PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node

Jidong Zhai Wenguang Chen Weimin Zheng
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, Beijing, China
dijd03@mails.tsinghua.edu.cn, {cwg,zwm-dcs}@tsinghua.edu.cn

Abstract

For designers of large-scale parallel computers, it is greatly desired that performance of parallel applications can be predicted at the design phase. However, this is difficult because the execution time of parallel applications is determined by several factors, including sequential computation time in each process, communication time and their convolution. Despite previous efforts, it remains an open problem to estimate sequential computation time in each process accurately and efficiently for large-scale parallel applications on non-existing target machines.

This paper proposes a novel approach to predict the sequential computation time accurately and efficiently. We assume that there is at least one node of the target platform but the whole target system need not be available. We make two main technical contributions. First, we employ deterministic replay techniques to execute any process of a parallel application on a single node at real speed. As a result, we can simply measure the real sequential computation time on a target node for each process one by one. Second, we observe that computation behavior of processes in parallel applications can be clustered into a few groups while processes in each group have similar computation behavior. This observation helps us reduce measurement time significantly because we only need to execute representative parallel processes instead of all of them.

We have implemented a performance prediction framework, called PHANTOM, which integrates the above computation-time acquisition approach with a trace-driven network simulator. We validate our approach on several platforms. For ASCI Sweep3D, the error of our approach is less than 5% on 1024 processor cores. Compared to a recent regression-based prediction approach, PHANTOM presents better prediction accuracy across different platforms.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Complexity Measures, Performance Measures

General Terms Performance, Measurement

Keywords Performance Prediction, Parallel Application, Deterministic Replay, Trace-driven Simulation

1. Introduction

1.1 Motivation

Today, large-scale parallel computers consist 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 10000 nodes connected by network *Z*?

Accurate answer to the above question enables designers to evaluate various design alternatives and make sure that the design can meet the performance goal. In addition, it also helps application developers to design and optimize applications even before the target machine is available.

However, accurate performance prediction of parallel applications[1] is difficult because the execution time of large parallel applications is determined by sequential computation time in each process, the communication time and their convolution. Due to the complex interactions between computation and communications, the prediction accuracy can be hurt significantly if either computation or communication time is estimated with notable errors.

In this paper, we focus on how to acquire sequential computation time accurately for large-scale parallel applications. This is because existing approaches address the communication time estimation and the convolution issues fairly well, such as BigNetSim and DIMEMAS [6, 9]. The bottleneck of current prediction framework is to estimate sequential computation time in each process accurately and efficiently for large-scale parallel applications on non-existing target parallel machines.

A lot of approaches have been proposed to estimate sequential computation time for parallel applications. For model-based methods [13, 19], the application signatures, including the number of integer and floating-point instructions, 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 *target platform* and give the estimation for each sequential computation unit.

However, with rising architecture and software complexity, the accuracy of model-based approaches is becoming increasingly compromised. For example, an out-of-order issue super-scalar processor can execute multiple instructions in parallel. Contention for shared resources, such as shared cache and memory bus, on the multi-core platform can result in complex program behavior. These factors make model-based approaches difficult to acquire accurate sequential computation performance.

Some researchers [8, 20] measured the time of sequential computation for weak-scaling parallel applications through executing the application on a *prototype system*, which has fewer processors than the *target system*. For weak-scaling applications, where problem size is fixed for each processor and the sequential computation

PPoPP'10, January 9–14, 2010, Bangalore, India.

[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.

does not change with the number of processors, a prototype system is sufficient to acquire accurate computation performance.

However, measurement-based approaches do not work for strong-scaling applications where the whole problem size is fixed and the sequential computation workload varies with the number of processors. For strong-scaling applications, a few work [4, 21] uses regression-based approaches to extrapolate computation performance of large problem size. Unfortunately, extrapolation is not always applicable due to the non-linear behavior in real applications [3].

To conclude, current 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 techniques to solve the problem. For readers who are not familiar with deterministic replay, please refer to Section 4.1 and [5, 10, 14, 24]. Our paper makes two main contributions:

1. Employing deterministic replay techniques to measure sequential computation time of strong-scaling applications without full-scale target machines The biggest challenge of measurement-based prediction approaches is unable to execute large strong-scaling applications on full-scale target machines because they are not available yet. We address this issue by employing deterministic replay techniques which enable us to execute any single process of a parallel application separately on a single node at real speed without the full-scale target machine. So we can simply measure the real sequential computation time on a target node for each process one by one.

It is obvious that this approach still requires long measurement time since we need to execute **all** processes on this single node one by one. However, the execution time can be reduced almost proportionally to the number of nodes used for performance prediction because we can replay any number of processes at the same time given sufficient number of target nodes. In fact, even in one node, we usually replay *number-of-cores* processes simultaneously instead of one process to gain accurate sequential computation time due to the effects of shared resources in modern computers.

Also it should be emphasized that the replayed processes are executed at full speed, which is at least three orders of magnitude faster than cycle-accurate simulation.

2. Employing representative replay to reduce measurement time Although with the approaches proposed above, we can accurately measure the sequential computation time for each process at an unprecedented speed, we still hope to reduce the measurement time further because target platforms may have thousands of nodes which can be translated into significant slowdown if we replay all processes on a small number of available target nodes. Given the fact that some applications execute for several days on thousands of nodes, there is still a great desire to reduce the measurement time notably further.

In this paper, we observe that computation behavior of processes in parallel applications can be clustered into a few groups while processes in each group have similar computation behavior. This observation helps us to reduce the measurement time significantly because we only need to replay representative parallel processes instead of all of them.

Our approach is based on the following two assumptions:

1. Message logs of the whole parallel application that will be used during the replay phase are available. In fact, this assumption is a limitation of our approach, since we need to run the parallel application on an existing system to collect the required message logs. If we want to predict an application's performance on a next-generation machine, we may not be able to find large enough existing systems to collect the logs for it.

Even with this limitation, we still believe that our approach is a solid step ahead of existing work because it can do cross-platform performance prediction. This is a greatly desired feature for HPC system vendors and customers when they design or purchase new parallel computers that are in the scale of current largest machine. Our technique can be employed by acquiring message logs from the existing large machines and predict the performance of the other one in the same or smaller scale, perhaps with different processors, interconnect and memory size/speed.

2. At least one node of the target platform is available. One node of the target platform is needed for our approach although the whole target system may not be available yet. We believe this assumption is reasonable because target-platform nodes are usually available several months earlier than the whole system.

We have implemented a performance prediction framework, called PHANTOM, which integrates the above computation-time acquisition approach with a trace-driven network simulator. We validate our approach on several platforms. For ASCI Sweep3D, the error of our approach is less than 5% on 1024 processor cores. We compare the prediction accuracy of PHANTOM with a recent regression-based prediction approach [4]. The results show that PHANTOM has better prediction accuracy across different platforms than the regression-based approach.

This paper is organized as follows. Section 2 gives our base performance prediction framework. In Section 3, we give two key definitions used in our approach. In Section 4 and 5, we present our idea that how we acquire sequential computation time. Section 6 describes the implementation of PHANTOM. Our experimental results are reported in Section 7. We discuss limitations and extension of our work in Section 8. The related work is discussed in Section 9. Finally, we conclude in Section 10.

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 two parts, predict computation and communication performance separately and finally use a simulator to convolute them to get the execution time of whole parallel applications. The framework includes the following key steps:

1. Collecting computation and communication traces We generate communication traces of parallel applications by intercepting all communication operations for each process, and mark the computation between communication operations as sequential computation units. The purpose of this step is to separate communications and computation in parallel applications to enable us to predict them separately. Figure 1 shows a simple MPI program and its computation and communication traces are given in Figure 2a when the number of processes is 2 (The elapsed time for the k^{th} computation unit of process x is denoted by `CPU_Burst(x,k)`.).

It should be noted that we only need the communication information (e.g. message type, message size, source and destination etc.) and the interleave of communication/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, we employ the FACT technique [26] which can generate traces of large-scale applications on small-scale systems fast.

2. Obtaining 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 deterministic replay techniques. We will elaborate it

```
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)
```

Figure 1: An example of Fortran MPI program.

in Sections 4 and 5. For now, we just assume that we obtain the accurate computation time for each MPI process which can be filled into the traces generated in step 1. Figure 2b shows obtained sequential computation time for process 0 of the program in Figure 1.

3. Use a trace-driven simulator to convolute communication and computation performance Finally, a trace driven simulator, called SIM-MPI [22], is used to convolute communication and computation performance. As shown in Figure 2c, the simulator reads trace files generated in step 1, the sequential computation time obtained in step 2, and network parameters of target platforms, to predict the communication performance of each communication operation and convolute it with sequential computation time to predict the execution time of the whole parallel application on the target platform. SIM-MPI is similar to DIMEMAS [9], but with a more accurate communication model, called LogGPO, which is an extension of LogGP model [1]. It can model the overlap between computation and communication more accurately than existing communications models. Details of LogGPO model and SIM-MPI simulator can be found in [22].

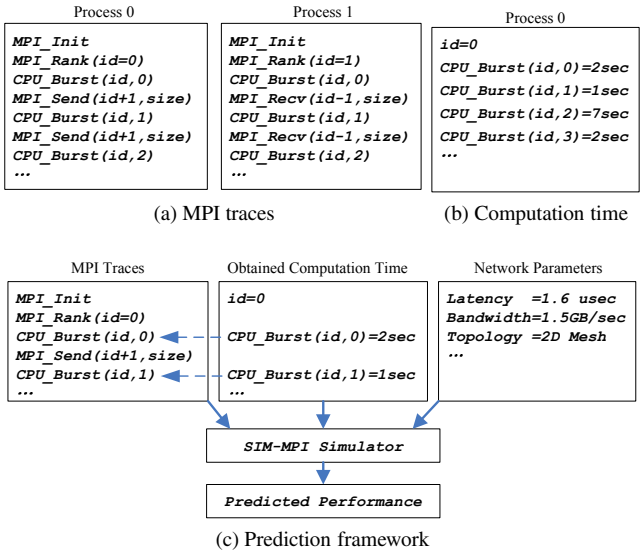


Figure 2: Base performance prediction framework.

3. Definitions

In order to illustrate our idea more clearly, we define two key concepts for MPI applications in this section. One is *communication sequence*, the other is *sequential computation vector*.

3.1 Communication Sequence

Communication sequence is first introduced by Shao *et al.* [17], which describes the intrinsic communication characteristics of parallel applications.

DEFINITION 1 (Communication Sequence). *Communication sequence is a representation of communication pattern for a given parallel program, which records message type of each communication operation in temporal sequence for each parallel process.*

Figure 3 gives an example of MPI program. The left part is the source code of program and the right part shows communications, computation and their interleave for this program with two processes: $P0$ and $P1$. $c0$, $c1(i)$, $c2(i)$ and $c3(i)$ mean the sequential computation time between communication operations. The communication sequence literal representation for processes $P0$ and $P1$ in Figure 3 is shown below (Square brackets means a loop of communication operations.). In this program, the processes with the same parity have identical communication sequence.

$\mathbb{C}(P0)$ = {Init, Barrier, [Send, Recv], Fina}.
$\mathbb{C}(P1)$ = {Init, Barrier, [Recv, Send], Fina}.

```
MPI_Init
c0    // Means computation
MPI_Barrier
for(i=0; i<N; i++){
  if(myid%2 == 1){
    MPI_Recv(..., myid-1, ...)
    c1(i)
    MPI_Send(..., myid-1, ...)
  }
  c2(i)
  if(myid%2 == 0){
    MPI_Send(..., myid+1, ...)
    c3(i)
    MPI_Recv(..., myid+1, ...)
  }
}
MPI_Fina
```

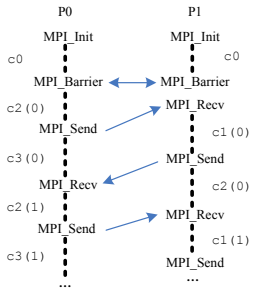


Figure 3: An example of MPI program and its execution model.

3.2 Sequential Computation Vector

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

The sequential computation vector for process x is denoted by c^x:

$$c^x = [t_0, t_1, \ldots, t_m] \tag{3}$$

where $t_k = (B_{k+1} - E_k), k \geq 0$, B_k and E_k are the timestamps of entry and exit points for the k^{th} communication operation respectively in process x. The dimension of the computation vector represents the number of the segmenting computation units for a given process, denoted by $dim(c)$. The computation pattern for the application A with n processes can be described with a n-tuple, denoted by $C(A)$:

$$C(A) = (c^0, c^1, \ldots, c^n) \tag{4}$$

For example, the sequential computation vectors for processes 0 and 1 in Figure 3 are listed below.

$c^0 = [c0, c2(0), c3(0), c2(1), c3(1), ..., c3(N-1)]$.
$c^1 = [c0, c1(0), c2(0), c1(1), c2(1), ..., c2(N-1)]$.

4. Obtaining Sequential Computation Time

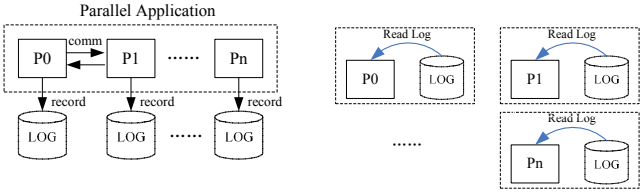
In this section, we present the basic approach that how we acquire the sequential computation time for a parallel application with the deterministic replay technique.

4.1 Deterministic Replay

Deterministic replay [5, 10, 14, 24] is a powerful technique for debugging parallel applications. Replay tools include two phases: record phase and replay phase. In the record phase, replay tools record orders and/or return values of irreproducible function calls, such as incoming messages, during the application execution. In the replay phase, they can replay the faulty processes to any state of the recorded execution. Data replay [5, 14] is an important type of deterministic replay technique for parallel applications. It records all incoming messages to each process during the application execution. With this approach, developers can execute any single process for debugging during the replay phase rather than having to execute the entire parallel application. The weakness of data replay is that replay tools must record all inter-process communications for each process.

4.2 Acquire Sequential Computation Time

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



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

Figure 4: Acquire the sequential computation time.

As shown in Figure 4, the main steps of our approach to acquire sequential computation time of applications include:
(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 these 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 the single node separately and collect the elapsed time for each sequential computation unit.

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

In our data-replay system, we record the data using MPI profiling interface (PMPI), which requires no modifications of either applications or MPI libraries. In the logging execution, our system intercepts each MPI computation operation, and then records the returned values and memory changes to log files, including all incoming messages to each process. For receiving communication operations, we record the contents of receiving messages, the return values from the function and the MPI status. For sending operations, only the returned values are recorded. For non-blocking receiving operations, we maintain a table to map the request handle to the receiving buffer corresponding to it and record real message contents at the invocation of `MPI_Wait` or `MPI_Waitall` routines. Figure 5 gives an example to record logs for `MPI_Recv` routine.

```
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
}
```

Figure 5: An example of recording logs for `MPI_Recv` routine.

Replaying Each Process Separately When we want to acquire the sequential computation time for a given process, we just need to execute that process rather than execute a full-scale parallel application. The data-replay technique makes it practical to execute particular process during the replay phase. The message-log database records the necessary information for replaying each process. To acquire the sequential computation time, we insert two timing functions at the entry and exist points of communication operations in the replay engine. A simple program is used to calculate the final sequential computation time for each process. Figure 6 shows an example for replaying `MPI_Recv` routine and recording the time-stamps. B_k and E_k are the time-stamps of entry and exist points for the k^{th} communication operations.

```
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
}
```

Figure 6: Replay `MPI_Recv` routine and record time-stamps.

4.3 Concurrent Replay

In current multi-core platforms or SMP (Symmetric Multi-Processor) servers, resource contention can affect the application performance significantly. To accurately measure the effects of resource contention on application performance, we propose the strategy of *concurrent replay* in PHANTOM. During the replay phase, we replay multiple processes simultaneously according to the number of processes running on one node of the target platform. Thus, the effects of resource contention can be captured during the concurrent execution.

To reduce overhead for recording message logs, we employ the SRR (Subgroup Reproducible Replay) technique [24] in PHANTOM. We treat the processes executing on the same node as a replay subgroup. Only the contents of communications crossing nodes are recorded, while the communications within a node are not recorded. During the replay phase, the processes on the same

node are executed simultaneously. Besides reducing logging overhead, another advantage of using SRR technique is that the execution mode during SRR replay is more close to the real execution. Because during the SRR replay the intra-node communications are the same with the real execution and only the inter-node communications need to be read from the message-log files.

On the accuracy of replay-based computation time collection Although the concurrent replay technique partially reproduces the cache/bus contentions occurred in the real execution of parallel applications, there are still sources for errors. For example, during the replay phase, all incoming messages crossing nodes are read from message logs directly without waiting for other processes to send them out. This may cause different resource contention patterns from the real execution. From our experiments, we find that the errors introduced by replay are acceptable for benchmarks we have evaluated.

5. Representative Replay

In this section, first we propose the challenges with the approach proposed in Section 4 when processing large-scale applications. Second, we present the attributes of computation similarity existing in most parallel MPI applications. At last, we present *representative replay* technique to address these challenges.

5.1 Challenges for Large-Scale Applications

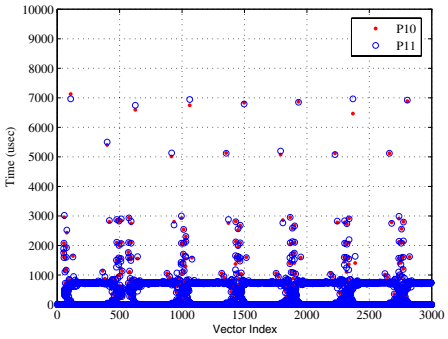
Basic approach can already acquire accurate sequential computation time for a strong-scaling application on a small-scale system. However, there are two main challenges for large-scale applications:

- *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, thus it will take nT to obtain all the computation performance. In fact, this time complexity is impractical for an application with thousands of processes.
- *Huge log size:* As data replay requires recording all incoming messages for each process, the log size will become more and more huge with rising of the number of processes.

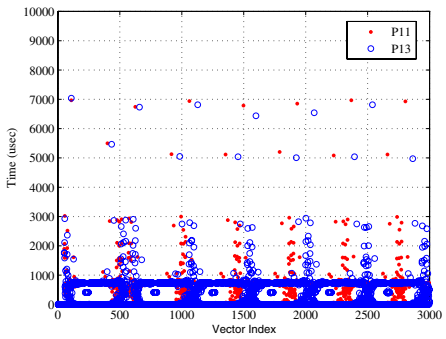
5.2 Computation Similarity

An important inspiration in this paper is that in parallel MPI applications computation behavior of different processes represents great similarity. Due to the SPMD (Single Program Multiple Data) nature of MPI programs, the program representation for each process is not necessarily distinct but rather most processes execute the same program with unique data. Such computation patterns are reasonable for most MPI applications. A number of studies for analyzing MPI applications have shown that most processes perform the same calculations and have the similar computation behavior [3, 7, 8, 15, 20]. For example, in the NPB MG program (CLASS=C) with 16 processes, the computation behavior of a group of processes 0-3, 8-11 represents great similarity, and that of another group of processes 4-7, 12-15 also represents great similarity. The computation behavior of processes between two groups has significant difference. For the purpose of clear presentation, we just list the computation behavior of processes 10, 11 in Figure 7a and processes 11, 13 in Figure 7b.

To measure the similarity degree of computation behavior between two processes, we use the distance of sequential computation vectors to characterize it. There are several ways of calculating the distance of two vectors, such as Euclidean distance and Manhattan distance. In this paper, we adopt Manhattan distance to compare two vectors. The Manhattan distance between two vectors is the



(a) Process10 vs. Process11



(b) Process11 vs. Process13

Figure 7: Sequential computation vectors for NPB MG program (CLASS=C, NPROCS=16). The computation behavior of processes 10, 11 represents great similarity, while the computation behavior of processes 11, 13 has great difference.

sum of absolute values of the differences of their corresponding elements, which has the advantage that it weights more heavily differences in each dimension. Thus, it is consistent with our aim of finding the processes that have the similar computation behavior as far as possible. For processes x and y, the distance is computed as:

$$\mathrm{Dist}(c^x, c^y) = \begin{cases} \sum_{i=1}^{m} |c_i^x - c_i^y| & \text{if } \mathbb{C}(x) = \mathbb{C}(y) \\ \infty & \text{if } \mathbb{C}(x) \neq \mathbb{C}(y) \end{cases} \quad (5)$$

where c^x and c^y are the sequential computation vectors for processes x and y. c_i^x and c_i^y are the i^{th} vector elements. m is the dimension of the vectors. $\mathbb{C}(x)$ and $\mathbb{C}(y)$ are the communication sequences. For processes with different communication sequences, we set their distance infinity, since the dimension of their sequential computation vectors may be not identical.

5.3 Select Representative Processes

Based on the key observation, we propose *representative replay* to address the challenges listed in Section 5.1. Our idea is to partition processes of applications into a number of groups so the computation behavior of processes in the same group are as similar as possible, and choose a representative process from the group to record and replay, whose sequential computation performance will be used instead of other processes in the same group.

To identify the similar processes, we employ clustering technique, which is an effective technique to analyze complex nature of multivariate relationships. There are two commonly used clustering techniques called K-means clustering and hierarchical clus-

tering. K-means clustering is computationally efficient, but it requires an *a priori* known number of classes. It is suitable for the users who know much about the computation patterns of the application. Hierarchical clustering is general method for most users who know little about the application. It forms the final class by hierarchically grouping sub-clusters according to predefined distance metric. Both clustering techniques are supported in PHANTOM.

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 5
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
```

The algorithm of hierarchical clustering used in PHANTOM is listed in Algorithm 1. Complete linkage is used to measure the inter-cluster distance in hierarchical clustering (The distance between two clusters is the distance between the furthest points in those clusters). The output of hierarchical clustering can be represented by a dendrogram, where each level indicates the merging of two closest sub-clusters. Figure 8 illustrates the dendrogram for NPB MG program with 16 processes. Depending on the expected accuracy of final prediction, a horizontal line can be drawn in the dendrogram to partition the processes into a number of groups. We use a $k\%$ factor to compute the linkage distance. Assuming that there are n processes of applications having identical communication sequence, the dimension of their sequential computation vectors is m. Hence, $k\%$-linkage distance, L, is given by Formula 6.

$$L = k\% \frac{1}{n} \sum_{x=1}^{n} \sum_{i=1}^{m} c_i^x \tag{6}$$

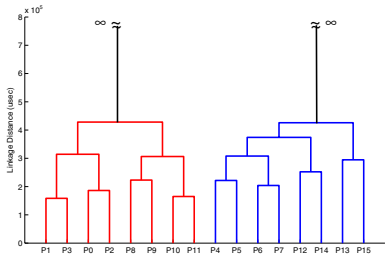


Figure 8: Dendrogram for NPB MG program (CLASS=C, NPROCS=16).

Representative process replay implemented in PHANTOM involves the following key steps:

1. Collecting the communication sequence and computation patterns for a given parallel application. The communication sequence described in Section 3 can be also acquired with FACT technique [26] on a small-scale system. The computation patterns for each process are represented with sequential computation vectors which can be collected on a host platform. The overhead in terms of time and space of this step is relatively little, since only a small quantity of data is collected.
2. Clustering the processes based on the degree of computation similarity. First, the processes that have identical communication sequence are put into the same group. Second, for each process group, the clustering technique is used to group similar processes further. Finally, all the processes are partitioned into a number of clusters.
3. Selecting representative processes and building the message-log database. For each cluster, the process that is closest to the center of the cluster is selected as the representative process. During the record phase, PHANTOM only records the message logs for these select processes.
4. Replaying representative processes on one node of the target platform. During the replay phase, the sequential computation time of these select processes are collected, which will be used instead of other processes in the same cluster for predicting application performance.

The main idea in *representative replay* is that for each process group, only the representative process should be recorded and replayed for performance prediction. Moreover, to capture the effects of resource contention, the processes executing on the same node with representative processes should be also recorded.

6. Implementation

In this paper, we implement a performance prediction framework for parallel applications based on *representative replay*, called PHANTOM. In fact, *representative replay* can be used in other prediction framework for improving prediction accuracy, such as PERC and macro-level simulation[19, 21]. PHANTOM is an automatic tool chain which requires little manpower for understanding the algorithm and implementation of the parallel application. Figure 9 gives an overview of PHANTOM. PHANTOM consists of three main modules, *CompAna*, *CommAna* and *NetSim*.

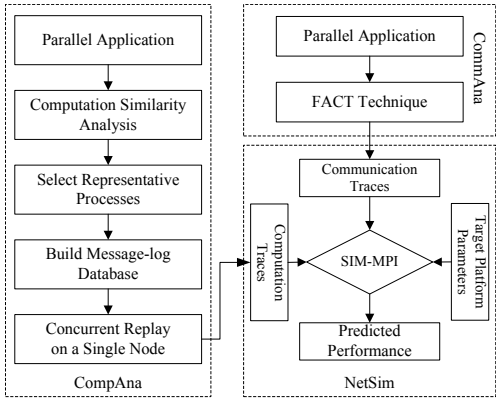


Figure 9: Overview of PHANTOM.

CompAna module is responsible for analyzing the computation similarity of parallel applications and building the message-log database on the host platform. When the users want to predict the application performance, at least one node of the target platform is used to replay representative processes. The collected sequential computation time for each process is stored with computation trace files as described in Section 2. *CommAna module* takes advantage of FACT technique [26] to collect the communication traces of parallel applications. The communication trace files record the message type, message size, message source and destination, etc. for each communication operation. More details about FACT can be found in [26]. In the *NetSim module*, the computation and communication traces generated by previous two modules are fed into a network simulator, SIM-MPI [22]. The SIM-MPI outputs the final performance prediction result of the application.

7. Evaluation

7.1 Experiment Platforms and Benchmarks

Table 1 gives a description of parallel platforms used in our evaluation. *Explorer platform* serves as the host platform used to collect message logs of applications. The other three platforms are used to validate the prediction accuracy of our approach.

Table 1: Parallel platforms used in the evaluation.

System	**Explorer**	**Dawning**	**DeepComp-F**	**DeepComp-B**
CPU type	Intel E5345	AMD 2350	Intel X7350	Intel E5450
CPU speed	2.33 GHz	2.0 GHz	2.93 GHz	3.0 GHz
#cores/node	8	8	16	8
#nodes	16	32	16	128
mem/node	8 GB	16 GB	128 GB	32 GB
Network	Infiniband	InfiniBand	InifiniBand	InfiniBand
Shared FS	NFS	NFS	StorNext	StorNext
OS	Linux	Linux	Linux	Linux

We evaluate our approach with 6 NPB programs [2], BT, CG, EP, LU, MG, SP and ASCI Sweep3D (S3) [12]. The version of NPB is 3.3 and input data set is Class C. For Sweep3D, both execution modes are used, strong-scaling and weak-scaling modes. For strong-scaling mode (S3-S), the total problem size of $512 \times 512 \times 200$ is used. For weak-scaling mode (S3-W), the problem size is $100 \times 100 \times 100$ which is fixed for each process.

7.2 Grouping Results

Table 2 shows the results of the number of process groups that have the similar computation behavior for each program with different numbers of processes (For BT and SP, the number of processes is 16, 36, 64, 144 and 256.). The grouping strategy is described in Section 5.3 and the factor used in hierarchical clustering is 10%. 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 almost similar computation behavior for different numbers of processes. For BT and SP with 36 processes, due to load imbalance the number of groups is 2. 2) For LU and Sweep3D (both strong-scaling and weak-scaling modes), the number of groups keeps constant with the number of processes. 3) For MG, the number of groups increases as the number of processes. However, the number of groups grows 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 that have identical communication sequence always have similar computation behavior. This finding gives us a hint that we can only use the communication sequence to group parallel processes in MPI applications approximately.

Table 2: The number of process groups that have similar computation behavior.

Proc. #	BT	CG	EP	LU	MG	SP	S3-S	S3-W
16	1	1	1	9	2	1	9	9
32	2	1	1	9	4	2	9	9
64	1	1	1	9	8	1	9	9
128	1	1	1	9	12	1	9	9
256	1	1	1	9	18	1	9	9

7.3 Performance Prediction

7.3.1 The accuracy of sequential computation performance

We have compared the sequential computation performance acquired using our approach with the real sequential computation performance on several platforms. The results show that our approach can get accurate sequential computation performance. Figure 11 gives a comparison result for process 0 of S3-S with 256 processes on *Dawning platform*. The main difference between replay-based execution with normal execution is that where the incoming messages are received. During the replay-based execution, the receiving messages are read from message-log files instead of network. We find that the operations of reading logs has little effect on the application performance. At the following section, we will give the detailed results of prediction accuracy using our acquired sequential computation time.

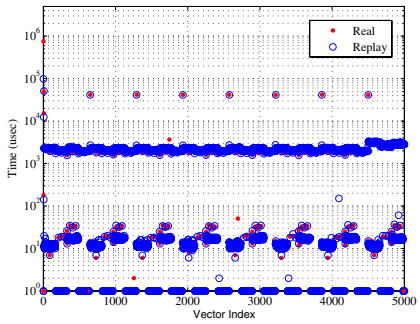


Figure 11: The real sequential computation performance vs. acquired with *representative replay* for process 0 of S3-S with 256 processes on *Dawning platform*.

7.3.2 Prediction results

Comparison In this paper, we compare the predicted time using PHANTOM with a recent prediction approach proposed by Barnes *et al.*: regression-based model [4]. This model predicts the execution time T of a given parallel application on p processors by using several instrumented runs of this program on q processors, where $q \in \{2, ..., p_0\}, p_0 < p$. Through varying the values of the input variables $(x_1, x_2, ..., x_n)$ on the instrumented runs, this model aims to calculate coefficients $(\beta_0, ..., \beta_n)$ by linear regression fit for $log_2(T)$:

$log_2(T) = \beta_0 + \beta_1 log_2(x_1) + \beta_2 log_2(x_2) + ... + \beta_n log_2(x_n) + g(q) + error$

where in this model $g(q)$ can be either a linear function or a quadratic function for the number of processors, q. Once these coefficients are determined, above equation can be used to predict the application performance on p processors. In this paper, we use three different processor configurations for training set: $p_0 = 16$, $p_0 = 32$ and $p_0 = 64$. For each program, we predict the performance with two forms of $g(q)$ function given by the authors, and the best results are reported.

In PHANTOM, all the sequential computation time of representative processes is 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, error is defined as *(measured time-predicted time)/(measured time*100)* and all the experiments are conducted for 5 times.

Figure 10 demonstrates the prediction results with both PHANTOM and the regression-based approach for seven programs on the *dawning platform*. As shown in Figure 10, the agreement between the predicted execution time with PHANTOM and the measured time is remarkably high. The prediction error with PHANTOM is less than 8% on average for all the programs. Note that EP is an embarrassing parallel program, which does not need communications. Its prediction accuracy actually reflects the accuracy of sequential computation time acquired with our approach. For EP, the prediction error is only 0.34% on average. Table 3 also lists the prediction errors with PHANTOM and the regression-based approach

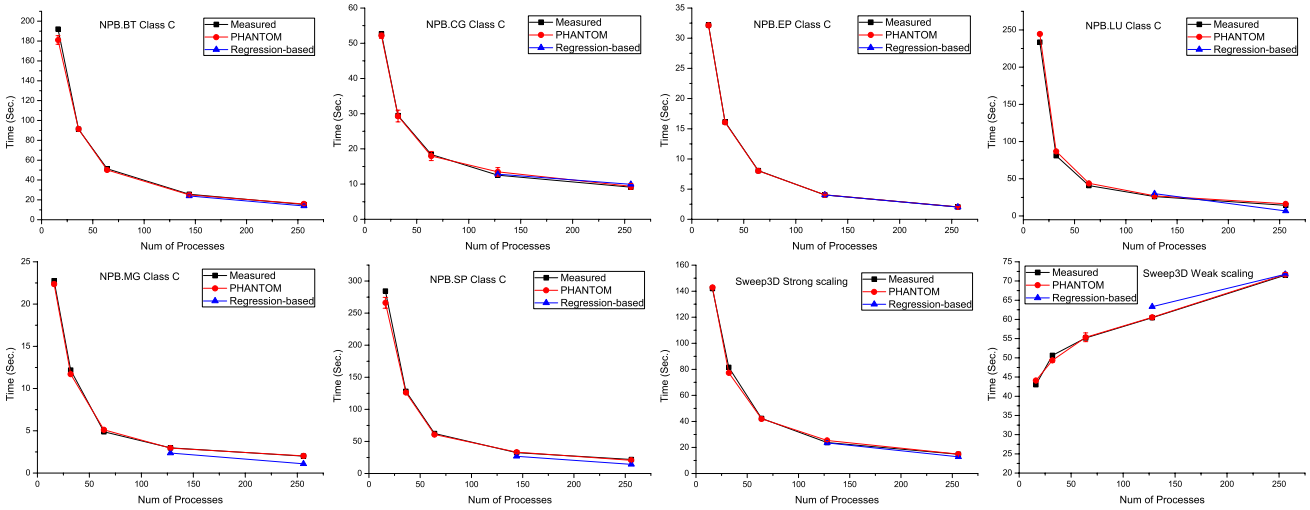


Figure 10: Predicted time with PHANTOM compared with that with Regression-based approach on *Dawning Platform*. *Measured* means the real execution time of applications. Error bars in predicted time of PHANTOM show the 90% confidence interval.

(Due to the limitations of the regression-based approach, only the performance of applications with p processes ($p > p_0$) can be predicted.). The prediction accuracy with PHANTOM is much higher than that with the regression-based approach for most programs.

Table 3: Prediction errors(%) with PHANTOM (P.T.) vs. Regression-Based approach (R.B.) on *Dawning platform*.

Proc. #		BT	CG	EP	LU	MG	SP	S3-S	S3-W
128	P.T.	2.22	-7.60	-0.34	-3.95	0.97	-2.29	-6.54	-0.15
	R.B.	6.32	-2.22	0.01	-15.02	20.58	17.72	1.30	-4.75
256	P.T.	-3.27	-2.65	-0.27	-14.28	-0.97	5.97	-0.52	-0.27
	R.B.	9.50	-8.95	0.76	53.20	45.32	34.38	13.41	-0.28

Figure 12 gives the breakdown of predicted execution time of process 0 for each program with 256 processes on *Dawning platform*. *comp* is the sequential computation cost, *comm* is the communication overhead and *syn* is the synchronization cost. We can find that synchronization overhead accounts for a large proportion of execution time for most of programs.

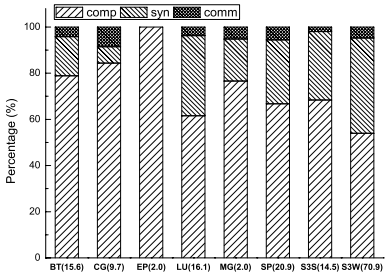


Figure 12: Breakdown of predicted time of process 0 (listed on the x-axis in second) for each program with 256 processes.

We predict the performance for Sweep3D on three target platforms. The real execution time is measured on each target platform to validate our predicted results. All the message logs are collected on *Explorer platform*. As shown in Figure 13, PHANTOM can get high prediction accuracy on these platforms. Prediction errors on *Dawning*, *DeepComp-F* and *DeepComp-B platforms* are on average 2.67%, 1.30% and 2.34% respectively, with only -6.54% maximum error on *Dawning platform* for 128 processes. PHANTOM has a better prediction accuracy as well as greater stability across different platforms compared to the regression-based approach. For example, on the *DeepComp-B platform*, the prediction error for PHANTOM is 4.53% with 1024 processes, while 23.67% for regression-based approach ($p_0 = 32, 64, 128$ used for training). Note that while *Dawning platform* has lower CPU frequency and peak performance than *DeepComp-F platform*, it has better application performance before 256 processes. *DeepComp-B platform* presents the best performance for Sweep3D among three platforms.

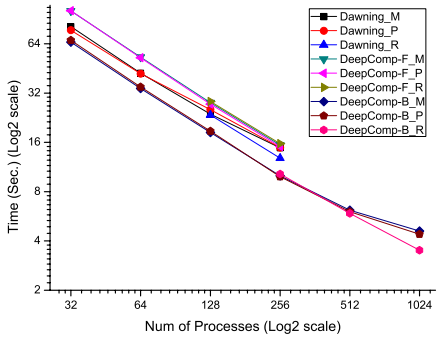


Figure 13: Performance prediction for Sweep3D on *Dawning*, *DeepComp-F* and *DeepComp-B platforms* (M means the real execution time, P means predicted time with PHANTOM, R means predicted time with Regression-based approach).

7.4 Message-Log Size and Replay Overhead

In PHANTOM, we just record the message logs for representative processes and those processes executing on the same node with them. As the number of process groups is far smaller than the number of processes presented in Section 7.2, the message-log size is reasonable for all the programs. As shown in Table 4, SP has the largest message logs while EP has the least due to little communication.

Figure 14 shows the replay-based execution time compared with normal execution for each program with 256 processes. Because most of the incoming messages are read from the log files and little synchronization overhead is introduced, the replay-based execution time is less than normal execution for most of programs. For

Table 4: Message-log size (in Giga-Byte except EP in Kilo-Byte).

Proc. #	BT	CG	EP	LU	MG	SP	S3-S	S3-W
16	3.01	2.12	1.03K	1.39	0.3	5.49	0.39	0.14
32	6.14	1.59	1.03K	2.79	0.62	11.2	0.78	0.55
64	3.5	1.85	1.03K	2.79	0.46	6.38	0.78	0.55
128	2.6	1.85	1.04K	3.32	0.58	4.75	0.92	1.29
256	1.99	1.99	1.06K	3.06	0.72	3.65	0.84	1.28

example, in weak-scaling Sweep3D with 256 processes, both communication cost and synchronization cost account for more than 46% of execution time, about 32.62 seconds. While the overhead introduced during the replay is 8.28 seconds. As a result, the replay-based execution time is much smaller than normal execution time.

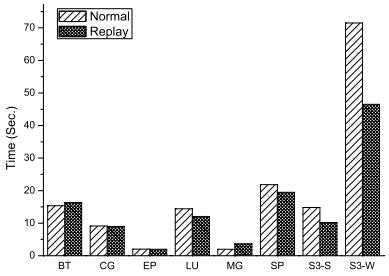


Figure 14: The elapsed time of replay-based execution compared with normal execution for each program with 256 processes.

7.5 Performance of SIM-MPI Simulator

SIM-MPI simulator has high efficiency since only the communication operations need to be simulated. All the simulation in this paper is executed on a server node equipped with 2-way quad-core Xeon E5504 processors (2.0GHz), 12GB of memory size. Table 5 gives the performance of SIM-MPI simulator for different programs. For most of programs, the simulation time is less than *1 minute*. Among these programs, LU has the longest simulation time due to frequent communication operations.

Table 5: Performance of SIM-MPI simulator (in Second).

Proc. #	BT	CG	EP	LU	MG	SP	S3-S	S3-W
16	0.49	0.79	0.04	4.17	0.20	0.93	0.36	0.19
32	1.51	1.98	0.09	8.57	0.60	2.82	0.79	0.43
64	3.24	4.20	0.15	17.33	0.75	6.21	1.62	0.92
128	9.97	11.27	0.25	34.19	1.35	19.36	2.75	1.97
256	30.38	21.19	0.49	66.47	2.73	39.41	5.24	4.24

8. Limitations and Discussions

Problem size A critical limitation of our approach is that the problem size we can deal with is limited by the scale of host platforms since we need to execute the parallel applications with the same problem size and the same number of parallel processes on them to collect message logs that are required at the replay phase.

It should be noticed that neither the CPU speed nor the interconnect performance of host platforms are relevant to the accuracy of performance prediction on target platforms in our framework. This implies that we can even generate message logs on a host platform with fewer number of processors/cores than the target platform. The only hard requirement for the host platform is its memory size.

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

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

Our approach can apply with a single node simulator which is usually ready years before the parallel machine. It is clearly that this will be much slower than measurement. Thanks to the representative replay technique we 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, especially in real large applications instead of kernel benchmarks we used in this paper. We think the framework of our approach can be extended to cope with I/O operations although there are many pending issues to investigate further.

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

9. Related Work

Performance prediction of parallel applications has a large body of prior work. There are two well-known approaches for performance prediction. One approach is to build an analytical model for the application on the target platform [3, 7, 8, 15, 20]. 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 which 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 their memory requirements. Existing simulators, such as BigSim, MPI-SIM [16, 23, 27], are still inadequate to simulate the very large problems that are of interest to high-end users.

Trace-driven simulation [9, 19] and macro-level simulation [21] 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 the Section 1. In this paper, our proposed representative replay technique can acquire more accurate computation time, which can be used in both trace-driven simulation and macro-level simulation. Our prototype system, PHANTOM, is also a trace-driven simulator integrated with the representative replay. Moreover, PHANTOM

adopts the FACT technique, which makes it feasible to collect the communication traces needed by the trace-driven simulator on a small-scale system.

Yang *et al.* propose a cross-platform prediction method based on relative performance between target platforms without program modeling, code analysis, or architecture simulation [25]. Their approach works well for iterative parallel codes that behave predictably. In order to measure partial iteration performance, their approach requires a full-scale target platform available, while our approach only requires a single node of the target platform. Lee *et al.* present piecewise polynomial regression models and artificial neural networks that predict application performance as a function of its input parameters [11]. Barnes *et al.* [4] employ the regression-based approaches 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 and their method only supports load-balanced workload.

Statistical techniques have been used widely for studying program behaviors from large-scale data [18, 28]. Our approach is inspired by these previous work and also adopts statistical clustering.

10. Conclusion

For designers of large-scale parallel computers, it has been long desired to predict performance of parallel applications on various design alternatives at the design phase.

In this paper, we extend existing trace-driven simulation framework by using deterministic replay techniques to measure computation time process by process on prototype of target systems. We further propose representative replay scheme which employs similarity of computation pattern in parallel applications to reduce time of prediction significantly. We verify our approach on several platforms and the prediction error is less than 5% for Sweep3D on 1024 processor cores.

The approach proposed in this paper is a combination of operating system and performance analysis techniques. We expect this paper will motivate more interactions between these two fields in the future.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. We thank Yunquan Zhang, Yuan Yu, Honghui Lu, Kang Chen, Ying Zhao, Wei Xue, Tianwei Sheng, Ruini Xue, Shiming Xu, Jin Zhang, Tian Xiao, Dehao Chen, Chuntao Hong, Jianian Yan, Dandan Song for their valuable feedback and suggestions. This research is supported by National High-Tech Research and Development Plan (863 plan) 2006AA01A105 and Chinese National 973 Basic Research Program 2007CB310900.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the logp model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1): 71–79, 1997.

[2] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. *The NAS Parallel Benchmarks 2.0*. NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 1995.

[3] K. J. Barker, S. Pakin, and D. J. Kerbyson. A performance model of the krak hydrodynamics application. In *ICPP'06*, pages 245–254, 2006.

[4] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *ICS'08*, pages 368–377, 2008.

[5] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic replay of MPI applications for interactive distributed debugging. In *EuroPVM/MPI*, pages 297–306, 2007.

[6] N. Choudhury, Y. Mehta, and T. L. W. et al. Scaling an optimistic parallel simulation of large-scale interconnection networks. In *WSC'05*, pages 591–600, 2005.

[7] A. Hoisie, O. Lubeck, and H. Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14(4):330–346, 2000.

[8] 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 *SC'01*, pages 37–48, 2001.

[9] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A parallel program development environment. In *Euro-Par'96*, pages 665–674, 1996.

[10] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987.

[11] 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 *PPoPP'07*, pages 249–258, 2007.

[12] LLNL. ASCI purple benchmark. URL `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks`.

[13] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS'04*, pages 2–13, 2004.

[14] M. Maruyama, T. Tsumura, and H. Nakashima. Parallel program debugging based on data-replay. In *PDCS'05*, pages 151–156, 2005.

[15] M. Mathias, D. Kerbyson, and A. Hoisie. A performance model of non-deterministic particle transport on large-scale systems. In *Workshop on Performance Modeling and Analysis*. ICCS, 2003.

[16] S. Prakash and R. Bagrodia. MPI-SIM: Using parallel simulation to evaluate MPI programs. In *Winter Simulation Conference*, pages 467–474, 1998.

[17] S. Shao, A. K. Jones, and R. G. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *IPDPS*, 2006.

[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57, 2002.

[19] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *SC'02*, pages 1–17, 2002.

[20] D. Sundaram-Stukel and M. K. Vernon. Predictive analysis of a wavefront application using LogGP. In *PPoPP*, pages 141–150, 1999.

[21] R. Susukita, H. Ando, and M. A. et al. Performance prediction of large-scale parallell system and application using macro-level simulation. In *SC'08*, pages 1–9, 2008.

[22] Tsinghua University. SIM-MPI simulator. URL `http://www.hpctest.org.cn/resources/sim-mpi.tgz`.

[23] T. Wilmarth, G. Zheng, and E. J. B. et al. Performance prediction using simulation of large-scale interconnection networks in POSE. In *Proc. 19th Workshop on Parallel and Distributed Simulation*, pages 109–118, 2005.

[24] 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 *PPoPP'09*, pages 251–260, 2009.

[25] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC'05*, page 40, 2005.

[26] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. FACT: fast communication trace collection for parallel applications through program slicing. In *SC'09*, 2009.

[27] G. Zheng, G. Kakulapati, and L. V. Kale. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *IPDPS'04*, pages 78–87, 2004.

[28] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI'04*, pages 255–266, 2004.