Trace Reduction for Performance Improvement Assessment of Message Passing Parallel Programs

Fumihiko Ino,[1] Yuki Kanbe,[2] Masao Okita,[1] and Kenichi Hagihara[1]
[1] Graduate School of Information Science and Technology, Osaka University, Toyonaka, 560-8531, Japan
[2] Toshiba Corporation, Tokyo, 105-0023, Japan

Abstract— **This paper proposes a trace reduction method for assessing the improvability of the performance of message passing parallel programs. This assessment is based on a what-if prediction approach that forecasts future program performance, for example, the execution time if the target program is modified according to typical tuning techniques. Our method reduces the size of trace files by aggregating records of communications that do not change the predicted execution time. In order to avoid recording such useless information, our method automatically identifies them during program execution by comparing the occurrence time of sends and receives. In case studies, our method reduces the analysis time for what-if predictions as well as the size of trace files roughly into half. We also discuss the usability of our method.**

Key words: message passing paradigm; performance improvement; trace; performance prediction; parallel processing.

I. Introduction

Message passing paradigm [1] is a programming method that is suited to distributed memory parallel systems such as the cluster [2] and the Grid [3]. In this paradigm, processors coordinately perform parallel computation by sending messages each other. For example, the Message Passing Interface (MPI) standard [1] enables us to develop portable, high-performance parallel programs. However, in order to obtain such successful programs, we must repeatedly improve their performance.

To assist developers in this time-consuming procedure, many research projects have developed trace-based performance analysis tools [4]. A trace here is a file generated during a program execution and contains a chain of events recorded with their information such as the time stamp when the event occurred, the process id where it occurred, and so on. An event occurs when a process executes a sequence of program statements. For example, Multi-Processing Environment (MPE) [5] provides predefined events, each corresponding to an MPI routine.

Most of these tools are capable of visualizing performance information recorded in traces. For example, ParaGraph [6], Vampir [7], and Jumpshot [8] can present the timeline view, allowing us to intuitively understand the process activities such as message passing along the time axis. This visualization approach is useful to locate performance bottlenecks in programs. In contrast to this visualization approach, PerWiz [9] is based on a what-if prediction approach that forecasts future program performance. This approach is useful to guide developers to places in a program only where a significant improvement can be achieved. To realize this guidance without any program modifications, PerWiz predicts what performance will be obtained if some events take shorter execution time. For example, by zeroing the execution time of a communication event, we can know a lower bound on the entire execution time achievable by some optimizations of the event.

Thus, trace-based tools provide developers useful capabilities to support their time-consuming procedure. One problem here is that such postmortem analysis tends to suffer from lower scalability, because the size of trace files increases with the number of occurred events. Furthermore, it usually takes longer time to analyze larger trace files.

In this paper, we describe a trace reduction method for assessing the improvability of MPI programs performance. Our method aims at minimizing the size of trace files for performing what-if predictions. To achieve this, our method reduces the trace size by aggregating records of communication events that do not change the PerWiz's prediction results. Such useless records are automatically identified and aggregated during program execution.

The rest of the paper is organized as follows. We begin in Section II by introducing related work. We then present a brief overview of PerWiz in Section III. Section IV defines aggregating conditions making clear what kinds of events can be aggregated in a trace file, and Section V describes our runtime reduction method based on this definition. Section VI shows case studies and Section VII discusses the usability of our method. Finally, Section VIII concludes the paper.

II. Related Work

Prior work on trace reduction employs one of the following three strategies.

- Information aggregation strategy: This strategy performs trace reduction with allowing the loss of information. For example, Nickolayev et al. [10] propose a runtime method that periodically evaluates the similarity of process behaviors in order to generate traces only for some processes that are representatives of the remaining similar processes.

  Pablo [11] focuses on the frequency of events in order to select the appropriate level of details for trace generation. To do this, it requires a threshold value on the frequency of events, so that records only the number of occurred events at the coarsest level.

  Yan et al. [12] reduces the number of recorded events by substituting averaged virtual events for precise actual events. For example, when the same statement placed in a loop generates a sequence of many events, their method records only two values, namely the occurrence number and an averaged execution time for these events. Thus, the precise execution times are not recorded for every event.

- Lossless data compression strategy: This strategy reduces the trace file size without the loss of information. In addition to the information aggregation strategy mentioned above, Yan et al. [12] further reduces the size by replacing repetitive sequences with formulae. For example, they record a formula of $0^3 1^2$ instead of a sequence of 00011. Because this method focuses on the repetition of data sequences, it is effective for the regular part of traces, such as the source/destination process and the communication tag, both recorded for communication events. However, it is not so effective for the irregular part, for example, the time stamp for events.
- Dynamic instrumentation strategy: This runtime strategy dynamically controls trace generation during program execution [13], [14]. Although this strategy is useful for large-scale programs, it cannot support critical path (CP) based analysis [15], which requires the entire trace from the beginning to the end of a program execution.

Thus, there are many trace reduction methods for analyzing the performance and behavior of programs. However, to the best of our knowledge, there exists no reduction method that supports what-if prediction. Except for lossless data compression strategy [12], the prior methods described above cannot be used for this purpose, because they possibly eliminate the key information necessary for what-if prediction (as described later in Section VII-B).

In contrast to the prior work, which focuses on minimizing traces for performance analysis, the objective of our work is to minimize them for predicting future program performance under specific assumptions. Summarizing the above description, the novelty of the paper is that we make clear the trace reducing condition for this purpose and realize this reduction during a program execution.

III. PerWiz: A What-If Prediction Tool

In this section we present an overview of what-if prediction provided by PerWiz [9].

A. Assessing the Improvability of Performance

To assess the improvability of a target program $\mathcal{M}$, PerWiz generates its trace file $\mathcal{L}$ and predicts the execution time of $\mathcal{M}$, assuming that any of events in $\mathcal{L}$ takes shorter execution time. Let T and T' be the measured and the predicted execution time of $\mathcal{M}$, respectively.

In order to predict time T', PerWiz reconstructs trace file $\mathcal{L}$ by using the LogGPS model [16]. This trace reconstruction generates a predicted trace file $\mathcal{L}'$ such that $\mathcal{L}'$ represents a program execution under the assumption of shortened events. Then, PerWiz computes the CP length for $\mathcal{L}'$ so that obtains time T', enabling us to know the possibility of performance improvement before modifying the target program $\mathcal{M}$.

Figure 1 shows a brief overview of asynchronous and synchronous communication behaviors under the LogGPS model. The occurrence and the completion of a communication event correspond to the call and the return of an MPI routine, respectively. In this figure, w denotes waiting time required for synchronization that can occur at both the sender and the receiver side. As illustrated in this figure, the waiting time at the receiver side is defined as the elapsed time from the call of a receive routine to the arrival of the message. On the other hand, at the sender side, the waiting time is defined as the elapsed time from the arrival of a send request REQ to the call of the matching receive routine. In the following discussion, let $a \to b$ be a communication from event a to event b. Let $e_{p,i}$ also denote the i-th event occurred on process p.

Figure 2 shows an example of trace reconstruction. In this example, we assume that the execution time of calculation event $e_{p,i-1}$ is reduced to zero second. Subject to this assumption, the succeeding event $e_{p,i}$ occurs on time $t'_{p,i}$, which is earlier than the original time $t_{p,i}$ in trace $\mathcal{L}$. Due to this earlier occurrence of the event, we must estimate its completion time, and this can be performed as follows.

- If $e_{p,i}$ is a communication event, its execution time usually includes the waiting time, which varies depending on the send/receive timing between communicating processes. Therefore, the execution time must be determined in consideration of this timing, which can be changed during prediction. PerWiz considers this by LogGPS, as presented later in Section III-B.
- Otherwise, namely if $e_{p,i}$ is a calculation event, PerWiz regards the measured time in $\mathcal{L}$ as its execution time in $\mathcal{L}'$. That is, PerWiz assumes that calculation events take the same time, for all what-if predictions.

Applying this to events from the head to the tail of the original trace $\mathcal{L}$ generates the predicted trace $\mathcal{L}'$.

In the following discussion, let $d_{p,i}$ denote *the difference* between the occurrence time of $e_{p,i}$ in $\mathcal{L}$ and that of the same event in $\mathcal{L}'$, where $d_{p,i} = t_{p,i} - t'_{p,i}$.

B. What-If Prediction by the LogGPS Model

In order to predict time T' for the target program $\mathcal{M}$, we must estimate the difference for the last completed event in trace $\mathcal{L}'$. To do this, PerWiz reconstructs trace $\mathcal{L}$ by estimating the difference $d_{p,i}$, for all event $e_{p,i} \in \mathcal{L}$, in the happened-before order [17].

Under the LogGPS model, the difference for a communication event can be determined according to four cases. These cases are classified in terms of the communication mode and the waiting time. Figure 3 shows the differences derived for a communication $e_{p,i} \to e_{q,j}$. In this figure, $w_{q,j}$ represents the waiting time for event $e_{q,j}$, and $x_{q,j}$ represents the elapsed time from the arrival of an asynchronous message to the call of the matching receive routine.

We now explain how the differences are derived for the asynchronous case presented in Figure 3(a). In this case, there is no waiting time at the sender, because this communication mode does not require synchronization before transmitting messages. Therefore, as same as for calculation events, the execution times for send events stay fixed during trace reconstruction. This means that $d_{p,i+1} = d_{p,i}$. On the other hand, the execution time for a receive event usually consists of waiting time. Therefore, the difference $d_{q,j+1}$ depends on $w'_{q,j}$, where $w'_{q,j}$ represents the waiting time for the receive event $e_{q,j}$ after trace reconstruction. If there is no waiting time $w'_{q,j}$ after reconstruction, namely if $d_{p,i} \geq d_{q,j} + w_{q,j}$, we then have $d_{q,j+1} = d_{q,j} + w_{q,j}$, because the execution time for $e_{q,j}$ decreases by the eliminated waiting time $w_{q,j}$. Otherwise, if $w'_{q,j} > 0$ ($d_{p,i} < d_{q,j} + w_{q,j}$), we then have $d_{q,j+1} = d_{p,i}$, because the execution time increases by $d_{q,j} - d_{p,i}$. In summary, $d_{q,j+1} = \min(d_{p,i}, d_{q,j} + w_{q,j})$, as presented in Figure 3(a).

The remaining three cases can be derived in the same manner. Thus, the difference for any event can basically be determined according to the difference for the previous event, its communication mode, and send/receive timing.

IV. Identifying Aggregatable Events

This section describes aggregating conditions for what-if prediction.

A. Conditions for Event Aggregation

Suppose that we focus on an event in $\mathcal{L}$ and regard its execution time as zero second to assess the improvability achievable by some modification of the event. Then, the aggregatable events $e_1, e_2, \ldots, e_n$ in $\mathcal{L}$ are defined such that $e_1, e_2, \ldots, e_n$ do not change the predicted time T' whether they are aggregated or not. To keep the same time T' after event aggregation, we must keep the same difference for every event. This is kept if the following two conditions R1 and R2 are satisfied.

R1: Events $e_1, e_2, \ldots, e_n$ are successive events occurred on the same process.

R2: For any given assumption (what-if prediction), the sum of the execution times for $e_1, e_2, \ldots, e_n$ stays fixed.

For example, given n calculation events that satisfy conditions R1 and R2, we then have the same predicted time T', even if they are aggregated into a calculation event.

On the other hand, when a communication event $e_{p,i}$ belongs to aggregatable events $e_1, e_2, \ldots, e_n$, it has to satisfy the following condition R3 to meet R2.

R3: There exists a communicating event $e_{q,j} \in \mathcal{L}$ $[(e_{p,i} \rightarrow e_{q,j}) \vee (e_{q,j} \rightarrow e_{p,i})]$ such that $d_{p,i} = d_{q,j}$.

That is, any communication event that keeps the same send/receive timing for any assumption satisfies R2.

Figure 4 shows an example of event aggregation for trace reduction. In this example, because communication events $e_{p,i}$ and $e_{q,j}$ satisfy condition R3, the number of recorded events can be reduced by aggregating $e_{p,i}$ ($e_{q,j}$) with its previous and next calculation events $e_{p,i-1}$ and $e_{p,i+1}$ ($e_{q,j-1}$ and $e_{q,j+1}$), respectively. In this case, by recording an aggregated event $e'_{p,i-1}$ with the sum of the execution times for events $e_{p,i-1}$, $e_{p,i}$, and $e_{p,i+1}$, we can reduce the trace file size without changing the predicted time T'. In addition, we regard $e'_{p,i-1}$ as a calculation event. The aggregation procedure mentioned above can be similarly applied to event $e'_{q,j-1}$ at the receiver side.

We now show that the predicted time T' stays fixed for the example presented in Figure 4, whether events are aggregated or not. To do this, we show that the last events have the same differences $d_{p,i+2}$ and $d_{q,j+2}$ after aggregation. In the following, we discuss on $d_{p,i+2}$, namely the sender side. Firstly, when $e_{p,i}$ and $e_{q,j}$ in the original trace satisfy condition R3, we have $d_{p,i+1} = d_{p,i}$, for all cases presented in Figure 3. Secondly, as mentioned before, we assume that traces $\mathcal{L}$ and $\mathcal{L}'$ record the same execution time for calculation events. Therefore, $d_{p,i} = d_{p,i-1}$. Similarly, $d_{p,i+2} = d_{p,i+1}$. These three equations give $d_{p,i+2} = d_{p,i-1}$ for the original trace. On the other hand, we have $d_{p,i+2} = d_{p,i-1}$ for the reduced trace, because $e'_{p,i-1}$ in this trace is a calculation event with difference $d_{p,i-1}$. Thus, $d_{p,i+2}$ stays fixed after event aggregation. In a similar manner, we can show that $d_{q,j+2}$ at the receiver side also stays fixed.

B. Aggregatable Communication Pattern

In Section III-B we have shown that the difference for an event is dependent on that for the previous event, its communication mode, and send/receive timing. This means that the aggregatable event depends on the communication pattern in the target program $\mathcal{M}$. In the following we present two examples of the communication pattern that causes an aggregatable communication $e_{p,i} \rightarrow e_{q,j}$.

C1: A communication $e_{p,i} \rightarrow e_{q,j}$ satisfies condition R3 if its previous communication $e_{p,i-2} \rightarrow e_{q,j-2}$ is a synchronous communication. This can be explained as follows. According to the results in Figures 3(c) and 3(d), the differences for calculation events $e_{p,i-1}$ and $e_{q,j-1}$, which occur immediately after the synchronous communication, satisfy $d_{p,i-1} = d_{q,j-1}$. Furthermore, because $e_{p,i-1}$ and $e_{q,j-1}$ are calculation events, we have $d_{p,i} = d_{q,j}$, satisfying R3.

C2: A communication $e_{p,i} \rightarrow e_{q,j}$ satisfies condition R3 if its previous communication $e_{p,i-2} \rightarrow e_{q,j-2}$ is an asynchronous communication with having waiting time $w_{q,j-2}$ at the receiver such that $d_{p,i-2} \leq d_{q,j-2} + w_{q,j-2}$. This can be explained as follows. According to Figure 3(a), the difference for calculation event $e_{q,j-1}$ at the receiver side is given by $d_{q,j-1} = \min(d_{p,i-2}, d_{q,j-2} + w_{q,j-2}) = d_{p,i-2}$. Furthermore, at the sender side, we have $d_{p,i-1} = d_{p,i-2}$ for calculation event $e_{p,i-1}$. These two equations show that $d_{p,i-1} = d_{q,j-1}$, and thereby $d_{p,i} = d_{q,j}$, as we presented for pattern C1.

Figure 5 shows an example of pattern C2. In this example, processes p and q communicate a round trip message. Note here that there is waiting time when receiving the message. The reason why this example satisfies pattern C2 can be explained as follows. To simplify the notation, we write $d_{p,i-4} = u$ and $d_{q,j-4} = v$ in the following explanation.

If $e_{q,j-4} \rightarrow e_{p,i-4}$ is a synchronous communication, it satisfies C2 because $d_{p,i-2} = d_{q,j-2}$ (according to C1). Otherwise, namely if it is an asynchronous communication, we have $d_{q,j-3} = v$ and $d_{p,i-3} = \min(v, u + w_{p,i-4})$ because $e_{q,j-4} \rightarrow e_{p,i-4}$ can be classified as the case in Figure 3(a). These two equations give $d_{p,i-3} \leq d_{q,j-3}$. Furthermore, because $e_{p,i-3}$ and $e_{q,j-3}$ are calculation events, we have $d_{p,i-2} \leq d_{q,j-2}$. Thus, $e_{q,j-4} \to e_{p,i-4}$ satisfies C2.

V. Aggregating Events

In this section we describe a trace generation method that locates aggregatable communication events and aggregates them during program execution. To make this description easier, we begin with a postmortem method, which performs event aggregation after program execution, and then present our runtime method.

A. Postmortem Aggregation Method

We have presented conditions for event aggregation in Section IV-A. Then, the remaining issues that must be solved by aggregation methods are as follows.

The first issue is how to avoid aggregating events that can be shortened during what-if prediction. These shortened events do not satisfy condition R2, because their execution time will be changed by PerWiz for what-if prediction. Therefore, aggregation methods must locate such shortened events in order to avoid aggregating them during trace generation.

To achieve this in advance of what-if prediction, our method locates such events according to whether they will yield better performance improvement or not. Because the improvement achievable by an event is limited by the execution time of the event, we locate them using a threshold value, for example, a value associated with the execution time or the message length.

The second issue is that some values of differences can be computed only when performing what-if prediction, while aggregatable events are located by comparing the differences between two communicating events, as shown in condition R3. These values depend on the assumptions specified by PerWiz. Therefore, aggregation methods must provide flexible representations for the differences, which cannot have precise values but need to be comparable during trace generation.

To do this, our method represents a difference as an equation with variables. Note here that this variable representation is applied only to aggregatable events, according to the threshold value mentioned before. Thus, when generating reduced trace files, our method represents a difference as an equation with variables rather than a value.

Figure 6 shows a postmortem aggregation method that generates the reduced trace $\mathcal{L}'$ from the original trace $\mathcal{L}$. To simplify the explanation, we assume that $\mathcal{L}$ consists of only communication events. Furthermore, we deal only with the case in Figure 3(a): asynchronous communication with waiting time at the receiver. The remaining three cases can be similarly processed by adding branch statements based on the equations in Figure 3, as we added them to lines 16 and 17 for the first case (see Figure 6).

By using the equation in Figure 3(a), our postmortem method computes the differences (lines 16 and 17) from earlier happened events (line 4). During this computation, the method omits recording occurred events (line 6) if they satisfy condition R3 (line 19). Otherwise, it regards them as unaggregatable, so that records them in the reduced trace $\mathcal{L}'$ (line 8). In addition, for events that can be shortened for what-if predictions, the method represents their differences as variables so that avoids aggregating them (lines 13 and 14). Repeating the above procedure for all events in the original trace $\mathcal{L}$ generates the reduced trace $\mathcal{L}'$.

B. Runtime Aggregation Method

As we mentioned before, we need to compare the differences between two communicating events in order to check whether they are aggregatable or not. Because the differences are dependent on both the sender and the receiver side, this requires additional communication to aggregate events during program execution.

For this reason, our runtime method adds some statements to MPI implementations in order to compare the differences whenever MPI routines are called in the target program. Figure 7 shows these statements added to the implementation of pure MPI routines.

In order to avoid increasing the occurrence of communication, our method realizes this comparison at the receiver side. In our method, the sender process p concatenates its current difference $d_{p,i}$ to the outgoing message (lines 3 and 4), so that the receiver q compares it with its own difference $d_{q,j}$ (line 11). In this receiver-compute rule, the sender p also has to send the information on event $e_{p,i}$ to the receiver q, because q determines aggregatable events. Therefore, our method also concatenates this information to the message, so that the receiver takes the responsibility for recording both events $e_{p,i}$ and $e_{q,j}$ (line 12).

Note here that variables representing a difference are required for every communication event that can be shortened for what-if prediction. Therefore, the number of variables increases when such communication occurs. This increase results in higher overhead, because it also increases the amount of information added to the original message. To keep this overhead lower, we restrict the number of additional variables. That is, our method communicates at most m variables, so that determines aggregatable events using the last m events.

VI. Case studies

We now show case studies to evaluate our method from the following four viewpoints:

- Reduction amount of trace size;
- Analysis time for what-if prediction;
- Overhead for trace generation;
- Scalability of trace files.

In these studies, we used the MPICH-SCore [18] and MPE [5] libraries, a fast MPI implementation and its trace generation library, respectively. MPE allows us to visualize a trace by using Jumpshot [8], a visualization tool widely used and distributed with MPE. We also used a cluster of 64 PCs interconnected by a Myrinet switch [19], yielding a full-duplex bandwidth of 2 Gb/s. Each node in the cluster has a Pentium III 1-GHz processor and 2 GB main memory.

A. Reduction Amount of Trace Size

We applied our method to an image registration application [20], which finds point correspondences between two different three-dimensional (3-D) images. Figure 8 presents a pseudo code for this application. In this application, every process has a portion of the images and operates control points overlaid on the image space. To move each of M control points to an appropriate place, processes perform an iterative optimization in a concurrent manner. During this optimization, processes are classified into groups such that processes in a group $\mathcal{P}_\phi$ have neighborhood pixels of a control point ϕ and participate in the computation and communication required for ϕ. Repeating the computation and communication phases K times aligns the images.

Table I shows the number of recorded events and its breakdowns on 64 processes. In this table, N_1 and N_2 denote the number for the unreduced (MPE) trace and that for our reduced trace, respectively. The breakdowns are shown for each i of the K iterations. We aggregated events by using the last three events ($m = 3$).

By comparing the total of N_1 to that of N_2, we can see that our method reduces the number of recorded events roughly into half. This reduction contributes to the reduction of trace size from approximately 9 MB to 5 MB.

Table I also indicates that the reduction rate σ_N of trace size reaches 50% when $i > 3$, whereas it results in a lower value when $i \leq 3$. This lower value is due to the communication pattern of the target program where many processes transmit messages with frequently changing destinations. In this situation, our method tends to locate less aggregatable events, so that the reduction rate results in a lower value.

Actually, this application has such a communication pattern. Because communication occurs between processes that belong to the same group, once all processes $p \in \mathcal{P}_\phi$ have the same difference, all the succeeding events occurred on all p can be aggregated according to condition R3. This indicates that the more processes belong to a group, the less aggregatable events will be. For example, our method achieves better reductions for the fourth and sixth iterations ($i = 4$ and 6), where groups consist of at most 4 and 2 processes, respectively. In contrast, groups at the first iteration ($i = 1$) consist at most 12 processes.

In addition to group size, frequent change of groups also blocks our event aggregation. Suppose that there is a group containing processes with the same difference. Once another group communicates to this coordinated group, the difference probably changes at the receiver. Thus, such incoming messages prevent processes from making the difference to be the same value. Therefore, when this happens frequently, our method cannot provide effective aggregation.

In summary, our reduction method is effective for programs where processes repeatedly communicate each other in a small group of processes.

B. Analysis Time for What-If Prediction

In order to make clear how reduced traces contribute to faster what-if prediction, we measured the analysis time required by PerWiz. Table I shows times T_1 and T_2, the analysis time for the unreduced trace and that for the reduced trace, respectively. The measurement was carried out on a node in the cluster.

In this table, we can see that the proposed method reduces the analysis time from 572.1 s to 275.5 s. Thus, reducing trace size contributes to faster what-if predictions. We also can see that $\sigma_N \leq \sigma_T$ for all breakdowns i. Therefore, the reduction effect of trace size exceeds that of analysis time. This is due to the time complexity of the what-if prediction because it takes $O(N^2)$ time in the worst case, where N denotes the number of events. Thus, we obtain $\sigma_N \leq \sigma_T$.

Although the reduced trace has less information, we obtained the same predicted results as we did from the unreduced trace. That is, PerWiz guides us to the same events with the same priority, allowing us to efficiently improve the program performance. Thus, our method reduces both trace size and analysis time without changing what-if prediction results, so that it provides developers a quick guidance.

C. Overhead for Trace Generation

There are three types of overheads in our method.

O1: The trace generation overhead (line 12 of Figure 7).

O2: The aggregation decision overhead (line 11 of Figure 7).

O3: The additional communication overhead (lines 3, 4, 9 and 10 of Figure 7).

Overhead O1 is involved in any trace generation method whether it aggregates events or not. For example, MPE takes 5μs to record an event with 20 bytes data including an event type ID, a time stamp, and source/destination process. Since our method reduces the number of recorded events, it reduces the total of overhead O1.

Note that MPE basically writes event information into a trace file after program execution. During program execution, every process stores the information to a buffer in own main memory.

In contrast to overhead O1, the remaining overheads O2 and O3 are unique to our reduction-based method. O2 requires $O(m)$ time in our method. When we use the last three events for aggregation decision ($m = 3$), O2 is shorter than 1 μs on our cluster. Therefore, O2 can be ignored as compared with O1.

On the other hand, O3 requires $O(m + l)$ time, where l denotes the length of the original message. This is due to the message concatenation mentioned in Section V-B (lines 3 and 4 in Figure 7). Therefore, O3 is not short enough to ignore its influence. This is especially true when we use more events m for aggregation decision on higher speed network, because message concatenation could take longer time than message transfer.

Actually, the entire overhead in this application was dominated by this message concatenation. When we execute this application with generating the reduced trace, the execution time was increased from 41 s to 45 s, while the unreduced trace increases it to 42 s. Thus, it took 3 s to aggregate events, which is mainly spent by memory copy operation for concatenation. Although this concatenation increased the total amount of communication by 2.8 MB, this additional amount was small enough compared to the entire amount of 6.9 GB.

D. Scalability of Trace files

The three overheads mentioned above prevent us from capturing accurate program behaviors because they can perturb the program behavior and the execution time. Therefore, to evaluate how our method tolerates this performance perturbation, we applied our method to two programs that generate many events. Evaluation is carried out in terms of event frequency and trace size.

For the evaluation of event frequency, we used a range of motion (ROM) estimation application [21], which computes the safe ROM for an artificial joint. This program rapidly detects rotational motions that avoid any impingements. To accelerate this estimation, it employs a master/worker paradigm, where master processes manage detection results for rotational motions while worker processes check each of assigned motions whether it causes impingements.

Figure 9 plots the execution time of this application with different task grain sizes γ, namely the number of rotational motions in a task. In this figure, T_3 and T_4 denote the execution time with reduced and unreduced trace generation, respectively. T_5 denotes the execution time without trace generation. When a task consists of less than 30 motions ($\gamma < 30$), the overhead $T_4 - T_5$ significantly increases as grain size γ decreases. At the minimum grain size, it takes 86 s and 88 s to execute the program with unreduced and reduced trace generation, respectively, whereas the original execute time is 32 s. This indicates that overhead O1 dominates the increased time, because master processes suffer from frequent occurrence of events, as tasks (γ) become smaller. As a result, O1 becomes larger relative to the execution time measured for the instrumented code, generating an inaccurate trace.

Note here that the rate of increased time $100(T_4/T_5 - 1)$ is below 11% when $\gamma \geq 30$. In these grain sizes, less than 30 thousands events are recorded every second. This means that, in order to achieve an accurate error of less than 10%, each of the instrumented code need to be executed in at least 33 μs.

As we mentioned in Section III-A, PerWiz estimates the execution time of communication events by using LogGPS. This estimation is accurate enough within 4% error [9]. Therefore, if the event order is kept as same in the original execution, overheads O1, O2, and O3 might become a trivial issue. However, these overheads might not be trivial if the execution time is dominated by calculation time, because O1 prevents us from generating accurate traces.

For the evaluation of trace size, we used a ping-pong program that iteratively exchanges a fixed length (128 KB) message between two processes until trace size exceeds the physical memory capacity of a node (2 GB).

Figure 10 shows how the generated trace grows during program execution. It plots the elapsed execution time along the trace size axis. In this figure, T_6 and T_7 denote the elapsed time measured with unreduced trace generation and that without trace generation, respectively. Although T_7 cannot be measured directly, we estimated it from the number of roundtrip and the message length. Note here that we did not aggregate events during this measurement.

When trace size is less than 1920 MB, we are allowed to measure the elapsed time T_7 with an error of approximately 15%. However, when the trace grows beyond the physical memory capacity, the error significantly increases, so that it takes 183 s to execute the program with trace generation, which is three times longer time than the original time, 50 s. This significant increase is due to frequent memory swapping. That is, as processes store more events to the buffer, it requires more physical memory resources. Once these resources are exhausted, application or trace data will frequently written to a swap file, increasing the trace generation overhead.

In summary, our method and PerWiz can accurately assess the improvability of programs such that:

- event frequency is low enough;
- no memory swapping during program execution;
- longer communication time with shorter calculation time.

The first two requirements are due to trace generation and the last one is due to our method.

VII. Discussion

In this section we discuss the usability of our method from a viewpoint of application developers who improve the program performance. To do this, we present how reduced traces contribute to the improvement of program performance. We also compare our method with prior methods.

A. Performance Improvement Procedure

PerWiz's assistances can reduce developer's efforts in the following two procedures.

W1: Improvability assessment. As we mentioned in Section III, PerWiz predicts the execution time of future program, assuming that any of events takes shorter execution time. This what-if prediction allows developers to avoid program modifications with less improvement. For example, when we improved the performance of the registration application by means of waiting time elimination, PerWiz predicted that eliminating the longest waiting time will not always be the best modification [9]. In fact, this modification increased the waiting time of the succeeding events, resulting in little improvement. In contrast, by eliminating the waiting time of events that PerWiz guided us, we reduced the execution time by 8.6%, as PerWiz predicted before program modification.

W2: Performance bottleneck search. One of the advantage of PerWiz is that it can compute a domino path (DP) for a trace. Here the DP is a path containing events that can yield a significant improvement of the program performance. PerWiz locates such events by repeating what-if prediction and presents in a graphical view, as shown in Figure 11(b). Computing the DP is useful to realize performance improvement with less effort, because it consists of events that can trigger a domino effect that shortens a chain of many events. Due to this guidance, developers are allowed to achieve more improvement with less effort. In the registration application case, we achieved 42.4% improvement by eliminating the waiting time of seven events in the DP. On the other hand, eliminating the longest seven waiting time results in 12.9% improvement. Thus, though we applied the same improvement method to the application, PerWiz guidance realizes three times better improvement.

By automating the above procedures, developers are allowed to know the improvability of programs in advance of their modification, so that can effectively improve the program performance.

Figure 11 presents a timeline view analyzed by PerWiz. In this timeline view, we have the process axis in vertical and the time axis in horizontal. A colored rectangle corresponds to a communication event. Figure 11(a) represents the process behavior of a program execution and Figure 11(b) shows the DP computed by PerWiz for this trace.

Note here that this visualization is not a visual assistance that considers the results of trace reduction. Such a assistance is left for future work. For example, because aggregated events will not be influenced by any program modification, we would intuitively understand the improvability of programs if we could see aggregated events in a timeline view.

B. Qualitative Comparison

To make clear the technical novelty of our method, we discuss whether the prior methods introduced in Section II can deal with what-if prediction.

As we presented in Section III-A, the execution time of the target program depends on the CP length. Therefore, to automate procedure W1 by PerWiz, it requires the following information (A): the CP length under arbitrary assumptions. Our method satisfies conditions R1, R2, and R3 so that this information (A) is always left in reduced traces, enabling what-if prediction. In the following, we mention the prior methods. Table II summarizes the comparison.

First, as we mentioned before, lossless data compression strategy [12] keeps all of the information in traces. Therefore, for lossless compressed traces, PerWiz can compute (A), so that allowed to assess the improvability of programs. Note here that this strategy does not reduce the number of recorded events. Therefore, combining it with our method will achieve further trace reduction. However, this strategy requires additional overheads for data compression and decompression. The data decompression overhead is the disadvantage of this method because it makes PerWiz's analysis time longer.

Next, information aggregation strategies [10]–[12] probably fail to keep information (A) in traces. Therefore, in many cases, PerWiz is unable to assess the improvability of programs by using their traces. The first method [10], which focuses on the similarly between process behaviors, stops recording events occurred on non-representative processes. Therefore, if the CP consists of such unrecorded events, PerWiz fails to compute the CP. The second method [12], which focuses on event frequency, omits recording the event type and the event order at the coarsest level. Therefore, PerWiz also fails to compute the CP. In contrast, the last method [11], which focuses on the program structure, allows PerWiz to compute the CP because it records the event type and the event order. However, because this method records averaged time rather than original time for each event, the CP length cannot be accurately computed if there is a large gap between them. Note here that PerWiz estimates communication time using LogGPS. Therefore, this accuracy issue is related only to calculation events.

Finally, dynamic instrumentation strategies [13], [14] stops trace generation during program execution. Therefore, generated traces lack the information during this uninstrumented period. This makes it impossible to compute the CP, so that PerWiz is unable to assess the improvability of programs. In contrast, PerWiz can assess the local improvability of procedures if generated traces include all events occurred in the target procedures. However, because the CP is unknown, it is unable to assess how this local improvement leads to the improvement of the entire program performance.

Summarizing the above comparison, the technical novelty of our method is that it reduces the number of recorded events with keeping the information (A) required for performing what-if prediction (see Table II).

VIII. Conclusions

We have presented a trace reduction method for assessing the improvability of MPI programs performance. Our method reduces the size of trace files by aggregating records of communications that do not change the execution time predicted by PerWiz. This aggregation is automatically performed during program execution. In case studies, our method reduces the analysis time for what-if predictions as well as the size of trace files roughly into half. Therefore, we believe that our method is useful for developers to assess the improvability of MPI programs performance without program modification.

Future work includes reducing the copy overheads for faster message concatenation and developing a trace generation method for large-scale programs. We are also planning on coupling our method with lossless data compression to present a qualitative evaluation.

IX. Acknowledgments

This work was partly supported by JSPS Grant-in-Aid for Young Scientists (B)(15700030) and NEC. We would like to thank the anonymous reviewers for their valuable comments.

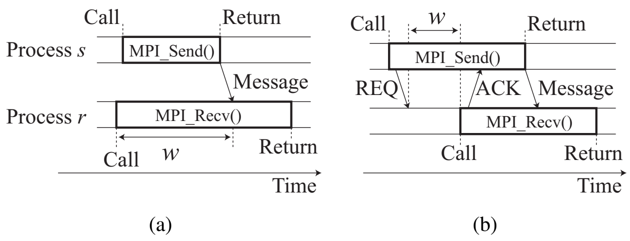


Fig. 1. Overview of (a) asynchronous and (b) synchronous communication behaviors under the LogGPS model. See [16] for more details.

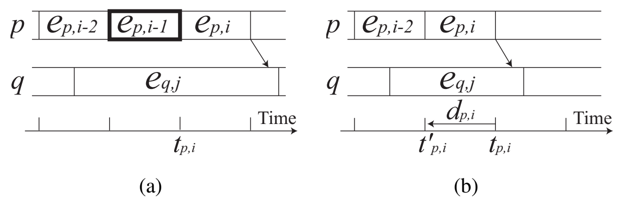


Fig. 2. Trace reconstruction for what-if prediction. Trace file (a) before and (b) after reconstruction. In this example, event $e_{p,i-1}$ in the original trace is assumed to take zero second in the reconstructed trace, so that the timestamps of the succeeding events are estimated by the LogGPS model.

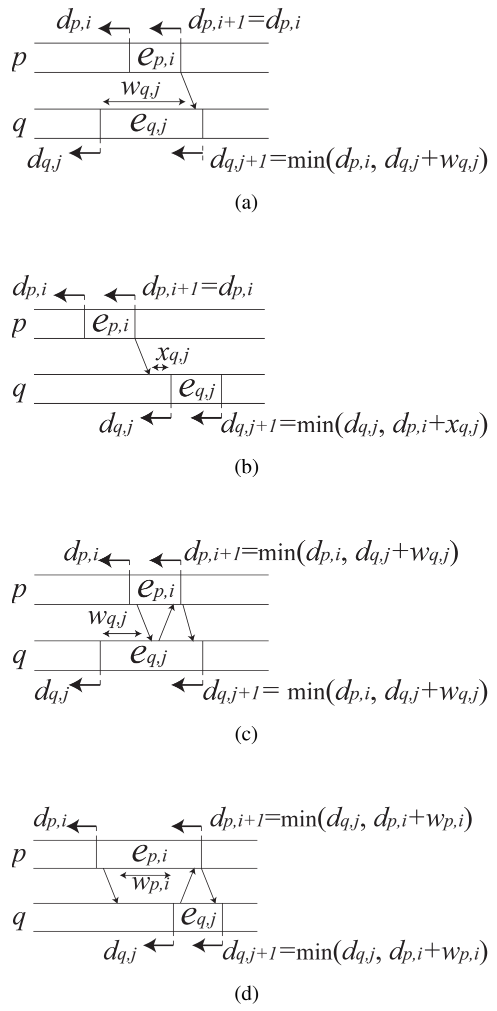

$d_{p,i}$
$d_{p,i+1}=d_{p,i}$
$e_{p,i}$
$w_{q,j}$
$e_{q,j}$
$d_{q,j}$
$d_{q,j+1}=\min(d_{p,i}, d_{q,j}+w_{q,j})$
$x_{q,j}$
$d_{q,j+1}=\min(d_{q,j}, d_{p,i}+x_{q,j})$
$d_{p,i+1}=\min(d_{p,i}, d_{q,j}+w_{q,j})$
$d_{p,i+1}=\min(d_{q,j}, d_{p,i}+w_{p,i})$
$w_{p,i}$
$d_{q,j+1}=\min(d_{q,j}, d_{p,i}+w_{p,i})$
(a)
(b)
(c)
(d)

Fig. 3. Definition of the difference under the LogGPS model. (a) Asynchronous communication mode with waiting time and (b) without waiting time. (c) Synchronous communication mode with waiting time at the receiver and (d) at the sender.

TABLE I

NUMBER OF RECORDED EVENTS ON 64 PROCESSES AND ANALYSIS TIME FOR PERFORMING WHAT-IF PREDICTION USING PERWIZ.

Breakdowns	# of events		Rate	Analysis time (s)		Rate
	MPE	Proposed	(%)	MPE	Proposed	(%)
	N_1	N_2	σ_N*	T_1	T_2	σ_T*
$i=1$	50,096	37,308	25.5	338.8	204.3	39.7
$i=2$	35,120	22,932	34.7	158.7	66.1	58.3
$i=3$	13,904	8,108	41.7	21.0	4.8	77.1
$i=4$	21,152	1,908	91.0	52.7	0.3	99.4
$i=5$	688	16	97.7	0.0	0.0	100.0
$i=6$	2,924	72	97.5	0.8	0.0	100.0
Others	58,876	28,084	52.3	—	—	—
Total	182,760	98,428	46.1	572.1	275.5	51.8

$* : \sigma_N = 100(1 - N_2/N_1), \sigma_T = 100(1 - T_2/T_1)$

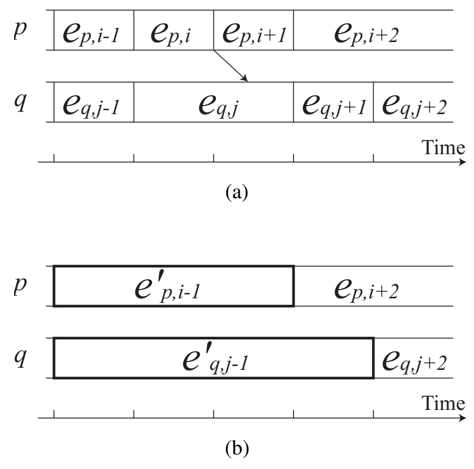
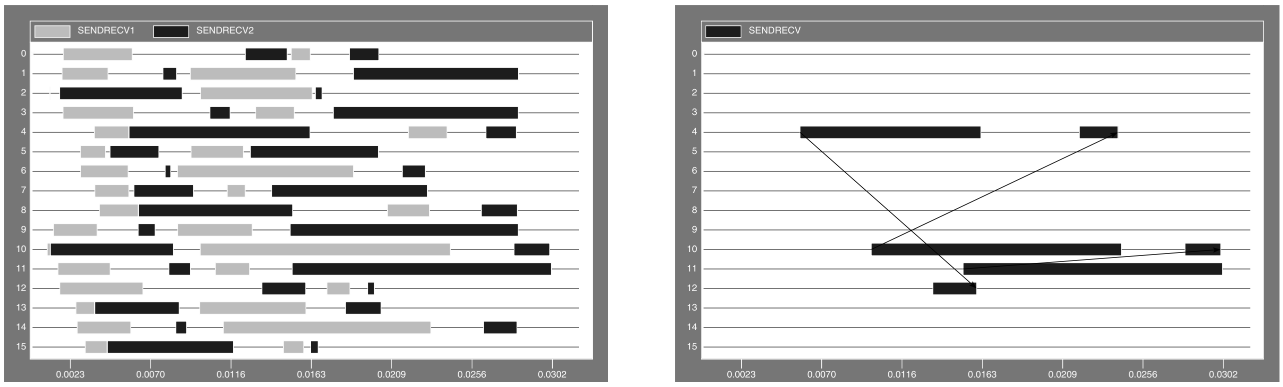


Fig. 4. Event aggregation for trace reduction. Three events (a) in the original trace are aggregated into an event (b) in the reduced trace.



(a) (b)

Fig. 11. Predicted results in timeline view visualized by Jumpshot [8]. (a) The original trace and (b) its domino path.

TABLE II

COMPARISON OF TRACE REDUCTION METHODS IN TERMS OF CAPABILITY.

Reduction method		Estimation of critical path length	Reduction of recorded events	Redution of trace size
Strategy	Method			
Information aggregation	Proposed	Yes	Yes	Yes
	[10]–[12]	No	Yes	Yes
Lossless data compression	[12]	Yes	No	Yes
Dynamic instrumentation	[13], [14]	No	Yes	Yes

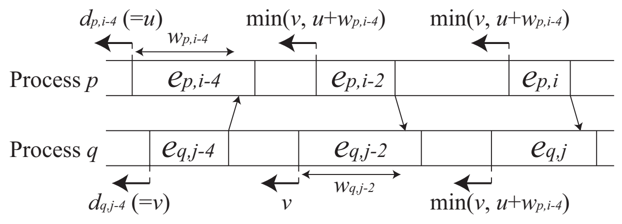


Fig. 5. Example of communication pattern that causes aggregatable communication.

```
Input: $\mathcal{L}$, a trace file containing communication events
Output: $\mathcal{L}'$, an aggregated trace file
1: Algorithm Aggregate($\mathcal{L}$) {
2:    $\mathcal{L}' := \emptyset$; // $\emptyset$: empty set
3:    while ($\mathcal{L} \neq \emptyset$) {
4:       Select $e_{p,i}, e_{q,j} \in \mathcal{L}$ such that $e_{p,i} \rightarrow e_{q,j}$ is the
           earliest happened communication;
5:       if (IsAggregatable($e_{p,i}$, $e_{q,j}$) = $true$) {
6:          Delete $e_{p,i}$ and $e_{q,j}$ from $\mathcal{L}$; // For aggregation
7:       } else {
8:          Add $e_{p,i}$ and $e_{q,j}$ to $\mathcal{L}$' and delete them from $\mathcal{L}$;
9:       }
10:   }
11: }
12: function IsAggregatable($e_{p,i}$, $e_{q,j}$) {
13:   if ($e_{p,i}$ or $e_{q,j}$ can be an event with assumptions) {
14:      Leave $d_{p,i+1}$ and $d_{q,j+1}$ as variables;
15:   } else { // For the case in Fig.3(a)
16:      $d_{p,i+1} := d_{p,i}$;
17:      $d_{q,j+1} :=$ min($d_{p,i}$, $d_{q,j} + w_{q,j}$);
18:   }
19:   if ($d_{p,i} = d_{q,j}$) { return $true$; } // Condition R3
20:   else { return $false$; }
21: }
```

Fig. 6. Postmortem aggregation method for trace reduction.

```
1: int MPI_Send($message$) {
2:    $t_{p,i}$ := PMPI_Wtime(); // occurrence time of $e_{p,i}$
3:    Copy $message$ to static_buf with $d_{p,i}$ and $t_{p,i}$;
4:    PMPI_Send(static_buf);
5:    $d_{p,i+1} := d_{p,i}$;
6: }
7: int MPI_Recv($message$) {
8:    $t_{q,j}$ := PMPI_Wtime(); // occurrence time of $e_{q,j}$
9:    PMPI_Recv(static_buf);
10:   Extract $message$, $d_{p,i}$ and $t_{p,i}$ from static_buf;
11:   if (IsAggregatable($e_{p,i}$, $e_{q,j}$) = $false$) {
12:      Record $e_{p,i}$ and $e_{q,j}$ with $t_{p,i}$ and $t_{q,j}$;
13:   }
14: }
```

Fig. 7. Runtime aggregation method for trace reduction.

```
1: for ($i$=1; $i \leq K$; $i$++) {
2:    for ($\phi$=1; $\phi \leq M$; $\phi$++) {
3:       if (MPI_Comm_rank() $\in$ process group $\mathcal{P}_\phi$) {
4:          Local computation;
5:          Communication in $\mathcal{P}_\phi$ using MPI_Send(),
              MPI_Recv(), and MPI_Sendrecv();
6:       }
7:    }
8: }
```

Fig. 8. Pseudo program for image registration.

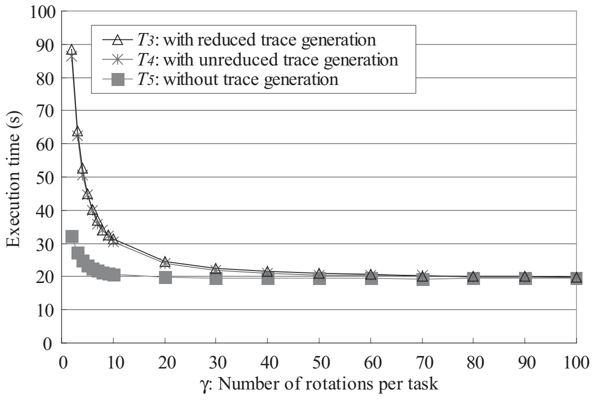


Fig. 9. Relationship between task grain size and measured execution time for range of motion estimation application.

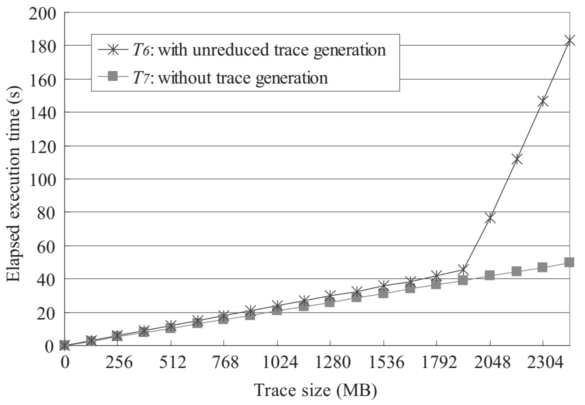


Fig. 10. Relationship between trace size and elapsed execution time for ping-pong program.

REFERENCES

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard," *Int'l J. Supercomputer Applications and High Performance Computing*, vol. 8, no. 3/4, pp. 159–416, 1994.

[2] R. Buyya, Ed., *High Performance Cluster Computing*. Upper Saddle River, NJ: Prentice Hall PTR, June 1999.

[3] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint of a New Computing Infrastructure*. San Mateo, CA: Morgan Kaufmann, July 1998.

[4] S. Browne, J. Dongarra, and K. London, "Review of performance analysis tools for MPI parallel programs," Dec. 1997, http://www.cs.utk.edu/~browne/perftools-review/.

[5] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp, "From trace generation to visualization: A performance framework for distributed parallel systems," in *Proc. High Performance Networking and Computing Conf. (SC2000)*, Nov. 2000.

[6] M. T. Heath and J. A. Etheridge, "Visualizing the performance of parallel programs," *IEEE Software*, vol. 8, no. 5, pp. 29–39, Sept. 1991.

[7] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *J. Supercomputing*, vol. 12, no. 1, pp. 69–80, Jan. 1996. [Online]. Available: http://www.pallas.de/pages/vampir.htm

[8] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *Int'l J. High Performance Computing Applications*, vol. 13, no. 2, pp. 277–288, June 1999. [Online]. Available: http://www-unix.mcs.anl.gov/perfvis/software/viewers/

[9] Y. Kanbe, M. Okita, F. Ino, and K. Hagihara, "A performance prediction tool for evaluating potential improvement of message passing programs," *IPSJ Trans. Advanced Computing Systems*, vol. 44, no. SIG11, pp. 101–110, Aug. 2003, (In Japanese).

[10] O. Y. Nickolayev, P. C. Roth, and D. A. Reed, "Real-time statistical clustering for event trace reduction," *Int'l J. Supercomputer Applications and High Performance Computing*, vol. 11, no. 2, pp. 144–159, 1997.

[11] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, "Scalable performance analysis: The Pablo performance analysis environment," in *Proc. 3rd Scalable Parallel Libraries Conf. (SPLC'96)*, Oct. 1996, pp. 104–113.

[12] J. C. Yan and M. A. Schmidt, "Constructing space-time views from fixed size trace files – getting the best of both worlds," in *Proc. Int'l Conf. Parallel Computing (ParCo'97)*, Sept. 1997.

[13] J. Yan, S. Sarukkai, and P. Mehra, "Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit," *Software: Practice and Experience*, vol. 25, no. 4, pp. 429–461, Apr. 1995.

[14] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn parallel performance measurement tool," *IEEE Computer*, vol. 28, no. 11, pp. 37–46, Nov. 1995.

[15] J. K. Hollingsworth, "Critical path profiling of message passing and shared-memory programs," *IEEE Transactions Parallel and Distributed Systems*, vol. 9, no. 10, pp. 1029–1040, Oct. 1998.

[16] F. Ino, N. Fujimoto, and K. Hagihara, "LogGPS: Modeling message-passing protocols in high-level communication libraries," *IPSJ Trans. High Performance Computing Systems*, vol. 42, no. SIG9, pp. 145–157, Aug. 2001, (In Japanese).

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[18] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa, "The design and implementation of zero copy MPI using commodity hardware with a high performance network," in *Proc. 12th ACM Int'l Conf. Supercomputing (ICS'98)*, July 1998, pp. 243–250. [Online]. Available: http://www.pccluster.org/

[19] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995. [Online]. Available: http://www.myri.com/

[20] J. A. Schnabel, D. Rueckert, M. Quist, J. M. Blackall, A. D. Castellano-Smith, T. Hartkens, G. P. Penney, W. A. Hall, H. Liu, C. L. Truwit, F. A. Gerritsen, D. L. G. Hill, and D. J. Hawkes, "A generic framework for non-rigid registration based on non-uniform multi-level free-form deformations," in *Proc. 4th Int'l Conf. Medical Image Computing and Computer-Assisted Intervention (MICCAI'01)*, Oct. 2001, pp. 573–581.

[21] Y. Kawasaki, F. Ino, Y. Sato, N. Sugano, H. Yoshikawa, S. Tamura, and K. Hagihara, "Real-time estimation of hip range of motion for total hip replacement surgery," in *Proc. 7th Int'l Conf. Medical Image Computing and Computer-Assisted Intervention (MICCAI'04), Part II*, Sept. 2004, pp. 629–636.

PLACE
PHOTO
HERE

Fumihiko Ino (member) received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently an Assistant Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation. He received the Best Paper Award at the 2003 International Conference on High Performance Computing (HiPC'03) and the Best Paper Award at the 2004 Symposium on Advanced Computing Systems and Infrastructures (SACSIS'04).

PLACE
PHOTO
HERE

Yuki Kanbe received the B.E. and M.E. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 2002 and 2004, respectively. He is currently a Software Engineer at Toshiba Corporation in Tokyo, Japan. His current research interests include performance prediction for high performance computing.

PLACE
PHOTO
HERE

Masao Okita received the B.E. and M.E. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 2001 and 2003, respectively. He is currently working toward the Ph.D. degree at the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. His current research interests include software development tools for high performance computing.

PLACE
PHOTO
HERE

Kenichi Hagihara (member) received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1974, 1976, and 1979, respectively. From 1994 to 2002, he was a Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. Since 2002, he has been a Professor in the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. From 1992 to 1993, he was a Visiting Researcher at the University of Maryland. His research interests include the fundamentals and practical application of parallel processing. He received the Best Paper Award at the 2003 International Conference on High Performance Computing (HiPC'03) and the Best Paper Award at the 2004 Symposium on Advanced Computing Systems and Infrastructures (SACSIS'04).