New Performance Modeling Methods for Parallel Data Processing Applications

JANKI BHIMANI, NINGFANG MI, MIRIAM LEESER, and ZHENGYU YANG, Northeastern University, USA

Predicting the performance of an application running on parallel computing platforms is increasingly becoming important because of its influence on development time and resource management. However, predicting the performance with respect to parallel processes is complex for iterative and multi-stage applications. This research proposes a performance approximation approach *FiM* to predict the calculation time with FiM-Cal and communication time with FiM-Com of an application running on a distributed framework. FiM-Cal consists of two key components that are coupled with each other: (1) a Stochastic Markov Model to capture nondeterministic runtime that often depends on parallel resources, e.g., number of processes, and (2) a machine-learning model that extrapolates the parameters for calibrating our Markov model when we have changes in application parameters such as dataset. Along with the parallel calculation time, parallel computing platforms consume some data transfer time to communicate among different nodes. FiM-Com consists of a simulation queuing model to quickly estimate communication time. Our new modeling approach considers different design choices along multiple dimensions, namely (i) process-level parallelism, (ii) distribution of cores on multi-processor platform, (iii) application related parameters, and (iv) characteristics of datasets. The major contribution of our prediction approach is that FiM can provide an accurate prediction of parallel processing time for the datasets that have a much larger size than that of the training datasets. We evaluate our approach with NAS Parallel Benchmarks and real iterative data processing applications. We compare the predicted results (e.g., end-to-end execution time) with actual experimental measurements on a real distributed platform. We also compare our work with an existing prediction technique based on machine learning. We rank the number of processes according to the actual and predicted results from FiM and calculate the correlation between the actual and predicted rankings. Our results show that FiM obtains a high correlation in the range of 0.80 to 0.99, which indicates considerable accuracy of our technique. Such prediction provides data analysts a useful insight of optimal configuration of parallel resources (e.g., number of processes and number of cores) and also helps system designers to investigate the impact of changes in application parameters on system performance.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Additional Key Words and Phrases: Performance modeling, queuing theory, Markov model, distributed systems, execution time, parallel calculation, communication network, prediction

This work was partially supported by National Science Foundation (NSF) award CNS-1452751, CNS-1717213, and AFOSR grant FA9550-14-1-0160.

Authors' address: J. Bhimani, N. Mi, M. Leeser, and Z. Yang, Northeastern University, 360 Huntington Ave, Boston, MA, 02115, USA; emails: {bhimani, ningfang}@ece.neu.edu, {mel, yangzy1988}@coe.neu.edu.

1049-3301/2019/06-ART15 $15.00
https://doi.org/10.1145/3309684

ACM Reference format:
Janki Bhimani, Ningfang Mi, Miriam Leeser, and Zhengyu Yang. 2019. New Performance Modeling Methods for Parallel Data Processing Applications. *ACM Trans. Model. Comput. Simul.* 29, 3, Article 15 (June 2019), 24 pages.
https://doi.org/10.1145/3309684

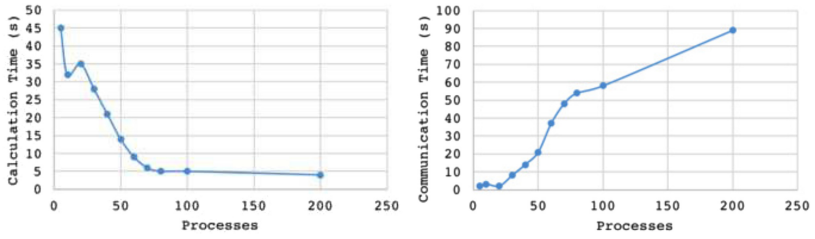
1 INTRODUCTION

High-Performance Computing (HPC) systems are ubiquitous in processing data for myriad applications involving huge datasets. How to achieve the best performance with an optimal configuration of parallel resources (e.g., number of processes and number of cores) is a challenging research problem. Currently, researchers run their application codes on a representative dataset, fix application parameters, and try different configurations of parallel resources to determine the optimal one. However, if we want to find optimal application parameters, then the investigation needs to consider all possible combinations of application parameters and parallel resources. Such an investigation becomes very expensive, requiring a significant amount of time and hardware resources. Besides, on parallel computing platforms, using more parallel resources does not always guarantee performance improvement. Hence, it is beneficial if we can approximate the optimal performance in terms of parallel resources and application parameters. The prediction of expected performance before the porting of an actual implementation on a hardware platform can save significant time and hardware resources spent in experimentally finding the optimal performance. The emergence of huge datasets as workloads and parallel computing has emphasized the importance of predictive analysis, and performance bottleneck identification. Designing efficient prediction tools thus becomes critically important to system designers and application programmers [24].

As motivation, we show an example with an iterative k-means clustering application running on a framework using Message Passing Protocol (MPI [19, 35]) in Figure 1. We observe that the calculation time decreases when we have more parallel MPI processes; however, the time to communicate data increases. Such an observation implies that speeding up parallel calculation time may not guarantee overall application speed-up. Also, the decrease in calculation time levels off after 70 parallel processes. Increasing the number of parallel processes further consumes more system resources, but may not improve overall application runtime. Thus, the capability of predicting such an optimal point (e.g., 70 in Figure 1) is vital to system designers for making good design choices.

In this article, we develop a new performance modeling approach, named FiM, to estimate both computation and communication times of iterative, multi-stage data processing applications using MPI. Each node is a CPU that has multiple cores, and each core can support multiple MPI processes. One of the nodes accesses an application dataset and determines the distribution of processing among the processes of other nodes. All these nodes then perform parallel computations using multiple processes. Such a parallel phase is known as one stage in our model. At the end of each stage, all processes synchronize to decide on the termination or launch next stage. In this research, we concentrate on predicting parallel processing time of such an iterative and multi-staged application running with global synchronizations. One of the key innovations in our work is that FiM relies only on small datasets for training but can predict the execution times for larger datasets. More specifically, this article aims to answer the following questions through our prediction models.

- Can we quickly estimate the parallel calculation and communication times of an application to identify the optimal number of processes?
- Can we use small datasets as training to predict performance of applications operating in parallel on large datasets?



(a) (b)

Fig. 1. Latency variation across different number of parallel processes for (a) calculation time and (b) communication time.

- How does the number of processes impact calculation and communication time?
- Is the application communication or compute bound?

To answer these questions, we introduce FiM, which consists of two main components: (1) FiM-Cal and (2) FiM-Com. The goal of FiM-Cal is to predict the calculation time by using a stochastic Markov model and a machine-learning model. The stochastic Markov model is built using the probabilistic technique to estimate the impact of an increase in the number of parallel processes. We first develop the base case of the parallel paradigm and then derive a generic model that applies to any number of parallel processes as well as any number of dependent stages (e.g., iterations) of an application. The base case of the Markov model is calibrated using the minimum number of system parameters. The machine-learning model is then designed to extrapolate the calibrated parameters for the Stochastic Markov model to adapt to changes in application parameters such as datasets.

The goal of FiM-Com is to predict the communication time using a set of simulation queuing models. Here our motive is to get a quick estimate using a simplified prediction model. Such an estimate of communication time along with calculation time can provide instant insight to users. Thus, our FiM approach can use the minimum possible calibration parameters to quickly predict the expected computation and communication time as well as the optimal number of processes for platform configuration. While comparing actual and predicted time, the worst prediction error of the overall application runtime by FiM is observed to be less than 20%. The source code of our calculation and communication time prediction model is available at GitHub (https://github.com/bhimanijanki/FiM). A preliminary version of the article, titled "FiM: Performance Prediction for Parallel Computation in Iterative Data Processing Applications," was published in *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD'17)* [8].

The remainder of this article is organized as follows. We present the two FiM components in Section 2 and Section 3.2, respectively. We evaluate our models on a distributed memory platform, see Section 4. In Section 5, we discuss some related work. Section 6 presents our conclusions and future works.

2 FIM-CAL: CALCULATION PREDICTION

In this section, we present *FiM-Cal*, an analytical approach to predicting the calculation time of an application running on a distributed multi-process platform. FiM-Cal consists of two key components: a stochastic Markov model and a machine-learning model. We first use the stochastic Markov model to represent the computational processing of an application in a parallel MPI framework. Then we design a machine-learning algorithm to estimate the parameters related to the system for calibrating our stochastic Markov model. This parameter extrapolation enables our model

Table 1. Notations Used in *FiM-Cal*

Notation	*Description*
S_j^i	Markov chain state with i active and j passive processes
Ps_i	Stage completion probability of i^{th} stage
P_{ij}	State transition probability of moving from i active to j active processes
P_{act}	Probability P_{11} of 1 process model
P_{p2a}	Probability P_{01} of 1 process model
P_{a2p}	Probability P_{10} of 1 process model
P_{pass}	Probability P_{00} of 1 process model
F	Frequency (GHz)
TC	Total cycles
SC	Total stall cycles
U	Utilization per process
T_i	Total time taken by stage i
α	Sensitivity constant
β	Regression constant
y_i	Dependent variables
$\vec{X_i}$	Vector of independent variables

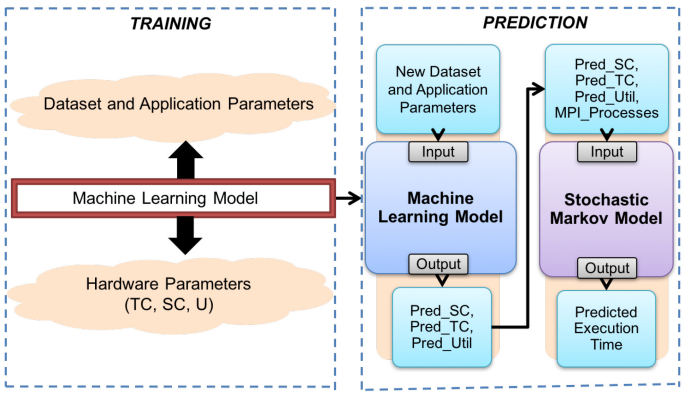


Fig. 2. Prediction procedure of FiM-Cal.

to predict an application's calculation time when we have a different number of parallel processes or variable application parameters (such as dataset size) without any system state instrumentation. Table 1 lists the notations used in this article for FiM-Cal. Figure 2 shows the overall workflow of our proposed FiM-Cal. We will introduce the details of each component in Figure 2 in the remainder of this section.

2.1 Stochastic Markov Model

Our stochastic Markov model is designed to model computational processing for an application running on a system with parallel multi-core CPUs deployed using MPI. Such a stochastic model allows us to capture a non-deterministic runtime that often depends on parallel resources, e.g., number of processes. If there exists a global synchronization call in an application, then all processes wait until the barrier. The processing of an application is partitioned into multiple stages

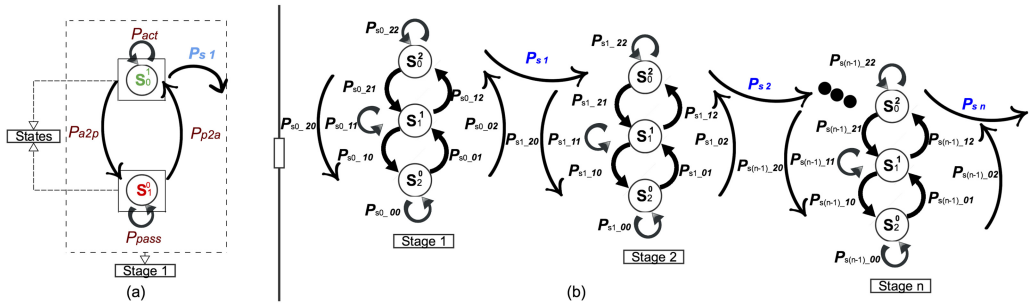


Fig. 3. Modeling (a) the nase case: single process with single stage and (b) generic case: two processes with multiple stages.

with respect to this global synchronization, such that each stage corresponds to a parallel phase until all processes have completed their tasks and are in an active state to proceed to the next stage. In this section, we first introduce our base case, which models a single stage for a single-process system, and then show its extension to the generic case with multiple processes and multiple stages.

2.1.1 Base Case. The base case model is built to represent a single process for a single stage application, see Figure 3(a). In our model, each process is considered to be either in *active* or *passive* state. As shown in Figure 3(a), when only a single process runs in the computing platform, we have two states for a stage such that state S_0^1 represents that one process is active while S_1^0 represents that one process is passive. We also introduce transition probabilities (e.g., P_{act}, P_{a2p}, P_{pass}, P_{p2a}) of switching between two states or staying in the same state, as well as the stage completion probability (Ps_1) of transferring from one stage to another. In the active state, the process performs constructive work and typically changes from the active state to the passive state when it is blocked by an event that would create a latency stall. Such a latency stall might be caused due to a cache miss that takes many CPU cycles. In this work, we do not model memory latencies, contentions, inter-dependencies and deadlocks individually for each process but rather treat the combined effect as a process remaining passive.

The probabilities in the base case model are parameterized by instrumenting the system details, which is further used to derive the probabilities of the generic case. To parameterize the probabilities of the base case, we use the `perf tool` [16] to instrument the required data, including the hardware clock rate (F), system CPU utilization factor per process (U), total number of cycles required for execution (TC), and stall cycles (SC). In particular, we run an application with a single stage on a single process and use the perf stat command to collect and report the required data as listed above.

In the base case (i.e., single-process and single-stage), the probability to remain in the active state (P_{act}) is primarily determined by the proportion of time that the process is performing useful work. Therefore, we use Equation (1) to get P_{act},

$$P_{act} = U, \tag{1}$$

where U is utilization per process. As shown in Figure 3(a), when the process is in an active state (i.e., S_0^1), there are three possible events for its next transition: (1) Remain in S_0^1 with probability P_{act}, (2) transition to with probability P_{a2p}, and (3) complete the stage with probability Ps_1. Now, if there are TC total cycles to be processed for the given dataset, then processing is completed only after completing the last cycle. This gives the probability of completion as $1/TC$. Probability P_{a2p}

for the process to transit to the passive state can then be calculated as shown in Equation (2),

$$P_{a2p} = 1 - (P_{act}) - (1/TC). \tag{2}$$

The probability of the process remaining passive (P_{pass}) is primarily determined by the ratio of stall cycles (SC) to total cycles (TC) as shown in Equation (3),

$$P_{pass} = SC/TC. \tag{3}$$

We can determine the probability of switching from passive to active (P_{p2a}) by applying the control flow equation to the passive state (S_1^0) as shown in Equation (4),

$$P_{p2a} = 1 - (P_{pass}). \tag{4}$$

We finally get the stage completion probability (Ps_1) by applying the control flow equation to the active state (S_0^1) as shown in Equation (5),

$$Ps_1 = 1 - (P_{act}) - (P_{a2p}). \tag{5}$$

Note that, for the base case with one stage and only one possible active phase, Ps_1 is the same as $1/TC$, because all active cycles can be spent only in one active state (S_0^1). Later, in generic cases, we discuss the calculation of Ps_1, which is then not the same as $1/TC$.

2.1.2 Generic Cases. Now we consider generic cases where we can have multiple processes operating on an application with multiple stages. This generic behavior can be modeled as an extension of the base case. The processing of an application may have multiple inter-dependent parallel stages. For example, an iterative application with 500 iterations can be divided into 500 parallel stages such that each stage represents an iteration and is entered only after the completion of all prior stages. Thus, the first stage corresponds to the parallel calculation phase by all processes in the first iteration and is followed by the remaining stages in the same order. In general a simple non-iterative and single stage application can be treated as having 1 iteration and 1 stage while predicting its runtime using our proposed technique.

Figure 3(b) shows an underlying Markov model for an application with two processes using n stages, each with two parallel processes. The entire workflow of an iterative, multi-stage, multi-process application can be mapped with a chain of n parallel stages, and Ps_1, Ps_2, . . ., Ps_n, are the completion probabilities for all n stages, see Figure 3(b). Note that these stage completion probabilities are non-uniform and dependent on all the completion probabilities of prior stages as well as intra-state transition probabilities of that stage. Also for every stage, all its state transition probabilities (P_{ij}) depend on the completion probability of the prior stage. Thus, the value of P_{ij} in a stage is different from that of P_{ij} in another stage even for the same i and j. Furthermore, a single stage can only complete when all of its processes are active, i.e., not being blocked by any events. A calculation phase of an application is completed when tasks assigned to all processes are completed in the last stage. Thus, the completion probability of an application is Ps_n.

To model an iterative, multi-stage paradigm with multiple processes, we use multiple states within each stage to represent activities (active or passive) of all processes. Consider t processes with i active processes and j passive processes, where $0 <= i <= t$, $0 <= j <= t$ and $t = i + j$. Each stage consists of a total of $M = t + 1$ states. Thus, the transition probabilities of jumping from any one of these M states to other states or itself can be divided into three types: (1) probability to remain in the same state (e.g., P_{22}, P_{11} and P_{00} in Figure 3(b)), (2) probability to increase active processes (e.g., P_{01}, P_{12}, P_{02} in Figure 3(b)), and (3) probability to increase passive processes (P_{10}, P_{21}, P_{20} in Figure 3(b)). Given M states, we have M probabilities to remain in the same state, $\sum_{i=1}^{M-1} i$ probabilities to increase active processes, and $\sum_{i=1}^{M-1} i$ probabilities to increase passive processes.

Thus, the total number of probabilities to be calculated for a single stage with M states is equal to $M + 2\sum_{i=1}^{M-1} i = M^2$.

2.1.3 Solving the Generic Model. We solve such a generic Markov model and derive its probabilities by relating them to the preliminary transition probabilities of the base case. That is, once we have transition probabilities for $M = 2$ (base case), we can calculate all probabilities for a generic case with $M > 2$. In Reference [27], a mathematical relation between the transition probabilities of a Markov model with two states and a Markov model with more than two states has been derived. We use their method to relate transition probabilities of the cases with $M = 2$ and $M > 2$. Initially, base case probabilities are calculated.

Specifically, let us consider to derive generic model probabilities for an application with $M = 3$ from the base case probabilities of $M = 2$. For $M = 3$, the application must have two parallel processes (see Figure 3(b)). At any given time during the execution, there are three states ($M = 3$): (1) Both of these processes can be active (state S_0^2), (2) one process can be active and the other passive (state S_1^1), and (3) both can be passive (state S_2^0). The probability of two active processes to continue as active is $P_{22} = P_{act} * P_{act}$. Similarly, the probability of two passive processes to continue as passive is $P_{00} = P_{pass} * P_{pass}$. If one process that was active in previous stage remains active and the other process that was passive remains passive, or if the active process becomes passive and the other process that was passive becomes active, then in both of the above cases the model stays in state S_1^1. Thus, the probability of such state transition is $P_{11} = (P_{act} * P_{pass}) + (P_{a2p} * P_{p2a})$. P_{01} is state transition probability from S_2^0 (i.e., both the passive processes) to S_1^1 (i.e., one active process and the other passive process).

Same as above, the state transition probabilities for any M can be derived. These derived equations can be mathematically reduced to a more generic form. Equations (6) to (8) give the state transition probabilities for $M > 2$, where i corresponds to the number of active processes in the previous state and j corresponds to the number of active processes in the targeted state. For example, in Figure 3(b), P_{21} indicates the state transition probability of moving from a state with 2 active processes (S_0^2) to a state with 1 active process (S_1^1). Substituting appropriate i and j in Equations (6) to (8) for $M = 3$, all the probabilities explained and derived above for 2 processes such as P_{00}, P_{11}, P_{22}, P_{01}, P_{02}, P_{12}, P_{10}, P_{20}, and P_{21} can be obtained. These equations represent the stochastic process of a Markov chain and can be calculated by mathematical induction after solving the Markov chain with a finite number of states. Particularly, as shown in Equations (6) to (8), we use the probabilities (P_{act}, P_{a2p}, P_{pass}, and P_{p2a}) that are obtained by the base case (Section 2.1.1) to calculate the state transition probabilities in generic cases.

If i == j, then

$$P_{ii}(t) = \sum_{k=0}^{min\{i,t-i\}} \binom{i}{k} \cdot \binom{t-i}{t-i-k} \cdot \left(P_{act}^{i-k}\right) \cdot \left(P_{a2p}^{k}\right) \cdot \left(P_{pass}^{t-i-k}\right) \cdot \left(P_{p2a}^{k}\right). \tag{6}$$

If i < j, then

$$P_{ij}(t) = \sum_{k=0}^{min\{i,t-j\}} \binom{i}{k} \cdot \binom{t-i}{t-j-k} \cdot \left(P_{act}^{i-k}\right) \cdot \left(P_{a2p}^{k}\right) \cdot \left(P_{pass}^{t-j-k}\right) \cdot \left(P_{p2a}^{j-i+k}\right). \tag{7}$$

If i > j, then

$$P_{ij}(t) = \sum_{k=0}^{min\{j,t-i\}} \binom{i}{i-j+k} \cdot \binom{t-i}{t-i-k} \cdot \left(P_{act}^{j-k}\right) \cdot \left(P_{a2p}^{i-j+k}\right) \cdot \left(P_{pass}^{t-i-k}\right) \cdot \left(P_{p2a}^{k}\right). \tag{8}$$

Additionally, after capturing the state transition probabilities of the first stage, we calculate the stage completion probability Ps_1 using Equation (9) and then use Ps_1 as an incoming probability for calculating the state transition probabilities of stage 2, and so on. This chaining process captures an iterative and multi-stage application running with multiple processes. Finally, Ps_n for the nth stage is calculated by Equation (9),

$$\begin{aligned} Ps_n &= P(X_s = n \mid X_{s-1} = n-1) \\ &= 1 - \sum_{j=0}^{j=i} (P_{ij} \mid X_{s-1} = n-1) \\ &for\ i = Max(\#Processes). \end{aligned} \tag{9}$$

We further use Equation (10) to calculate the time (T_n) spent in performing parallel calculation for n stages, given the completion probability (Ps_n) and CPU frequency (F),

$$T_n = \frac{1}{(Ps_n).(F)}. \tag{10}$$

Consequently, our stochastic Markov model can predict the computation time required to process any particular dataset using different levels of parallelism such as different number of processes in MPI. Next, we present our machine-learning technique that assists to extrapolate the data (such as, F, U, TC, and SC) required for calibrating the base case model.

2.2 Machine-learning Model

Our stochastic Markov model allows us to predict the calculation time of an application when we have a different number of processes in the system. However, the required hardware parameters (i.e., TC, SC, U) need to be instrumented for every new dataset and a new setting of application parameters. This limits the scope of the model to predict for a particular set of datasets and fixed application parameters. Most analytical models suffer from this lack of flexibility. Therefore, we develop our machine-learning model to avoid additional instrumentation for a new dataset or a new set of application parameters. For a new application, we need first to train to derive a new model. However, for new sets of application parameters and new datasets of the same application, the derived model can be used to predict the runtime. To reduce the complexity of the machine-learning model, we also assume that the application calculation time is dependent on the fewest possible hardware parameters. Our evaluation results, shown in Section 4, demonstrate the feasibility of this assumption by showing the fairly accurately predicted results obtained by our approach that is good to give a quick approximation. Here we introduce a two-stage machine-learning model that emulates hardware behaviors without performing actual instrumentation for required hardware-related data. Such a hybrid emulation of hardware is the key to allowing the approach to be able to predict parameters for datasets with sizes much larger than those of the training datasets.

2.2.1 Regression Mapping. The focus of regression is to find the relationship between a dependent variable (such as the hardware parameters that we want to emulate) and one or more independent variables (such as application parameters and datasets). This analysis estimates the conditional expectation of a dependent variable given values of all related independent variables. We find that the generalized linear regression model performs the best when compared to others (quadratic, Poisson model, and gradient decent) for modeling all desired hardware parameters (U, TC, and SC). We show the validation of a linear regression model in Section 4.3.

The linear regression equation for learning variable y_i is shown in Equation (11), where $\vec{X_i}$ is a vector of p-independent variables related to application parameters and datasets, $\vec{\beta_i}$ consists of a vector of $p + 1$ constants, and n is total number of scalar-dependent variables. Suppose for the

k-means application, elements of $\vec{X_i}$ would consist of the number of desired clusters (K), the number of iterations (I), and size (N). For our model, we have three scalar-dependent variables, i.e., U, TC, and SC, which can be predicted after building this linear regression model. Thus, we have three equations for y_1, y_2, and y_3 with $n = 3$,

$$\begin{aligned} y_i &= \beta_{i0} + \beta_{i1}x_{i1} + \cdots + \beta_{ip}x_{ip} \\ &= \vec{\beta_i}(1 + \vec{X_i}^T), \ \textit{for } i = 1, 2, \ldots, n. \end{aligned} \tag{11}$$

This linear regression model is used to find values for constants $\vec{\beta_i}$ using the training data for which both dependent variables and independent variables are known. We obtain the regression curve and regression constants $\vec{\beta_i}$ by building our machine-learning model in MATLAB.

ALGORITHM 1: Calibration of α

```
1  Input: ϵ, τ y_i, X_i,Output: α
2  Initialize: β_i, TC, SC, U using regression mapping, α = 0 and iter = 0
3  if 0 <= U <= 1, TC >0, SC >0, SC <TC then
4      Predict comp. time using stochastic Markov model
5      Calculate RMS error (Actual, Predicted)
6      if iter == 0 then
7          Calculate error (Actual - Predicted)
8          Decide OP = + or -, depending on positive or negative error
9      if RMS error <τ then
10         return
11     else
12         α = α {OP} ϵ
13         iter ++
14         Calculate TC, SC, and U using Equation (12)
15         goto line 3
16 else
17     Neglect bad values
18     goto line 4
```

2.2.2 Iterative Improvement Model. We found non-negligible errors between the actual and predicted calculation times when we pair the linear regression model described above with our stochastic Markov model, to predict calculation time of large datasets based on small training datasets. To handle this issue, we develop an iterative improvement model that uses a sensitivity parameter α to tune the constant factors $\vec{\beta_i}$ with respect to the predicted results (i.e., calculation time) of our stochastic Markov model as shown in Equation (12). Note that in this equation the constants in vector $\vec{\beta_i}$ are obtained from the regression between hardware parameters and application parameters, but constant α is obtained by using both Markov model and regression model as described in Algorithm 1,

$$\begin{aligned} y_i &= \alpha(\beta_{i0} + \beta_{i1}x_{i1} + \cdots + \beta_{ip}x_{ip}) \\ &\textit{for } i = 1, 2, \ldots, n. \end{aligned} \tag{12}$$

Initially, we use hardware parameters (U, TC, and SC) and application parameters of training data with regression mapping (Equation (11)) to obtain constants of vector $\vec{\beta}_i$. The hardware parameters are passed to our stochastic Markov model to predict calculation time. In the first iteration, the error between actual and predicted time is used to decide the adjust direction of α, see lines 6 to 8. That is, if the actual value is greater than the predicted one, then the algorithm increases α and vice versa. In the following iterations, Algorithm 1 increases or decreases α by a small value ϵ (e.g., ϵ = 1e-5), and the hardware parameters (U, TC, and SC) are predicted using constants of vector $\vec{\beta}_i$ and α with Equation (12) (see line 14). Then the computation time is predicted using U, TC, and SC as the inputs to our stochastic Markov model (see line 4). The adjustment process continues until the root mean square (RMS) error becomes smaller than a predefined threshold (e.g., τ = 0.01), see line 9. The value of τ needs to be set such that we avoid underfitting as well as overfitting the training data. From our experiments, we observe that τ = 0.01 is a good value on an average across different choices of applications and training datasets. Thus, the algorithm adjusts the value of α until the predicted calculation time becomes close to the actual time. We tune α while training the model and then use the exact tuned α value throughout the prediction stage. Note that our machine-learning model is used to calculate the constants of vector $\vec{\beta}_i$ and α in the training phase, which are after that used for extrapolation of hardware parameters.

In summary, Figure 2 shows the overall procedure of our prediction model *FiM-Cal*, which includes the training and prediction phases. In the prediction phase, our machine-learning model extrapolates the dependent variables (such as hardware parameters: SC, TC, U) for new datasets and new sets of application parameters. These predicted hardware details can then be used as an input to our stochastic Markov model to predict the calculation time.

3 FIM-COM: COMMUNICATION PREDICTION

Apart from computation time [7, 8], applications spend some time in the communication of data to various processes. With the increase in the number of processes, this communication time keeps increasing [37]. Therefore, it is essential to roughly estimate communication time, i.e., the runtime of *data transfer*. For applications running on a distributed multi-process system, the data transfer time for processes lying on the same node is different from that between processes lying on different nodes. The framework considered in this work consider all inter-node communication.

3.1 Communication Patterns

It is non-trivial to predict communication time, due to various communication patterns and the non-deterministic latency of the communication network. There exist many, more complex models to predict accurate network communication time [10, 20, 23, 30], but here our motive is to get a quick estimate using a simplified prediction model. In this work, we investigate three types of collective communication patterns including downlink (scatter and broadcast) and uplink (gather). We build different queuing models to capture those patterns when running a data processing application such as k-means clustering on a multi-process system with MPI. Because the data transfer time along with computation time is significant, our model aims to provide very quick rough estimates of communication time to make fast decisions.

Specifically, we consider communications from the one to many nodes as the *downlink*. Such a downlink communication can have either the scatter pattern or the broadcast pattern. Under the scatter pattern, the data are distributed among the multiple nodes by the one node such that each node gets a unique part of the data. Under the broadcast pattern, the data are broadcast to all nodes, and each node receives the same copy of the data. The scatter communication is usually undertaken by communicating data to one after another process in MPI. We also consider communications

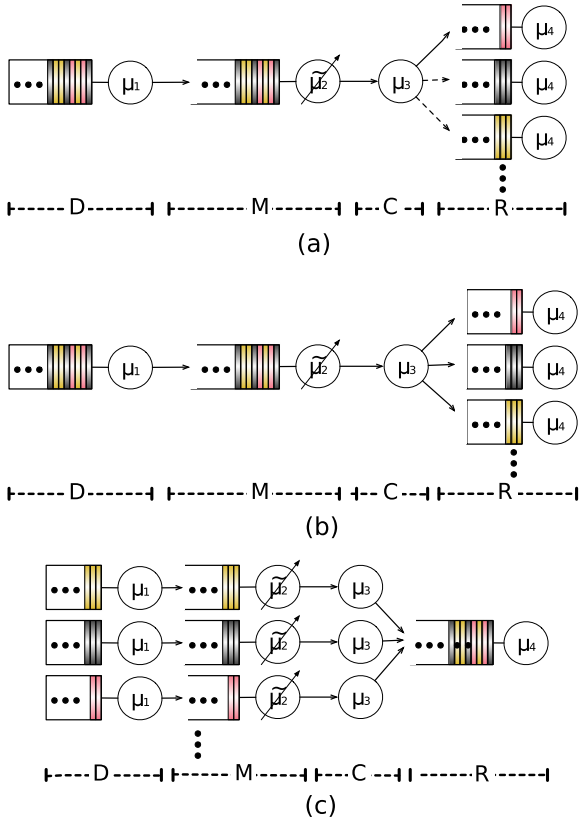


Fig. 4. Queuing models for the (a) scatter, (b) broadcast, (c) gather communication patterns.

from multiple nodes to the one node as the uplink gather pattern. When gathering, each node sends their own data to a shared buffer, and the managing node reads those data one by one from the buffer.

3.2 Communication Models

We develop a set of queuing models to capture these three collective communication patterns. Such queuing models can give a high-level abstraction of real communication systems in terms of packet arrival rates, delay/waiting time, and packet transfer rate, which helps to estimate the overall data communication time. Note that we try to keep these queuing models simple to enable fast estimation.

Figure 4 presents our queuing models to represent the scatter, broadcast, and gather patterns, respectively. Typically, communication time depends on various network properties such as initialization cost, maximum network bandwidth, network load, and the amount of metadata (e.g., headers, acknowledgments, and flags). Therefore, in each of these models, we use four components, Data transfer (D), Metadata (M), Cost of network initialization (C), and Receive (R) in Figure 4, in series to capture different properties of network communication. Each job in the queue of D, M, C, and R represents a data packet e.g., a pixel for k-means clustering.

First, D is used to capture the actual data transfer rate through the communication network, where mean service rate μ_1 indicates the available network bandwidth. We assume that all data

packets are available in memory. Thus, all jobs are assumed to wait in the queue of part D before data transfer begins.

The second part M, in the models of Figure 4 captures the effect of metadata on overall communication time. Such metadata can include headers, acknowledgments, addresses, offsets, padding, and flags that need to be transferred over the network along with the original data packets as part of the network protocol. The time consumed by such metadata transfers depends on the number of actual data packets and instantaneous network load. To capture this, we design component M with a variable mean service rate $\tilde{\mu_2}$, which depends on the instantaneous queue length of part M. The arrivals to this queue are the departures from part D.

Part C in the model represents network initialization. Usually, the cost for network initialization is a network latency that is added to overall communication time. We use a delay server with mean service rate μ_3 to emulate such network latency. The last part of our models (R in Figure 4) is used to estimate the data fetching time on the receiver side. We use part R to capture the receiving, while the first three parts (D, M, and C) capture the sending. We assume that the receiving process is homogeneous across different nodes. Therefore, multiple servers and queues are used to represent multiple receivers, and the mean service rate of each server is μ_4.

We present the corresponding models for scatter and broadcast in Figure 4(a) and (b), to predict communication times for the downlink, i.e., from the one node to multiple nodes. Under the scatter pattern, the one node sends different pieces of data to other nodes one by one. We use the round robin policy to distribute jobs into queues in part R, and thus there is only one solid arrow in R to indicate data transfer to one process at a time, see Figure 4(a). In Figure 4(b), there are multiple solid arrows to R, which represents that we duplicate jobs (data points) and distribute them to all queues of R in parallel to capture the broadcast pattern. The third model, shown in Figure 4(c), represents the gather pattern of the uplink communication, where multiple nodes send their data to one node. We have multiple servers running in parallel in parts D, M and C. The First-In-First-Out (FIFO) discipline is used in the last queue to emulate data fetching by the managing node.

3.3 Communication Model Calibration

The main tasks in communication model calibration are (1) to determine an appropriate queuing model and (2) to derive service processes for all servers in the model based on its collective communication pattern. Our models only need to be calibrated when we have a new application or run on a new hardware platform. We then leverage the knowledge of actual transfer processes on a new platform to calibrate the service rates of all servers in the models. The calibrated models can be used directly to predict communication times for new datasets, new application parameters (e.g., number of iterations in k-means) and different number of MPI processes.

In particular, we calibrate μ_1 using the maximum network bandwidth obtained from the provided network configuration of a cluster (e.g., 10Gb/s backplane). We observe from conducting experiments on multiple applications for both 10Gb/s and faster 56Gb/s TCP/IP backplane that the actual data transfer rates are often within 90% to 100% of the maximum network bandwidth, with a uniform distribution. Therefore, we calculate the lower bound (ζ_{min}) and the upper bound (ζ_{max}) of transfer rate per unit job (i.e., each data point) and derive the service rate μ_1 in part D using Equation (13),

$$\mu_1 = Uniform(\zeta_{min}, \zeta_{max}). \tag{13}$$

To get the transfer time of the metadata, we first measure the actual total communication time for sending a unit job (a single data point) and then deduct the measured runtime of the other three parts, i.e., network bandwidth, network initialization, and receiver's data fetch time. We find that the derived transfer rates of metadata are logarithmic to the instantaneous network load (refer

Table 2. Platform Configurations Labelled as C1 to C5 (2-E52670—Two Multi-core, Hyper-threaded Intel Xeon E5 2670 CPU's @ 2.60GHz and 256GB of RAM) (2-E52650—Two Multi-core, Hyper-threaded Intel Xeon E5 2650 CPU's @ 2.00GHz and 128GB of RAM)

	C1	C2	C3	C4	C5
CPU	2-E52670	2-E52670 2-E52650	2-E52670 4-E52650	2-E52670 6-E52650	2-E52670 8-E52650
Cores	32	64	96	128	160
Network	10Gb/s Ethernet backplane TCP/IP				
Shared FS	NFS				
OS	Linux				

Section 4.3). Therefore, we use Equation (14) to generate $\tilde{\mu_2}$ for metadata, where χ represents the number of jobs currently waiting in the queue of `M`. The tilde on μ_2 represents the variable mean service rate of component `M`, which depends on the instantaneous queue length of part `M` (χ) to symbolize variance as explained above,

$$\tilde{\mu_2} = log(\chi). \tag{14}$$

In part C, μ_3 captures the network initialization latency using a delay server. Specifically, the sending node first activates itself (`comselfcreate`) and then activates the receiving nodes (`comworldcreate`). The `MPE` tool [19] is used to collect the `mpilog` log files. We observe that the initialization latency follows an exponential distribution (refer Section 4.3). Thus, we use Equation (15) to get the service rate μ_3, where η is the mean of the service rate distribution,

$$\mu_3 = exp(\eta). \tag{15}$$

Finally, we observe that the data receiving rate is exponentially distributed (refer Section 4.3). Additionally, we consider a multiplicative factor ϕ to model the number of receivers that simultaneously fetch data. For example, ϕ is equal to one under the scatter pattern, since only one node can receive data at any time. For the broadcast pattern, ϕ is equal to the number of nodes. Thus, Equation (16) is used to draw μ_4 for all the servers in part `R`, where ψ is the mean service rate distribution,

$$\mu_4 = \phi * exp(\psi). \tag{16}$$

4 EVALUATION

We evaluate our FiM prediction approach (a combination of FiM-Cal and FiM-Com) by comparing the predicted results (e.g., end-to-end execution time) with actual experimental measurements on a real distributed platform. We also compare our prediction approach with an existing work, named RBASP [5], which is a regression-based approach to extrapolate execution time. In our evaluation, we consider the end-to-end execution time of data processing applications as the sum of the runtime spent in communicating the required data to parallel processing units, performing the calculations in parallel, and transferring the results back to the managing node. We assume that no overlap exists between the calculation and communication in the application implementation. We use the Discovery Cluster at Northeastern University [1] to build our experimental platform. Table 2 describes five parallel platform configurations we used in our evaluation, where each CPU belongs to different nodes.

We evaluate our approach with six NAS Parallel Benchmarks (NPB—version *NPB3.3.1-MPI*) [2], with the large size dataset of *Problem Class C*. Table 3 lists the six benchmarks we used in our

Table 3. NPB Benchmarks

BT	Block Tri-diagonal solver	Compute Bound
EP	Embarrassingly Parallel	Compute Bound
SP	Scalar Penta-diagonal solver	I/O Bound
LU	Lower-Upper Gauss-Seidel solver	I/O Bound
IS	Integer Sort	Memory Bound
CG	Conjugate Gradient	Memory Bound

evaluation. For each benchmark, we train our model using three small size datasets of *Problem Class S*. Note that we use the trained model to predict the performance for datasets of *Problem Class C*, which are much larger than the datasets of *Problem Class S*. We reprogram the NPB benchmarks to implement iterative, multi-stage paradigm versions in MPI using C. The time spent for computation in all iterations (or multiple phases) is the total computation time. For each iteration, we measure the time from the start of parallel processes to the completion of all the processes. We also evaluate FiM with two iterative data processing applications: k-means [6] and Pagerank. For each of these applications, we run experiments on 15 different datasets and choose one dataset as a representative to show the results, i.e., N_L (the large dataset with 13 million data points). For both applications, we train our model using three small datasets, i.e., N_S (the small datasets each with 3 thousand data points).

k-means Clustering (KM): Our k-means clustering implementation [6] takes color images as input datasets. We cluster pixels in an image based on five features, including three *RGB* channels and the position (x, y) of each pixel. We choose random data points to initialize each cluster centroid and then use the Euclidean distance to calculate the nearest cluster for each data point. The parameters of k-means include a number of desired clusters (K), number of iterations (I), and size of input dataset (N).

Pagerank (PR): The Pagerank application takes a network of directed vertexes and edges as an input dataset. The output of the Pagerank application is a probability distribution representing the weights of each vertex (page). We choose a damping factor of 0.85 and initialize all vertices (pages) with the same probability weights. The parameters of this application include a number of vertexes (V), number of iterations (I), size of input dataset (N), and network nature (dense or sparse).

4.1 Performance Evaluation

In our evaluation, we consider a regression-based approach named RBASP [5] to compare with our FiM approach. RBASP is well known for its simplicity and accuracy in extrapolating execution time of multi-process applications. In our FiM approach, we combine our stochastic Markov modeling with machine-learning regression to predict calculation time and use our queueing models to predict communication time. Our prediction model with this combination of popular techniques helps to predict accurately in most cases and gives a quick estimation. We choose RBASP to compare our prediction accuracy, because, similarly to FiM, it also gives a quick prediction and can estimate runtime of datasets larger than the training datasets. RBASP is a pure regression-based approach; unlike FiM, it directly estimates total time (calculation + communication) using regression. RBASP model predicts the execution time (y) of a given parallel application on p processes by using several instrumented runs of an application on q processes, where $q \in \{1, \ldots, p_0\}$ and $p_0 < p$. By varying the values of independent variables $(x_1, x_2, \ldots, x_n)$, this model aims to calculate coefficients $(\beta_0, \ldots, \beta_n)$ by the linear regression fit for $log_2(y)$ (Equation (17)), where $g(q)$ can

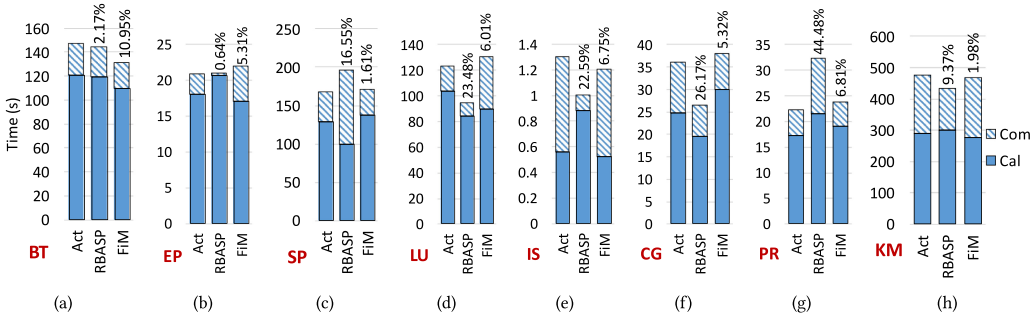


Fig. 5. Actual and predicted execution time using FiM and RBASP with the relative prediction error listed on top of each bar.

Table 4. Summary of Results for All Applications (Rel. Er.-Relative Error) (Act.-Actual) (App.-Application) (Opt.-Optimal)

	Best Rel. Er. %		Average Rel. Er. %		Worst Rel. Er. %		Opt. # MPI Process		
App.	RBASP	FiM	RBASP	FiM	RBASP	FiM	RBASP	FiM	Act.
BT	1.98	1.04	2.17	10.95	66.45	18.07	136	360	360
EP	0.44	2.13	0.64	5.31	76.74	15.32	309	252	248
SP	1.98	0.21	16.55	1.61	94.21	19.14	212	64	56
LU	0.14	0.06	23.48	6.01	79.15	26.88	82	28	20
IS	1.73	1.3	22.59	6.75	90.94	34.36	70	70	70
CG	2.63	2.99	26.17	5.32	77.65	40.73	156	102	102
PR	2.02	0.91	44.48	6.81	57.65	31.46	21	56	64
KM	1.61	0.42	9.37	1.98	30.43	10.87	64	192	192

be either a linear function or a quadratic function,

$$log_2(y) = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + g(q). \tag{17}$$

While reproducing the RBASP model, we use $p_0 = 1, 2, 4$ as the training set and predict the performance with two forms of $g(q)$ function as suggested in Reference [5]. The RBASP approach directly predicts the execution time using regression, which requires performing training with data points processed for a different number of multiple processes. In contrast, FiM extrapolates the hardware parameters for a given computing platform and then uses these hardware parameters as the inputs to the stochastic Markov model for predicting execution time for a different number of processes. FiM does not need to be trained again when we change the number of processes used in a given computing platform.

Figure 5 shows the predicted results using RBASP and FiM for six NPB benchmarks and two iterative data processing applications. We run these experiments on the $C5$ platform (see its configuration in Table 2), using the actual optimal number of processes listed in Table 4. As shown in Figure 5, our FiM approach achieves a pretty good agreement between the predicted and actual results across all the six benchmarks and two applications. We also observe that RBASP has lower relative prediction error than FiM for only BT and EP. Both BT and EP are compute-intensive benchmarks, so a pure regression technique is sufficient to predict their execution time. However, prediction using only regression is not good enough for I/O and memory intensive applications as observed from RBASP prediction results in Figure 5 for the rest of the applications. As shown in

Figure 5, for all the remaining benchmarks and applications, FiM performs better. Many real-time applications are I/O or memory intensive, for whom simple regression model is not sufficient to give accurate predictions. For the results shown in Figure 5, the calculation time reported includes all background delay that occurs due to I/O latency to access the data that are required to perform the computations. The time required for data transfer among different parallel processes is communication time. We also observe that the prediction error of FiM remains less than 20% for individual prediction of calculation time and communication time. Furthermore, RBASP's prediction is limited to the fixed application parameters on which the model is trained, because if the application parameters are changed, then RBASP needs to be re-trained. In contrast, FiM can predict execution time without any prior training for new application parameters, because we do not regress the execution time directly, but instead, we train our model to learn the change in system counters like total cycles and stalled cycles when application parameters are changed. Thus, this also advanced our approach to avoid over-fitting to the execution time of the training datasets.

Table 4 lists the best, average and worst prediction errors as well as the actual and predicted optimal numbers of processes using RBASP and FiM. Apart from having a lower average error, also having a tight prediction error range is important for these prediction approaches, because such a range can be used to provide a quick approximation before conducting actual experiments. We observe that FiM has a relatively tight prediction error range from the best to the worst, compared to RBASP. Despite the lower prediction error under the best case, RBASP obtains higher prediction errors in the worst case for all benchmarks and applications. This is because the pure regression model used in RBASP has poor adaptability to changes in the values of attributes (e.g., number of processes). FiM provides tighter error bounds, which is very important for such a quick estimation modeling technique. We further observe that the optimal number of processes predicted by FiM is very close to the actual one. We also rank the number of processes according to the actual and predicted results from FiM and calculate the correlation[1] between the actual and predicted rankings. We obtain a high correlation in the range of 0.80 to 0.99, which indicates considerable accuracy of our FiM estimation technique.

We further use the Chi-square goodness-of-fit test [29] to evaluate how well the predicted runtime distribution matches with the actual measured runtime distribution with varying application parameters and the number of processes. Our null hypothesis is that the predicted data are not consistent with actual distribution. The significance level of our test is 0.1, meaning that the deviation of the predicted value is not more than 10%. The p-value[2] that we obtain for our test is equal to 0.08. Since the p-value (0.08) is less than the significance level (0.1), we cannot accept the null hypothesis. This indicates reasonably high confidence of less than 10% error between the actual and predicted values.

4.1.1 Individual Prediction Results. We carefully study the effectiveness of our model for predicting execution time with the increasing number of processes. Figure 6 shows the results for FiM per data point (e.g., per pixel in k-means) when operated with a dataset consisting of 13 million data points after getting trained using three datasets with less than 1,000 data points. These experiments are performed using the C5 hardware configuration of Table 2. We conduct all experiments for 1,000 times and measure the average runtime as well as the minimum and maximum actual runtime obtained in these 1,000 runs, which are shown by the range bars. From Figure 6(a), we observe that the actual calculation time keeps decreasing when we have more simultaneous processes running. We also observe that 192 is the optimal number of processes. Increasing the

[1]A correlation between actual and predicted ranks describes the degree of agreement between them. Correlation ranges between −1 and 1 with 1 being the best; higher correlation signifies better accuracy of predicted results.
[2]The p-value is a statistical measure of the deviation of the actual distribution from the hypothesis.

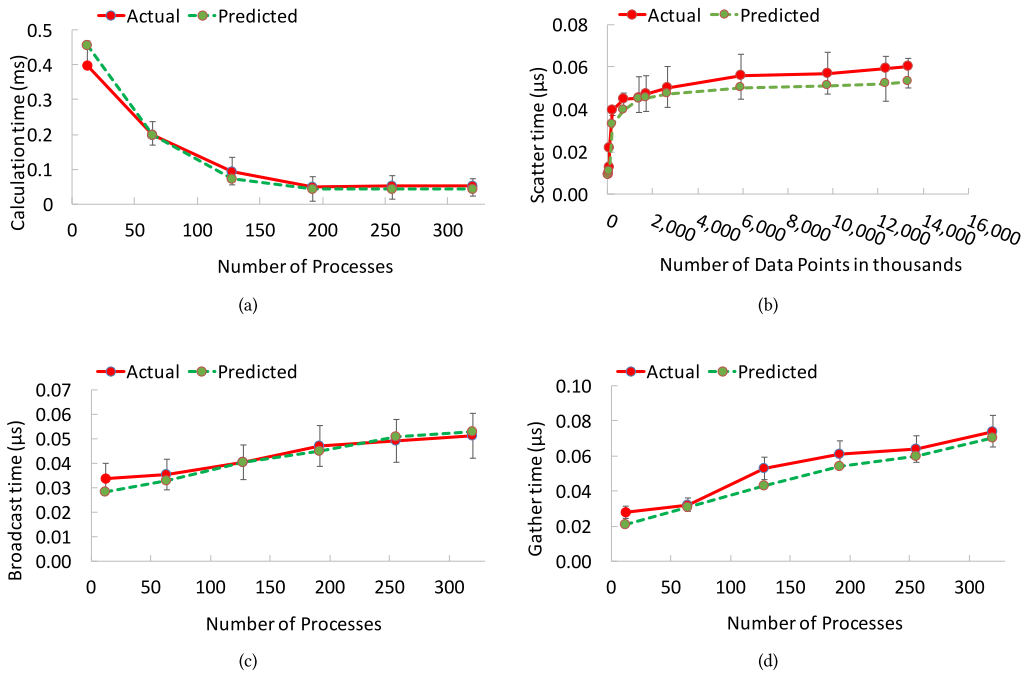


Fig. 6. Actual and predicted time - (a) Calculation (b) Scatter (c) Broadcast (d) Gather.

number of processes further above 192 does not give any performance improvement. Our model can accurately predict this optimal number of processes.

Figure 6(b) shows the results of FiM-Com for the scatter pattern. For scatter of "n" data points over "p" processes, "n/p" data points need to be transferred to each process. For scatter, each process receives a unique subset of the dataset and thus, with the increase in the number of processes, the number of data points to be transferred in total does not increase. Hence, the runtime of the scatter communication depends mainly on the size of input datasets and not on the number of processes. So, we evaluate the FiM-Com scatter model under different datasets of different sizes. The results are shown in Figure 6(b) are obtained using 192 processes and for datasets with the different number of data points mentioned on the x-axis in thousands. The smallest dataset consists of one thousand data points and the largest dataset consists of 13 million data points. We observe that, when we have relatively small datasets, the scatter time per data point increases rapidly with the increase in dataset size until 600 thousand data points scattered to 192 processes. However, for larger datasets, the scattering time per data point remains almost constant or increases very slowly. Each data point is of 20B, thus for less than or equal to 600 thousand data points scattered over 192 processes, less than or equal to 64KB is required to be transferred to each process. Note that the eager protocol message size is set to 64KB with environment variable `MP_EAGER_LIMIT`. Accordingly, the default receive buffer size is increased with environment variable `MP_BUFFER_MEM`. Hence, for a data transfer smaller than 64KB, the communication network follows the eager protocol. However, for a large data transfer, the communication network follows the rendezvous protocol [35] to perform sender-receiver handshake. Our scatter model is designed to capture the effect of this protocol shift.

Figure 6(c) shows the results of FiM-Com for broadcast as a function of the number of processes. We see that data transfer of each data point consumes more time as the number of processes

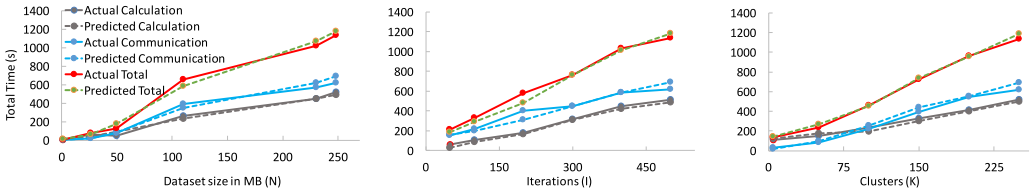


Fig. 7. Sensitivity analysis - (a) Dataset size (b) Number of Iterations (c) Number of Clusters.

increases. This is because for broadcast, with each additional process, the number of data points to be transmitted also increases unlike scatter. The increase in data packets incurs an additional load that in turn increases the average broadcast time of each data point. Last, the results under the uplink gather pattern are plotted in Figure 6(d) with respect to the number of processes for each data point. The gathering time increases linearly with an increase in the number of processes due to the increased congestion in the communication network, with more data points to be received by the data collecting node. From Figure 6, we observe that our FiM models accurately predict the calculation and communication times under different communication patterns. Also, the predicted results mostly remain in the range bars (i.e., the minimum to the maximum) of the actual observations. All these graphs show that FiM estimates are very accurate.

4.2 Sensitivity Analysis

One significant contribution of our modeling techniques is to accurately predict for large datasets by using only small datasets to train and calibrate the models. In our experiments, we use three small datasets as the training ones to collect data for model calibration. For a new application, we need first to train to derive a new model. However, for new sets of application parameters and new datasets (including both small and large ones) of the same application, the derived model can be used to predict the runtime. Therefore, we perform a sensitivity analysis of different dataset sizes and application parameters. We argue that if the user has the flexibility to choose application parameters for achieving optimal performance, our model then can provide useful guidance by helping the user to decide appropriate parameters.

For example, k-means processing with more iterations and more clusters can provide better clustering results and accuracy but consume more time. Therefore, it would be useful to use FiM to estimate the execution time with respect to the increase in the number of iterations and number of clusters to determine how much extra latency is needed to achieve better accuracy. The results of execution time for the k-means algorithm as a function of (a) dataset size, (b) the number of iterations, and (c) number of clusters are plotted in Figure 7. Figure 7 also shows the actual and predicted calculation and communication time. We experiment with different hardware configurations as shown in Table 2, and present the results of the C5 configuration here. In these experiments, we also use the predicted optimal number of processes listed in Table 4. For each plot in Figure 7, we do a sensitivity analysis on one parameter and fix the remaining two parameters with dataset size of 250MB, 500 iterations and 250 clusters. We observe that our models can accurately predict the execution time even when the datasets become large, see Figure 7(a). Note that we only use small datasets to train our models. Figure 7(b) shows a linear increase of execution time with increasing number of iterations. Figure 7(c) further shows execution time with respect to the increase in number of clusters. Summarizing from Figure 7, we can see that FiM consistently achieves predicted results in good agreement with actual ones under different application parameters like dataset size, number of iterations and number of clusters.

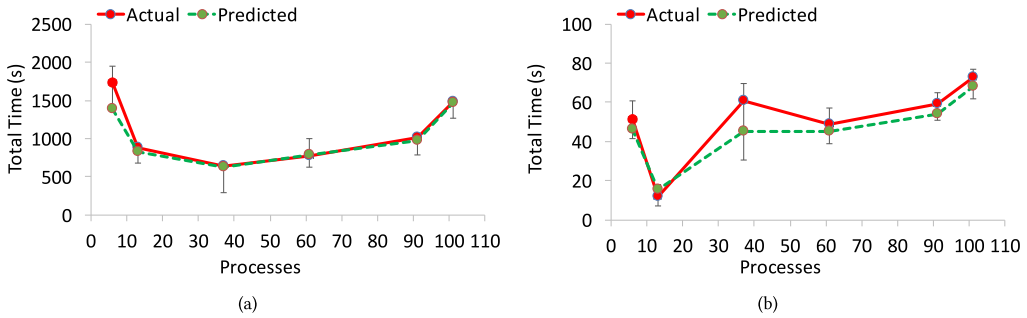


Fig. 8. Sensitivity analysis w.r.t. number of MPI processes for (a) k-means and (b) Pagerank.

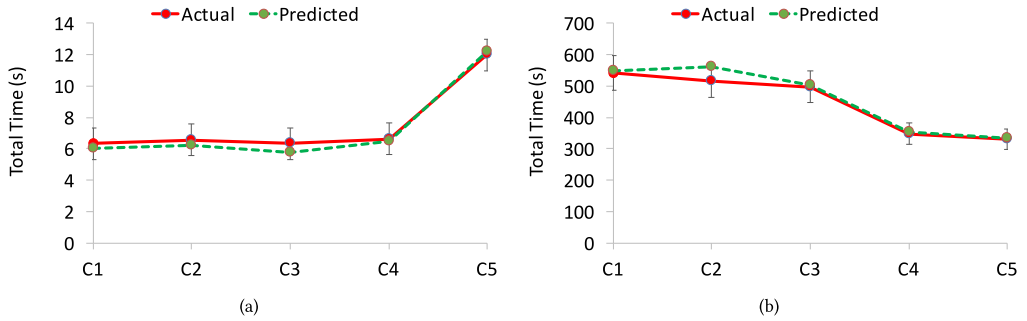


Fig. 9. Sensitivity analysis w.r.t. distribution of cores under (a) a small dataset with 40 vertices and (b) a large dataset with 4,039 vertices for Pagerank application.

We further evaluate the prediction of our models under different hardware configurations. A distributed computing platform deployed using MPI offers different choices in the number of parallel processes and the distribution of cores (e.g., C1-C5 listed in Table 2). Figure 8 shows the actual and predicted execution times with respect to the number of parallel processes for (a) k-means and (b) Pagerank, respectively. We can see that the best performance (i.e., the shortest execution time) is achieved in the middle range of processes, e.g., 38 for k-means and 12 for Pagerank. FiM is able to predict the optimal performance for both applications accurately. Predicted results match well with actual ones across the different number of processes.

Figure 9 plots the predicted results for Pagerank under five different hardware configurations listed in Table 2. To be able to predict across such heterogeneous platforms, we calibrate our model for data compute and data transfer among each type of available hardware. We observe that the first three configurations (C1, C2, and C3) achieve a shorter execution time for a small dataset (see Figure 9(a)). When we have large datasets, C4 and C5 with more distributed cores are better (see Figure 9(b)). FiM can accurately predict such performance trends, i.e., Pagerank becomes more scalable on the higher number of distributed cores for larger datasets. These estimation results can thus provide us with insightful data regarding the scalability of an application on a multi-core computing platform.

Here, we evaluate our technique by individually varying each variable parameter. From the results presented above, we see that this sensitivity analysis helps to observe that prediction accuracy of our model remains intact with changing application parameters, number of parallel processes and different hardware platforms.

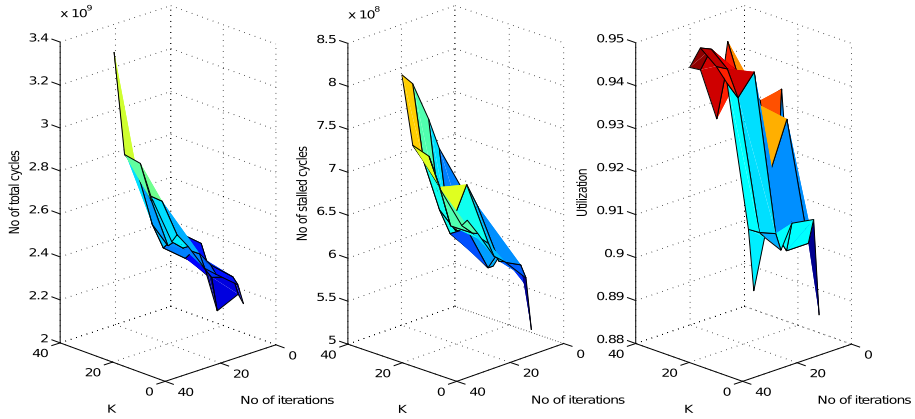


Fig. 10. Regression mapping (K - number of clusters, I - number of iterations).

4.3 Validation

In this section, we present the validation for considering the linear regression model to extrapolate hardware variables such as TC, SC, and U. In particular, we show results using k-means clustering as a representative. Recall, for k-means, we chose linear regression to extrapolate hardware variables as a function of application parameters (I, N, and K), see Section 2. There are a variety of regression models that can be used. It is not straightforward to choose the right regression model that is best suited to our requirement. Even a complex model might over-fit the training data, generating a substantial prediction error on other datasets. However, a simple model may underestimate the learning trends and produce incorrect predicted results [39]. Considering these two cases, we first investigate the learning trend of three hardware variables (TC, SC, and U) under different settings of application parameters. We choose the linear regression model after examining other regression models such as a piece-wise linear model, Poisson models and quadratic models with different degrees of the polynomial.

We investigate the learning trend of three hardware variables under different settings of application parameters. We show the hardware variables such as TC, SC, and U for k-means clustering as a representative. Figure 10 depicts the resulting surface of each hardware variable as a function of application parameters (e.g., I and K for the k-means application). Similar results can be obtained for other combinations of application parameters, such as (I, N) and (N, K). In Figure 10, linear surfaces can be found for different hardware variables. Thus, we conclude that hardware variables TC, SC, and U, linearly depend on the increment in application parameters (such as K, I, and N). These results confirm the use of the linear regression mapping in our machine-learning approach.

We also investigate the calibration of our communication models, i.e., the training results of the service rates for each server in the queuing models, see Section 3.2. In Figure 11, we plot the CDFs of the measured communication service times (i.e., $1/\mu_2$, $1/\mu_3$ and $1/\mu_4$) for the three components (i.e., metadata transfer (M), network initialization (C) and data fetch time (R)) in the communication. The predicted service times drawn from our calibrated service processes (i.e., Equations (14), (15), and (16)) are also plotted in Figure 11. We can see that the predicted service time distributions well capture the actual service times.

The Chi-square goodness-of-fit test [29] is used to evaluate how the three derived statistical distributions (i.e., uniform, logarithmic and exponential) for μ_2, μ_3, and μ_4 fit the actual measurements. The chi-squared test is used to determine whether there is a significant difference between

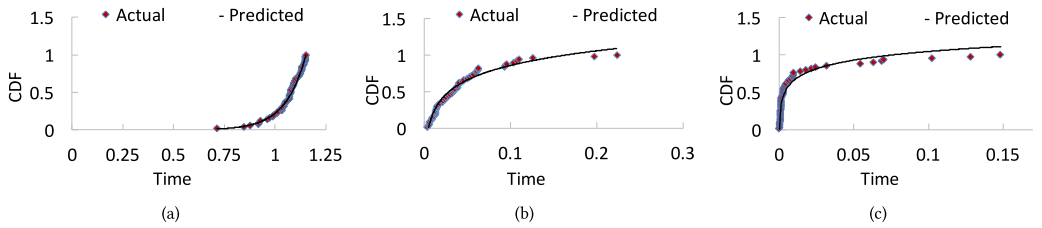


Fig. 11. Cumulative distributed functions (CDFs) of the actual and predicted service times (per job) for (a) metadata transfer, (b) network initialization, and (c) data fetching.

the expected distribution and the observed distribution. Our null hypothesis is that the derived statistical distributions are not consistent with actual distribution. The significance level of our test is 0.15, meaning that the maximum deviation of the predicted distribution is not more than 15%. The p-values[3] that we obtain for service rates μ_2, μ_3, and μ_4 are equal to 0.08, 0.11, and 0.14, respectively. Since the p-values are less than the significance level (0.15), we reject the null hypothesis. This indicates reasonably high confidence of not more than 15% error between the actual and predicted values.

5 RELATED WORK

Modeling helps to shorten the development cycle by providing the necessary insights to obtain optimal performance. Performance modeling can be approached in several different ways, including empirical evaluation [17, 18], simulation [9, 26, 28] and analytical modeling [4, 15]. Empirical evaluation is a technique for gaining knowledge about a system from observations of experiments. This requires the exact implementation as well as similar hardware to the target, because results are based on observed ground truth. Empirical techniques were popular in the past when computer hardware was stable over long periods. They give result fast and accurately for the datasets similar to training.

Simulators model hardware such as memory hierarchy, communication buses, parallel ports, and accelerators [31, 33, 40]. They evaluate each block of the given codes, similar to the manner that it is executed on the target machine. Thus, they require source codes while performing prediction. Although simulators like SimpleScalar [3] and CACTI [38] can predict with high accuracy, they also consume a long time to give predicted results. Their slow running time and infrastructure cost are major drawbacks.

Analytical modeling is the technique of building a set of equations to show the high-level abstraction of the behavior of an application and a hardware architecture. This type of modeling can be evaluated quickly and easily as it is reduced to the form of the set of equations to be solved. Analytical models can be flexible and scalable but are comparatively less accurate than empirical and simulation-based models, because they lack accurate hardware machine models. The major drawback of analytical modeling is that it limits the scope of prediction. These models will give high prediction errors if tested with parameters that were not captured when building the model equations. An innovative idea is the combination of the above models, e.g., COMPASS [25]. It gives good accuracy but requires the compilation of source code for each new test dataset as well as the conversion of source codes to ASPEN [36]. COMPASS requires the hardware machine model to be formed not only for training but also for testing, which is quite time consuming, especially for large datasets.

[3]The p-value is a statistical measure of deviation of the actual distribution from the hypothesis.

Predicting the performance of any parallel processing platform consists of two major phases, i.e., calculation and communication. The Markov chain modeling using probabilistic distribution assists in predicting the calculation load of multi-process and multi-core architectures [12, 27, 32]. These approaches model systems in the form of equations, where changes to the code or data require changes to the equations. The above process can be time-consuming when we desire to predict for a range of parameters. Some analytical models [13, 14, 21, 22, 34] require the conversion of source code into a control flow chart for the ease of framing equations. Our model predicts the performance of an application on a range of input parameters without requiring a new set of equations, as FiM uses a machine-learning model to learn hardware parameters. Ad hoc analytical models and structured analytical models have been developed to predict the network communication behavior of an application [11]. They use predefined models such as timeline diagrams showing various network overheads. This type of models cannot efficiently capture different patterns like scatter, broadcast and gather. The concept of using queuing theory to model such communication and predict the expected communication time is a novel idea described in our article.

6 CONCLUSIONS

We present a novel performance modeling technique (FiM) to predict the execution time of iterative multi-stage data processing applications running on parallel computing paradigm. We combine different modeling techniques, such as stochastic Markov modeling, machine-learning techniques and queuing theory, to predict the end-to-end execution time. FiM estimates the time required for both data calculation and data communication across a range of input datasets, application configurations and parallel hardware parameters such as number of processes. We demonstrate that FiM helps system designers and application programmers choose optimal hardware parameters and application parameters. More importantly, our prediction models are parameterized using small datasets but can predict accurately for large datasets. We evaluated our approach using NAS Parallel Benchmarks and real iterative data processing applications. We rank the number of processes according to the actual and predicted results from FiM and calculate the correlation between the actual and predicted rankings. Our results show that FiM obtains high correlation in the range of 0.80 to 0.99, which indicates considerable accuracy of our technique. In the future, we plan to expand the scope of our prediction model to investigate other communication patterns like all-to-all communication. We will also capture scenarios where there is an overlap between the communication and computation phases. Large-scale computing platforms, including GPUs, will further be considered as a target environment.

REFERENCES

[1] [n.d.]. Information Technology Services—Research Computing. Retrieved from https://www.northeastern.edu/rc/.

[2] [n.d.]. NASA Advanced Supercomputing Division, NAS Parallel Benchmarks. Retrieved from http://www.nas.nasa.gov/publications/npb.html.

[3] T. Austin, E. Larson, and D. Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer* 2 (2002), 59–67.

[4] David H. Bailey and Allan Snavely. 2005. Performance modeling: Understanding the past and predicting the future. In *Proceedings of the European Conference on Parallel Processing*. Springer, 185–195.

[5] Bradley J. Barnes, Barry Rountree, David Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. 2008. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing*. ACM, 368–377.

[6] Janki Bhimani, Miriam Leeser, and Ningfang Mi. 2015. Accelerating K-means clustering with parallel implementations and GPU computing. In *Proceedings of the High Performance Extreme Computing Conference (HPEC'15)*. IEEE, 1–6.

[7] Janki Bhimani, Ningfang Mi, and Miriam Leeser. 2016. Performance prediction techniques for scalable large data processing in distributed MPI systems. In *Proceedings of the IEEE 35th International Performance Computing and Communications Conference (IPCCC'16)*. IEEE, 1–2.

[8] Janki Bhimani, Ningfang Mi, Miriam Leeser, and Zhengyu Yang. 2017. FiM: Performance prediction for parallel computation in iterative data processing applications. In *Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD'17)*. IEEE, 359–366.

[9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Arch. News* 39, 2 (2011), 1–7.

[10] Henri Casanova. 2001. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001*. IEEE, 430–437.

[11] WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. 2009. LogGPO: An accurate communication model for performance prediction of MPI programs. *Sci. Chin. Ser. F: Inf. Sci.* 52, 10 (2009), 1785–1791.

[12] Xi E. Chen and Tor M. Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. IEEE, 329–340.

[13] Gopinath Chennupati, Nandakishore Santhi, Robert Bird, Sunil Thulasidasan, Abdel-Hameed A. Badawy, Satyajayant Misra, and Stephan Eidenbenz. 2017. A scalable analytical memory model for CPU performance prediction. In *Proceedings of the 8th International Workshop on High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, PMBS (PMBS@SC'17)*. Springer, 114–135.

[14] G. Chennupati, N. Santhi, S. Eidenbenz, and S. Thulasidasan. 2017. An analytical memory hierarchy model for performance prediction. In *Proceedings of the 2017 Winter Simulation Conference (WSC'17)*. 908–919.

[15] Gopinath Chennupati, Nanadakishore Santhi, Stephen Eidenbenz, Robert Joseph Zerr, Massimiliano Rosa, Richard James Zamora, Eun Jung Park, Balasubramanya T. Nadiga, Jason Liu, Kishwar Ahmed, and Mohammad Abu Obaida. 2017. *Performance Prediction Toolkit (PPT)*. Los Alamos National Laboratory (LANL). Retrieved from https://github.com/lanl/PPT.

[16] Arnaldo Carvalho de Melo. 2010. The new linux perf tools. In *Proceedings of Linux Kongress*, vol. 18.

[17] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*. ACM, 114–118.

[18] Phillip Gibbons, Yossi Matias, and Vijaya Ramachandran. 1998. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. Comput.* 28, 2 (1998), 733–769.

[19] A. Chan, W. Gropp, and E. Lusk. User guide for MPE: Extensions for MPI programs. Technical report, ANL-98. Argonne National Laboratory.

[20] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. LogGOPSim: Simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 597–604.

[21] Tanzima Z. Islam, Jayaraman J. Thiagarajan, Abhinav Bhatele, Martin Schulz, and Todd Gamblin. 2016. A machine learning framework for performance coverage analysis of proxy applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 46:1–46:12.

[22] Nikhil Jain, Abhinav Bhatele, Michael P. Robson, Todd Gamblin, and Laxmikant V. Kale. 2013. Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 95:1–95:12.

[23] Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. 2012. A simulator for large-scale parallel computer architectures. *International Journal of Distributed Systems and Technologies (IJDST)* 1, 2 (2010), 57–73.

[24] Darren Kerbyson, Henry Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey Wasserman, and Mike Gittings. 2001. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. ACM, 37–37.

[25] Seyong Lee, Jeremy Meredith, and Jeffrey Vetter. 2015. COMPASS: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 405–414.

[26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[27] Reshmi Mitra, Bharat S. Joshi, Arun Ravindran, Arindam Mukherjee, and Ryan Adams. 2012. Performance modeling of shared memory multiple issue multicore machines. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*. IEEE, 464–473.

[28] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*. ACM, 1050–1055.

[29] D. Cox. 2002. Karl pearson and the chi-squared test. In *Goodness-of-Fit Tests and Model Validity*. Springer, 3–8.

[30] Juan-Antonio Rico-Gallego, Juan-Carlos Díaz-Martín, and Alexey L. Lastovetsky. 2016. Extending τ-Lop to model concurrent MPI communications in multicore clusters. *Fut. Gener. Comput. Syst.* 61 (2016), 66–82.

[31] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. 2011. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (2011), 37–42.

[32] Rafael H. Saavedra-Barrera and David E. Culler. 1991. *An Analytical Solution for a Markov Chain Modeling Multithreaded.* Technical Report. Citeseer, Berkeley, CA.

[33] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 97–108.

[34] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. 2002. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02).* IEEE, 1–17.

[35] Marc Snir. 1998. *MPI—The Complete Reference: The MPI Core.* Vol. 1. MIT Press.

[36] Kyle Spafford and Jeffrey Vetter. 2012. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE Computer Society Press, 84.

[37] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[38] Steven Wilton and Norman Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circ.* 31, 5 (1996), 677–688.

[39] Ming Yuan and Yi Lin. 2006. Model selection and estimation in regression with grouped variables. *J. Roy. Statist. Soc. B* 68, 1 (2006), 49–67.

[40] G. Zheng, Gunavardhan Kakulapati, and L. V. Kale. 2004. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004.*

Received January 2018; revised December 2018; accepted January 2019