FiM: Performance Prediction for Parallel Computation in Iterative Data Processing Applications

Janki Bhimani, Ningfang Mi, Miriam Leeser, and Zhengyu Yang
Dept. of Electrical & Computer Engineering, Northeastern University, 360 Huntington Ave., Boston, MA 02115
bhimani@ece.neu.edu, ningfang@ece.neu.edu, mel@coe.neu.edu, and yangzy1988@coe.neu.edu

***Abstract*—Predicting performance of an application running on high performance computing (HPC) platforms in a cloud environment 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, multi-stage applications. This research proposes a performance approximation approach *FiM* to model the computing performance of iterative, multi-stage applications running on a master-compute framework. FiM consists of two key components that are coupled with each other: 1) Stochastic Markov Model to capture non-deterministic runtime that often depends on parallel resources, e.g., number of processes. 2) Machine Learning Model that extrapolates the parameters for calibrating our Markov model when we have changes in application parameters such as dataset. Our new modeling approach considers different design choices along multiple dimensions, namely (i) process level parallelism, (ii) distribution of cores on multi-core processors in cloud computing, (iii) application related parameters, and (iv) characteristics of datasets. The major contribution of our prediction approach is that FiM is able to provide an accurate prediction of parallel computation time for the datasets which have much larger size than that of the training datasets. Such calculation 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.**

***Keywords*—Performance Modeling, Markov Model, Regression, Distributed Systems, Cloud Computing, Big Data Infrastructure**

I. Introduction

High Performance Computing (HPC) services in cloud are ubiquitous in processing scientific 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 typical dataset, fix application parameters, and try different configurations of parallel resources to determine the optimal one. However, if we further want to find optimal application parameters, then the investigation needs to consider all possible combinations of application parameters and parallel resources. Such an investigation in a cloud environment becomes very expensive, requiring a large amount of time and hardware resources. In addition, in HPC, using more parallel resources does not always guarantee performance improvement. Hence, it could be beneficial if we can approximate the optimal performance point in terms of parallel resources, application parameters, and datasets. The prediction of expected performance prior to the porting of an actual implementation on a cloud platform can save time and resources spent in experimentally finding the optimal performance point.

This work was partially supported by National Science Foundation Career Award CNS-1452751 and AFOSR grant FA9550-14-1-0160.

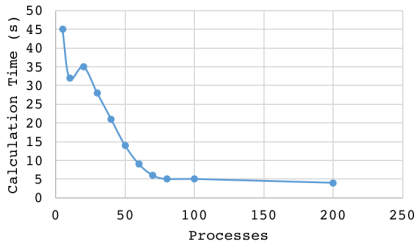

Fig. 1: Latency variation across different number of parallel processes for calculation time

As a motivation, we plot an example with an iterative K-means clustering application running on a "master-compute" (also called "master-worker") framework using Message Passing Protocol (MPI [1], [2]) in Figure 1. We observe that the calculation time considerably decreases when we have more parallel MPI processes until 70 parallel processes. This implies that increasing the number of parallel processes after 70 only consumes more resources, but does not decrease the overall application runtime. Thus, the capability of predicting such an optimal point (e.g., 70 in Figure 1) is important to system designers for making good design choices. More specifically, this paper aims to answer the following questions through our prediction models.

- Can we quickly estimate the parallel calculation 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?
- How does the distribution of parallel cores impact the optimal number of processes?

To answer these questions, we introduce FiM which consists of two main components: (1) Stochastic Markov Model, and (2) Machine Learning Model. The Stochastic Markov Model is built using the probabilistic technique to estimate the impact of increase in the number of parallel processes. We first develop the base case of the considered parallel paradigm and then derive a generic model that is applicable 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 by using the minimum number of system parameters. The Machine Learning Model is then designed to extrapolate the calibrated parameters for the Stochastic Markov model in order to adapt to the changes in application parameters such as datasets. Thus, our FiM approach can use the minimum possible calibration parameters to accurately estimate the computation time as well as the optimal number of processes. While comparing actual and predicted compute time, the worst prediction error of FiM is less than 40%. One of the key innovations in our work is that FiM relies only on small datasets for training but can estimate the execution times

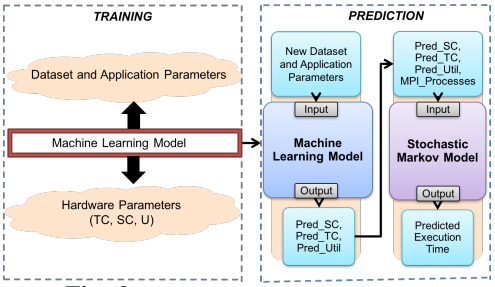

Fig. 2: Prediction procedure of FiM

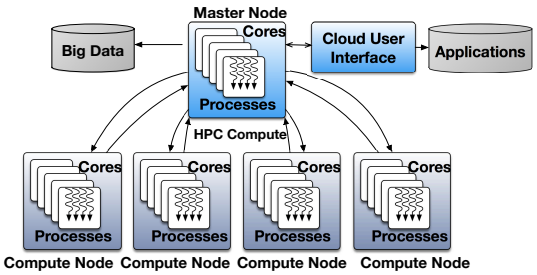

Fig. 3: High Level Architecture of HPC on Cloud

for larger datasets.

The remainder of this paper is organized as follows. We present the two FiM components in Section II. In Section III, we evaluate our model on a distributed memory platform. We discuss some related work in Section IV and summarize our conclusions and future plan in Section V.

II. FiM: Calculation Prediction

In this section, we present FiM, an analytical approach to predicting the calculation time of an application running on a distributed multi-process platform. FiM consists of two key components, i.e., a stochastic Markov model and a machine learning model. Generally, we first use the stochastic Markov model to represent the computational processing of an application in a parallel master-compute 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 thus enables our model to predict an application's calculation time when we have different number of parallel processes or variable application parameters (such as dataset size) without any system state instrumentation. Table I lists the notations that are used in this paper. Figure 2 shows the overall work flow of our proposed FiM. We will introduce the details of each component in Figure 2 in the remaining part of this section.

TABLE I: Notations used in FiM

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 fraction 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

A. System Description

We model the parallel computation in iterative, multi-stage data processing applications, running on a master-compute framework as shown in Figure 3. Each compute node is a CPU which has multiple cores and each core can support multiple MPI processes. One of the nodes is declared as the master to handle the processing control. The remaining nodes are all compute nodes which perform parallel computations. The master node accesses an application and its dataset to determine the distribution of its processing among the processes of compute nodes. These processes on compute nodes operate in parallel to calculate their local mean values. Such a parallel phase is known as *one stage* in our model. At the end of each stage, the master node gathers the results from each process and broadcasts the new task to each process on compute nodes for the next stage. In this research, we concentrate on predicting parallel computation time of such an iterative and multi-stage application running in a master-compute platform with global synchronizations.

B. Stochastic Markov Model

Our stochastic Markov model is designed to model a 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 non-deterministic runtime that often depends on parallel resources, e.g., number of processes. In real systems, due to the global synchronization, all processes wait until each process has completed its own work. So, each stage is modelled to represent a parallel phase until all processes have completed their tasks and are in an active state to proceed to the next stage. The processing of an application is partitioned into multiple stages with respect to its global synchronization, such that each stage corresponds to a parallel phase. 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.

1) Base Case: The base case model is built to represent a single process for a single stage application, see Figure 4 (a). In our model, each process is considered to be either in *active* state or *passive* state. As shown in Figure 4(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, interdependencies and deadlocks individually for each process but alternatively treat the combined effect as a

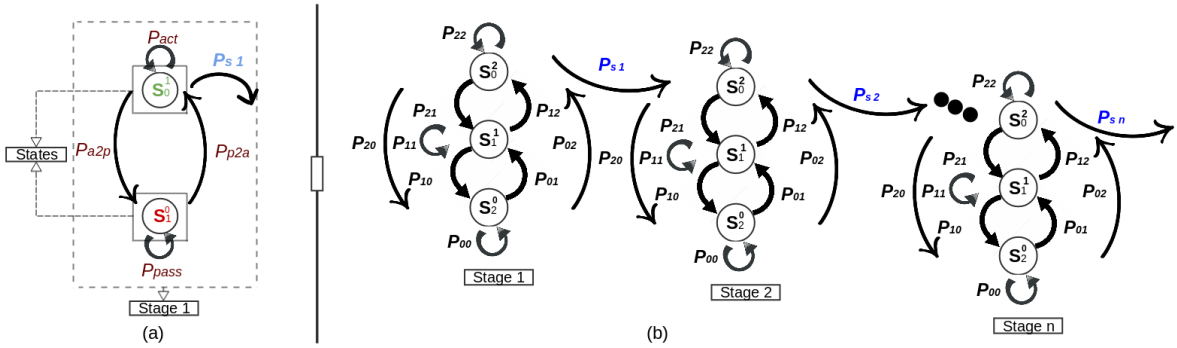

Fig. 4: Modeling (a) Base Case: single process with single stage, (b) Generic Case: two processes with multiple stages

process remaining passive.

The probabilities in the base case model are parameterized by instrumenting the system details, which will further be used to derive the probabilities of the generic case. In order to parameterize the probabilities of base case, we use `perf tool` [3] 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 constructive work is being performed by the process. Therefore, we use Eq. 1 to get P_{act},

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

where U is utilization per process. As shown in Figure 4 (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 S_1^0 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 Eq. 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 Eq. 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 Eq. 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 Eq. 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 will discuss the calculation of Ps_1, which is then not the same as $1/TC$.

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.

Figure 4 (b) shows an underlying Markov model for an application with two processes using n multiple stages. Thus, in a similar way, the entire work flow 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 4 (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 $\mathbb{M} = t + 1$ states. Thus, the transition probabilities of jumping from any one of these $\mathbb{M}$ states to other states or itself can be divided into 3 types: 1) probability to remain in the same state (e.g., P_{22}, P_{11} and P_{00} in Figure 4 (b)), 2) probability to increase active processes (e.g., P_{01}, P_{12}, P_{02} in Figure 4 (b)), and 3) probability to increase passive processes (P_{10}, P_{21}, P_{20} in Figure 4 (b)). Given $\mathbb{M}$ states, we have $\mathbb{M}$ probabilities to remain in the

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

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 $\mathbb{M} = 2$ (base case), we can calculate all probabilities for a generic case with $\mathbb{M} > 2$. In [4], a mathematical relation between 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 $\mathbb{M} = 2$ and $\mathbb{M} > 2$.

Eq.s 6 - 8 give the state transition probabilities for $\mathbb{M} > 2$, 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 4 (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). 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 Eq.s 6 - 8, we use the probabilities (P_{act}, P_{a2p}, P_{pass} and P_{p2a}) that are obtained by the base case (Sec. II-B1) to calculate the state transition probabilities in generic cases.

$$P_{ii} = \sum_{\substack{x+y<=t-i,\\ x>=0,\\ y<=i\\ x=t-i,\\ y=t-i-x,\\ x--,\\ y++}} \binom{i}{y}.\binom{t-i}{x}.(P_{act}^{i-y}).(P_{a2p}^{y}).(P_{pass}^{x}).(P_{p2a}^{t-i-x}) \quad (6)$$

$$P_{ij}(i<j) = \sum_{\substack{x+y<=t-i,\\ x>=0,\\ y<=t-j\\ x=t-j,\\ y=0,\\ x--,\\ y++}} \binom{i}{y}.\binom{t-i}{x}.(P_{act}^{i-y}).(P_{a2p}^{y}).(P_{pass}^{x}).(P_{p2a}^{t-i-x}) \quad (7)$$

$$P_{ij}(i>j) = \sum_{\substack{x+y<=t-j,\\ x>=0,\\ y<=t\\ x=t-i,\\ y=i-j,\\ x--,\\ y++}} \binom{i}{y}.\binom{t-i}{x}.(P_{act}^{i-y}).(P_{a2p}^{y}).(P_{pass}^{x}).(P_{p2a}^{t-i-x}) \quad (8)$$

Additionally, after capturing the state transition probabilities of the first stage, we calculate the stage completion probability Ps_1 using Eq. 9 and then use Ps_1 as an incoming probability for calculating the state transition probabilities of stage 2, and so on. This chain process captures an iterative and multi-stage application running with multiple processes. Finally, Ps_n for the n^{th} stage is calculated by Eq. 9, where Z represents a random variable over stages.

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

We further use Eq. 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).

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

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 which assists to extrapolate the data (such as, F, U, TC and SC) required for calibrating the base case model.

C. Machine Learning Model

Our stochastic Markov model allows us to predict the calculation time of an application when we have 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. To reduce the complexity of the machine learning model, we also assume that the application execution time is dependent on the fewest possible hardware parameters. Our evaluation results shown in Section III demonstrate the feasibility of this assumption by showing the fairly accurate predicted results obtained by our approach which 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 an encapsulated emulation of hardware is the key for allowing the approach to be able to predict parameters for datasets with sizes much larger than those of the training datasets.

1) Regression Mapping: The focus of regression is to find the relationship between a dependent variable (such as the hardware parameters which 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, gradient decent, etc.) for modeling all desired hardware parameters (U, TC and SC). We will show the validation of linear regression model in Sec.III-C. The linear regression equation for learning variable y_i is shown in Eq. 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 number of desired clusters (K), 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.

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

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.

2) Iterative Improvement Model: However, 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 execution time of large datasets based on small training datasets. In order to handle this issue, we develop an iterative improvement model which uses a sensitivity parameter α to tune the constant factors $\vec{\beta_i}$ with respect to the predicted results (i.e., exeution time) of our stochastic Markov model as shown in Eq. 12. Note that in this equation the constants in vector $\vec{\beta_i}$ are obtained from 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.

$$y_i = \alpha(\beta_{i0} + \beta_{i1}x_{i1} + ... + \beta_{ip}x_{ip}) \quad for\ i = 1, 2, ..., n \tag{12}$$

Initially, we use hardware parameters (U, TC and SC) and application parameters of training data with regression mapping (Eq. 11) to obtain constants of vector $\vec{\beta_i}$. The hardware parameters are passed to our stochastic Markov model to predict execution time. In the first iteration, the absolute error between actual and predicted time is used to decide the adjust direction of α, see lines 6 to 8. That is, if actual value is greater than the predicted one, then the algorithm plans to increase α and vice verse. 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 Eq. 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. Thus, the algorithm adjusts the value of α until the predicted execution time becomes close to the actual one. 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 thereafter used for extrapolation of hardware parameters.

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 absolute error (Actual, Predicted)
8          Decide OP = + or −, depending on positive or
           negative absolute error
9      if RMS error <τ then
10         return
11     else
12         α = α {OP} ε
13         iter ++
14         Calculated TC, SC and U using Eq. 12
15         goto line 3
16 else
17     Neglect bad values
18     goto line 4
```

In summary, Figure 2 shows the overall procedure of our prediction model FiM, 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.

III. Evaluation

We built a distributed cluster using MPI installed HPC nodes as shown in Figure 3. We evaluate our FiM prediction approach (a combination of stochastic Markov modeling and machine learning regression), by comparing the predicted results (e.g., parallel calculation 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. We use Discovery Cluster at Northeastern University [6] to build our experimental platform. Table II gives a description of five parallel platform configurations we used in our evaluation, where each CPU belongs to different compute nodes.

We evaluate our approach with six different NAS Parallel Benchmarks (NPB - version *NPB3.3.1-MPI*) [7], with the large size dataset of *Problem Class C*. The benchmarks, Block Tri-diagonal solver (BT) and Embarrassingly Parallel (EP) are compute bound, Scalar Penta-diagonal solver (SP) and Lower-Upper Gauss-Seidel solver (LU) are I/O bound, and Integer Sort (IS) and Conjugate Gradient (CG) are memory bound. 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 datasets of *Problem Class C* which are much larger than the datasets of *Problem Class S*. We reprogram the NPB benchmarks to implement its iterative or multi-stage, master-compute paradigm version 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 [8] and Pagerank. For each of these two applications, we run experiments on 15 different datasets and choose one dataset as 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).

TABLE II: Platform Configurations (2-E52670 - Two Multi-Core, Hyper-Threaded Intel Xeon E5 2670 CPU's @ 2.60 GHz and 256 GByte of RAM) (2-E52650 - Two Multi-Core, Hyper-Threaded Intel Xeon E5 2650 CPUs @ 2.00 GHz and 128 GByte 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	10 Gb/s Ethernet backplane TCP/IP				
Shared FS	NFS				
OS	Linux				

K-means Clustering (KM): Our K-means clustering implementation [8] 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. The parameters of K-means include number of desired clusters (K), number

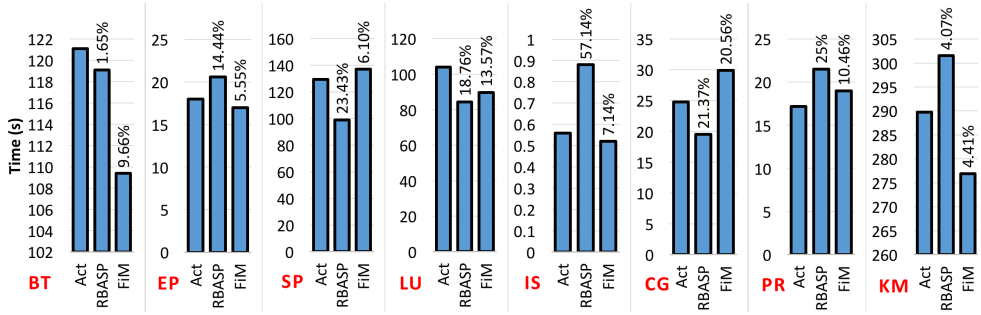

Fig. 5: Actual and predicted execution time using FiM and RBASP with the relative prediction error listed on top of each bar

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). The parameters of this application include number of vertexes (V), number of iterations (I), size of input dataset (N) and network nature (dense or sparse).

A. Performance Evaluation

In our evaluation, we consider a regression-based approach named RBASP [5] to compare with our FiM approach. We choose RBASP because it is well known for its simplicity and accuracy in extrapolating execution time of multi-process applications. In our FiM approach, we combine the stochastic Markov modeling with the machine learning regression to reduce prediction error in the most cases. The 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 $q \in \{1, ..., p_0\}$ and $p_0 < p$. By varying the values of independent variables $(x_1, x_2, ..., x_n)$, this model aims to calculate coefficients $(\beta_0, ..., \beta_n)$ by the linear regression fit for $log_2(y)$ (Eq. 13), where $g(q)$ can be either a linear function or a quadratic function.

$$log_2(y) = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n + g(q) \qquad (13)$$

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 [5]. The RBASP approach directly predicts the execution time using regression, which requires to perform training with data points processed for different number of multiple processes. But, 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 different number of processes. That is, FiM does not required to be trained again when we change the number of processes in that 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 II), using the actual optimal number of processes listed in Table III. 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 KM. For all the remaining benchmarks and applications, FiM performs better with the prediction error less than 20%. Furthermore, we find that RBASP's prediction is limited to the fixed application parameters on which the model is trained, while FiM can predict execution time without any prior training for new application parameters.

TABLE III: Summary of results for all applications (Rel. Er.-Relative Error) (Act.-Actual) (App.-Application) (Opt.-Optimal)

	Best Rel. Er. %		Worst Rel. Er. %		Opt. # MPI Process		
App.	RBASP	FiM	RBASP	FiM	RBASP	FiM	Act.
BT	1.62	1.5	59.99	20.25	104	320	320
EP	0.59	2.41	76.74	18.96	300	242	256
SP	1.75	0.39	89.40	22.86	256	64	32
LU	0.01	0.04	81.96	23.74	64	32	16
IS	1.36	1.42	89.71	35.41	70	70	70
CG	1.96	3.74	61.83	39.14	192	96	96
PR	1.45	0.85	49.36	27.23	38	52	60
KM	1.01	0.51	28.78	11.21	72	192	192

Table III lists the best and worst prediction errors as well as the actual and predicted optimal numbers of processes using RBASP and FiM. We notice that 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 the poor adaptability to the change in the values of attributes (e.g., number of processes). 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 high correlation in the range of 0.81 to 0.99, which indicates considerable accuracy of our FiM estimation technique.

B. Sensitivity Analysis

One important 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. The derived model is then used to predict the runtime for new datasets (including both small and large ones) and new sets of application parameters. Therefore, we perform sensitivity analysis on different dataset sizes and

[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.

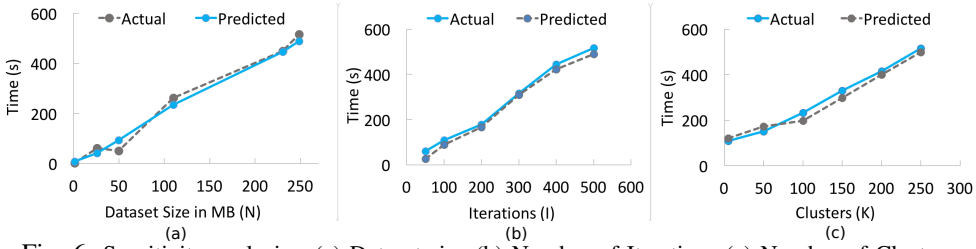

Fig. 6: Sensitivity analysis - (a) Dataset size (b) Number of Iterations (c) Number of Clusters

application parameters. We argue that if the user has flexibility to choose application parameters for achieving optimal performance, our model then can provide the useful guidance. Our model can predict the performance under different sets of application parameters and help 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 quite useful to use our 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 calculation time for the K-means algorithm as a function of (a) dataset size, (b) number of iterations, and (c) number of clusters are plotted in Figure 6. We experiment with different hardware configurations as shown in Table II, and present the results of the C5 configuration here. In these experiments, we also use the predicted optimal number of processes listed in Table III. For each plot in Figure 6, we do a sensitivity analysis on one parameter and fix the remaining two parameters with dataset size of 250 MB, 500 iterations and 250 clusters. We observe that our models can accurately predict the execution time even when the datasets become large, see Figure 6(a). Note that we only use small datasets to train our models. Figure 6 (b) shows a linear increase of execution time with increasing number of iterations. Figure 6 (c) further shows execution time with respect to the increase in number of clusters. Summarizing from Figure 6, we can see that FiM consistently achieves predicted results in a good agreement with actual ones under different application parameters like dataset size, number of iterations and number of clusters.

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 II). Figure 7 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., 192 for K-means and 60 for Pagerank. FiM is able to accurately predict the optimal performance point for both applications. Predicted results match well with actual ones across different number of processes.

Figure 8 plots the actual and predicted optimal number of processes for Pagerank using the five different hardware configurations listed in Table II. The model of FiM is calibrated under each of these hardware configurations. We observe that when the platform consists of more distributed cores i.e., from C1 to C5, the optimal number of processes tends to increase in order to reduce calculation time. But, we also

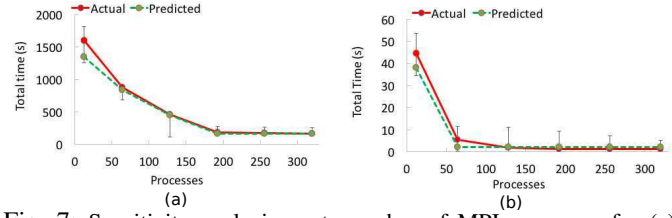

Fig. 7: Sensitivity analysis w.r.t. number of MPI processes for (a) K-means and (b) Pagerank

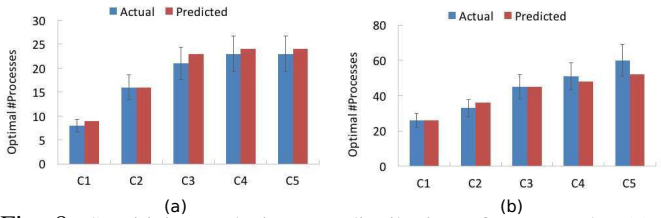

Fig. 8: Sensitivity analysis w.r.t. distribution of cores under (a) a small dataset with 40 vertices and (b) a large dataset with 4039 vertices

notice that when the dataset is small (i.e., Figure 8(a)), the number of optimal processes converges (i.e., the execution time is not decreasing) even when more distributed cores are added. These estimation results can thus provide us insight thoughts regarding the scalability of an application on a multi-core computing platform.

C. 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 the results by using K-means clustering as a representative. Recall, for K-means, we choose the linear regression to extrapolate hardware variables as a function of application parameters (I, N and K), see Section II. Actually, there are a variety of regression models that can be used. It is not straightforward to choose the right regression model that are best suitable for our requirement. Even a too complex model might over-fit the training data, generating a very large prediction error on other datasets. On the other hand, a simple model may under-estimate the learning trends and thus produce incorrect predicted results [9]. Considering these two cases, we choose the linear regression model after examining other regression models such as piece-wise linear model, Poisson models and quadratic models with different degrees of polynomial.

We investigate the learning trend of three hardware variables under different settings of application parameters. Figure 9 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 9, linear surfaces can be found for different hardware variables. We can thus conclude that

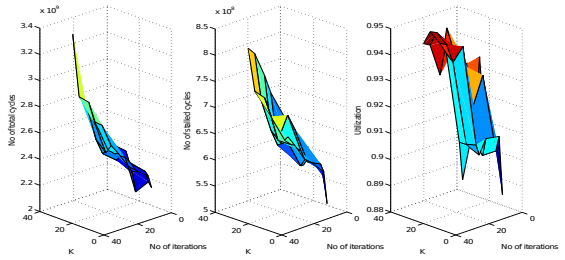

Fig. 9: Regression mapping (K - number of clusters, I - number of iterations)

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.

IV. Related Work

Modeling helps to shorten the development cycle by providing the necessary insights to obtain optimal performance. Some recent researches [10], [11] on performance prediction are more concerned to improve database workloads, but with increase in scientific data computations, we also need to explore performance prediction of HPC workloads. It can be approached in several different ways, including empirical evaluation, simulation and analytical modeling. Empirical evaluation requires the exact implementation as well as similar hardware to the target because results are based on observed ground truth. This technique was popular in the past [12], when computer hardware was stable over long periods.

Although simulators like SimpleScalar [13] and CACTI [14] can predict with high accuracy, they also consume a long time in order to give predicted results. Their slow running time and infrastructural 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 architecture [15], [16]. This type of model can be evaluated quickly and easily, but are comparatively less accurate than empirical and simulation based models because they lack accurate hardware machine modeling.

Predicting the performance of any parallel processing platform consists of a parallel computation phase. Markov chain modeling using probabilistic distribution assists in predicting the calculation load of multi-process and multi-core architectures [4], [17], [18]. 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 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.

V. Conclusions

In this paper, we present a novel performance modeling technique (FiM) to predict the computation time of iterative, multi-stage scientific data processing applications running on high performance cloud computing platforms. We combine two modeling techniques, specifically stochastic Markov modeling and machine learning techniques, to predict the parallel computation time. FiM estimates the time required for parallel data calculation across a range of input datasets, application configurations and parallel hardware parameters such as number of processes. We demonstrate that FiM can assist system designers and application programmers to choose optimal processing parameters and application parameters. More importantly, our prediction model is trained using small datasets but can predict accurately for large datasets. In the future, we plan to extend FiM's ability to predict computation for other platforms and accelerators, such as Spark, GPUs, etc. We also plan to evaluate our models for MPI applications in Amazon EC2 and consider different performance interference parameters to evaluate the effectiveness of our prediction technique in virtualized cloud systems.

References

[1] M. Snir, *MPI–the Complete Reference: The MPI core*. MIT press, 1998, vol. 1.

[2] W. Gropp and E. Lusk, "User guide for mpe: Extensions for mpi programs," *Argonne National Laboratory*, 1998.

[3] A. C. de Melo, "The new linux perf tools," in *Slides from Linux Kongress*, 2010.

[4] R. Mitra, B. Joshi, A. Ravindran, A. Mukherjee, and R. Adams, "Performance modeling of shared memory multiple issue multicore machines," in *ICPPW*. IEEE, 2012, pp. 464–473.

[5] B. J. Barnes, B. Rountree, D. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*. ACM, 2008, pp. 368–377.

[6] "Discovery cluster overview," May 2015. [Online]. Available: http://nuweb12.neu.edu/rc/?page_id=27

[7] "Nasa advanced supercomputing division." [Online]. Available: http://www.nas.nasa.gov/publications/npb.html

[8] J. Bhimani, M. Leeser, and N. Mi, "Accelerating k-means clustering with parallel implementations and gpu computing," in *HPEC*. IEEE, 2015, pp. 1–6.

[9] M. Yuan and Y. Lin, "Model selection and estimation in regression with grouped variables," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 68, no. 1, pp. 49–67, 2006.

[10] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, 2016, pp. 363–378.

[11] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan, "Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs," in *35th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2016.

[12] P. Gibbons, Y. Matias, and V. Ramachandran, "The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms," *SIAM Journal on Computing*, vol. 28, no. 2, pp. 733–769, 1998.

[13] Austin, Larson, and Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[14] S. Wilton and N. Jouppi, "CACTI: An enhanced cache access and cycle time model," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 5, pp. 677–688, 1996.

[15] W. Chen, J. Zhai, J. Zhang, and W. Zheng, "LogGPO: An accurate communication model for performance prediction of MPI programs," *Science in China Series F: Information Sciences*, vol. 52, no. 10, pp. 1785–1791, 2009.

[16] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman, "LogGP: incorporating long messages into the LogP modelone step closer towards a realistic model for parallel computation," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*. ACM, 1995, pp. 95–105.

[17] X. Chen and T. Aamodt, "A first-order fine-grained multithreaded throughput model," in *HPCA*. IEEE, 2009, pp. 329–340.

[18] R. Saavedra-Barrera and D. Culler, *An analytical solution for a markov chain modeling multithreaded execution*. Citeseer, 1991.