Performance Prediction for Large-Scale Parallel Applications Using Representative Replay

Jidong Zhai, Wenguang Chen, Weimin Zheng, and Keqin Li, *Fellow, IEEE*

Abstract—Automatically predicting performance of parallel applications has been a long-standing goal in the area of high performance computing. However, accurate performance prediction is challenging, since the execution time of parallel applications is determined by several factors, such as sequential computation time, communication time and their complex interactions. Despite previous efforts, accurately estimating the sequential computation time in each process for large-scale parallel applications remains an open problem. In this paper, we propose a novel approach to acquiring accurate sequential computation time using a parallel debugging technique called deterministic replay. The main advantage of our approach is that we only need a single node of a target platform but the whole target platform does not need to be available. Therefore, with this approach we can simply measure the real sequential computation time on a target node for each process on by one. Moreover, we observe that there is great computation similarity in parallel applications, not only within each process but also among different processes. Based on this observation, we further propose *representative replay* that can significantly reduce replay overhead, because we only need to replay partial iterations for representative processes instead of all of them. Finally, we implement a complete performance prediction system, called PHANTOM, which combines the above computation-time acquisition approach and a trace-driven simulator. We validate our approach on both traditional HPC platforms and the latest Amazon EC2 cloud platform. On both types of platforms, prediction error of our approach is less than 7 percent on average up to 2,500 processes.

Index Terms—Deterministic replay, high performance computing, MPI, parallel applications, performance prediction, trace-driven simulation

1 INTRODUCTION

1.1 Motivation

AUTOMATICALLY predicting and modeling performance of parallel applications has been a long-standing goal in the area of high performance computing (HPC) [1], [2], [3], [4], [5], [6], [7], [8]. Accurate performance prediction of parallel applications has many important uses.

Today, large-scale parallel computers consist of tens of thousands of processor cores and cost millions of dollars which take years to design and implement. For designers of these computers, it is critical to answer the following question at the design phase.

What is the performance of application X on a parallel machine Y with 10,000 nodes connected by network Z?

Accurate answer to the question above enables designers to evaluate various design alternatives and make sure which design can meet the performance goal.

Recently, with introduction of cloud platforms targeting HPC applications, such as the Amazon EC2 cluster compute instances (CCIs) [9], public clouds have become a cost-effective choice for many scientific application users and developers [10]. To fit different use cases, public clouds provide a wide selection of optimized instance types. For example, Amazon EC2 provides a series of instances varying in CPU, memory and network capacity and gives users flexibility to choose the most appropriate mix of resources for their applications. However, for users on clouds, they always ask the following question to make a cost-effective decision.

Which type of cloud instances should I choose or how many instances should I apply for to execute my application?

However, accurate performance prediction of parallel applications[1] is difficult, because execution time of large parallel applications is determined by sequential computation time in each process, communication time, and their convolution. Due to complex interactions between computation and communication, prediction accuracy can be hurt significantly if either computation or communication time is estimated with notable errors. Despite previous efforts, it remains an open problem to estimate the sequential computation time in each process accurately and efficiently for large-scale parallel applications.

A lot of approaches have been proposed to estimate the sequential computation time for parallel applications. For model-based methods [1], [2], application signatures, including integer and floating-point instruction count, memory access patterns, etc., are collected on a *host platform* through instrumentation or hardware performance counters. Then a parameterized model is constructed to estimate the time for each of these operations according to the parameters of a *target platform* and give the estimation for each sequential computation unit.

However, with rising architecture and software complexity, the prediction accuracy of model-based approaches is

1. Because message passing interface (MPI) is the dominant programming model in large-scale high performance computing, we use *parallel applications* to indicate parallel applications written in MPI in this paper. However, our approach is applicable to other message passing programming models.

• J. Zhai, W. Chen, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {zhaijidong, cwg, zwm-dcs}@tsinghua.edu.cn.
• K. Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.

Manuscript received 21 Sept. 2014; revised 23 Aug. 2015; accepted 25 Aug. 2015. Date of publication 16 Sept. 2015; date of current version 15 June 2016. Recommended for acceptance by J.D. Bruguera.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2479630

0018-9340 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

becoming increasingly compromised. For example, an out-of-order issue super-scalar processor can execute multiple instructions in parallel. Also, contention for shared resources, such as shared cache and memory bus, on multi-core platforms can result in complex program behavior. These factors make model-based approaches difficult to estimate the accurate sequential computation time.

Some researchers [3], [4] measure the sequential computation time of weak-scaling applications through executing the applications on a small-scale system of the *target platform*. For the weak-scaling applications, where the problem size is fixed for each process and the sequential computation does not change with the number of processes, the measurement above is sufficient to acquire the accurate computation performance.

However, the measurement-based approach fails to deal with large strong-scaling applications without the full-scale target platform, where the whole problem size is fixed and the sequential computation varies with the number of processes. For the strong-scaling applications, a few work [5], [11] uses regression-based approaches to extrapolating the computation performance for a large problem size. Unfortunately, the extrapolation is not always effective due to the non-linear behavior in real applications [12].

To conclude, previous approaches are not able to perform accurate sequential computation time estimation at affordable cost and time, especially for large strong-scaling parallel applications.

1.2 Our Approach and Contributions

In this paper, we propose a novel approach based on deterministic replay to solve the problem above. For readers not familiar with deterministic replay, please refer to Section 4.1 and [13], [14], [15], [16]. Our paper makes four main contributions.

1. *We first introduce deterministic replay to measure the sequential computation time of strong-scaling applications only using a single node of the target platform.* A main challenge of the previous measurement-based method is its inability to separately execute partial processes of a large strong-scaling application for effective measurement. As a result, the traditional method cannot obtain accurate sequential computation time without a full-scale target platform. To address this problem, we employ a parallel debugging technique, called deterministic replay, which enables us to execute any single process of a parallel application on a single target node without a full-scale target platform. So we can simply measure the real sequential computation time only with a target node for each process one by one.
2. *We employ sub-group replay to capture the effect of resource contention.* On currently multi-core servers, resource contention can significantly affect the application performance. To capture the contention effect of shared resources within a server, we propose sub-group replay, which replays *number-of-cores* processes simultaneously instead of one process to obtain accurate sequential computation time. Moreover, it should be emphasized that the replayed processes are executed at full speed. According to our experimental results, the replay-based execution is about two orders of magnitude faster than cycle-accurate simulation.
3. *We further propose representative replay to significantly reduce replay overhead.* Although with the approach proposed above, we can obtain accurate sequential computation time with a single node of the target platform, it still requires long measurement time if we replay a program with thousands of processes on a small number of target nodes. To address this problem, we further propose *representative replay*, which is based on our observation that computation behavior in parallel applications shows great similarity, not only within each process but also among different processes. Based on this observation, we partition the processes of a parallel application into a few groups, where processes in each group have similar computation behavior, and select representative processes from each group. For each selected process, we also replay partial iterations if there is also computation similarity between different iterations. *Representative replay* can significantly reduce replay overhead because we only need to execute partial iterations for selected parallel processes instead of all of them.
4. *We integrate the computation time acquisition approach above with a trace-driven network simulator for effective performance prediction.* We implement an automatic performance prediction system, called PHANTOM, which can predict application performance without a full-scale target platform. We validate our system on both traditional HPC platforms and the latest Amazon EC2 cloud platform [9]. On both types of platforms, prediction error of our approach is less than 7 percent up to 2,500 processes on average. We also compare PHANTOM with a cross-platform prediction method [8] and a regression-based prediction approach [5].

A preliminary version of this work has been published in PPoPP [17]. In this version, we improve the representative replay within a computing server node. We further reduce the replay overhead through exploring repetitive computation patterns within each process and propose a method of partial recording and replaying. Moreover, we extend our experimental results to 10 times the scale of the earlier version. We also validate our approach on the latest cloud platform with more applications. At last, we demonstrate the usage of PHANTOM with two applications.

2 Base Prediction Framework

We use a trace-driven simulation approach for the performance prediction. In our framework, we split the parallel applications into computation and communication parts, predict computation and communication performance separately and finally use a simulator to convolve them to get the execution time of the whole parallel application. It includes the following key steps.

1. *Collecting computation and communication traces.* We generate communication traces of parallel applications by intercepting all communication operations

```
real A(MAX,MAX), B(MAX,MAX), C(MAX,MAX), buf(MAX,MAX)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid...)
DO iter=1, N
    if (myid .gt. 0) then
        call MPI_RECV(buf(1, 1),num,MPI_REAL,myid-1,...)
    endif
    DO i=1, MAX
        DO j=1, MAX
            A(i,j)=B(i,j)*C(i,j)+buf(i,j)
        END DO
    END DO
    if (myid .lt. numprocs-1) then
        call MPI_SEND(A(1, 1),num,MPI_REAL,myid+1,...)
    endif
END DO
call MPI_FINALIZE(rc)
```

Fig. 1. An example of Fortran MPI program.

for each process, and mark computations between communication operations as sequential computation units. The purpose of this step is to separate communications and computations in parallel applications to enable us to predict them separately. Fig. 1 shows a simple MPI program and its computation and communication traces are given in Fig. 2a for two processes (The elapsed time for the kth computation unit of process x is denoted by `CPU_Burst (x,k)`.).

It should be noted that we only need partial communication information (e.g., message type, message size, source and destination etc.) and interleave of communication and computation in this step. All temporal properties are not used in later steps of performance prediction. A common approach of generating these traces is to execute parallel applications with instrumented MPI libraries. To further reduce the overhead in this step, ScalaExtrap [18] and FACT [19] can be used to generate traces of large-scale applications on small-scale systems.

2. *Acquiring the sequential computation time for each process.*The sequential computation time for each MPI process is measured through executing each process separately on a node of the target platform with replay techniques. We will elaborate it in Sections 4 and 5. For now, we just assume that we acquire the accurate computation time for each MPI process that can be filled into the traces generated in step 1. Fig. 2b shows the acquired sequential computation time for process 0 of the program in Fig. 1.
3. *Using a trace-driven simulator to convolve communication and computation performance.* Finally, a trace driven simulator, called SIM-MPI, is used to convolve communication and computation performance. As shown in Fig. 2c, the simulator reads trace files generated in step 1, the sequential computation time acquired in step 2, and the network parameters of the target platform, predicts the communication performance of each communication operation and convolves it with the sequential computation time to predict the execution time of the whole parallel application. We will elaborate this step in Section 6.

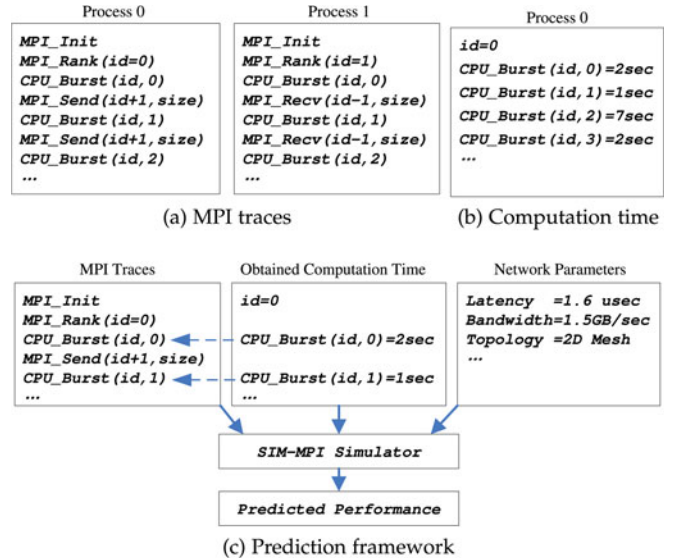


Fig. 2. Base performance prediction framework.

3 DEFINITIONS

To illustrate our method more clearly, we give two key definitions for parallel applications. One is *communication sequence*, the other is *sequential computation vector*.

Definition 1 (Communication Sequence). *A communication sequence is a representation of communication patterns for a given parallel program, which records the main message information of each communication operation in chronological order for each parallel process.*

Communication sequence is first introduced by Shao et al. [20], which describes intrinsic communication characteristics of parallel applications. In our earlier conference version [17], we give an example of communication sequence for a simple parallel application.

Definition 2 (Sequential Computation Vector). *A sequential computation vector is a time vector that is used to record the sequential computation performance for a given process of a parallel application. Each element of the vector is the elapsed time of the corresponding computation unit.*

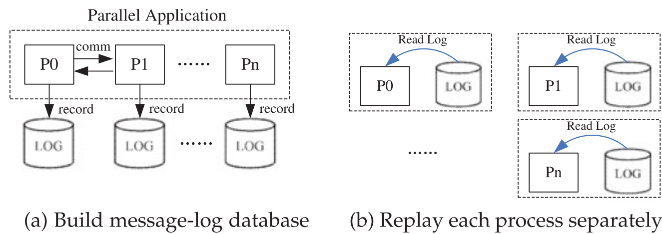
The sequential computation vector for process x is denoted by c^x: $c^x = [t_0, t_1, \ldots, t_m]$, where $t_k = (B_{k+1} - E_k)$, $k \geq 0$, and B_k and E_k are the time-stamps of entry and exit points for the kth communication operation respectively in process x. The dimension of the computation vector is the number of segmenting computation units for a given process, denoted by $dim(c)$.

4 SEQUENTIAL COMPUTATION TIME

In this section, we present our basic approach to acquiring the sequential computation time for a parallel application with deterministic replay.

4.1 Deterministic Replay

Deterministic replay [13], [14], [15], [16] is a powerful technique for debugging parallel applications. Deterministic replay normally includes two phases: *record* and *replay*. During the record, it records all return values and/or orders for



(a) Build message-log database (b) Replay each process separately

Fig. 3. Acquire the sequential computation time.

irreproducible function calls, such as incoming messages, during the application execution. During the replay, it replays faulty processes to any state of the recorded execution. Data replay [14], [16] is an important type of the deterministic replay for parallel applications. It records all incoming messages for each process during the application execution. With this approach, developers can execute **any single process separately** for debugging during the replay rather than have to execute the whole parallel application. We leverage this merit of data replay to execute any single process on one node of the target platform for performance prediction.

4.2 Acquiring Sequential Computation Time

In contrast to previous methods, our approach is based on data-replay techniques to acquire sequential computation time. Our replay-based approach requires two platforms. One is the *host platform*, which is used to collect message logs of applications like traditional data-replay techniques. The other is *one single node* of the target platform on which we will predict application performance. For homogeneous HPC systems, only one node of the target platform is sufficient for our approach. More nodes can be used to accelerate the replay. If the target platform is heterogeneous, at least one node for each architecture type should be available.

As shown in Fig. 3, there are two main steps to acquire the sequential computation time for a given application.

1) *Building message-log database:* Record all necessary information as in the data-replay tools when executing the application on the host platform and store this information to a message-log database. This step is only done once and the message-log database can be reused in the future prediction.
2) *Replaying each process separately:* Replay each process of the application on a single node separately and collect the elapsed time for each sequential computation unit. We will show more details below.

Building message-log database: This step is the same as the record phase in the data replay. All irreproducible information should be recorded during the application execution. We maintain a message-log database to record the data for different applications, which can be reused in the future prediction. This step can also be done during application development, which is reasonable when data-replay techniques are used for debugging.

In our data-replay system, we record the information above using the MPI profiling interface (PMPI), which requires no modifications of either applications or MPI libraries. During the record, our system intercepts each MPI computation operation, and then records the returned values and memory changes into log files, including all

```
int MPI_Recv (buf, count, type, src, tag, comm, status){
    int retVal = PMPI_Recv (buf, count, type, src,
                            tag, comm, status)
    Write retVal to log
    Write buf to log
    Write status to log
    return retVal
}
```

Fig. 4. An example of recording logs for `MPI_Recv`.

incoming messages of each process. For receive operations, we record contents of received messages, the return values of functions, and MPI routine status. For send operations, only the returned values of functions are recorded. For non-blocking receive operations, we maintain a table to map request handler to corresponding receive buffer and record message contents until invocation of corresponding `MPI_Wait` or `MPI_Waitall`. Fig. 4 gives an example to record logs for `MPI_Recv` routine.

Replaying each process separately: To acquire the sequential computation time for a given process, we just need to execute the process separately rather than execute the whole parallel application. The data-replay technique is able to execute particular process during the replay. The message-log database records the necessary information for replaying each process. To collect the sequential computation time, we insert two timing functions at the entry and exit points of communication operations in the replay system. An analysis program is used to calculate the final sequential computation time for each process. Fig. 5 shows an example for replaying `MPI_Recv` routine and recording time-stamps. B_k and E_k are the time-stamps of entry and exit points for the kth communication operations.

4.3 Subgroup Replay

On multi-core platforms or symmetric multi-processor (SMP) servers, resource contention can significantly affect the application performance. For example, Fig. 6 shows sequential computation performance of process 0 for NPB CG with eight processes when there are different numbers of processes executing on one server node. The server is equipped with two-way quad-core Intel Xeon E5345 processors. Note that when there are more processes executing on one server node, the sequential computation performance of process 0 changes dramatically.

To accurately capture the effect of resource contention on the application performance, we propose a method of *subgroup replay* in PHANTOM. During the replay, we replay a subgroup of processes simultaneously according to the number of processes executing on one node of the target platform. To reduce cost of recording message logs, we also integrate the subgroup reproducible replay technique [13] into

```
int MPI_Recv (buf, count, type, src, tag, comm, status){
    Record time-stamp(B_k)
    Read log to retVal
    Read log to buf
    Read log to status
    Record time-stamp(E_k)
    return retVal
}
```

Fig. 5. Replay `MPI_Recv` and record time-stamps.

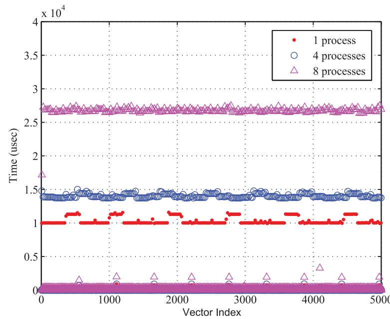
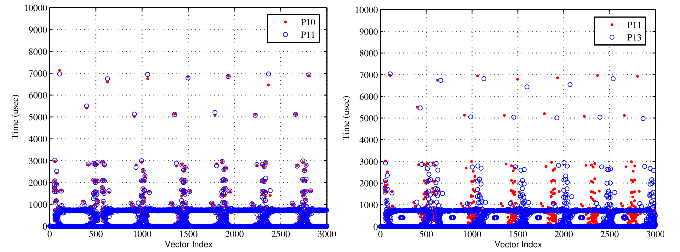


Fig. 6. Sequential computation performance of process 0 for NPB CG (CLASS $=$ B, NPROCS $= 8$) when there are different numbers of processes executing on one server node.



(a) Process10 vs. Process11 (b) Process11 vs. Process13

Fig. 7. The sequential computation time for NPB MG (CLASS $=$ C, NPROCS $=$ 16). The computation behavior of processes 10, 11 shows great similarity, while that of processes 11, 13 shows great difference.

PHANTOM. We let the processes executing on one node as a replay subgroup. Only the communications crossing subgroups are recorded, while the communications within a subgroup are not recorded.

Besides reducing the cost of recording the message log, a main advantage of *subgroup replay* is that the program execution mode during the replay is very similar to the real execution. The reason is that in *subgroup replay* intra-node communications are the same with the original execution and only the inter-node communications need to be read from the message log. As a result, the effect of resource contention can be accurately captured during the replay.

5 REPRESENTATIVE REPLAY

5.1 Challenges for Large-Scale Applications

The basic approach can already acquire the accurate sequential computation time for a strong-scaling application on a single node. However, there are two main challenges for large-scale parallel applications.

1. *Large time overhead:* For a parallel application with n processes, assuming that we replay one process at a time and the average time for replaying one process is T, it will take nT to obtain all the computation performance. However, this time complexity is impractical for an application with tens of thousands of processes.
2. *Huge log size:* As the data replay technique needs to record all the contents of incoming messages for each process, the log size will become increasingly large with the rising number of processes.

5.2 Observation

In this paper, we observe that there are two important computation properties in MPI-based parallel applications.

1. *Intra-process repetitive Computation:* For most of MPI-based parallel applications, iterative programming models are always used to solve a complex scientific problem. For each iteration, the application always shows periodic computation behavior.
2. *Inter-process similar computation:* Due to the single program multiple data (SPMD) characteristic of most MPI applications, program processes can be clustered into a few groups and the processes in each group execute the same program branch but with unique input data. These processes in the same group always have very similar computation behavior.

Based on our observation above, computation patterns are very common for most MPI applications. Several previous studies for analyzing MPI applications have also shown the similar computation behavior [3], [4], [6], [12]. For example, in NPB MG (CLASS = C) with 16 processes, the sequential computation time within a process shows great repetitiveness and has a periodic pattern every 500 computation units. Also, the computation behavior of a group of processes 0-3, 8-11 shows great similarity, and that of another group of processes 4-7, 12-15 also shows great similarity. The computation behavior of processes between two groups shows great difference. For the purpose of clear presentation, we only list the computation behavior of processes 10, 11, 13 in Fig. 7.

5.3 Measuring Computation Similarity

To measure the similarity degree of computation behavior for two given sequential computation vectors, we use vector distance to quantify them. There are several methods to calculate the distance of two vectors, such as euclidean distance and Manhattan distance. In this paper, we adopt Manhattan distance to measure the distance of two vectors. The reason is that it weights the difference for each dimension of two vectors more heavily. Hence, it is consistent with our objective of identifying computation patterns with the most similar behavior. For sequential computation vectors c^x and c^y, the distance is calculated as below:

$$\mathrm{Dist}(c^x, c^y) = \begin{cases} \sum_{i=1}^{m} |c_i^x - c_i^y| & \text{if } dim(c^x) = dim(c^y) \\ \infty & \text{if } dim(c^x) \neq dim(c^y), \end{cases} \quad (1)$$

where c_i^x and c_i^y are the ith elements of the sequential computation vectors. $dim(c^x)$ and $dim(c^y)$ are vector dimensions. When the dimensions of two sequential computation vectors are not equal, our trace-driven simulator regards them as different computation patterns. Hence, we set their distance infinity.

5.4 Representative Replay

Based on the observation above, we further propose *representative replay* to address the challenges listed in Section 5.1. Our idea is to partition the processes of applications into a number of groups so that the computation behavior of processes in the same group are as similar as possible, and select a representative process from each group. For the selected process, we only record and replay partial iterations according to applications' computation patterns. The acquired sequential computation time during the replay will be used for other processes in the same group.

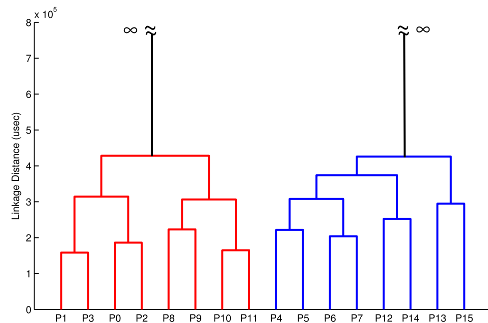


Fig. 8. Acquired dendrogram for NPB MG (CLASS = C).

5.4.1 Selecting Representative Processes

To identify the similar processes, we employ clustering techniques, which are very effective to analyze the complex nature of multivariable relationships. There are two main clustering approaches, i.e., K-means clustering and hierarchical clustering. K-means is efficient in computing, but it requires an *a priori* number for classification. It is suitable for users who have well understood applications' computation patterns. Hierarchical clustering is a general method for users with little knowledge about the application, which forms a final cluster through hierarchically grouping sub-clusters with a predefined distance metric. Both clustering techniques are supported in our framework.

The algorithm of the hierarchical clustering used in PHANTOM is listed in Algorithm 1. Complete linkage is used to measure inter-cluster distance in the hierarchical clustering (The complete linkage denotes the distance between the furthest points in two clusters.). The hierarchical clustering finally outputs a dendrogram, in which each level indicates a merge of the two closest sub-clusters. Fig. 8 shows the dendrogram for NPB MG with 16 processes. Depending on expected accuracy of final prediction, a horizontal line is drawn in the dendrogram to partition processes into a number of groups. For each group, the process that is closest to the center of the cluster is selected as representative process. Combining with the subgroup replay, we also need to replay those adjacent processes within the same node. Normally, we replay number-of-cores processes simultaneously to capture the resource contention. Although we replay more processes than representative processes, it does not introduce extra overhead due to concurrent execution.

Algorithm 1. Hierarchical Clustering in PHANTOM

1: **procedure** CLUSTERING
2: *Assign each process to its own cluster*
3: *Compute the inter-cluster distance matrix by Formula* 1
4: **repeat**
5: *Find the most closest pair of clusters (have minimal distance) and merge them into a new cluster*
6: *Re-compute the distance matrix between new cluster with each old cluster using complete linkage*
7: **until** *Only a single cluster is left*
8: **end procedure**

If the target platform has more processor cores than the host platform, we need to use the communication traces of the large-scale target platform acquired on the small-scale host platform to choose the representative processes. Our approach is to partition parallel processes into a number of groups so that the processes in the same group have completely identical communication sequences (e.g., message type, message size, message source and destination, and message tag etc.), which can be extracted from the collected communication traces. Moreover, we also need to perform computation similarity analysis on the host platform for the small-scale processes using the clustering algorithms and validate the above partition results.

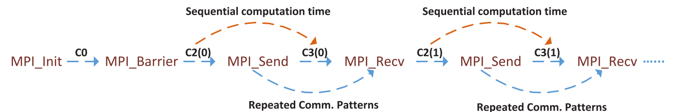


Fig. 9. Identifying periodic computation patterns.

5.4.2 Partial Recording and Replaying

To further reduce the cost of recording and replaying representative processes, we explore repetitive computation patterns within each process and propose a method of *partial recording and replaying*. Our idea is that we only record partial iterations of the representative processes when periodic computation patterns exist in these processes and use the acquired computation time of these iterations to predict the whole application performance.

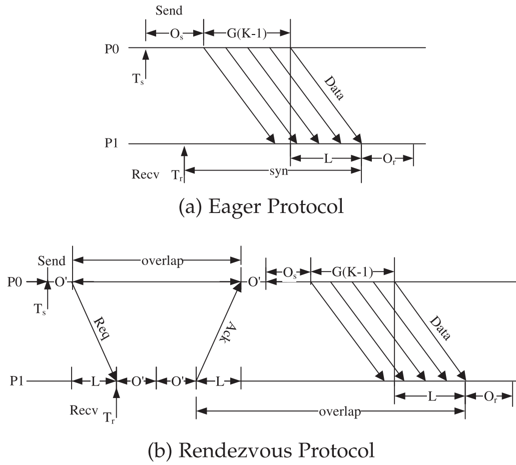
Our method includes the following key steps. First, we collect communication traces for the representative processes, which record the message type, size, source, destination, and so on. Second, we use a *sliding window* to traverse the communication traces and identify repeated communication patterns. Third, we calculate computation similarity for those sequential computation vectors within the repeated communication patterns. Finally, according to the prediction precision, we decide whether the partial recording mechanism is enabled. We also need to record total iteration count and recorded iteration count for performance prediction.

We use an example to describe the method above. In Fig. 9, we list communication traces for one process of a simple MPI program. $c0$, $c2(k)$, and $c3(k)$ $(0 \leq k < n)$ denote the sequential computation time between communication operations. We can identify that `[MPI_Send, MPI_Recv]` is a repetitive communication pattern in this process. c^k is the computation vector within this communication pattern, $c^k = [c2(k), c3(k)]$ $(0 \leq k < n)$. We calculate computation similarity for continuous computation vectors using Equation (1). We enable *partial recording and replaying*, if the sum of the vector distances is less than a predefined threshold defined in Equation (2), i.e., $\sum_{k=0}^{n-1} Dist(c^k, c^{k+1}) \leq L$, where L is defined below:

$$L = k\% \frac{1}{n} \sum_{x=1}^{n} \sum_{i=1}^{m} c_i^x. \tag{2}$$

6 CONVOLVING COMPUTATION AND COMMUNICATION PERFORMANCE

In order to convolve computation and communication performance, we design and implement a trace-driven simulator, called SIM-MPI. SIM-MPI consumes computation and communication traces of a given parallel application and



(a) Eager Protocol

(b) Rendezvous Protocol

Fig. 10. Convolve computation and communication in SIM-MPI.

the network parameters of the target platform and outputs the execution time of the application.

For the underlying machine model, we extend LogGP [21] model, considering synchronization overhead and overlap between computation and communication in MPI applications. The LogGP model abstracts the communication performance by five parameters: communication latency (L), overhead (o), bandwidth for small messages ($1/g$), bandwidth for large messages ($1/G$), and the number of processors (P). The gap (g) has little effect on the communication cost for high-level communication routines. Therefore, we disregard g in SIM-MPI. We add two additional parameters in LogGP model, the synchronization overhead (syn) and the overlap length ($overlap$).

SIM-MPI supports two important communication protocols, i.e., eager and rendezvous. For the eager protocol, messages are transmitted without regard to the receive process's state. If a matching receive operation has not been posted, messages need to be buffered at the receive process. In SIM-MPI, we distinguish send overhead and receive overhead, o_s and o_r. Therefore, the minimal communication overheads incurred by the send and receive processes are o_s and o_r respectively. The total end-to-end communication cost of sending and receiving a message can be modeled as $o_s + G(K-1) + L + o_r$ (K is message size). In Fig. 10a, T_s and T_r are arrival times for send and receive operations. If the receive operation arrives early, the synchronization overhead, syn, is incurred at the receive process. In both send and receive processes, there is $G(K-1) + L$ communication time that can be overlapped with useful computation.

For the rendezvous protocol, it needs a negotiation for the buffer availability before messages are actually transferred. We assume that communication overhead for sending or receiving a control message is o'. Therefore, the communication overheads introduced at send and receive processes are $2(o' + L + o') + o_s$ and $o' + o' + L + o' + o_s + G(K-1) + L + o_r$ respectively. If synchronization overhead is not introduced, the total end-to-end communication cost is $2(o' + L + o') + o_s + G(k-1) + L + o_r$. Similarly, if the receive operation arrives early, the synchronization overhead, syn, is incurred at the receive process. Otherwise, the synchronization overhead is introduced at the send process. There is $L + o' + o_s + G(K-1) + L$ communication time that can be overlapped at the receive process and $2(o' + L) + G(K-1) + L$ at the send process (denoted by *overlap* in Fig. 10b).

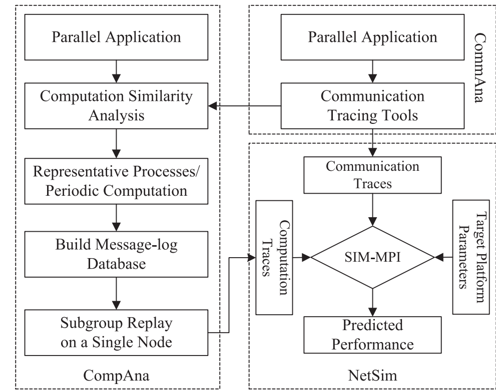


Fig. 11. Overview of PHANTOM.

For the sequential computation time, SIM-MPI replicates the computation time of representative processes for other processes in the same group. For each process, SIM-MPI maintains a virtual clock, stating from zero. SIM-MPI adds the sequential computation time and the communication overhead to the virtual clock based on aforementioned communication protocols. When *partial recording and replaying* is enabled, SIM-MPI needs to calculate the execution time with the recorded iteration count and the total iteration count. Finally, it outputs the execution time for each process.

In most of MPI implementations, non-blocking communications are used to achieve computation and communication overlap. However, different MPI implementations can cause different overlap ratios. These factors have been considered in SIM-MPI. Furthermore, collective communications are simulated through decomposing them into a series of point-to-point communications according to their algorithms [22].

7 PHANTOM PREDICTION FRAMEWORK

Fig. 11 shows overall performance prediction framework, called PHANTOM. PHANTOM is an automatic tool chain not requiring users to understand the detailed algorithm or implementation of a given parallel application, and it consists of three main modules, *CompAna*, *CommAna*, and *NetSim*.

CompAna module is responsible for acquiring the sequential computation time for each process. First, it acquires communication traces and computation patterns for a given parallel application. The computation patterns of each process are collected on a host platform. Second, it analyzes computation similarity and selects representative processes. Note that if the target platform has more processor cores than the host platform, we need to combine both the computation similarity patterns acquired on the host platform and the communication sequences for the large-scale target platform to decide the representative processes. Third, for the selected representative processes, it identifies periodic computation patterns within these processes and determines whether partial recording and replaying is enabled. Fourth, it builds the message-log database for the representative processes on the host platform. Finally, it uses a single node of the target platform to replay representative processes and outputs the computation traces. *CommAna module* is

TABLE 1
Experimental Platforms Used in the Evaluation

System	Amazon EC2	Nebulae	Dawning	DeepComp-F	DeepComp-B	Explorer	Explorer-100
CPU type	Intel E5-2670	Intel X5650	AMD 2350	Intel X7350	Intel E5450	Intel E5345	Intel E5-2670
CPU speed	2.6 GHz	2.66 GHz	2.0 GHz	2.93 GHz	3.0 GHz	2.33 GHz	2.6 GHz
# cores/node	16	12	8	16	8	8	16
# nodes	8	209	32	16	128	32	100
Memory	60.5 GB	24 GB	16 GB	128 GB	32 GB	8 GB	32 GB
Network	10 G.E.	IB QDR	IB DDR	IB DDR	IB DDR	IB DDR	IB QDR
Shared FS	NFS	Lustre	NFS	StorNext	StorNext	NFS	NFS

responsible for collecting communication traces. The communication traces can be acquired using the traditional tracing methods or any recent work such as ScalaExtrap [18] and FACT [19]. In the *NetSim module*, the computation and communication traces generated by previous two modules are fed into SIM-MPI simulator. SIM-MPI outputs the final performance prediction result of the application.

8 EVALUATION

8.1 Methodology

Experimental Platforms. Table 1 shows the experimental platforms used to evaluate our approach. For traditional HPC platforms, we use six platforms with a variety of system configurations, Nebulae, Dawning, DeepComp-F, DeepComp-B, Explorer, and Explorer-100. Among these systems, Nebulae was ranked number 2 in the June 2010 TOP500 list. For cloud platforms, we use the latest Amazon EC2 platform, with node type *cc2.8xlarge* of CCIs [9]. Both *Explorer* and *Explorer-100* are used as the host platforms to collect the message logs and the message traces.

Benchmarks. We evaluate PHANTOM with 6 NPB programs [23], eight applications of the SPEC MPI2007 benchmark suite [24], ASCI Sweep3D [25], and NWChem [26]. For NPB programs, the version is 3.3 and input data set is CLASS E. For SPEC MPI2007, we use the medium-sized reference configuration. For Sweep3D, the grid size is $512 \times 512 \times 200$. For NWChem, the input set is *c240_pbe0.nw*.

Comparison. In this paper, we compare the prediction accuracy of PHANTOM with two state-of-the-art approaches, a cross-platform prediction proposed by Yang et al. [8] and a regression-based model proposed by Barnes et al. [5].

Cross-platform performance prediction utilizes partial execution for a limited number of timesteps to compute the relative performance across platforms for an application. This novel approach is observation-based. On real platforms, this method demonstrates very high accuracy at a low cost for several real-world applications. In this paper, we implement two strategies of the cross-platform prediction to compute the relative performance. One is using a full target platform and the other is using a subset of the target platform (256 processors). We use *Explorer-100* as the host platform and *Nebulae* as the target platform.

Regression-based model uses several program executions on a small subset of the processors to predict the execution time on larger numbers of processors. They explore three different regression techniques for effective prediction and show that the lowest root-mean-squared-error generally provides the best prediction. In this paper, we repeat different regression approaches and only report their best result for each case.

8.2 Sequential Computation Time

8.2.1 The Number of Representative Replay Groups

Table 2 shows the results of the number of process groups that have similar computation behavior for each program with different numbers of processes (For BT and SP, the number of processes is 256, 400, 1,024, 1,600, and 2,500.). The grouping strategy is described in Section 5.4.1. The computation patterns of the applications are collected on *DeepComp-B platform*. The results can be classified into three categories: 1) For BT, CG, EP and SP, all the processes have the similar computation behavior for different numbers of processes. 2) For LU and Sweep3D, the number of groups keeps constant with the number of processes. 3) For MG, the number of groups increases with the number of processes. However, the number of groups grows much slower than the number of processes. The experimental results confirm our observation that most processes in parallel programs have the similar computation behavior.

An interesting finding observed in our experiments is that the processes having completely identical communication sequences always process the same amount of computation and therefore have similar computation behaviors. For example, Fig. 12 lists partial communication sequences for LU with 16 processes. We put the processes that have identical communication sequences into the same group. There are nine groups in total. After measuring computation similarity for each group, we find that the processes in the same group also have similar computation behavior. Based on this observation, when the host platform does not have enough processor cores to obtain the sequential computation vector, we can leverage the communication sequences to group parallel processes in MPI applications.

8.2.2 Validation of Sequential Computation Time

We validate all the sequential computation time acquired using our approach with the real sequential computation performance measured on the target platform. Results show that our approach can get accurate sequential computation time. The average error for the acquired sequential

TABLE 2
The Number of Representative Replay Groups

Proc. #	BT	CG	EP	LU	MG	SP	Sweep3D
128	1	1	1	9	12	1	9
256	1	1	1	9	18	2	9
512	1	1	1	9	27	1	9
1,024	1	1	1	9	36	1	9
2,048	1	1	1	9	48	1	9

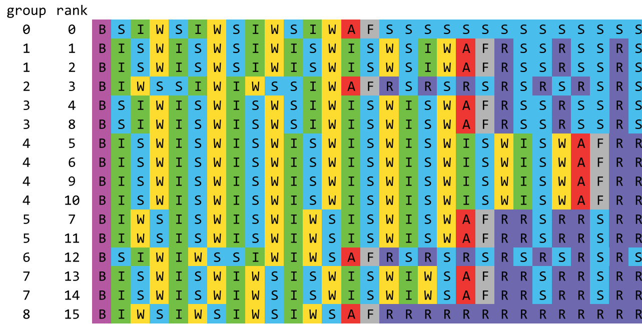


Fig. 12. Partial communication sequences for LU with 16 processes. (B=MPI_Bcast, S=MPI_Send, I=MPI_Irecv, W=MPI_Wait, A=MPI_Allreduce, F=MPI_Barrier, R=MPI_Recv).

computation time is less than 5 percent for all the programs. The main difference between the replay-based execution with the normal execution is the sources of incoming messages. During the replay-based execution, received messages are read from the message logs. We find that the operations of reading logs has little effect on the application performance. The *subgroup replay* can effectively capture computation contention within one server node. In the next Section, we will further show detailed results of prediction accuracy using the acquired sequential computation time.

8.2.3 Analysis of Sequential Computation Time

To study the characteristics of the sequential computation time in parallel applications, we present the acquired sequential computation time of process 0 for BT, LU and SP on *Nebulae platform* (vector size = 2,000) in Fig. 13. We find that the sequential computation time within each process for these programs shows great repetitiveness. For example, the sequential computation vector in BT has a periodic pattern every 250 computation units. Through analyzing its source codes, we find that the periodic computation is generated by repeated loop iterations. We leverage this computation property and enable *partial recording and replaying* to reduce replay time for representative processes.

8.3 Performance Prediction for HPC Platforms

In PHANTOM, all the sequential computation times of representative processes are acquired using a single node of the target platform. The network parameters needed by SIM-MPI are measured with micro-benchmarks on the network of the target platform. In this paper, prediction error is defined as *(predicted time—measured time)/(measured time) × 100 percent* and all experiments are conducted for five times.

Fig. 14 shows prediction results with PHANTOM, the cross-platform prediction, and the regression-based approach for

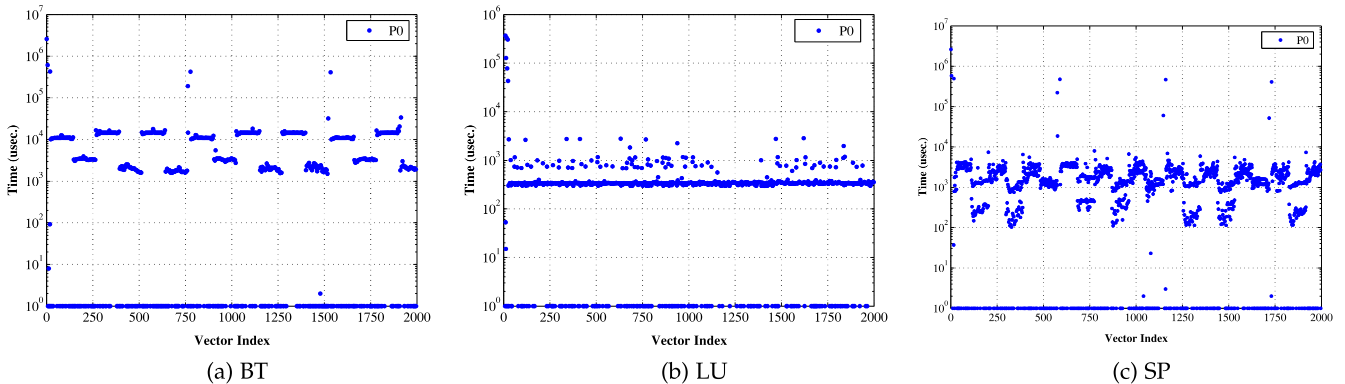


Fig. 13. Acquired sequential computation time for BT, LU and SP on *Nebulae*.

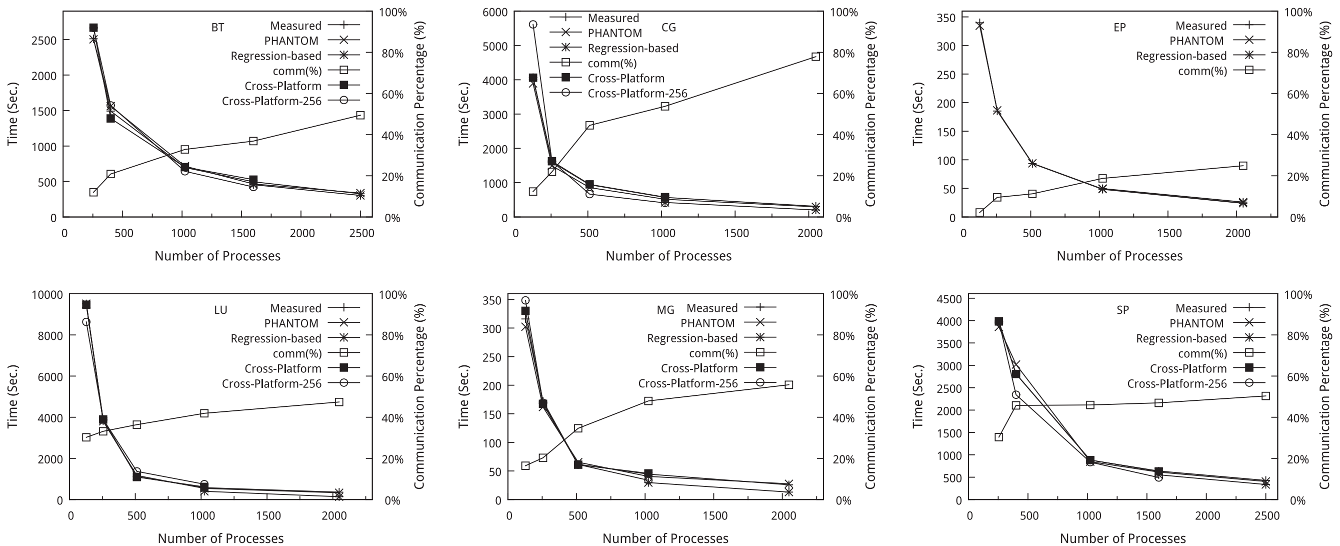


Fig. 14. Predicted time with PHANTOM, cross-platform prediction, and regression-based approach on *Nebulae*. *Cross-Platform* uses a full target platform to compute the relative performance and *Cross-Platform-256* only uses 256 processors to compute the relative performance. *Measured* means the real execution time. *comm* means the communication time percentage.

TABLE 3
Prediction Errors (Percent) of PHANTOM (P.T.), Regression-Based Approach (R.B.), and Cross-Platform Prediction (C.P. Uses a Full Target Platform to Compute the Relative Performance and C.P.-256 Only Uses 256 Target Processors to Compute the Relative Performance) on *Nebulae*

Proc.#		BT	CG	EP	LU	MG	SP
1,024	P.T.	-6.22	-9.36	0.95	-6.07	-9.17	-3.68
	R.B.	-8.38	-25.83	2.62	-30.60	-33.37	-13.31
	C.P.	0.72	1.45	N.A.	6.27	2.91	0.25
	C.P.-256	-7.59	-28.14	N.A.	28.80	-24.21	-5.02
2,048	P.T.	2.83	-5.33	1.20	-8.15	6.77	-5.58
	R.B.	-7.26	-34.89	9.00	-61.77	-49.23	-21.29

NPB programs on *Nebulae*. We validate our approach up to 2,500 processes. As shown in Fig. 14, the agreement between the predicted execution time of PHANTOM and the measured time is remarkably high. The prediction error with PHANTOM is less than 5 percent on average for all the programs. The maximum error is 9.36 percent for CG with 1,024 processes. As the number of processes increases, the network contention becomes a significant bottleneck and the communication time accounts for a large proportion of the execution time in CG, about 53.76 percent for 1,024 processes and 77.88 percent for 2,048 processes. Note that EP is an embarrassing parallel program, which has very little communication. Its prediction accuracy actually reflects the accuracy of the sequential computation time acquired with our approach. For EP, the prediction error is only 0.95 percent on average.

Table 3 also lists prediction errors of PHANTOM, the regression-based approach, and the cross-platform prediction for 1,024 and 2,048 processes (Due to the limitations of the regression-based approach, only the large processor configurations can be predicted. We use the *Explorer-100* as the host platform in the cross-platform prediction, so we can only predict the performance up to 1,600 processes). The maximum absolute prediction error of PHANTOM is less than 10 percent for all the programs up to 2,500 processes (for BT and SP), while the absolute errors for the regression-based approach vary from 2.62 to 61.77 percent. The cross-platform prediction shows very high prediction accuracy using a full target platform to compute the relative performance and the average prediction error is 2.32 percent. However, the prediction has relatively large errors for some programs when using a subset of the target platform and the average prediction error is 18.75 percent. Also, EP is not an iteration-based application, so we cannot use the cross-platform approach to predict its performance.

We also analyze the average communication time percentage for these programs on Infiniband network in Fig. 14. For most of these programs, communication time increases with the number of processes. Among these programs, CG is the most communication-intensive with the maximum communication percentage of 77.88 percent for 2,048 processes.

8.4 Performance Prediction for Amazon Cloud Platform

Fig. 15 shows prediction results on the Amazon cloud platform with PHANTOM. The sequential computation time of representative processes is acquired using a single node of the Amazon EC2 platform. We can find that PHANTOM shows very high prediction accuracy for most of the applications. The prediction error of PHANTOM is less than 7 percent on average for all the applications. For both 113.GemsFDTD and 128.GAPgeofem, the prediction errors are relative high. This is because the communication time accounts for most of the execution time in these programs and communication contention becomes more serious with the number of processes. For example, the communication time percentage in 113.GemsFDTD is more than 80 percent for 128 processes. In addition, we can find that the applications of 113. GemsFDTD, 127.wrf2 and 128.GAPgeofem are not scalable on the cloud.

Through comparing the communication time between the HPC platform and the Amazon cloud platform in Figs. 14 and 15, we can find that the communication becomes a main bottleneck on the cloud platform. On the Amazon cloud platform using 10-Gigabit Ethernet, the communication time increases rapidly. For some applications, the communication time is more than 40 percent only for 128 processes. While on the HPC platform, the communication time is moderate for most of applications. Fig. 16 shows the latency and bandwidth on HPC platforms and the Amazon cloud platform. The HPC platform has very lower latency for small messages than the cloud platform, while the bandwidth of the cloud platform is very close to the Dawning platform using IB DDR network. Our previous

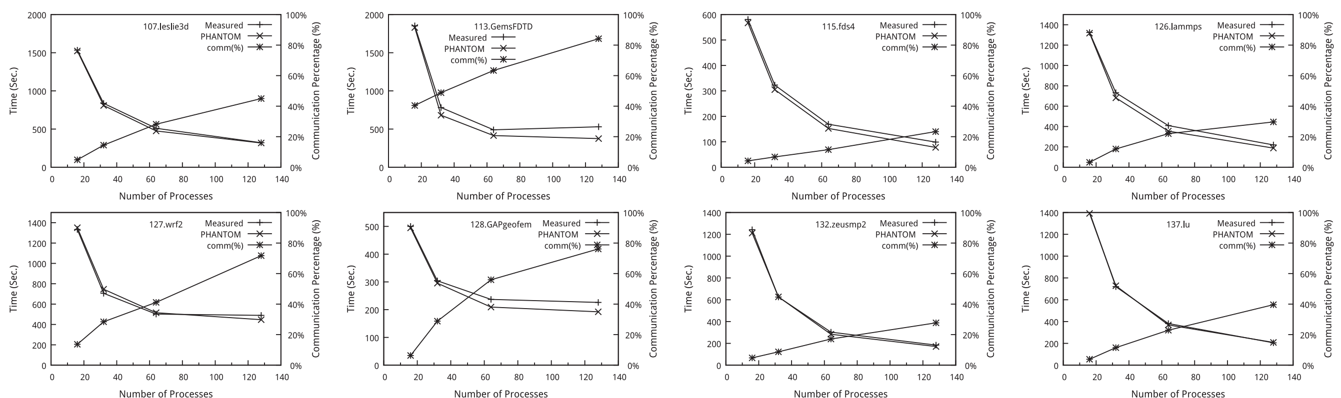


Fig. 15. Predicted time with PHANTOM on Amazon EC2 cloud platform. *Measured* means the real execution time of applications. *comm* means the communication time percentage of total execution time.

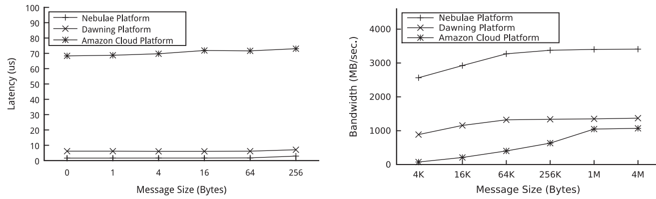


Fig. 16. Latency and bandwidth on HPC platforms and Amazon cloud platform.

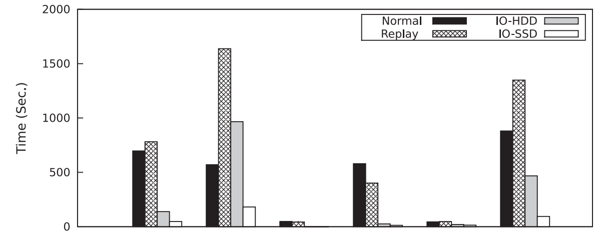


Fig. 17. The replayed execution versus the normal execution, and the replayed I/O time on HDD and SSD for 1,024 processes on *Nebulae*.

study [10] shows that the communication latency is a main reason hindering tightly-coupled parallel applications running on the current cloud.

8.5 Message-Log Size and Replay Overhead

Table 4 shows the message log size of the representative processes for NPB programs. Because the number of the representative processes is far smaller than the total number of processes described in Section 8.2.1, the message-log size is reasonable for all the programs. Results also show that the message log size is highly dependent on the application communication patterns. Although the number of process groups in MG increases with the number of processes, the message log size does not present the similar trend. EP has the least message logs due to its little communication.

Fig. 17 shows the replayed execution compared with the normal execution with 1,024 processes. For some programs such as CG and SP, the replayed execution is much longer than the normal execution. There are two main reasons about it. First, we employ the sub-group replay to execute a subgroup of processes on the same node, so all the inter-node communications need to be read from the message logs stored in local disks. Therefore, the replayed execution time is highly dependent on the message-log size and the underlying I/O performance of the local disks. Second, during the sub-group replay, all the intra-node communications use the traditional message-passing interfaces, so large synchronization overhead (waiting for the incoming messages) can be introduced due to slower processes. However, based on our experimental results, these aspects have little impact on the sequential computation time.

On *Nebulae*, both the lower bandwidth and longer latency of the local disks compared with IB QDR network significantly slow down the replayed execution. Fig. 17 also lists the disk I/O time for reading the message logs for each program (denoted by IO-HDD). We find that the I/O time accounts for a large proportion of the replayed execution time. We also measure the replayed I/O time on a RAID-0 built with five solid-state drive (SSD) disks on a local server (denoted by IO-SSD). We can find that the I/O time is significantly reduced. Moreover, the sub-group replay can eliminate partial inter-node synchronization, so the replayed execution may be shorter than the normal execution for synchronization-intensive programs such as LU and MG.

8.6 Performance of SIM-MPI Simulator

Table 5 shows the performance of SIM-MPI simulator. SIM-MPI is a parallel simulator and each thread simulates an MPI process, but the I/O time used to read message trace files is limited to the underlying disk performance. All simulations in this paper are executed on a server node (two-way Xeon E5504 processors, 12 GB of memory). Table 5 gives the performance of SIM-MPI simulator for different programs. For most of programs, the simulation time is from several seconds to several minutes up to 2,500 processes (BT and SP). Among these programs, LU has the longest simulation time due to its frequent communication operations.

8.7 Case Study

8.7.1 Analyzing Program Behavior

PHANTOM provides various what-if analysis for application developers through changing the input parameters of SIM-MPI. With this feature, application developers can identify applications' potential performance bottleneck. We take NWChem as an example. NWChem [26] is a large computational chemistry suite supporting electronic structure calculations using a variety of chemistry models. We use NWChem to perform a PBE0 calculation on the C_{240} system with the *c240_pbe0.nw* input set. Fig. 18 shows performance prediction and what-if analysis for NWChem on Explorer-100. PHANTOM gets very high prediction accuracy and the average prediction error is 3.3 percent.

To analyze the program behavior, we perform two what-if analysis with PHANTOM. First, we improve the sequential computation performance by two times (labeled with *comp/2*), and then the performance of NWChem will improve 22.7 percent on average. Second, if we double the network performance (labeled with *comm/2*), it has little impact on the application's performance. To identify the potential performance bottleneck of NWChem, we break down the detailed

TABLE 4
Message-Log Size (in GB Except EP in Byte)

Proc. #	BT	CG	EP	LU	MG	SP
128	15.7	31.6	143B	31.8	8.5	20.7
256	13.0	31.6	146B	30.4	13.2	16.8
512	8.1	21.1	146B	18.3	8.0	10.7
1,024	6.7	21.1	146B	12.2	9.5	8.6
2,048	5.4	13.2	146B	9.1	12.7	6.9

TABLE 5
Performance of SIM-MPI Simulator (Sec.)

Proc. #	BT	CG	EP	LU	MG	SP
128	8.2	8.1	0.1	100.1	1.8	15.7
256	15.9	15.4	0.2	195.0	3.3	29.6
512	64.0	37.4	0.3	349.5	5.8	119.0
1,024	127.5	75.4	0.6	642.9	10.8	233.4
2,048	255.2	182.4	1.2	1196.0	21.0	481.1

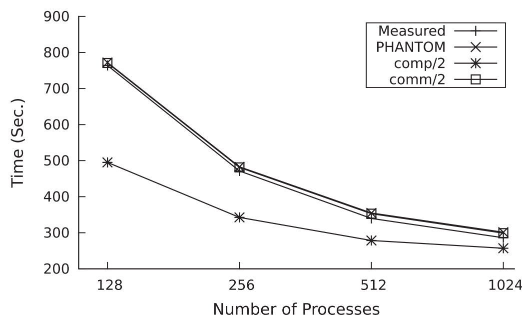


Fig. 18. Performance prediction and what-if analysis for NWChem on *Explorer-100*.

communication overhead of NWChem in Fig. 19. We can find that the synchronization overhead accounts for a large proportion of communication overhead. So the network performance has little impact on the performance.

8.7.2 Platform Selection

We demonstrate another example of using PHANTOM to help users select a suitable platform. Fig. 20 shows the prediction results of PHANTOM for Sweep3D on three target platforms. The real execution time is measured on each target platform to validate our predicted results. As shown in Fig. 20, PHANTOM gets very high prediction accuracy on these platforms. Prediction errors on *Dawning*, *DeepComp-F*, and *DeepComp-B* are 2.67, 1.30, and 2.34 percent respectively, with the maximum absolute error of −6.54 percent on *Dawning* for 128 processes. We can use the prediction results to help users compare these platforms. For example, although *Dawning* has much lower CPU frequency and peak performance than *DeepComp-F*, it has better application performance before 256 processes. *DeepComp-B* demonstrates the best performance for Sweep3D among these platforms.

9 DISCUSSIONS

Problem size. The problem size we can deal with is limited by the scale of host platforms since we need to execute the parallel application with the same problem size and the same number of parallel processes on them to collect message logs that are required during the replay phase. It should be noticed that neither the CPU speed nor the interconnect performance of host platforms is relevant to the accuracy of performance prediction on target platforms in our framework. This implies that we can generate message logs on a host platform with fewer number of processors/cores than the target platform. In fact, in our evaluation we have collected our message logs on a small-scale system. The only hard requirement for the host platform is its memory size.

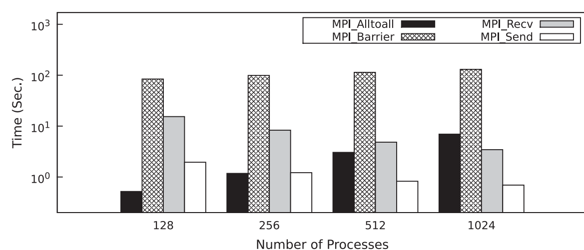


Fig. 19. Breaking down the communication overhead in NWChem.

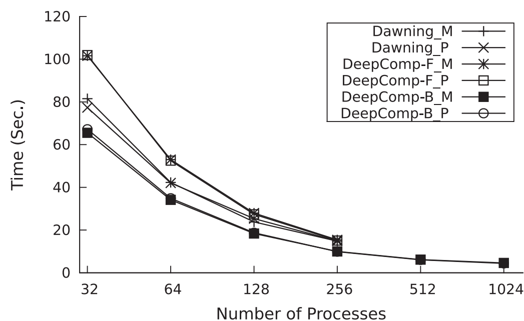


Fig. 20. Performance prediction for Sweep3D on *Dawning*, *DeepComp-F*, and *DeepComp-B* (M means the real execution time, P means predicted time with PHANTOM).

There are several potential ways to address this limitation. One is to use grid computing techniques through executing applications on grid systems that provide larger memory size than any single host platform. Another promising way is to use SSD devices and virtual memory to trade speed for cost. Note that the message logs only need to be collected once for one application with a given problem size, which is a favorable feature of our approach to avoid high cost for message log collection.

Even so, we still believe that our approach is a solid step ahead of existing work because it can acquire accurate sequential computation time with one single node of the target platform. This is a greatly desired feature for HPC system vendors and customers when they design or purchase a new parallel computer that is in the scale of current largest machine. Our approach is also important for users of public clouds to make more cost-effective decision when selecting suitable cloud instances.

Node of target platforms. We assume that we have at least one node of the target platform that enables us to measure computation time at real execution speed. This raises a problem of how we can predict performance of the target platform even without a single node.

Our approach can apply with a single node simulator that is usually ready years before the parallel machine. It is clear that this will be much slower than the measurement. Thanks to the representative replay proposed in this paper, we only need to simulate a few representative processes and the simulation can be also performed in parallel.

I/O operations. Our current approach only models and simulates communication and computation of parallel applications. However, I/O operations are also an important factor of parallel applications. In this paper, we focus on how to acquire the sequential computation time. We believe that the framework of our approach can be extended to cope with I/O operations although there are many pending issues to investigate.

Non-deterministic applications. As a replay-based framework, PHANTOM has limitations in predicting performance for applications with non-deterministic behaviors. PHANTOM can only predict the performance of one possible execution of a non-deterministic application. However, we argue that for *well-behaved* applications, non-deterministic behaviors should not cause significant impact on their performance because it means poor performance portability. So we believe that it is acceptable to use the predicted performance of one execution to represent the performance of *well-behaved* applications.

Irregular parallel applications. For irregular parallel applications such as parallel graph applications, if these programs are written with message passing interfaces such as MPI, our approach can also deal with this type of programs. But we need to emphasize that the load balance of these irregular applications is highly dependent on the input set and sometimes there is little computation similarity among different processes. In the worst case, we need to replay more process groups for accurate performance prediction.

10 Related Work

There are two types of approaches for performance prediction. One approach is to build an analytical model for the application on the target platform [3], [4], [6], [12], [27], [28]. Spafford and Vetter also proposed a domain specific language for performance modeling [29]. The main advantage of analytical methods is low-cost. However, constructing analytical models of parallel applications requires a thorough understanding of the algorithms and their implementations. Most of such models are constructed manually by domain experts, which limits their accessibility to normal users. Moreover, a model built for an application cannot be applied to another one. Phantom is an automatic framework that requires little user intervention.

The second approach is to develop a system simulator to execute applications on it for performance prediction. Simulation techniques can capture detailed performance behavior at all levels, and can be used automatically to model a given program. However, an accurate system simulator is extremely expensive, not only in terms of simulation time but especially in the memory requirements. A lot of methods have been explored to improve the accuracy and efficiency of predicting parallel performance for system simulators, such as BigSim [7], [30], MPI-SIM [31].

Trace-driven simulation [1], [32] and macro-level simulation [11] have better performance than detailed system simulators since they only need to simulate the communication operations. The sequential computation time is usually acquired by analytical methods or extrapolation in previous work. We have discussed their limitations in Section 1. In this paper, our proposed representative replay can acquire more accurate computation time, which can be used in both trace-driven simulation and macro-level simulation.

Yang et al. [8] proposed a novel cross-platform prediction method without program modeling, code analysis, or architecture simulation, which utilizes partial execution for a limited number of timesteps to compute the relative performance across platforms for a given application. This novel approach is observation-based and has demonstrated very high accuracy at extremely low cost for several real-world applications on multiple large parallel computers. In fact, we can also use the representative replay to compute the relative performance and integrate it with the cross-platform approach for effective prediction. Lee et al. presented piecewise polynomial regression models and artificial neural networks that predict application performance as a function of its input parameters [33]. Barnes et al. [5] employed the regression-based approach to predict parallel program scalability and their method shows good accuracy for some applications. However, the number of processors used for training is still very large for better accuracy.

Wu and Mueller [18] proposed a set of novel algorithms to extrapolate communication traces of a large scale application with information gathered from smaller executions. Arnold et al. [34] proposed equivalence classes for quick identification of errors from thousands of processes. Laguna et al. [35] used scalable sampling-based clustering and nearest-neighbor techniques to detect abnormal processes in large-scale systems. Gioachin et al. [36] employed the record-replay technique to debug large-scale parallel applications. Statistical techniques have been used widely for studying program behaviors from large-scale data [37], [38]. Our approach is inspired by these work but to our best knowledge we are the first to employ determine replay to acquire the sequential computation time for performance prediction.

11 Conclusion

In this paper, we demonstrate the benefit of an automatic and accurate prediction method for large-scale parallel applications. We propose a novel technique of using replay techniques to acquire the accurate sequential computation time for large-scale parallel applications on a single node of the target platform and integrate this technique into a trace-driven simulation framework to accomplish effective performance prediction. We further propose the representative replay scheme that employs the similarity of computation patterns in parallel applications to significantly reduce the replay overhead. We verify our approach on traditional HPC platforms and the latest Amazon EC2 cloud platform. The experimental results show that our approach can get high prediction accuracy on both types of platforms.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable comments and suggestions. We thank Dandan Song, Bowen Yu, Haojie Wang, and Feng Zhang for their valuable feedback. This work has been partially sponsored by NSFC project 61472201 and the National High-Tech Research and Development Plan (863 project) 2012AA01A302.

References

[1] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A framework for application performance modeling and prediction," in *Proc. ACM Conf. Supercomput.*, 2002, pp. 1–17.

[2] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, 2004, pp. 2–13.

[3] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Proc. ACM Conf. Supercomput.*, 2001, pp. 37–48.

[4] D. Sundaram-Stukel and M. K. Vernon, "Predictive analysis of a wavefront application using LogGP," in *Proc. 7th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 1999, pp. 141–150.

[5] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proc. 22nd Annu. Int. Conf. Supercomput.*, 2008, pp. 368–377.

[6] M. Mathias, D. Kerbyson, and A. Hoisie, "A performance model of non-deterministic particle transport on large-scale systems," in *Proc. Workshop Performance Model. Anal.*, 2003, pp. 905–915.

[7] G. Zheng, G. Kakulapati, and L. V. Kale, "Bigsim: A parallel simulator for performance prediction of extremely large parallel machines," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2004, pp. 78–87.

[8] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Proc. ACM Conf. Supercomput.*, 2005, p. 40.

[9] Amazon Inc. (2011). High Performance Computing (HPC) [Online]. Available: http://aws.amazon.com/ec2/hpc-applications/

[10] Y. Zhai, M. Liu, J. Zhai, X. Ma, and W. Chen, "Cloud versus in-house cluster: Evaluating amazon cluster compute instances for running MPI applications," in *Proc. ACM Conf. Supercomput.*, 2011, p. 11.

[11] R. Susukita, H. Ando, and M. Aoyagi, et al., "Performance prediction of large-scale parallell system and application using macro-level simulation," in *Proc. ACM Conf. Supercomput.*, 2008, pp. 1–9.

[12] K. J. Barker, S. Pakin, and D. J. Kerbyson, "A performance model of the Krak hydrodynamics application," in *Proc. Int. Conf. Parallel Process.*, 2006, pp. 245–254.

[13] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker, "MPIWiz: Subgroup reproducible replay of MPI applications," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2009, pp. 251–260.

[14] M. Maruyama, T. Tsumura, and H. Nakashima, "Parallel program debugging based on data-replay," in *Proc. Int. Conf. Parallel Distrib. Comput.*, 2005, pp. 151–156.

[15] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.*, vol. C-36, no. 4, pp. 471–482, Apr. 1987.

[16] A. Bouteiller, G. Bosilca, and J. Dongarra, "Retrospect: Deterministic replay of MPI applications for interactive distributed debugging," in *Proc. 14th Eur. Conf. Parallel Virtual Mach. Message Passing Interface*, 2007, pp. 297–306.

[17] J. Zhai, W. Chen, and W. Zheng, "Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node," *ACM Sigplan Notices*, vol. 45, no. 5, pp. 305–314, 2010.

[18] X. Wu and F. Mueller, "Scalaextrap: Trace-based communication extrapolation for SPMD programs," in *Proc. 7th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2011, pp. 113–122.

[19] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng, "FACT: Fast communication trace collection for parallel applications through program slicing," in *Proc. ACM Conf. Supercomput.*, 2009, pp. 1–12.

[20] S. Shao, A. K. Jones, and R. G. Melhem, "A compiler-based communication analysis approach for multiprocessor systems," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2006, pp. 1–12.

[21] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating long messages into the logp model for parallel computation," *J. Parallel Distrib. Comput.*, vol. 44, no. 1, pp. 71–79, 1997.

[22] J. Zhang, J. Zhai, W. Chen, and W. Zheng, "Process mapping for MPI collective communications," in *Proc. 15th Int. Euro-Par Conf. Parallel Process.*, 2009, pp. 81–92.

[23] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "The NAS parallel benchmarks 2.0," NAS Syst. Division, NASA Ames Res. Center, Moffett Field, CA, USA, Tech. Rep. RNR-91-002, 1995.

[24] S. P. E. Corporation. (2007). SPEC MPI2007 benchmark suite [Online]. Available: http://www.spec.org/mpi2007/

[25] LLNL. ASCI purple benchmark [Online]. Available: https://asc.llnl.gov/computing_resources/purple/archive/benchmarks, 2003.

[26] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus, et al., "Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Comput. Phys. Commun.*, vol. 181, no. 9, pp. 1477–1489, 2010.

[27] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme, "A general predictive performance model for wavefront algorithms on clusters of SMPS," in *Proc. Int. Conf. Parallel Process.*, 2000, pp. 219–228.

[28] J. Meng, V. A. Morozov, V. Vishwanath, and K. Kumaran, "Dataflow-driven GPU performance projection for multi-kernel transformations," in *Proc. ACM Conf. Supercomput.*, 2012, pp. 82:1–82:11.

[29] K. L. Spafford and J. S. Vetter, "Aspen: A domain specific language for performance modeling," in *Proc. ACM Conf. Supercomput.*, 2012, pp. 84:1–84:11.

[30] G. Zheng, "Achieving high performance on extremely large parallel machines: performance prediction and load balancing," Ph.D. dissertation, Univ. Illinois at Urbana-Champaign, Champaign, IL, USA, 2005.

[31] S. Prakash and R. Bagrodia, "MPI-SIM: Using parallel simulation to evaluate MPI programs," in *Proc. Winter Simul. Conf.*, 1998, pp. 467–474.

[32] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "DiP: A parallel program development environment," in *Proc. 2nd Int. Euro-Par Conf. Parallel Process.*, 1996, pp. 665–674.

[33] B. C. Lee, D. M. Brooks, and B. R. de Supinski, et al., "Methods of inference and learning for performance modeling of parallel applications," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2007, pp. 249–258.

[34] D. Arnold, D. Ahn, B. De Supinski, G. Lee, B. Miller, and M. Schulz, "Stack trace analysis for large scale debugging," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2007, pp. 1–10.

[35] I. Laguna, T. Gamblin, B. R. de Supinski, S. Bagchi, G. Bronevetsky, D. H. Anh, M. Schulz, and B. Rountree, "Large scale debugging of parallel tasks with automaded," in *Proc. ACM Conf. Supercomput.*, 2011, pp. 50:1–50:10.

[36] F. Gioachin, G. Zheng, and L. V. Kalé, "Robust non-intrusive record-replay with processor extraction," in *Proc. 8th Workshop Parallel Distrib. Syst.: Testing, Anal., Debugging*, 2010, pp. 9–19.

[37] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, "Array regrouping and structure splitting using whole-program reference affinity," in *Proc. ACM SIGPLAN Conf. Programm. Language Design Implementation*, 2004, pp. 255–266.

[38] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proc. Int. Conf. Archit. Support Programm. Lang. Oper. Syst.*, 2002, pp. 45–57.

Jidong Zhai received the BS degree in computer science from the University of Electronic Science and Technology of China in 2003, and the PhD degree in computer science from Tsinghua University in 2010. He is currently an assistant professor in the Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis, and modeling of parallel applications.

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. He was the CTO of Opportunity International Inc. from 2000 to 2002. Since January 2003, he joined Tsinghua University. He is currently a professor and associate head in the Department of Computer Science and Technology, Tsinghua University. His research interest is in parallel and distributed computing and programming model.

Weimin Zheng received the BS and master's degrees from Tsinghua University in 1970 and 1982, respectively. He is currently a professor in the Department of Computer Science and Technology, Tsinghua University. He is currently the director of China Computer Federation (CCF). His research interests include parallel and distributed computing, compiler technique, grid computing, and network storage.

Keqin Li is a SUNY Distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published more than 350 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *Journal of Parallel and Distributed Computing*. He is a fellow of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.