DOI: 10.2298/CSIS120118046L

SMP-SIM: an SMP-based Discrete-event Execution-driven Performance Simulator

Yufei Lin, Xinhai Xu, Yuhua Tang, Xin Zhang, and Xiaowei Guo

State Key Laboratory of High Performance Computing
National University of Defense Technology, Changsha, China
{linyufei,xuxinhai,yhtang}@nudt.edu.cn,zx2010@jfjb.com.cn, xiaow_g@126.com

Abstract. Designing and implementing a large-scale parallel system can be time-consuming and costly. It is therefore desirable to enable system developers to predict the performance of a parallel system at its design phase so that they can evaluate design alternatives to better meet performance requirements. Before the target machine is completely built, the developers can always build an symmetric multi-processor (SMP) for evaluation purposes. In this paper, we introduce an SMP-based discrete-event execution-driven performance simulation method for message passing interface (MPI) programs and describe the design and implementation of a simulator called SMP-SIM. As the processes share the same memory space in an SMP, SMP-SIM manages the events globally at the granularity of central processing units (CPUs). Furthermore, by re-implementing core MPI point-to-point communication primitives, SMP-SIM handles the communication virtually and sequential computation actually. Our experimental results show that SMP-SIM is highly accurate and scalable, resulting in errors of less than 7.60% for both SMP and SMP-Cluster target machines.

Keywords: simulator, SMP, MPI, performance prediction.

1. Introduction

Nowadays, large-scale parallel computers that comprise thousands of processors cost millions of dollars and take years to design and build. For system developers, it is greatly desired that the performance of a parallel system can be predicted efficiently and accurately at its design phase. This can help them evaluate different design alternatives to better meet performance requirements [25].

There are two popular performance prediction methods: model-based and discrete event simulation. The former [3,11] builds a parameterized model based on the signatures extracted from the given system, such as the number of cores, the number of instructions executed, and memory access patterns. Then the model is used to estimate the system performance. However, a model built for an application cannot be applied to another one. Moreover, as the system complexity increases, the factors that affect the system performance become too complex to be extracted thoroughly. So it can be difficult to obtain accurate predictions for large-scale parallel computers with the model-based method.

Discrete event simulation is capable of simulating large-scale parallel systems. The simulation procedure jumps from one state to another upon the occurrence of an event [5]. A simulator updates the simulation time and state information by processing the events. Events are generated either by the pre-extracted trace (trace-driven) or by the execution of the program (execution-driven). For trace-driven techniques, such as Dimemas [12], PERC [19] and SIM-MPI [18], the event trace is obtained with instrumentation tools by running the program before the simulator runs. For execution-driven techniques, such as WWT [17], WWT II [14], MPI-SIM [15] and BigSim [26], the events are generated when the program is running.

In discrete event simulation, the large-scale parallel machine to be conducted is called the ***target machine***, the machine on which the simulator executes is called the ***host machine***, and the program whose performance is to be predicted is called the ***target program***. Through analysis, we find that before the target machine is completely constructed, the system developers can always build a symmetric multi-processor (SMP), whose central processing units (CPUs) are the same as those in the target machine. If the target machine is an SMP, such as NEC SX-9 [20], CRAY CX1000-S [4] or IBM SP-SMP [8], the developers can use some of the target machine's CPUs to build a smaller SMP. If the target machine is an SMP-Cluster, a cluster whose nodes are SMPs, such as Roadrunner [2], Tianhe-1A [24] or K Computer [1], the developers can have at least one of the SMP nodes at the design phase. Therefore, we have opted to use the smaller SMP (or the SMP node) as the host machine to predict the performance of the target program on the target machine.

SMPs are well suited to discrete event simulation since the processes share the memory and are controlled by the same operating system. However, there has not been a simulation method utilizing these characteristics. Thus, in this paper, for message passing interface (MPI) [22], the de facto standard for parallel computing, we propose and design SMP-SIM, an SMP-based discrete-event execution-driven performance simulator. Our main contributions are as follows:

– We introduce a discrete-event execution-driven simulation method based on SMPs.
– We describe the design of SMP-SIM, a discrete-event execution-driven performance simulator, which consists of three modules, primitive decomposer, communication model, and event management.
– By re-implementing MPI core point-to-point communication primitives, we have implemented SMP-SIM based on MPICH2 [13].
– We show that SMP-SIM is highly accurate and scalable, with errors of less than 7.60% for both SMP and SMP-Cluster target machines.

The rest of this paper is organized as follows. Section 2 reviews the related works. Section 3 describes the basic ideas of SMP-SIM. Section 4 introduces the framework of SMP-SIM. Section 5 discusses the implementation of SMP-SIM based on MPICH2. Our experiments are presented in Section 6. Finally, Section 7 summarizes the paper.

2. Related Work

Discrete event simulation has been researched extensively in academia and industries. A number of simulators have been developed, including WWT [17] and WWT II [14] of University of Wisconsin, Dimemas [12] of CEPBA (European Center for Parallelism in Barcelona), MPI-SIM [15] of University of California, PERC [19] of San Diego Supercomputer Center, and BigSim [26] of University of Illinois at Urbana-Champaign. Among these simulators, MPI-SIM and BigSim are the most similar with SMP-SIM. All the three predict the performance while executing the target MPI applications.

In MPI-SIM, each process of the target program is simulated by a thread. MPI-SIM uses a portable library MPI-LITE to translate the target program into a multithreaded program, and measures the execution time of sequential computation codes when the multithreaded program is executing. However, threads differ from processs in terms of such program behaviours such as cache replacement policy and execution pattern. As a result, the sequential computation time cannot be measured accurately. Moreover, because MPI-SIM is not based on the SMP host machine, messages must carry the simulation timestamps that are used to calculate communication overheads.

BigSim is a simulator developed for BlueGene/C. The simulator contains two parts, a parallel-function emulator and a parallel-network simulator BigNetSim [27]. BigSim defines a set of application interfaces, such as addMessage and sendPacket, which are used to implement the MPI interfaces. All the application interfaces are executed by the simulator, and the other codes are directly executed on the host machine. In BigSim, the sequential computation time is calculated by heuristic approaches, which cannot lead to high accuracy. Several parallel programming languages are implemented on BigSim, including MPI, CHARM++ [10] and Adaptive MPI [7].

Zhai et al. [25] proposed a new method, called Phantom, to estimate the sequential computation time, which is used in a trace-driven simulator SIM-MPI [18]. Phantom needs to execute the target program twice. During the first execution, the communication traces of parallel applications are generated by intercepting all communication operations for each process and the computation between communication operations is marked as sequential computation unit. During the second execution, the real sequential computation time is measured on a target processing node for each process one by one. However, executing the target program twice is time-consuming. And it cannot deal with the uncertain programs because an inaccurate trace is usually generated.

The above simulation approaches are not based on SMP. To the best of our knowledge, there has not been an SMP-based performance simulator for MPI programs.

3. Basic Ideas of SMP-SIM

To design a discrete event simulator, three key questions need to be answered: what events exist in the simulator; how are the events generated; and how the events are processed for performance prediction. In this section, we introduce the basic idea behind SMP-SIM by addressing these questions. Without loss of generality, we assume that the core count of the target machine is equal to the process count of the target program. It should be noticed that as mentioned in Section 1, the CPUs in the host and target machines are the same.

3.1. Event Definition

For an MPI program, at any point during execution, a process is in either the *sequential computation state* or the *communication state*. So there are four types of events during the simulation of an MPI program: ***simulation start events***, ***simulation end events***, ***communication start events*** and ***sequential computation start events***. A simulation start event means that the simulation starts, i.e. a process enters the sequential computation state. A simulation end event means that the simulation ends, i.e. a process ends. A communication start event means that the process enters the communication state. A sequential computation start event means that the process enters the sequential computation state.

It is not difficult to find out that the simulation start event and the simulation end event for an MPI process are the first event and the last event, respectively, of the process. Communication start events and sequential computation start events are generated alternately, as shown in Fig. 1.

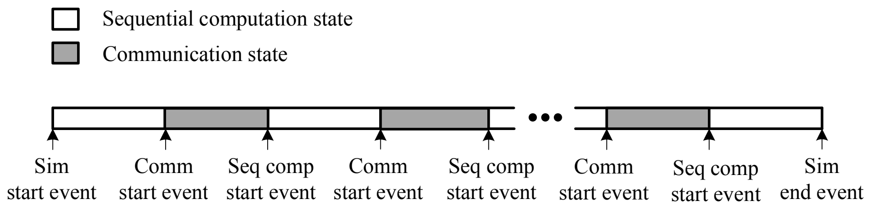

Fig. 1. The relationship between events and states.

3.2. Event Generation

Events are generated either by the pre-extracted trace (trace-driven) or by the execution of the program (execution-driven). For a trace-driven method, the event trace is extracted with instrumentation tools while running the program. Then the simulator uses the trace to drive the events and predict the execution time of the program. However, when predicting the performance of a large-scale parallel computing system, the time to extract the trace and the space to store it become intolerable. Moreover, due to some uncertain factors (e.g. branches, dynamic instruction generations and non-deterministic communications in the program), trace-driven simulation may be inaccurate with the incorrect trace acquired.

For an execution-driven method, events are generated during program execution. Thus, program behaviours such as branch prediction and dynamic instruction generation can all be simulated. Moreover, an execution-driven method can make use of available system resources to directly execute portions of the application code and simulate the features that are of specific interest or unavailable [16]. We prefer an execution-driven method over a trace-driven method, because the former is closer to the program execution reality and has higher efficiency. Therefore, in SMP-SIM, the events are generated when the target program is running on the host machine:

- The first executable statement in an MPI process is MPI_Init and the last one is MPI_Finalize. So the execution time of a process under consideration is the time period between MPI_Init and MPI_Finalize. Therefore, when the MPI process encounters MPI_Init and MPI_Finalize, the simulation start event and simulation end event are generated respectively.
- When encountering an MPI communication primitive, such as MPI_Ibsend or MPI_Irecv, a process enters the communication state and a communication start event is generated.
- When an MPI communication primitive finishes (i.e. a process returns from an MPI communication primitive), the process enters the sequential computation state and a sequential computation start event is generated.

3.3. Processing the Events

The aim of processing the events is to estimate the execution time of the target program running on the target machine. For this purpose, SMP-SIM maintains the simulation time for each process of the target program, denoted as t^s. When an event is generated when the target program is executing on the host machine, the simulation time of this process will be updated to the generation time of the event. The generation time of an event is the time when the event is generated if the target program executes on the target machine. Therefore, when a simulation end event is being processed, the simulation time of its corresponding process (i.e. the generation time of the simulation end event) represents the execution time of the process when it runs on the target machine.

As a parallel simulator that can be run on a small-scale SMP, SMP-SIM utilizes the characteristics of SMP to deal with the events efficiently and accurately:

- Manage the events globally at the granularity of CPUs. Because the processes allocated to a CPU on the host machine may outnumber the cores in the CPU and all the processes share the memory in SMP, SMP-SIM globally manages the events that are generated by the processes allocated to the same CPU by using a shared-segment. The event with the smallest generation time will always be selected to be processed first.

– Use a virtual-actual combined method to process the selected event. If it is a communication start event, it will be processed virtually, i.e. the communication overhead is estimated by the communication model; if it is a sequential computation start event, it will be processed actually, i.e. the sequential computation time is measured by direct execution.

It should be mentioned that for the MPI programs with non-deterministic communications (e.g. a receive request contains MPI_ANY_SOURCE as the source), the simulator needs a synchronization mechanism to make sure that the right messages (i.e. the messages that are received when the program executes on the target machine) are accepted during the simulation on the host machine. The synchronization mechanism used in SMP-SIM is optimistic mechanism [9] [21], which allows to process the earliest available event with no regard to safety. When an older message arrives, a rollback mechanism is needed to undo an earlier out of order execution and re-execute the events to guarantee the correct sequence of event processing. However, synchronization mechanism is not the focus of this work. The interested readers please refer to [9] for a detailed description.

4. Framework of SMP-SIM

Due to its good configuration, high performance and portability, MPICH [6] has become one of the most popular MPI libraries. SMP-SIM is designed based on MPICH. We have modified the MPICH library and integrated all the functionalities of SMP-SIM into the modified MPICH library.

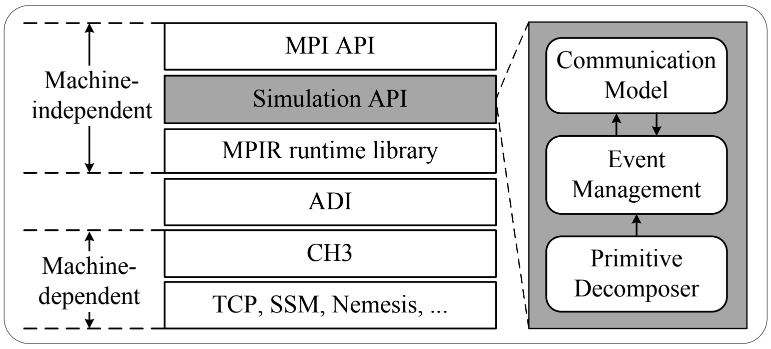

Fig. 2. The framework of SMP-SIM.

MPICH consists of two layers, a machine-independent layer and a machine-dependent layer, separated by an abstract device interface (ADI), as shown in Fig. 2. The machine-independent layer consists of an MPI application programmer interface (API) layer and an MPIR runtime library layer. The MPI API provides the user with programming interfaces and handles the MPI structures irrelevant to environments. The MPIR runtime library translates the complex MPI primitives in MPI API into point-to-point communication operations. The main functionalities of SMP-SIM are implemented by adding a ***Simulation API Layer*** between the MPI API and MPIR runtime library layers. As shown in Fig. 2, the simulation API layer comprises three modules: ***primitive decomposer***, ***communication model*** and ***event management***.

In the primitive decomposer module, all the MPI communication primitives are reconstructed by using core point-to-point communication primitives. This is the base of the other modules. When a process encounters a communication primitive as the target program runs on the host machine, the primitive decomposer module will decompose it into several core point-to-point communication primitives and then these core point-to-point communication primitives will be invoked one by one. The invocation and the return of a core point-to-point communication primitive will generate the corresponding events. When an event is generated, the event management module globally schedules the events, processes the selected event and updates the simulation time of the event's corresponding process. When processing a communication start event, the event management module will interact with the communication model module to calculate the time overheads of the corresponding core point-to-point primitive. Next, we will introduce these three modules in detail.

4.1. Primitive Decomposer

MPI provides users with a lot of communication primitives, including collective communication primitives and point-to-point communication primitives. In MPICH, all collective communication operations are implemented in terms of point-to-point communication operations. So, we choose four core point-to-point communication primitives from the MPI library: MPI_Ibsend (non-blocking buffered send), MPI_Issend (non-blocking synchronous send), MPI_Irecv (non-blocking receive) and MPI_Wait [15]. Using these four core point-to-point communication primitives, we can reconstruct the other point-to-point and collective communication primitives.

Table 1. Point-to-point communication primitives reconstruction in SMP-SIM

Primitive	**Functionality**	**Re-constructed by**
MPI_Bsend	Blocking buffered send	Both MPI_Ibsend and MPI_Wait
MPI_Ssend	Blocking synchronous send	Both MPI_Issend and MPI_Wait
MPI_Rsend	Blocking ready send	MPI_Ssend
MPI_Send	Blocking standard send	Either MPI_Bsend or MPI_Ssend
MPI_Recv	Blocking standard receive	Both MPI_Irecv and MPI_Wait

Table 1 lists the ways to reconstruct the other five point-to-point communication primitives in MPI. The collective communication primitives can be reconstructed by four core point-to-point communication primitives and the five primitives listed in the first column of Table 1. When a process of the target program encounters a communication primitive on the host machine, the primitive decomposer module will decompose it into several core point-to-point communication primitives and then these core point-to-point communication primitive will be invoked one by one. Consequently, the corresponding events will be generated. Therefore, the communication start events mentioned in Section 3.1 can be further divided into four types, *non-blocking buffered send start events*, *non-blocking synchronous send start events*, *non-blocking receive start events* and *wait start events*, which are generated due to the invocation of MPI_Ibsend, MPI_Issend, MPI_Irecv and MPI_Wait, respectively. All the events that will appear in SMP-SIM are listed in Table 2, where the four items in the third row are all communication start events.

Table 2. All the events in SMP-SIM

Event	Symbol	Generated by
Simulation start event	E_{Start}	Invocation of MPI_Init
Non-blocking buffered send start event Non-blocking synchronous send start event Non-blocking receive start event Wait start event	E_{Ibsend} E_{Issend} E_{Irecv} E_{Wait}	Invocation of MPI_Ibsend Invocation of MPI_Issend Invocation of MPI_Irecv Invocation of MPI_Wait
Sequential computation start event	E_{Seq}	Return of MPI_Init, MPI_Ibsend, MPI_Issend, MPI_Irecv, MPI_Wait
Simulation end event	E_{End}	Invocation of MPI_Finalize

4.2. Communication Model

This module is responsible for calculating the time overheads of the point-to-point communication primitives. First, we list all the symbols to be used in Table 3. Then we discuss how to calculate the time overheads of core point-to-point communication primitives. The way to calculate the time overhead of an core point-to-point communication primitive is related to the two communicating processes' physical distributions in the target machine.

MPI_Ibsend. MPI_Ibsend starts a non-blocking buffered send. A buffer space for the data needs to be reserved at the sender side, and the message is always sent no matter whether there is a matching receive at the receiver side.

Table 3. Symbols to be used in Section 4.2

Symbol	**Meaning**
$t^{send}_{generate}$	Invocation time of MPI_Ibsend or MPI_Issend[*]
t^{send}_{return}	Return time of MPI_Ibsend or MPI_Issend
t^{send}_{safe}	The time when MPI_Ibsend or MPI_Issend is safe[**]
$t^{recv}_{generate}$	Invocation time of MPI_Irecv
t^{recv}_{return}	Return time of MPI_Irecv
t^{recv}_{safe}	The time when MPI_Irecv is safe
$t^{wait}_{generate}$	Invocation time of MPI_Wait
t^{wait}_{return}	Return time of MPI_Wait
t_{arrive}	The time when the message arrives at the receiver
L_{mem}	Memory latency
B_{mem}	Memory bandwidth
L_{net}	Network latency
B_{net}	Network bandwidth
m	Message (data) size
δt	The time overhead of invocation a primitive
O_a	The time overhead of reserving a buffer space

[*] The invocation time of an primitive is the generation time of the event generated by the primitive.
[**] The time when a communication primitive is safe is the time when its corresponding data are safe.

As shown in Fig. 3(a), if two communicating processes are physically distributed, MPI_Ibsend returns as soon as the buffer at the sender side is available. The data can be sent to the network while they are being copied to the buffer (generally speaking, $B_{mem} > B_{net}$), according to the configurations of the current machines. The data are safe when they have been completely copied to the buffer. We can calculate t^{send}_{return}, t^{send}_{safe} and t_{arrive} as in Equations 1 – 3:

$$t^{send}_{return} = t^{send}_{generate} + \delta t + O_a \tag{1}$$

$$t^{send}_{safe} = t^{send}_{generate} + \delta t + O_a + m/B_{mem} \tag{2}$$

$$t_{arrive} = t^{send}_{generate} + \delta t + O_a + L_{net} + m/B_{net} \tag{3}$$

As shown in Fig. 3(b), if two communicating processes are physically centralized, the buffer will not be reserved and MPI_Ibsend returns as soon as it is invoked. Then, we have:

$$t^{send}_{return} = t^{send}_{generate} + \delta t \tag{4}$$

$$t^{send}_{safe} = t^{send}_{generate} + \delta t + m/B_{mem} \tag{5}$$

$$t_{arrive} = t^{send}_{generate} + \delta t + L_{mem} + m/B_{mem} \tag{6}$$

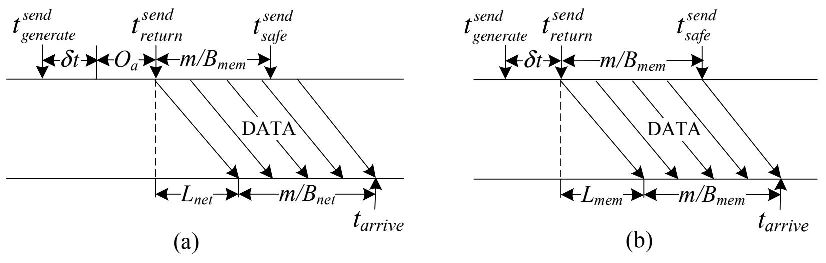

Fig. 3. Non-blocking buffered send. (a) Physically distributed. (b) Physically centralized.

MPI_Issend. MPI_Issend starts a non-blocking synchronous send. A handshake, which is used to make sure the matching receive has started, happens between the sender and the receiver via REQ and ACK messages before the data message is sent to the receiver. MPI_Issend will return when ACK from the receiver is received. The return time of MPI_Issend is affected by the relationship between $t^{recv}_{generate}$ and the REQ's arrival time (denoted as t_{REQ}).

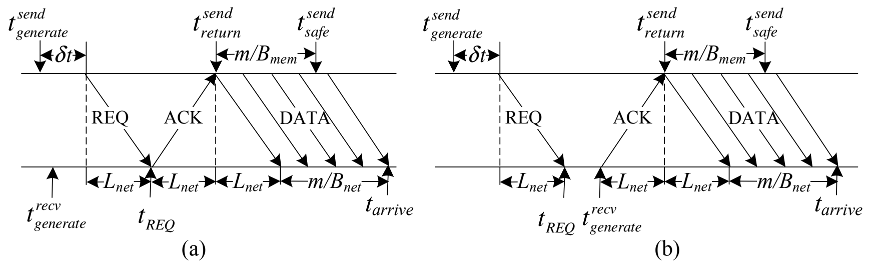

Fig. 4. Non-blocking buffered send (the sender and receiver are physically distributed) (a)$t^{recv}_{generate} \leqslant t_{REQ}$. (b)$t^{recv}_{generate} > t_{REQ}$.

Fig. 4(a)(b) shows the case where two communicating processes are physically distributed in target machine. The data can be sent to network while they are being copied to the buffer, and the data are safe after they have been copied to the buffer. t^{send}_{return}, t^{send}_{safe} and t_{arrive} can be calculated as in Equations 8 – 10:

$$t_{REQ} = t^{send}_{generate} + \delta t + L_{net} \tag{7}$$

$$t^{send}_{return} = \begin{cases} t^{send}_{generate} + \delta t + L_{net}, & t^{recv}_{generate} \leqslant t_{REQ} \\ t^{recv}_{generate} + L_{net}, & t^{recv}_{generate} > t_{REQ} \end{cases} \tag{8}$$

$$t^{send}_{safe} = \begin{cases} t^{send}_{generate} + \delta t + 2L_{net} + m/B_{mem}, & t^{recv}_{generate} \leqslant t_{REQ} \\ t^{recv}_{generate} + L_{net} + m/B_{mem}, & t^{recv}_{generate} > t_{REQ} \end{cases} \tag{9}$$

$$t_{arrive} = \begin{cases} t^{send}_{generate} + \delta t + 3L_{net} + m/B_{net}, & t^{recv}_{generate} \leqslant t_{REQ} \\ t^{recv}_{generate} + 2L_{net} + m/B_{net}, & t^{recv}_{generate} > t_{REQ} \end{cases} \tag{10}$$

Due to the space limitation, we will not analyze the case where the communicating processes are physically centralized. We can calculate t^{send}_{return}, t^{send}_{safe} and t_{arrive} of this case by replacing the L_{net} and B_{net} in Equations 8 – 10 with L_{mem} and B_{mem}, respectively.

MPI_Irecv. MPI_Irecv returns as soon as it is invoked. The return of MPI_Irecv means that the message can be received, rather than having been received. Then, we have:

$$t^{recv}_{return} = t^{recv}_{generate} + \delta t \tag{11}$$

MPI_Wait. MPI_Wait is used to guarantee the safety of the data. It will block the process until the data sent or received by its related non-blocking communication are safe. The return time of MPI_Wait t^{wait}_{return} is the maximum value of the time when the data are safe (i.e. t^{send}_{safe} or t^{recv}_{safe}) and the invocation time of MPI_Wait $t^{wait}_{generate}$. So, we have:

$$t^{wait}_{return} = \begin{cases} max\{t^{send}_{safe}, t^{wait}_{generate}\}, \text{if the waiting object is a non-blocking send} \\ max\{t^{recv}_{safe}, t^{wait}_{generate}\}, \text{if the waiting object is a non-blocking receive} \end{cases} \tag{12}$$

t^{send}_{safe} in Equation 12 can be calculated as shown in Equations 2, 5 and 9. Then, we analyze the way to calculate t^{recv}_{safe}.

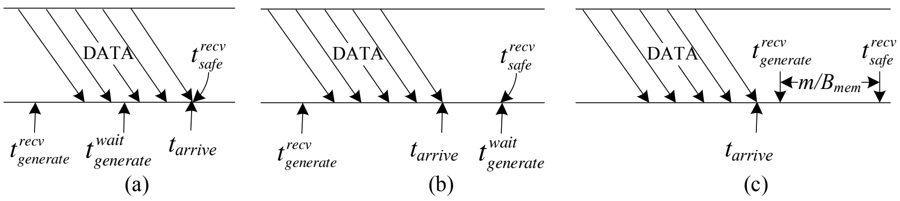

Fig. 5. Non-blocking receive. (a) $t^{recv}_{generate} < t_{arrive}$, $t^{wait}_{generate} \leqslant t_{arrive}$. (b) $t^{recv}_{generate} < t_{arrive}$, $t^{wait}_{generate} > t_{arrive}$. (c) $t^{recv}_{generate} \geqslant t_{arrive}$.

t^{recv}_{safe} is affected by the relationships among $t^{recv}_{generate}$, $t^{wait}_{generate}$ and t_{arrive}, as shown in Fig. 5. It is worth mentioning that the case shown in Fig. 5(c) only exists when the send primitive is MPI_Ibsend. From Fig. 5, we can derive that

$$t^{recv}_{safe} = \begin{cases} max\{t_{arrive}, t^{wait}_{generate}\}, & t^{recv}_{generate} < t_{arrive} \\ t^{recv}_{generate} + m/B_{mem}, & t^{recv}_{generate} \geqslant t_{arrive} \end{cases} \tag{13}$$

where t_{arrive} can be calculated as shown in Equations 3, 6 and 10.

4.3. Event Management Module

SMP-SIM is a parallel simulator, and the processes are fixed to their own CPUs while the target programs are running on the host machine. The target machine has more CPUs than the host machine and the core count of the target machine is equal to the process count of the target program. So the processes allocated to a host machine's CPU outnumber the cores in it. The event management module maintains a *waiting event queue* for each CPU, i.e. the processes allocated to the same CPU share a waiting event queue. At the beginning of running a target program on the host machine, this module inserts an E_{Start} event for each process into its corresponding waiting event queue and sets the generation time of this event as -1. During the execution of the target program, the module manages the events at the granularity of CPUs using the workflow:

Step 1. The event management module selects the *un-processed executable event* e that has the smallest generation time from the CPU's waiting event queue. It should be noticed that E_{Start}, E_{End}, E_{Seq}, E_{Ibsend} and E_{Irecv} events are always executable; E_{Issend} is executable only when its corresponding receive primitive has been invoked; and E_{Wait} is executable only when the data of its related send or receive primitive are safe. For convenience, for a given event e, we use $e.t_{generate}$ to represent the generation time of e, $e.doing$ to determine whether e is being processed and $e.executable$ to determine whether e is executable at present.

Step 2. The event management module switches the process that is related to e onto an idle core and then processes e. As shown in Algorithm 1, the detailed dealing methods are different among different types of events. The *completing event queue* is used to store the E_{Ibsend}, E_{Issend} and E_{Irecv} events, which have already been processed, and the information of these events is necessary when processing the corresponding E_{Wait} events. For convenience, for a given communication event e, we use $e.t_{return}$, $e.t_{arrive}$ and $e.t_{safe}$ to refer to the return time, arrival time and the data's safe time of e's corresponding communication primitive respectively.

Step 3. After accomplishing the processing of the selected event e, the event management module deletes e from the waiting event queue and may insert a new event e_{new} into the waiting event queue as follows:

- $e.type$ is E_{Start}, E_{Ibsend}, E_{Issend}, E_{Irecv} or E_{Wait}: after executing the communication operations corresponding to event e, the process will deal with sequential computation codes, so $e_{new}.type = E_{Seq}$ and $e_{new}.executalbe = True$.
- $e.type$ is E_{Seq}: denote the MPI primitive behind the sequential computation codes as S.
  - S is MPI_Ibsend: $e_{new}.type = E_{Ibsend}$ and $e_{new}.executalbe = True$.
  - S is MPI_Issend: $e_{new}.type = E_{Issend}$; if the E_{Irecv} event that matches e_{new} is in the waiting/completing event queue, $e_{new}.executalbe = True$, otherwise $e_{new}.executalbe = False$.
  - S is MPI_Irecv: $e_{new}.type = E_{Irecv}$ and $e_{new}.executalbe = True$.

Algorithm 1 The Algorithm of EventHandler

Input: Event e
Output: Void

```
1.  $e.doing \leftarrow True$
2.  if $e.type = E_{Start}$ then
3.      execute the body codes of MPI_Init;
4.      $t^s \leftarrow 0$;
5.  else if $e.type = E_{End}$ then
6.      print $t^s$;
7.      execute the body codes of MPI_Finalize;
8.  else if $e.type = E_{Seq}$ then
9.      get the current system time $t_1$;
10.     execute the sequential computation codes;
11.     get the current system time $t_2$;
12.     $t^s \leftarrow t^s + (t_2 - t_1)$;
13. else if $e.type = E_{Ibsend}$ then
14.     execute the body codes of MPI_Ibsend;
15.     calculate $e.t_{return}$, $e.t_{safe}$ and $e.t_{arrive}$ based on Equations 1 – 3 and 4 – 6;
16.     if there is the $E_{Irecv}$ event $e'$ matched with $e$ in waiting event queue then
17.         $e'.t_{arrive} \leftarrow e.t_{arrive}$
18.     else if there is the $E_{Irecv}$ event $e'$ matched with $e$ in completing event queue then
19.         $e'.t_{arrive} \leftarrow e.t_{arrive}$
20.         if there is the $E_{Wait}$ event $e''$ matched with $e'$ in waiting event queue then
21.             $e''.executable \leftarrow True$;
22.         end if
23.     end if
24.     $t^s \leftarrow e.t_{return}$;
25.     insert $e$ into its completing event queue;
26. else if $e.type = E_{Issend}$ then
27.     execute the body codes of MPI_Issend;
28.     get $E_{Irecv}$ event $e'$ which matches with $e$ from the completing event queue;
29.     calculate $e.t_{return}$, $e.t_{safe}$ and $e'.t_{arrive}$ based on Equations 8 – 10;
30.     if there is the $E_{Wait}$ event $e''$ matched with $e'$ in waiting event queue then
31.         $e''.executable \leftarrow True$;
32.     end if
33.     $t^s \leftarrow e.t_{return}$;
34.     insert $e$ into its completing event queue;
35. else if $e.type = E_{Irecv}$ then
36.     execute the body codes of MPI_Irecv;
37.     $e.t_{arrive} \leftarrow -1$;
38.     if there is the $E_{Ibsend}$ event $e'$ matched with $e$ in completing event queue then
39.         $e.t_{arrive} \leftarrow e'.t_{arrive}$;
40.     else if there is the $E_{Issend}$ event $e'$ matched with $e$ in waiting event queue then
41.         $e'.executable \leftarrow True$;
42.     end if
43.     $t^s \leftarrow t^s + \delta t$;
44.     insert $e$ into its completing event queue;
45. else if $e.type = E_{Wait}$ then
46.     execute the body codes of MPI_Wait;
47.     get event $e'$ which matches with $e$ from the completing event queue;
48.     calculate $e.t_{return}$ based on Equations 12 and 13;
49.     delete $e'$ from the completing event queue;
50. end if
```

  - S is MPI_Wait: $e_{new}.type = E_{Wait}$; if $e'.t_{arrive} > 0$ (e' is the E_{Irecv} event that matches e_{new}), $e_{new}.executalbe = True$, otherwise $e_{new}.executalbe = False$.
  - S is MPI_Finalize: $e_{new}.type = E_{End}$ and $e_{new}.executalbe = True$.
- $e.type$ is E_{End}: the process finishes after executing MPI_Finalize, so event management module will not insert any new event into the waiting event queue, i.e. no event e_{new} will be created.

Whatever the type of e_{new} is, $e_{new}.doing = False$ always holds after inserting e_{new} into the waiting event queue.

After the work of step 2 and step 3, if there is any un-processed event in the waiting event queue, the event management module will start the work of step 1 again and deal with the new selected event as described in step 2 and step 3. It is not difficult to find out that for each process, there is only one related event in the waiting event queue at any time during the simulation. Because only the processes whose events are selected to be processed can run, the unselected processes must be suspended. Processes allocated to the same CPU may be switched when a new event is selected to be processed, and the detailed implementation will be introduced in Section 5.

5. MPICH2-based Implementation of SMP-SIM

This section describes the MPICH2-based implementation of SMP-SIM. By recognizing the fact that the processes share the memory in SMPs, SMP-SIM creates event queues in the shared segment for all the processes at the granularity of CPUs. SMP-SIM re-implements the core MPI primitives and integrates the functionalities of event management into them.

Fig. 6 shows the directory of the source codes of MPICH2. The major modification of the implementation of SMP-SIM is in the sub-directory /src/mpi. The implementations of MPI_Init and MPI_Finalize are in /src/mpi/init; the implementations of all point-to-point MPI primitives are in /src/mpi/pt2pt; and the implementations of all collective MPI primitives are in /src/mpi/coll.

5.1. Data Structure

Event is the core of SMP-SIM, and we implement the data structure of an event as shown in Fig. 7.

- Attributes $type$, $processID$, $doing$, $executable$ and $t\_generate$ are used for all types of events. $type$, $processID$ and $t\_generate$ stand for the type, process ID and generation time of the event respectively; $doing$ and $executable$ judge whether the event is being and can be processed respectively.
- Attributes t_return, t_arrive, t_safe, $pairID$ and tag are used for three types of events: E_{Ibsend}, E_{Issend} and E_{Irecv}. These attributes stand for the return time, message arrival time, safe time, matched process ID and tag of the corresponding communication primitive respectively. Given an E_{Ibsend},

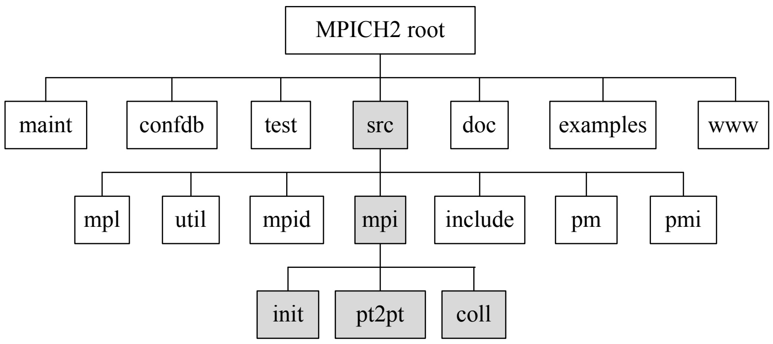

Fig. 6. The directory of the source codes of MPICH2

```
struct Event{
    int type;
    int processID;
    bool doing;
    bool executable;
    double t_generate;
    double t_return;
    double t_arrive;
    double t_safe;
    int pairID;
    int tag;
    MPI_Request *request;
}
```

Fig. 7. The data structure of event

E_{Issend} or E_{Irecv} event, we can find the matched E_{Irecv}, E_{Ibsend} or E_{Issend} event with $pairID$ and tag.

– Attribute $request$ is used for four types of events: E_{Ibsend}, E_{Issend}, E_{Irecv} and E_{Wait}. This attribute stands for the object of the corresponding non-blocking communication. Given an E_{Wait} event, we can find the matched E_{Ibsend}, E_{Issend} or E_{Irecv} event with $request$.

Based on the definition of the data structure of an event, we create two event queues, $Wait\_Event\_Queue$ and $Complete\_Event\_Queue$, at the granularity of CPUs, and store them in the shared segment so that all the processes can access them conveniently. It should be noticed that in order to guarantee correctness, all the operations on the event queues must be protected by the lock mechanism, i.e. only one process is permitted to access a given event queue at one time. In order to briefly describe our implementation, all the operations on the event queues described in the rest of this section are encapsulated by $lock()$ and $unlock()$ implicitly.

5.2. Re-implementations of Core Primitives

This subsection introduce the re-implementations of six core MPI primitives MPI_Init, MPI_Finalize, MPI_Ibsend, MPI_Issend, MPI_Irecv and MPI_Wait. These re-implementations realize the functionalities of event management described in Section 4.3.

```
MPI_Init(){
   Codes of original body;

   // codes added for SMP-SIM
   new a event e;
   e.type=$E_{Seq}$; e.processID=myRank; e.doing=False;
   e.t_executable=True; e.t_generate=0; $t^s$=0;
   Insert e into its Wait_Event_Queue;
   MPI_Barrier();
   lock();
   if(Count<CORE_NUM){
      Count++;
      set the priority of this process HIGH;
      if(Count == CORE_NUM)
         suspend all the processes allocated to this CPU without HIGH priority;
   }
   unlock();
   e.doing=True;
   $t_1$=gettime();

   return;
}
```

Fig. 8. The re-implementation of MPI_Init

MPI_Init As shown in Fig. 8, besides the codes of the original MPI_Init body, the re-implementation of MPI_Init does the work as follows: firstly, each process inserts an E_{Sqe} event into its corresponding $Wait\_Event\_Queue$; secondly, make sure that only one process is running on a given processing core and the other processes are suspended; thirdly, the process that has not been suspended starts to deal with the E_{Sqe} event.

In the re-implementation of MPI_Init, $Count$ is a global variable shared by all the processes allocated to the same CPU, and it is used for logging the count of the processes whose E_{Sqe} events have been started to be processed. $lock()$ and $unlock()$ are two functions used to protect the critical section between them. While the target program is running, the process switching mechanism is the priority scheduling mechanism instead of the time-sliced round-robin mechanism, so the line *set the priority of this process HIGH* in Fig. 8 guarantees that the current running process can not be switched.

```
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request){
    // codes added for SMP-SIM: remove $E_{Sqe}$ type event
    $t_2$=gettime();
    remove the event whose processID is myRnak from the Wait_Event_Queue;

    // codes added for SMP-SIM: generate $E_{Ibsend}$ type event
    new a event e;
    e.type=$E_{Ibsend}$; e.processID=myRank; e.doing=False;
    e.t_executable=True; e.t_generate=$t^s$+($t_2$-$t_1$);  $t^s$=e.t_generate;
    e.pairID=dest; e.tag=tag; e.request=request;
    Insert e into its Wait_Event_Queue;
    pID = findNextEvent();
    if (pID != myRank)
        suspend the current process and switch to the process whose ID is pID;

    Codes of original body;

    // codes added for SMP-SIM: deal with event e
    execute the codes whose function is same as line 15-25 in Algorithm 1;

    // codes added for SMP-SIM: generate $E_{Seq}$ type event
    new a event e1;
    e1.type=$E_{Seq}$; e1.processID=myRank; e1.doing=False;
    e1.t_executable=True; e1.t_generate=$t^s$;
    Insert e1 into its Wait_Event_Queue;
    pID = findNextEvent();
    if (pID != myRank)
        suspend the current process and switch to the process whose ID is pID;
    $t_1$=gettime();

    return;
}
```

Fig. 9. The re-implementation of MPI_Ibsend

MPI_Ibsend, MPI_Issend, MPI_Irecv and MPI_Wait The re-implementations of MPI_Ibsend, MPI_Issend, MPI_Irecv and MPI_Wait are similar, and the major work is divided into four steps as follows:

- Firstly, remove the E_{Seq} event that has just been processed from the $Wait\_Event\_Queue$.
- Secondly, insert a new communication event according to this MPI communication primitive into the $Wait\_Event\_Queue$; select the next event to process from the $Wait\_Event\_Queue$; switch to the process corresponding to the selected event.
- Thirdly, after this process is switched on again, execute the codes of the original body and process the event related to this primitive.
- Fourthly, insert a new E_{Seq} event into the $Wait\_Event\_Queue$; select the next event to process from the $Wait\_Event\_Queue$; switch to the process corresponding to the selected event.

The re-implementation of MPI_Ibsend is shown in Fig. 9, where the function $findNextEvent()$ is used for selecting the un-processed executable event e with the minimum generation time, and the return value of $findNextEvent()$ is $e.processID$. The major difference between MPI_Issend(/MPI_Irecv/MPI_Wait) and MPI_Ibsend is the work of processing the event related to the primitive, and the detailed processing methods can be found in Algorithm 1.

```
MPI_Finalize(){
   // codes added for SMP-SIM
   $t_2$=gettime();
   remove the event whose processID is myRnak from the Wait_Event_Queue;
   $t^s = t^s + (t_2 - t_1)$;
   print $t^s$;

   Codes of original body;
   return;
}
```

Fig. 10. The re-implementation of MPI_Finalize

MPI_Finalize As shown in Fig. 10, in the re-implementation of MPI_Finalize, before executing the codes of the original MPI_Finalize body, we first remove the E_{Seq} event that has just been processed from the $Wait\_Event\_Queue$, and then update and print the current simulation time of this process.

6. Experiments

We demonstrate the validation and accuracy of SMP-SIM in this section. Section 6.1 introduces the benchmarks and experimental platform used. Section 6.2 describes the methodology used for evaluating our work. Section 6.3 presents and analyzes the experimental results.

6.1. Benchmarks and Platform

We select three parallel kernels EP, CG, FT from NPB3.3-MPI benchmarks and Sweep3D-2.2d to evaluate SMP-SIM. The problem sizes of EP, CG and FT are all Class C, and the problem size of Sweep3D is $100 \times 100 \times 100$. All the benchmarks are executed in Red Hat Enterprise Linux Server Release 5.5, and MPICH2-1.3.1 is used.

Our platform is a cluster with eight nodes, and each node is an SMP equipped with two 2.93G Intel Xeon X5670 CPUs and 24GB RAM. The interconnection is described in [23]. Notice that although there are 6 cores in Xeon X5670 CPU, we only use it as a 4-core CPU because most of the benchmarks must be executed by 2^n processes.

6.2. Evaluation Methodology

In order to validate the accuracy of SMP-SIM for different target machines, our experiments include two parts:

– SMP target machine: for a single SMP node in our platform, firstly, we use the two CPU as the target machine and only one CPU as the host machine, and test the accuracy of SMP-SIM; secondly, we use only one core as the host machine, and test the scalability of SMP-SIM with varying the scale of the target machine.
– SMP-Cluster target machine: for the whole platform, we use all the eight nodes as the target machine and only one node as the host machine, and test the accuracy of SMP-SIM.

6.3. Results and Analysis

SMP Target Machine As shown in Fig. 11, when using two CPUs as the target machine and only one CPU as the host machine, for all the four benchmarks, the errors of SMP-SIM are between 2.56% and 6.14%. The accuracy of SMP-SIM for EP benchmark is the highest, because EP has the fewest communications. The sequential computation time is measured by direct execution, so the simulation of sequential computation is more accurate than that of communication. Consequently, the more communications a benchmark contains, the lower accuracy of SMP-SIM is.

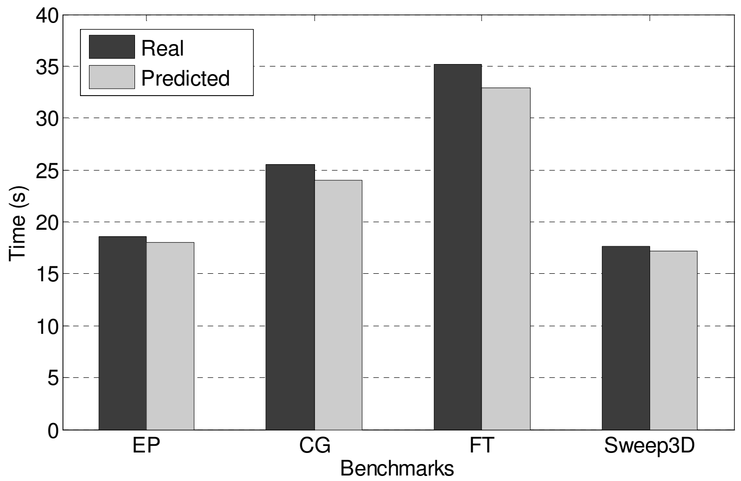

Fig. 11. The accuracy of SMP-SIM for SMP target machine

In order to test the scalability of SMP-SIM, we fix the host machine as a single processing core, and vary the scale of the target machine as 2, 4, and

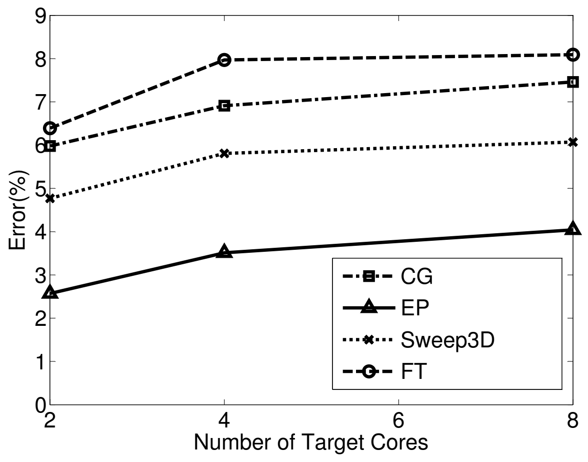

Fig. 12. The scalability of SMP-SIM

8 processing cores. As shown in Fig. 12, the results show that SMP-SIM has good scalability.

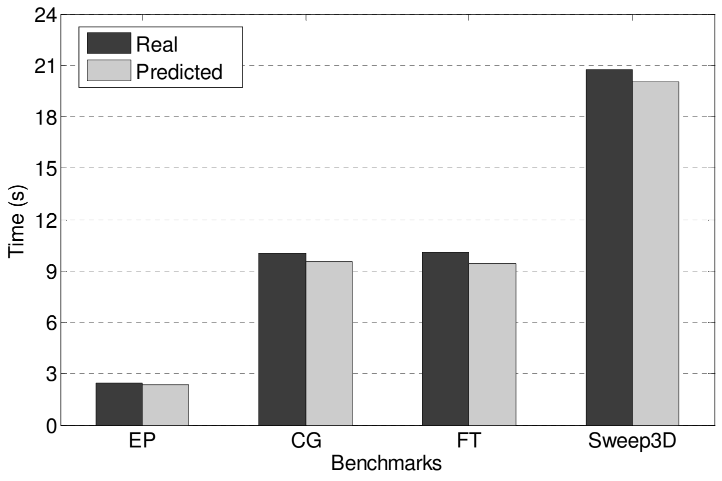

Fig. 13. The accuracy of SMP-SIM for SMP-Cluster target machine

SMP-Cluster Target Machine As shown in Fig. 13, when using all the eight nodes as the target machine and only one node as the host machine, for all the four benchmarks, the accuracies of SMP-SIM are high and the errors are between 2.67% and 7.60%.

7. Conclusion

For the system developers, it has been long desired that the performance of a parallel system can be predicted at the design phase. Before the target machine is completely constructed, the developers can always build an SMP machine used as a host machine. In this paper, we introduce SMP-SIM, an SMP-based discrete-event execution-driven performance simulator that exploits the characteristics of SMP to achieve accurate and scalable performance prediction.

In SMP-SIM, we have integrated three modules, primitive decomposer, communication model and event management, by adding a simulation API layer on top of the MPICH library. By executing the target program on the host machine with the modified MPICH library, SMP-SIM manages the events globally, handles the communication start events virtually and sequential computation start events actually. In our experimental evaluation, SMP-SIM shows high accuracy and good scalability with prediction errors of less than 7.60%.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (NSFC) No.60921062.

References

1. Ajima, Y., Takagi, Y., Inoue, T., Hiramoto, S., Shimizu, T.: The tofu interconnect. In: High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on. pp. 87–94 (aug 2011)
2. Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., Sancho, J.C.: Entering the petaflop era: the architecture and performance of roadrunner. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. pp. 1:1–1:11. SC '08, IEEE Press, Piscataway, NJ, USA (2008)
3. Barker, K., Pakin, S., Kerbyson, D.: A performance model of the krak hydrodynamics application. In: Parallel Processing, 2006. ICPP 2006. International Conference on. pp. 245 –254 (aug 2006)
4. Website, http://www.cray.com/Products/CX1000/Chassis/CX1000S.aspx
5. Fujimoto, R.M.: Parallel discrete event simulation. Commun. ACM 33, 30–53 (October 1990)
6. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the mpi message passing interface standard. Parallel Comput. 22, 789–828 (September 1996)
7. Huang, C., Lawlor, O., Kal, L.: Adaptive mpi. In: Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science, vol. 2958, pp. 306–322. Springer Berlin / Heidelberg (2004)
8. Website, http://static.msi.umn.edu/tutorial/scicomp/sp/intro-Power3SP/index.html
9. Jefferson, D., Beckman, B., Wieland, F., L., B., Diloreto, M.: Time warp operating system. SIGOPS Oper. Syst. Rev. 21, 77–93 (November 1987)
10. Kale, L.V., Krishnan, S.: Charm++: a portable concurrent object oriented system based on c++. SIGPLAN Not. 28(10), 91–108 (Oct 1993)
11. Kerbyson, D.J., Alme, H.J., Hoisie, A., Petrini, F., Wasserman, H.J., Gittings, M.: Predictive performance and scalability modeling of a large-scale application. In: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM). pp. 37–37. Supercomputing '01, ACM, New York, NY, USA (2001)

12. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: Dip: A parallel program development environment. Lecture Notes in Computer Science 1124, 665 – 665 (1996)
13. Website, http://www.mcs.anl.gov/research/projects/mpich2/
14. Mukherjee, S.S., Reinhardt, S.K., Falsafi, B., Litzkow, M., Hill, M.D., Wood, D.A., Huss-Lederman, S., Larus, J.R.: Wisconsin wind tunnel ii: A fast, portable parallel architecture simulator. IEEE Concurrency 8, 12–20 (October 2000)
15. Prakash, S., Bagrodia, R.L.: Mpi-sim: using parallel simulation to evaluate mpi programs. In: Proceedings of the 30th conference on Winter simulation. pp. 467–474. WSC '98, IEEE Computer Society Press, Los Alamitos, CA, USA (1998)
16. Prakash, S., Deelman, E., Bagrodia, R.: Asynchronous parallel simulation of parallel programs. IEEE Trans. Softw. Eng. 26, 385–400 (May 2000)
17. Reinhardt, S.K., Hill, M.D., Larus, J.R., Lebeck, A.R., Lewis, J.C., Wood, D.A.: The wisconsin wind tunnel: virtual prototyping of parallel computers. In: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems. pp. 48–60. SIGMETRICS '93, ACM, New York, NY, USA (1993)
18. Website, http://sourceforge.net/projects/sim-mpi/
19. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A.: A framework for performance modeling and prediction. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing. pp. 1–17. Supercomputing '02, IEEE Computer Society Press, Los Alamitos, CA, USA (2002)
20. Soga, T., Musa, A., Shimomura, Y., Egawa, R., Itakura, K., Takizawa, H., Okabe, K., Kobayashi, H.: Performance evaluation of nec sx-9 using real science and engineering applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. pp. 28:1–28:12. SC '09, ACM, New York, NY, USA (2009)
21. Steinman, J.S.: Breathing time warp. SIGSIM Simul. Dig. 23, 109–118 (July 1993)
22. Website, http://www-unix.mcs.anl.gov/mpi
23. Xie, M., Lu, Y., Liu, L., Cao, H., Yang, X.: Implementation and evaluation of network interface and message passing services for tianhe-1a supercomputer. In: Proceedings of the 2011 IEEE 19th Annual Symposium on High Performance Interconnects. pp. 78–86. HOTI '11, IEEE Computer Society, Washington, DC, USA (2011)
24. Yang, X.J., Liao, X.K., Lu, K., Hu, Q.F., Song, J.Q., Su, J.S.: The tianhe-1a supercomputer: Its hardware and software. Journal of Computer Science and Technology 26(3), 344–351 (2011)
25. Zhai, J., Chen, W., Zheng, W.: Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node. In: Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP. pp. 305 – 314. Bangalore, India (2010)
26. Zheng, G., Kakulapati, G., Kale, L.: Bigsim: a parallel simulator for performance prediction of extremely large parallel machines. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International. p. 78 (april 2004)
27. Zheng, G., Wilmarth, T., Lawlor, O., Kale, L., Adve, S., Padua, D., Guebelle, P.: Performance modeling and programming environments for petaflops computers and the blue gene machine. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International. p. 197 (april 2004)

Yufei Lin was born in 1985. She received her B.S. degree in automation from the University of Science and Technology of China (USTC) in 2006 and M.S. degree in computer science from the National University of Defense Technology (NUDT) in 2008. She is now a Ph.D. candidate in Computer Science from State Key Laboratory of High Performance Computing at NUDT. Her research interest lies in high performance computing and performance evaluation. Her email address is linyufei@nudt.edu.cn.

Xinhai Xu was born in 1984. He received his B.S., M.S. and Ph.D. degrees in computer science from NUDT in 2006, 2008 and 2012. He is now an assistant professor from School of Computer Science at NUDT. His research interest lies in high performance computing and fault tolerance. His email address is xuxinhai@nudt.edu.cn.

Yuhua Tang was born in 1962. She received her B.S. and M.S. degrees in computer science from NUDT in 1983 and 1985. She is now a professor from School of Computer Science at NUDT. Her research interest lies in high performance computing and its router design. Her email address is yhtang@nudt.edu.cn.

Xin Zhang was born in 1986. He received his B.S. degree in computer science from NUDT in 2009. He is now an M.S. candidate in Computer Science from School of Computer Science at NUDT. His research interest lies in performance evaluation. His email address is zx2010@jfjb.com.cn.

Xiaowei Guo was born in 1986. He received his B.S. and M.S. degrees in computer science from NUDT in 2009 and 2011. He is now a Ph.D. candidate in Computer Science from State Key Laboratory of High Performance Computing at NUDT. His research interest lies in high performance computing and its applications. His email address is xiaow_g@126.com.

Received: January 18, 2012; Accepted: December 4, 2012.