdots b_{j_i}^s x_i^1 x_i^2 \cdots x_i^p]$ and j_i is the machine from which observation Y_i was obtained. Given this set, a parameter vector $X = [x^1 x^2 \cdots x^p]$ describing the input data set, and the reduced machine space $\mathbb{R}_s$ containing a point $B_j = [b_j^1 b_j^2 \cdots b_j^s]$ for each machine j, pseudocode for this algorithm can be constructed as follows.

```
Execution Time Estimation Algorithm:
 begin
     For each candidate machine $j$ with
        benchmark vector $B_j = [b_j^1 b_j^2 \cdots b_j^s]$
     begin
          Compute $\hat{m}(Y_j)$ and $\hat{\sigma}^2$, where
              $Y_j = [b_j^1 b_j^2 \cdots b_j^s x^1 x^2 \cdots x^p]$.
     end
     Give estimates computed above to
        matching and scheduling algorithm.
        The algorithm will return a
        machine $j$ chosen to execute the task.
     Execute task on machine $j$, and
        measure the execution time $t_{n+1}$.
     Add observation $(t_{n+1}, Y_{n+1})$ to the
```

```
            set of previous observations, where
            $Y_{n+1} = [b_j^1 b_j^2 \cdots b_j^s x^1 x^2 \cdots x^p]$.
        $n = n + 1$.
    end
```

It can be seen that every time a given task is run on a machine in the system, a new observation is added to the set of previous observations. Thus, the quality of the predictions improves with time.

One issue which has not been addressed is the source of an initial set of observations. Since the execution time estimate is a function of the set of previous observations, at least one initial observation is required when a new task/application is introduced into the system. Thus, the task must be executed on a few selected machines in order to obtain a few initial estimates. These values are easily obtained during the development, testing, and debugging of the application.

4 Evaluation

To evaluate the performance of the methods presented in this paper, two sets of experiments were performed using real data. A machine space was constructed using the 10 benchmarks from the Byte benchmark suite [1]. These benchmarks consist of a variety of integer and floating point benchmarks. These benchmarks were executed on 16 different machines running different flavors of UNIX. The results from these benchmarks were normalized, giving all benchmarks equal weight. This 10-dimensional space was then reduced to a 3-dimensional space using the algorithm outlined in Section 3.2. The normalized result of this embedding is pictured in Figure 5. In this 16 machine environment, experiments were performed using real data obtained from the Cholesky decomposition and trial division algorithms presented in Section 2.

The first set of experiments emulates the situation where a new machine is added to the network. This experiment compares the performance of the execution time estimation algorithm when observations can and cannot be shared between machines. In this experiment, the execution time of the Cholesky decomposition algorithm was estimated on a single machine for 5 different matrix sizes. The number of observations on the machine was varied between 1 and 50. The average prediction error is compared for three different simulations. The results of these three simulations are given in Figure 6, which show how the average error in $\hat{m}(X)$ changes as the number of observations increases.

The first simulation shows the performance of the method when observations are not shared between machines. The execution time was computed to be a function of a scalar parameter: the size of the matrix. The second and third simulations show the performance of the algorithm when it is able to use observations on other machines, taking advantage of an additional 350 observations uniformly distributed across the remaining 15 machines. In the second simulation, the 3-dimensional reduced machine space is used. Thus, by using the size of the input data set and the 3-dimensional embedding as parameters, the execution time was a function of a 4-dimensional parameter vector. The third simulation used the full 10-dimensional machine space in the multidimensional algorithm. In this case, the execution time was a function of an 11-dimensional parameter vector. In all of these simulations, the value of k used to make an estimate of the execution time on a given machine j was defined to be $\lceil (0.1)(n_j)^{4/5} \rceil$, where n_j is the number of observations for machine j, which satisfies the bias-variance tradeoff requirements outlined in Section 3.1.

As shown in Figure 6, the ability to share observations between machines gives the algorithms used in the second and third simulations a significant performance advantage over the first algorithm when there are few observations from which to compute an estimate. The latter simulations produce prediction errors around 50%, versus errors around 500% using the first method. The performance difference between the two observations-sharing methods is negligible. For larger numbers of observations, all three methods perform equally, with prediction errors around 15% using a few 10's of observations. To compare the computational costs of these three algorithms, Figure 7 shows the measured CPU time of each algorithm as a function of n (the size of the entire observations set). This figure shows that using a reduced parameter space is considerably more efficient than using the full parameter space, making the reduced parameter space approach the best choice when considering both accuracy and efficiency. The algorithm was implemented using MATLAB's scripting language, and the CPU time was measured on an HP B180L workstation. While the measured CPU times are small, it is likely that a more efficient implementation could result from a conventional programming language.

In the second set of experiments, estimates were computed using the trial division algorithm on a single machine (no observation sharing was done in this experiment). As shown in Figure 2, the execution time of this algorithm is very loosely correlated to the parameter vector X. Thus, in this extreme example, the error in the execution time estimates will always be large, regardless of the number of past observations. However, these experiments demonstrate the utility of estimating the sample variance of the execution time, and using this value to bound the execution time. Two different simulations were performed, where $\hat{m}(X)$ and $\hat{\sigma}^2$ were

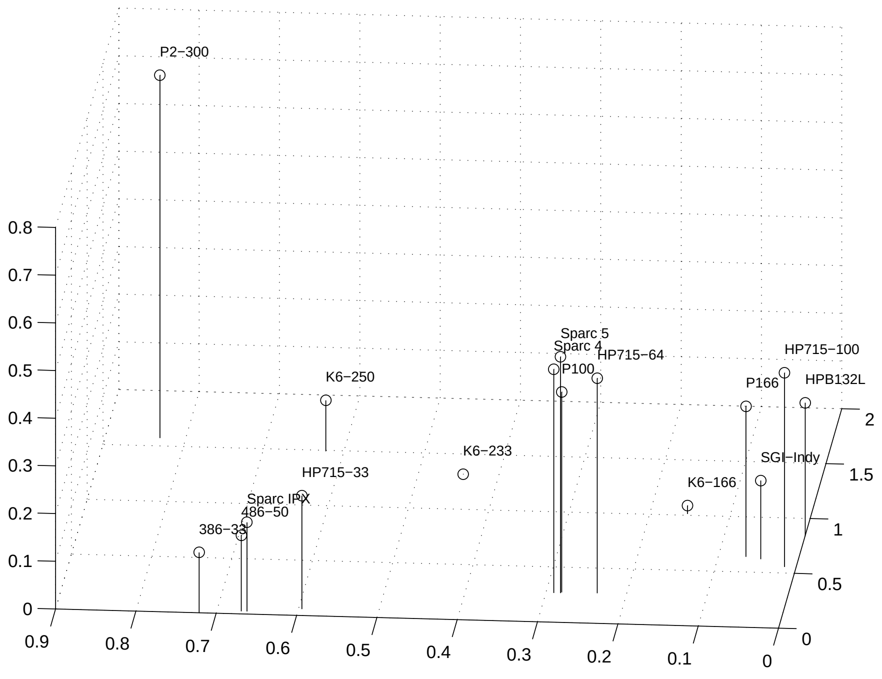

Figure 5. 3-Dimensional Distance Embedding of Machine Space.

computed for 50 evenly-spaced problem sizes using 20 and 200 observations, respectively. These results are presented in Figures 8 and 9, which show $\hat{m}(X)$ and $\hat{m}(X)+3\hat{\sigma}$. It can be seen in the figures that $\hat{m}(X)+3\hat{\sigma}$ does act as a reasonable upper bound on the execution time. Thus, the combination of both the estimated execution time and the uncertainty, σ, could be used by an appropriate matching and scheduling algorithm to make good scheduling decisions, despite the fact that the parameter vector may not contain sufficient information to obtain an accurate execution time estimate. Furthermore, Figure 8 illustrates how the execution time estimate conforms to the set of past observations, where the estimated curves form a distinct “hump” around the two observations with large execution times.

5 Conclusions

This paper presents a statistical execution time estimation algorithm for use in a heterogeneous distributed computing environment. This algorithm treats the execution time as a random variable, and makes predictions using past observations of the execution time. These estimates compensate for the properties of the input data set and the machine type, without requiring any direct knowledge of the internal operation of the task or machine. The random model allows the algorithm to determine the probable execution time of the task, even in situations where the estimate has a large amount of uncertainty. This algorithm is unique in that it is able to use observations from dissimilar machines when making predictions, through the process of analytic benchmarking. This ability greatly simplifies the process of adding new machines to the system. Furthermore, an algorithm is presented which can be used to reduce the number of parameters introduced by the analytic benchmarking process. As shown in Figure 6, experimental results indicate that this method can make accurate execution time estimates over a wide range of parameter values using a few dozen past observations.

A Appendix: k-NN Regression

The *k-NN* regression algorithm, and other statistical techniques shown in this section, are derived from the methods surveyed in the books by Härdle [8] and Eubank [6], unless noted otherwise. Given a parameter vector $X = [x^1 x^2 \cdots x^p]$ and a set of n previous observations $\{(t_i, X_i)\}_{i=1}^{n}$, the *k-NN* method can be formally defined as follows. Let

$$J_X = \{i : X_i \text{ is one of the } k \text{ nearest neighbors of } X\}. \quad (4)$$

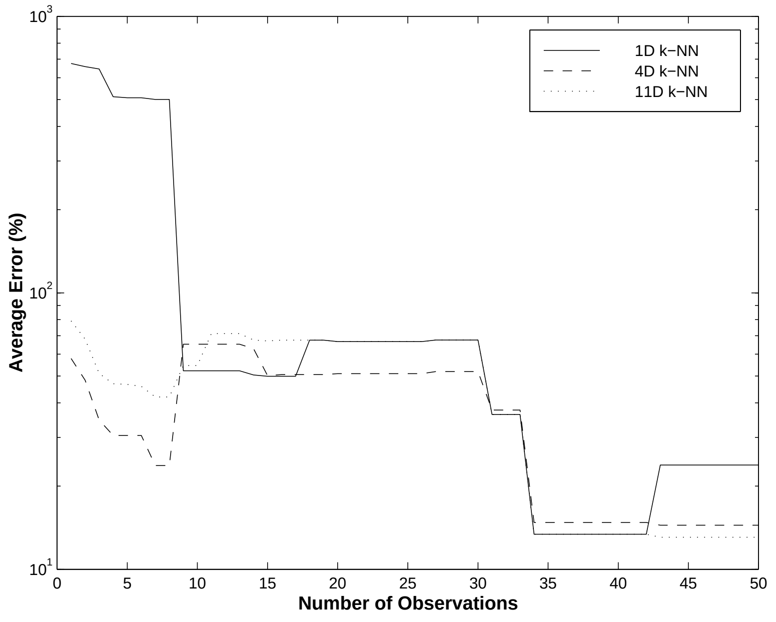

Figure 6. Average Prediction Error vs. Number of Observations.

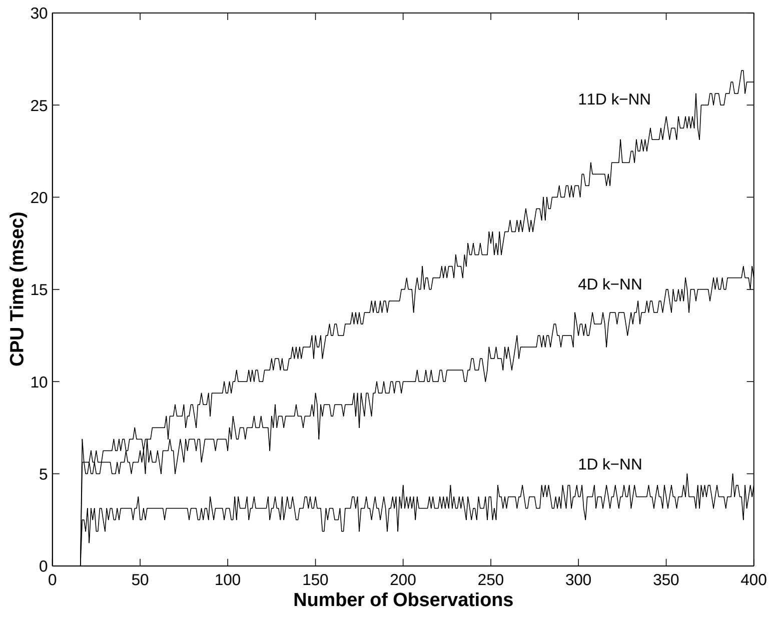

Figure 7. Computation Costs of Prediction Algorithm.

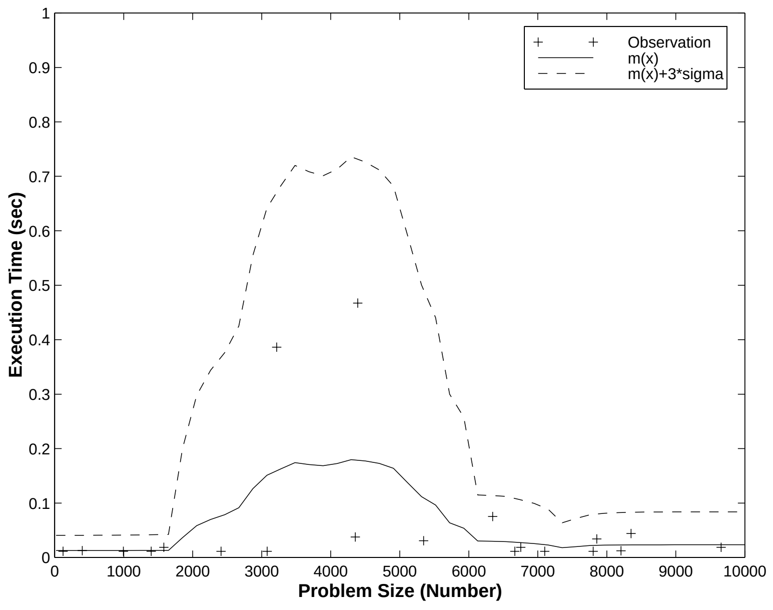

Figure 8. $\hat{m}(X)$ **and** $\hat{\sigma}^2$ **with 20 observations.**

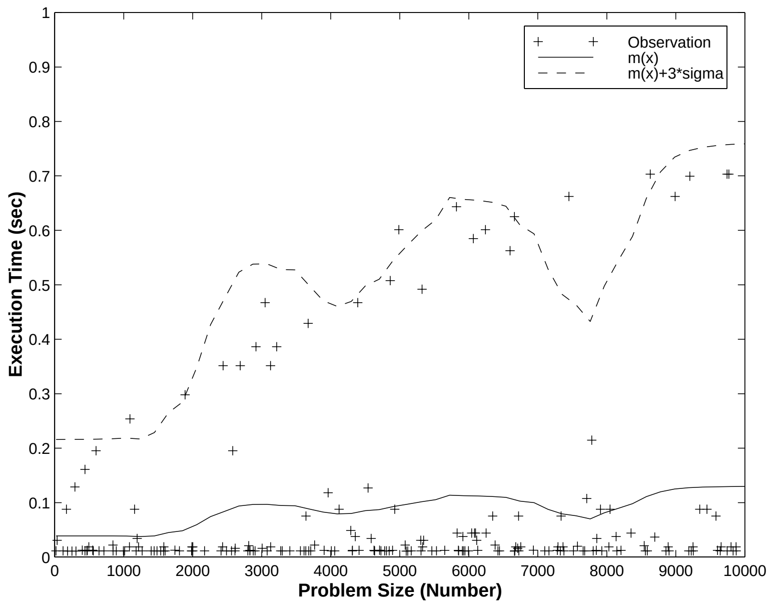

Figure 9. $\hat{m}(X)$ **and** $\hat{\sigma}^2$ **with 200 observations.**

The *k-NN* method uses the elements in J_X to form a weighted average of observations, similar to equation 2.

L-Smoothing [8] is used to make the weighted average robust (i.e., able to tolerate outliers in the data set). A fixed percentage of the observations with the largest and smallest t values are not included in the local average. *L-Smoothing* can be implemented by sorting the observations $\{(X_i, t_i)\} \in J_X$ by t_i, then computing $\hat{m}(X)$ to be

$$\hat{m}(X) = \frac{1}{k - 2\lfloor\gamma k\rfloor} \sum_{i=\lfloor\gamma k\rfloor}^{k-\lfloor\gamma k\rfloor} W_i(X) t_i \qquad (5)$$

and $\hat{\sigma}^2$ to be

$$\hat{\sigma}^2 = \frac{1}{k - 2\lfloor\gamma k\rfloor} \sum_{i=\lfloor\gamma k\rfloor}^{k-\lfloor\gamma k\rfloor} W_i(X)(t_i - \hat{m}(X))^2. \qquad (6)$$

The value of γ, where $0 \leq \gamma < 0.5$, controls the percentage of observations excluded from the average.

Next, the weight $W_i(X)$ assigned to each observation will be defined. First, consider a weighting function (also called a kernel function) for a single dimension j. It is desirable for this weighting function to give higher weights to the observations closer to the parameter value x^j. A weighting function which satisfies this condition is the Epanechnikov Kernel $K(u)$ [5, 6], where

$$K(u) = \frac{3}{4}(1 - u^2) \qquad (7)$$

and $|u| < 1$. This function is symmetric, with a maximum at $u = 0$, and can be shifted, scaled and normalized such that the point $u = 0$ corresponds to the parameter value x^j. In this way, the function gives higher weights to observations near the parameter value, and lower weights to more distant observations. To formally present this concept, the scaled Epanechnikov kernel, for dimension j, is defined to be

$$K_R^j(u) = \frac{1}{R^j} K^j(\frac{u}{R^j}). \qquad (8)$$

The Epanechnikov kernel is scaled by the factor R^j, which, for the k points in J_X, is defined to be

$$R^j = \max_{J_X}(x^j - x_i^j). \qquad (9)$$

Given these definitions, a scaled kernel function $K_R^j(u)$ can be computed for each dimension j, where $1 \leq j \leq p$. Each of these functions can then be combined into a single multidimensional kernel function $K_R(U)$, where $U = [u^1 u^2 \cdots u^p]$ and

$$K_R(U) = \prod_{j=1}^{p} K_R^j(u_j). \qquad (10)$$

Finally, the scaled multidimensional kernel can be shifted and normalized to form our weighting sequence $W_i(X)$, where, for $i \in J_X$

$$W_i(X) = \frac{K_R(X - X_i)}{\hat{f}_R(X)}, \qquad (11)$$

where $\hat{f}_R(X)$ is a normalizing factor, defined to be

$$\hat{f}_R(X) = \frac{1}{k - 2\lfloor\gamma k\rfloor} \sum_{i=\lfloor\gamma k\rfloor}^{k-\lfloor\gamma k\rfloor} K_R(X - X_i). \qquad (12)$$

This factor ensures that the sum of the weights will be one.

A modified kernel function is used for parameter values near the boundary, in order to compensate for boundary effects. This method, first presented by Rice [19], parameterizes the kernel function, which eliminates the bias near the boundaries, and yields variance near the boundaries that is the same order of magnitude as for points in the interior [19].

To define this method, first assume observations in dimension j are bounded to the interval $[a^j, b^j]$. Now, after computing the set J_X, if an observation $X_i = [x_i^1 x_i^2 \cdots x_i^p]$ with $x_i^j \leq a^j$ or $x_i^j \geq b^j$ is in the set J_X, a boundary kernel will need to be used for dimension j in equation 10. Otherwise, the regular kernel function is used. To define the boundary kernel, first define the parameter ρ^j

$$\rho^j = \begin{cases} \frac{x^j - a^j}{R^j} & \text{if } \exists X_i \in J_X : x_i^j \leq a^j \\ \frac{b^j - x^j}{R^j} & \text{if } \exists X_i \in J_X : x_i^j \geq b^j \end{cases} \qquad (13)$$

where R^j is as defined in equation 9, and x_j is the jth element of the parameter vector X. Next, define

$$\alpha^j = 2 - \rho^j. \qquad (14)$$

For the Epanechnikov Kernel, let

$$Y(\rho) = \frac{3}{4}\frac{(\rho - 1)^2}{(\rho - 2)}. \qquad (15)$$

Then, let

$$\beta^j = \frac{Y(\rho^j)}{Y(\rho^j) - \alpha^j Y(\rho^j/\alpha^j)}. \qquad (16)$$

Now a new kernel function can be defined to be

$$K_\rho^j(u) = (1 - \beta^j)K^j(u) + (\beta^j/\alpha^j)K^j(u/\alpha^j). \qquad (17)$$

The function K_ρ^j can be directly substituted for the function K^j defined in equation 8.

B Appendix: Embedding Points in an s-dimensional Space

Potter and Chiang [17] present an algorithm to embed points in a lower dimensional space in a manner which attempts to preserve the distance relationship between the points. This method begins with the r-dimensional machine space defined above, which contains m points $B_1, B_2, \dots, B_m \in \mathbb{R}_r$ representing the available machines. As mentioned above, the distance relationship between these m points will be represented using a Euclidean distance matrix $D_r = \{d_{ij}\} \in \mathbb{R}_{m\times m}$ (a real $m \times m$ matrix) in $\mathbb{R}_r$, where

$$d_{ij} = -\frac{1}{2} \parallel B_i - B_j \parallel \tag{18}$$

and $\parallel Z \parallel = (\sum_{l=1}^{r} z_l^2)^{1/2}$.

The goal of this algorithm is to find a set of m points in $\mathbb{R}_s$, where $s < r$, with distance matrix D_s providing the best-fit to D_r. The algorithm operates as follows.

1. Compute the Euclidean distance matrix D_r from the m points in $\mathbb{R}_r$.

2. Compute the orthogonal projection matrix P, which is defined to be

   $$P = I - \frac{1}{m} ee^T, \tag{19}$$

   where $e = [11 \cdots 1]$ is a vector in $\mathbb{R}_m$, and I is an $m \times m$ identity matrix.

3. Construct the matrix

   $$A = PDP^T. \tag{20}$$

4. Diagonalize A with an orthogonal matrix U and a diagonal matrix Λ, such that

   $$A = U\Lambda U^T. \tag{21}$$

5. Form $\hat{\Lambda}$ by retaining the s largest eigenvalues in Λ and setting the rest to zero.

6. Compute the matrix

   $$C = U\hat{\Lambda}^{1/2}. \tag{22}$$

The rows of the matrix C will give the coordinates of the m points in $\mathbb{R}_s$ which have a distance relationship closest to that of the original points [17].

References

[1] BYTEmark benchmark documentation. *BYTE Magazine Web Page (http://www.byte.com/ bmark/bdoc.htm)*, 1998.

[2] R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Proc. of the 1998 Heterogeneous Computing Workshop*, pages 79–87, Orlando, FL, Mar. 1998. IEEE Computer Society Press.

[3] M. V. Devarakonda and R. K. Iyer. Predictability of process resource usage: A measurement-based study on UNIX. *IEEE Trans. Software Engineering*, 15(12):1579–1586, Dec. 1989.

[4] L. P. Devroye. The uniform convergence of nearest neighbor regression function estimators and their application in optimization. *IEEE Trans. Information Theory*, IT-24(2):142–151, Mar. 1978.

[5] V. A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability and Its Applications*, 14:153–158, 1969.

[6] R. L. Eubank. *Spline smoothing and nonparametric regression*. M. Dekker, 1988.

[7] R. Freund. Optimal selection theory for superconcurrency. In *Proceedings of the 1989 Supercomputing Conference*, pages 13–17. IEEE Computer Society Press, 1989.

[8] W. Härdle. *Applied nonparametric regression*. Cambridge University Press, 1990.

[9] C.-J. Hou and K. G. Shin. Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems. *IEEE Trans. Computers*, 43(9):1076–90, Sept. 1994.

[10] M. A. Iverson, F. Özgüner, and G. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. In *Proc. of the 1996 High Performance Distributed Computing Conference*, pages 263–270, Syracuse, NY, Aug. 1996.

[11] A. Khokhar, V. Prasanna, M. Shaaban, and C.-L. Wang. Heterogeneous supercomputing: Problems and issues. In *Proc. of the 1992 Workshop on Heterogeneous Processing*, pages 3–12. IEEE Computer Society Press, Mar. 1992.

[12] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang. Heterogeneous computing: Challenges and opportunities. *IEEE Computer*, 26(6):18–27, June 1993.

[13] T. Kidd, D. Hensgen, L. Moore, R. Freund, D. Charley, M. Halderman, and M. Janakiraman. Studies in the useful predictability of programs in a distributed and homogeneous environment. *The Smartnet Home Page (http://papaya.nosc.mil:80/SmartNet/)*, 1995.

[14] W. R. King. A stochastic personnel-assignment model. *Operational Research*, 13(1):67–81, Jan. 1965.

[15] Y. A. Li, J. K. Antonio, H. J. Siegel, M. Tan, and D. K. Watson. Estimating the distribution of execution times for SIMD/SPMD mixed-mode programs. In *Proc. of the 1995 Heterogeneous Computing Workshop*, pages 35–46. IEEE Computer Society Press, Apr. 1995.

[16] D. Pease, A. Ghafoor, I. Ahmad, D. L. Andrews, K. Foudil-Bey, T. E. Karpinski, M. A. Mikki, and M. Zerrouki. PAWS: A performance evaluation tool for parallel computing systems. *IEEE Computer*, 24(1):18–29, Jan. 1991.

[17] L. C. Potter and D. Chiang. Distance matrices and modified cyclic projections for molecular conformations. In *Proceedings of the 1992 Inter. Conf. on Acoustics, Speech, and Signal Processing*, pages 173–176. IEEE Press, 1992.

[18] B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *Proc. of the 1994 ACM Conference on LISP and functional programming*, pages 65–78. ACM Press, June 1994.

[19] J. Rice. Boundary modification for kernel regression. *Communications in Statistics—Theory and Methods*, 13(7):893–900, 1984.

[20] H. J. Siegel. Heterogeneous computing. *Annual Research Summary 5.92, http://ece.www.ecn.purdue.edu/Researchsummary/Section5/sec5_92.html*, 1994.

[21] M. Tan and H. J. Siegel. A stochastic model of a dedicated heterogeneous computing system for establishing a greedy approach to developing data relocation heuristics. In *Proc. of the 1997 Heterogeneous Computing Workshop*, pages 122–34, Geneva, Apr. 1997. IEEE Computer Society Press.

[22] J. Yang, I. Ahmad, and A. Ghafoor. Estimation of execution times on heterogeneous supercomputer architectures. In *the 1993 Inter. Conf. on Parallel Processing*, volume 1, pages 219–226. CRC Press, Aug. 1993.

[23] T. Yang and A. Gerasoulis. DSC: Scheduling tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed Systems*, 5(6):951–967, Sept. 1994.

Michael Iverson received the B.S. degree in Computer Engineering at Michigan State University in 1992, and the M.S. degree in Electrical Engineering at The Ohio State University in 1994. He is currently researching topics in heterogeneous distributed computing for his Ph.D. dissertation. Mr. Iverson has also developed Internet video conferencing systems and wireless networking systems for Ohio State. Upon completion of his degree, he will be employed at Iverson Industries Inc. of Wyandotte, Michigan.

Füsun Özgüner received the M.S. degree in electrical engineering from the Istanbul Technical University in 1972, and the Ph.D. degree in electrical engineering from the University of Illinois, Urbana-Champaign, in 1975. She worked at the I.B.M. T.J. Watson Research Center with the Design Automation group for one year and joined the faculty at the Department of Electrical Engineering, Istanbul Technical University in 1976. Since January 1981 she has been with The Ohio State University, where she is presently a Professor of Electrical Engineering. Her current research interests are parallel and fault-tolerant architectures, heterogeneous computing, reconfiguration and communication in parallel architectures, real-time parallel computing and parallel algorithm design. She has served as an associate editor of the IEEE Transactions on Computers and on several conference program committees.

Lee C. Potter received the B.E. degree from Vanderbilt University and M.S. and Ph.D. degrees from the University of Illinois, Urbana, all in electrical engineering. Since 1991 he has been with the Department of Electrical Engineering at The Ohio State University where he is currently Associate Professor. His research interests include statistical signal processing, inverse problems, detection, and estimation, with applications in radar target identification and ultra wide-band systems. Dr. Potter is a 1993 recipient of the Ohio State College of Engineering MacQuigg Award for Outstanding Teaching.