PerWiz: A What-If Prediction Tool for Tuning Message Passing Programs*

Fumihiko Ino, Yuki Kanbe, Masao Okita, and Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
{ino,y-kanbe,m-okita,hagihara}@ist.osaka-u.ac.jp

Abstract. This paper presents PerWiz, a performance prediction tool for improving the performance of message passing programs. PerWiz focuses on locating where a significant improvement can be achieved. To locate this, PerWiz performs a post-mortem analysis based on a realistic parallel computational model, LogGPS, so that predicts what performance will be achieved if the programs are modified according to typical tuning techniques, such as load balancing for a better workload distribution and message scheduling for a shorter waiting time. We also show two case studies where PerWiz played an important role in improving the performance of regular applications. Our results indicate that PerWiz is useful for application developers to assess the potential reduction in execution time that will be derived from program modification.

1 Introduction

Message passing paradigm [1] is a widely employed programming paradigm for distributed memory architectures such as clusters and Grids. This paradigm enables writing high performance parallel applications on these architectures. However, it assumes implicit parallelism coded by application developers. Therefore, developers have to take responsibility for detecting the bottleneck code of sequential applications and for determining which code should be parallelized, according to the performance analysis of the applications. Thus, performance tuning is an important process for developing high performance parallel applications.

One issue for tuning parallel programs is an enormous amount of performance data, which makes this process difficult and time-consuming task. Earlier research addresses this issue by visualizing the performance data collected during program execution [2–5]. Although this visual approach enables intuitive understanding of program behavior, it has a scalability problem. It is not easy for developers to detect performance bottlenecks from the complicated visualizations rendered for large-scale applications.

Therefore, some research projects address the issue by automating program instrumentation [6], performance problem search [7–10], and performance prediction [11]. These automatic approaches are attractive for developers who want to locate performance bottlenecks and adapt applications to other computing environments. However,

* This work was partly supported by JSPS Grant-in-Aid for Young Scientists (B)(15700030), for Scientific Research (C)(2)(14580374), and Network Development Laboratories, NEC.

these projects raise another question that must be answered: how much improvement in execution time will be derived with what kind of tuning techniques?

Hollingsworth [12] gives a preliminary answer to this question by developing a run-time algorithm to compute a variant of critical path (CP), called CP zeroing. CP zeroing provides an upper bound on the reduction in the length of CP possible by tuning a specific procedure. Because CP zeroing directs developers to procedures expected to provide a significant improvement, it is useful to prevent spending time tuning procedures that will not give an improvement. A similar analysis [13] is also useful to balance the workload in a coarse-grain manner. This analysis predicts the potential improvement in execution time when changing the assignment of processes to processors.

Thus, many tools focus on helping performance tuning. However, to the best of our knowledge, there is no tool that gives an answer to the further question: what performance will be achieved if various tuning techniques are applied to the programs? This *what-if prediction* support is important for developers because it enables them to prevent wasted effort with no improvement. Note here that CP zeroing predicts the improvement if a procedure is removed, so that considers no tuning technique during prediction. Thus, addressing what-if prediction with various tuning techniques is lacking, for instance, what-if predictions for fine-grain load balancing and message scheduling, aiming to obtain a better workload distribution and a shorter waiting time, respectively.

In this paper, we present Performance Wizard (PerWiz), a what-if prediction tool for tuning Message Passing Interface (MPI) programs [1]. PerWiz reveals hidden bottleneck messages in a program by predicting the program's execution time if zeroing the waiting time of a message. Furthermore, by specifying the program's *parallel region*, where developers intend to parallelize, PerWiz presents a lower bound on the execution time if balancing the workload in a parallel region. These what-if predictions are based on a post-mortem analysis using LogGPS [14], a parallel computational model that models the computing environment by several parameters. Predicted results are presented in a timeline view rendered by logviewer [4], a widespread visualization tool.

The paper is organized as follows. We begin in Section 2 by abstracting the execution of message passing programs with LogGPS. In Section 3, we describe a method for assessing the potential improvement of message passing programs. Section 4 describes PerWiz, which implements the method. Section 5 shows two case studies and Section 6 discusses related work. Finally, Section 7 concludes this paper.

2 Modeling Message Passing Programs

2.1 Definition of Parallel Region

Let $\mathcal{A}$ denote a message passing program. In order to analyze the workload distribution of $\mathcal{A}$, the program code that developers intend to execute in parallel must be identified. We call this code a parallel region, $\mathcal{R}$, composed of a set of blocks in $\mathcal{A}$. That is, any block b such that $b \in \mathcal{R}$ can be executed in parallel (or even in sequential, against developer's intension). Note here that any $b \in \mathcal{R}$ can include a routine call for message passing. Figure 1 shows two examples of a parallel region specified by two directives: PRGN_BEGIN and PRGN_END. Developers aim to parallelize the entire N iterations in Figure 1(a) and each of N iterations in Figure 1(b).

```
PRGN_BEGIN;
for (k=0; k<N; k++) {
    MPI_Sendrecv();
    Calculation;
}
PRGN_END;
```

(a)

```
for (k=0; k<N; k++) {
    PRGN_BEGIN;
    MPI_Sendrecv();
    Calculation;
    PRGN_END;
}
```

(b)

Fig. 1. Parallel regions specified (a) for the entire N iterations and (b) for each of N iterations.

In the following discussion, we call a logical step that corresponds to an execution of a parallel region as a *parallel step*.

2.2 Modeling Program Execution

An execution of $\mathcal{A}$ can be represented as a directed acyclic graph (DAG), $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ denote a set of vertexes and that of edges, respectively. While a vertex corresponds to an event occurred during an execution, an edge corresponds to *a happened-before relation* [15], $\rightarrow$, a precedence relationship defined between events.

An event occurs when a process executes a sequence of statements in $\mathcal{A}$. Events can be classified into two groups: communication and calculation events. A communication event corresponds to an execution of a communication routine, from its call to return. On the other hand, a calculation event corresponds to an execution of other statements processed between two successive communication events. Communication events can be more classified into send and receive events according to whether their corresponding routine sends or receives a message.

An execution time for $\mathcal{A}$ can be represented as the CP length of weighted $\mathcal{G}$, which has a weight associated with each vertex and edge. One method for weighting $\mathcal{G}$ is to use a realistic parallel computational model such as the LogP [16] family of models [14, 17]. LogGPS [14] is an extension of LogGP [17] and captures both synchronous and asynchronous messages by using the following seven parameters (see Figure 2).

- L: an upper bound on the latency, incurred in sending a message from its source processor to its target processor.
- o: the overhead, defined as the length of time that a processor is engaged in the transmission or reception of each message; During this time the processor cannot perform other operations.
- g: the gap between messages, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor.
- G: the Gap per byte for long messages, defined as the time per byte for a long message.
- P: the number of processor/memory modules.
- S: the threshold for message length, above which messages are sent in a synchronous mode.
- s: the threshold for message length, above which messages are sent in multiple packets.

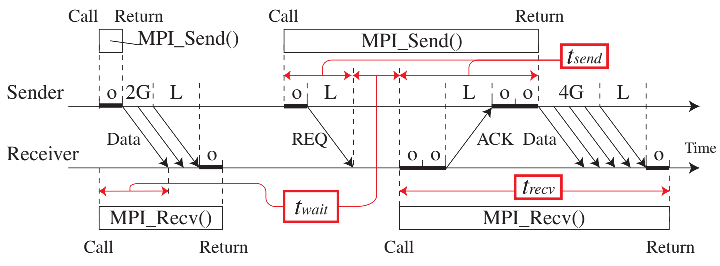

Fig. 2. Asynchronous and synchronous messages under LogGPS. An arrow from the sender to the receiver represents a 1-byte message.

In a precise sense, o should be distinguished for the send and receive overheads and also be represented by a linear function on message length. However, because the following explanation requires no precise notation, this paper uses a simple notation, o.

As illustrated in Figure 2, the execution times for send and receive events include waiting time t_{wait}, in addition to sending and receiving times, t_{send} and t_{recv}, respectively. Here, the waiting time for a receive event is defined as the receiver's time from the call of a receive routine to the arrival of the message. The waiting time for a send event, which appears only in a synchronous mode, is also defined as the sender's time from the arrival of the send request REQ to the call of a matching receive routine.

In this paper, waiting time zeroing for communication event e means obtaining $t_{wait} = 0$ for e. This is achieved by making a hypothesis in which e's matching routine is scheduled so that called for t_{wait} earlier than present.

3 Potential Improvement Assessment

This section describes a method for assessing the potential improvement of message passing programs, which perform communication and calculation.

3.1 Assessing Communication Bottlenecks

Tuning methods for communication bottlenecks can be classified into two groups.

T1: Message scheduling, aiming to minimize t_{wait}.
T2: Message reduction, aiming to minimize t_{send} and t_{recv}.

CP zeroing analysis is effective for T2 because applying this analysis to communication routines shows an upper bound on the potential improvement by assuming the amount of messages be zero byte. Therefore, this paper tackles on T1.

In general, focusing on a message with maximum t_{wait} does not always result in a significant improvement. That is, even if the message is scheduled to obtain $t_{wait} = 0$, the program will not reduce its execution time unless the length of CP is shortened. By contrast, scheduling a message with small t_{wait} can trigger *domino effect*, which can significantly improve the overall performance by reducing the waiting time of its subsequent messages, like as domino toppling. Thus, to evaluate the potential improvement

```
Inputs: (1) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a direct acyclic graph; (2) $\mathcal{P}$, a set of processes.
Outputs: (1) $\mathcal{D}$, a set of domino paths; (2) $T$, Predicted execution time if zeroing waiting time.
 1: Algorithm ComputingDominoPath($\mathcal{G}$, $\mathcal{P}$, $\mathcal{D}$, $T$);
 2: begin
 3:   $T := \infty$;
 4:   foreach process $p \in \mathcal{P}$ do begin
 5:     $D_p := \phi$;      // $D_p$: A domino path terminated at $e_{p,i}$
 6:     Select event $e_{p,i} \in \mathcal{V}$ such that $\neg\exists e_{p,j} \in \mathcal{V}\ (e_{p,i} \rightarrow e_{p,j})$;
 7:     $w(D_p)$ := ComputingForEachProcs($\mathcal{G}$, $e_{p,i}$, $\infty$);
 8:     if $(T > w(D_p))$ then $T := w(D_p)$;      // Select the minimum execution time
 9:   end
10:   $\mathcal{D} := \{D_p \mid w(D_p) = T\}$;
11: end
12: Function ComputingForEachProcs($\mathcal{G}$, $e_{p,i}$, $T_p$);
13: begin
14:   foreach event id $j \in \{1, 2, \cdots, i\}$ do begin
15:     $T_{p,j}$ := WaitingTimeZeroingSimulation($e_{p,j}$);      // See Section 4
16:   end
17:   Select event id $m \in \{1, 2, \cdots, i\}$ such that $\forall j \in \{1, 2, \cdots, i\}\ (T_{p,j} \geq T_{p,m})$;
18:   if $((e_{p,m} = e_{p,i})\ ||\ (T_{p,m} > T_p))$ then
19:     return $T_p$;
20:   else begin
21:     Select event $e_{q,j} \in \mathcal{V}$ such that $(p \neq q) \wedge ((e_{q,j} \rightarrow e_{p,m}) \vee (e_{p,m} \rightarrow e_{q,j}))$;
22:     Add events $e_{p,m}$ and $e_{q,j}$ to $D_p$;
23:     return ComputingForEachProcs($\mathcal{G}$, $e_{q,j}$, $T_{p,m}$);      // Recursive call
24:   end
25: end
```

Fig. 3. Algorithm for computing the domino paths of a DAG.

in terms of T1, our method searches a message that triggers the domino effect rather than a message with maximum t_{wait}.

We now propose an algorithm for locating a communication event that triggers the domino effect, aiming to achieve a significant improvement with little effort. Figure 3 describes our algorithm, which requires a DAG and a set of processes, $\mathcal{G}$ and $\mathcal{P}$, respectively, then returns a set of domino paths, $\mathcal{D}$, with the minimum of predicted execution time, T. To locate the event, our algorithm recursively backtracks happened-before relations $\rightarrow$ in $\mathcal{G}$, and repeatedly predicts the execution time with a hypothesis in which a specific communication event $u \in \mathcal{V}$ is scheduled to have $t_{wait} = 0$. As a result, it computes a *domino path* (DP), a set of events that participate in the domino effect. As presented later in Section 4, our prediction is based on LogGPS, which clearly defines t_{wait} for MPI routines so that improves the prediction accuracy for synchronous messages, compared to LogP and LogGP [14].

Let $e_{p,i}$ denote the i-th event occurred on process p. The relation backtracking is performed in three steps as follows. First, the algorithm starts with event $e_{p,i}$ last occurred on process p (line 6), then searches event number m, where $1 \leq m \leq i$, such that $e_{p,m}$ accompanies the minimum execution time $T_{p,m}$ if assuming $t_{wait} = 0$ for $e_{p,m}$

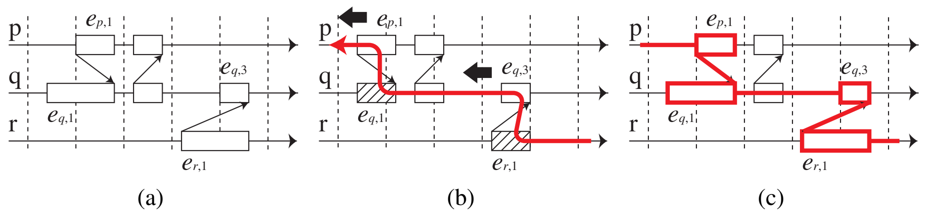

Fig. 4. Computing process for a domino path. (a) Given a DAG, the algorithm (b) recursively backtracks happened-before relations so that (c) locates a domino path.

(line 14–17). Here, the algorithm in Figure 3 is simplified to obtain such m in unique, however, our actual algorithm performs round robin prediction on every m with minimum $T_{p,m}$. In the next step, it terminates this backtracking (1) if current computed DP D_p already includes $e_{p,m}$ or (2) if current predicted time $T_{p,m}$ exceeds to T_p, the minimum execution time predicted before (line 18). Otherwise, the algorithm searches $e_{p,m}$'s matching event $e_{q,j}$ (line 21), then add $e_{p,m}$ and $e_{q,j}$ to D_p (line 22). Finally, applying this backtracking recursively to $e_{q,j}$ (line 23) computes a DP terminated on p, and performing this for all processes $p \in \mathcal{P}$ (line 4) returns $\mathcal{D}$ and T (lines 8 and 10).

Figure 4 illustrates an example of computing process for a DP. Our algorithm backtracks $e_{r,1} \rightarrow e_{q,3}$, then $e_{q,1} \rightarrow e_{p,1}$, so that points out a DP containing $e_{p,1}$, $e_{q,1}$, $e_{q,3}$ and $e_{r,1}$. The waiting times of $e_{q,1}$ and $e_{r,1}$ can be reduced by calling $e_{p,1}$'s corresponding send routine earlier in order to cause the occurrence of $e_{p,1}$ forward.

3.2 Assessing Calculation Bottlenecks

As we did for communication bottlenecks, tuning methods for calculation bottlenecks can also be classified into two groups.

T3: Load balancing, aiming to obtain a better workload distribution.
T4: Time complexity reduction, aiming to minimize calculation time.

Like for T2, CP zeroing is also effective for T4 because it predicts an upper bound on the potential improvement by assuming the execution time of a specific procedure be zero second. Therefore, we focus on T3.

The concept of parallel region is useful to evaluate the potential improvement that will be derived by T3, because it enables predicting the execution time if the workload is balanced among processors. For example, the most imbalanced parallel step can be detected by computing the standard deviation $\sigma(t_k)$ of execution time t_k, where t_k represents the time that a process requires in the k-th parallel step. However, as we mentioned for T1, focusing on a parallel step with maximum $\sigma(t_k)$ does not always derive a significant improvement. Therefore, by using LogGPS simulation, our method searches a parallel step that derives the minimum execution time if its workload is assumed to be balanced ($\sigma(t_k) = 0$).

Recall here that parallel regions can include routine calls for message passing. Therefore, the above method for calculation bottlenecks can also be applied to communication bottlenecks in order to predict the execution time if the amount of messages is assumed to be balanced in a parallel region.

4 PerWiz: A What-If Prediction Tool

PerWiz has the following three functions.

– **F1: Function for modification choice search.**
  To assist developers in achieving a significant improvement with little effort, PerWiz presents modification choices sorted by their significances. For communication bottlenecks, it shows a DP computed by the algorithm presented in Section 3.1. For calculation bottlenecks, as mentioned in Section 3.2, it locates a parallel step that reduces the program's execution time by load balancing.
– **F2: Function for what-if prediction.**
  This function is a basis for function F1 and predicts the program's execution time for the following what-if questions: what performance will be derived if making a hypothesis in which
  • the waiting time or the execution time of a specific event becomes zero second;
  • all processes pass the same amount of messages or require the same execution time in a specific parallel step.

  The predicted results for the above questions can be visualized by logviewer, as shown in Figure 5(c). In addition, developers can give PerWiz a combination of the questions, allowing them to consider various tuning techniques.
– **F3: Function for lower bound prediction.**
  In order to assist developers in changing the parallel algorithm employed in the target program, this function computes a lower bound on execution time for each parallel region. Since different algorithms produce different DAGs, it predicts a lower bound by simply summing up the breakdown of execution time. That is, F3 needs only the breakdown of calculation, sending, receiving, and waiting times while F2 requires a DAG in order to simulate a program execution with proper precedence. Developers can give and combine the following what-if questions: what performance will be derived if making a hypothesis in which
  • the waiting time or the execution time spent for a specific parallel region becomes zero second;
  • all processes pass the same amount of messages or require the same execution time in a specific parallel region.

To tune MPI programs with PerWiz, developers first must specify parallel regions in their code, as presented in Figure 1. After this instrumentation, compiling the instrumented code generates an object file and linking it with an instrumentation library generates an executable binary file. By executing the binary file in parallel, PerWiz generates a trace file, $\mathcal{L}$, logged in ALOG format [18]. The instrumentation library for ALOG is widely distributed with MPICH [19], a basis of many MPI implementations. Finally, giving a what-if question and $\mathcal{L}$ generates a predicted result in a text file and

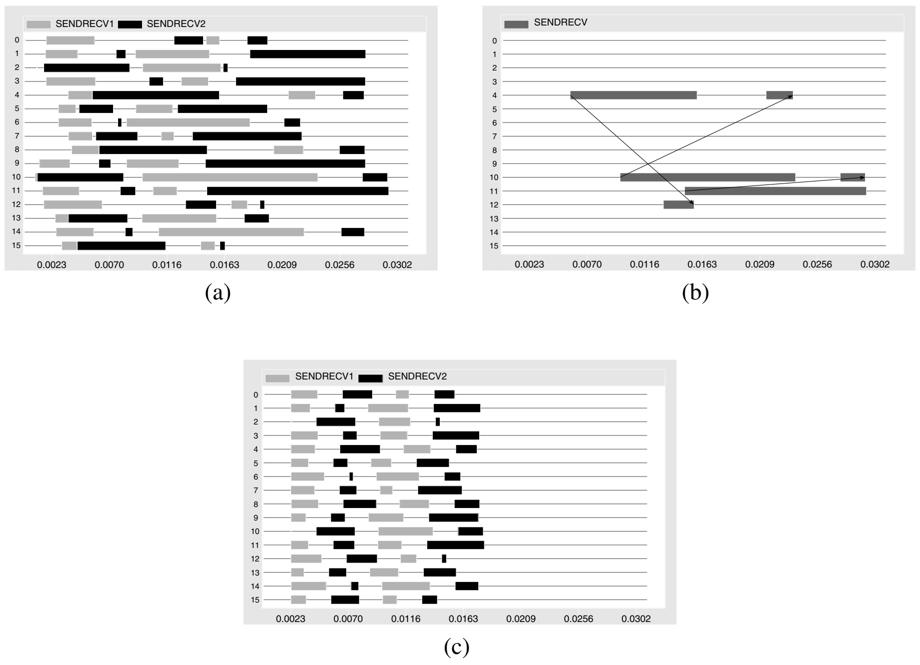

Fig. 5. Predicted results in timeline views rendered by logviewer. (a) The original behavior on 16 processes, (b) its computed domino path, and (c) a predicted behavior if the workload in parallel steps is assumed to be balanced. A colored box represents an execution of MPI_Sendrecv(). Orange and red boxes correspond to an execution in odd and even parallel steps, respectively.

a reconstructed ALOG file, $\mathcal{L}'$. Note here that giving $\mathcal{L}'$ to PerWiz enables iterative prediction without additional program execution.

We now present how PerWiz reconstructs predicted trace file $\mathcal{L}'$ from recorded trace file $\mathcal{L}$. In order to realize accurate performance predictions, PerWiz utilizes the measured execution time in $\mathcal{L}$ as much as possible. For calculation events, PerWiz applies the measured time in $\mathcal{L}$ to the predicted time in $\mathcal{L}'$. For communication events, PerWiz employs two approaches according to whether the event passes a synchronous message or an asynchronous message:

– For asynchronous communication events, PerWiz uses the LogGPS execution time, computed for every length of messages [14];
– For synchronous communication events, PerWiz estimates both sending time t_{send} and receiving time t_{recv} from measured $\mathcal{L}$, then utilizes them for $\mathcal{L}'$. For example, given send event u, it first decomposes its measured time t_{all} recorded in $\mathcal{L}$ into $t_{wait} = max(t_v - t_u - (o + L), 0)$ and $t_{send} = t_{all} - t_{wait}$ (see Figure 2), where t_u and t_v denote the occurrence time of u and that of a matching receive event v recorded in $\mathcal{L}$, respectively, and $o + L$ represents the estimated time defined from the occurrence of v to the arrival of REQ. PerWiz then regards $t'_{all} = max(t'_v -$

```
PRGN_BEGIN;
for ($\phi$=1; $\phi \leq$ N; $\phi$++) {
    if (MPI_Comm_rank() $\in$ group $g(\phi)$) {
        Local gradient calculation;
        Communication within $g(\phi)$ by MPI_Send(),
            MPI_Recv(), and MPI_Sendrecv();
    }
}
PRGN_END;
```

(a)

```
for (k=1; k $\leq$ log P; k++) {
    PRGN_BEGIN;
    Local calculation for image splitting;
    MPI_Sendrecv();
    Local calculation for image compositing;
    PRGN_END;
}
```

(b)

Fig. 6. Pseudo code of (a) image registration and (b) image compositing applications.

$t'_u - (o + L), 0) + t_{send}$ as the predicted execution time for u, where t'_u and t'_v denote the predicted occurrence time of u and that of v in $\mathcal{L}'$, respectively. The same approach determines the predicted execution time for receive events.

5 Case Studies

This section shows two case studies in order to demonstrate the usefulness of PerWiz. We used a cluster of 64 Pentium III 1-GHz PCs interconnected by a Myrinet switch [20], yielding a full-duplex bandwidth of 2 Gb/s. We also used the MPICH-SCore library [21], a fast MPI implementation.

The values of LogGPS parameters were $L = 9.11$ μs, $o = 2.15$ μs, and $S =$ 16 383 bytes, measured using the method presented in [14]. Note here that the remaining LogGPS parameters had no influence on the predicted results because the applications employed for the studies passed only synchronous (long) messages, which require only L and o for prediction, as presented in Section 4.

5.1 Case 1: Performance Improvement by Message Scheduling

We applied PerWiz to an image registration application [22], which aligns a pair of three-dimensional (3-D) images. We improved its performance by reducing waiting time. Figure 6(a) shows its pseudo code with its parallel region. In this application, every process holds a portion of the images and computes the gradient of a function that represents the similarity between the images. This gradient has to be calculated for each point ϕ, placed dynamically in the images. In addition, the gradient calculation at ϕ requires the neighborhood pixels of ϕ. To parallelize this application, developers dynamically organized processes into groups, aiming at concurrent processing of gradient calculations. All processes that hold the neighborhood pixels of ϕ compose group $g(\phi)$, and all processes in $g(\phi)$ participate in the calculation at ϕ.

We first detected the hotspot of this application by using gprof [23], a profiling tool. The hotspot was the gradient calculation repeated for 14 times. Therefore, we decided to instrument only one of the 14 repetitions. We executed the instrumented code on 16 dedicated processors, each with one process, so that generated 720 KB of a trace file that contains 4634 communication events. The execution time was 23.8 s including a run-time overhead of 0.1 s for trace generation.

Table 1. Lower bound prediction for image registration and image compositing applications by PerWiz (function F3). Q1, Q2, and Q3 denote questions if zeroing waiting time, balancing the workload and the amount of messages in its parallel region, respectively.

What-if question	Predicted execution time		What-if question	Predicted execution time	
	Registration T_1 (s)	Compositing T_2 (ms)		Registration T_1 (s)	Compositing T_2 (ms)
—	23.8	30.6	Q1 $\wedge$ Q2	5.3	16.5
Q1	8.2	22.3	Q1 $\wedge$ Q3	8.0	16.8
Q2	23.4	34.9	Q2 $\wedge$ Q3	23.6	38.2
Q3	21.4	25.7	Q1 $\wedge$ Q2 $\wedge$ Q3	5.0	11.0

Table 2. Predicted and measured times for image registration application after l-th modification, where $0 \leq l \leq 7$. Code modification is based on the events located by the domino path approach and the longest waiting time approach.

l	Domino path approach by PerWiz			Longest waiting time approach	
	Execution time (s)		Waiting time (s)	Execution time (s)	Waiting time (s)
	$T_{P,l}$: Predicted	$T_{M,l}$: Measured	$t_{wait,l}$	$T_{M,l}$: Measured	$t_{wait,l}$
0	—	23.8	—	23.8	—
1	21.6	21.8	11.8	23.9	12.5
2	19.5	19.5	10.7	22.6	14.7
3	18.2	18.2	7.5	22.6	11.9
4	15.9	15.8	5.8	21.5	13.5
5	14.8	14.8	3.4	21.5	11.4
6	14.4	14.4	0.4	20.7	12.3
7	13.7	13.7	0.7	20.8	10.7

Table 1 shows the execution time, T_1, predicted by function F3, the lower bound prediction. We obtained $T_1 < 10$ s if Q1 is specified: what performance will be derived if zeroing the waiting time in the parallel region? Therefore, reducing waiting time is necessary to improve the performance of this application.

In order to clarify the usefulness of PerWiz, we now compare two approaches for reducing waiting time in MPI programs: (1) the DP approach, which locates an event that triggers the domino effect computed by PerWiz and (2) the longest waiting time (LWT) approach, which locates an event with LWT. Because the waiting time in this application was due to the processing order of points (loops at line 2), we repeatedly modified the code to obtain an appropriate order that reduces the waiting time of the event located by each approach. To compute a DP, PerWiz took approximately 10 s on a Pentium III 1-GHz system.

Table 2 compares predicted time $T_{P,l}$ and measured time $T_{M,l}$ after the l-th modification, where $0 \leq l \leq 7$. PerWiz enables developers to reduce $T_{M,l}$ from 23.8 to 13.7 s after the seventh modification. By contrast, the LWT approach results in 20.8 s. Thus, our DP approach allows developers to efficiently improve this application and to derive a performance improvement of 42% while the LWT approach gives that of 13%.

Furthermore, our DP approach successfully gives a performance improvement for every modification, whereas the LWT approach fails to reduce execution time $T_{M,l}$ at

$l = 1, 3, 5$, and 7. This is due to the LWT approach, which lacks the guarantee on the reduction of the length of CP. By contrast, our approach guarantees it by LogGPS simulation. For example, the LWT approach reduces $t_{wait,l}$, where $l = 1, 3, 5$, and 7, however, this reduction increases the waiting times of the succeeding events, resulting in a similar total performance.

The domino effect appears at $t_{wait,l}$, the waiting time of the event located at the l-th modification. In our approach, $t_{wait,l}$ decreases as l increases. This is due to the domino effect, which reduces the waiting time of many events that compose the computed DP. Actually, at every modification, our DP approach reduces the total amount of the waiting time by approximately 25 s, whereas the LWT approach reduces it by 0–10 s. Thus, our DP approach directs developers to events with a shorter waiting time but with a promised improvement, and the LWT approach directs them to events with a longer waiting time but with an uncertain improvement. Therefore, PerWiz enables developers to efficiently improve the application's performance.

5.2 Case 2: Performance Improvement by Load Balancing

We also applied PerWiz to an image compositing application [24] for 3-D volume rendering systems (see Figure 6(b)). We improved its performance by load balancing. On P processes, it merges P locally rendered images into the final image in $\log P$ stages. At each stage, all processes are paired up, and the two processes involved in a compositing split the image plane into two pieces. Each process then takes responsibility for one of the two pieces and exchanges the other piece. Repeating this splitting and exchanging with different pairs of processes for $\log P$ times produces the final image in a distributed manner. Since every process takes responsibility for $1/2^k$ image at the k-th stage, where $1 \leq k \leq \log P$, developers expected a good workload distribution. However, in practice, it had a load imbalance issue due to transparent pixels that can omit calculation for compositing. The trace file generated on 64 dedicated processors was 100 KB in size, containing 384 communication events.

The lower bound on execution time, T_2, presented in Table 1 indicates that its execution time can be reduced from 30.6 to 11.0 ms by applying all of the following three tuning techniques: reducing waiting time, and balancing the workload and the amount of messages in its parallel region. However, the most effective technique is unclear because T_2 decreases by approximately 5 s when adding these techniques successively to the question given to PerWiz (22.3, 16.5, and 11.0 ms).

In this application, because every process synchronizes at every stage, we first intended to reduce its waiting time by balancing the workload. We then used function F2 to predict the execution time, T_3, possible by balancing the workload in the k-th parallel step, where $1 \leq k \leq 6$. Table 3 shows the results, which give two hints toward a performance improvement: (1) load balancing in all parallel steps reduces T_3 from 30.6 to 11.0 ms; and (2) this load balancing is effective especially for parallel step S_2.

First, we discuss on hint (1). The minimum time of $T_3 = 11.0$ ms equals to that of $T_2 = 11.0$ ms predicted for Q1 $\wedge$ Q2 $\wedge$ Q3, as presented in Table 1. This indicates that, as we intended before, load balancing in all parallel steps successfully reduces the waiting time in this application, because it yields the same performance possible by

Table 3. What-if prediction for image compositing program by PerWiz (function F2). Q4(k) represents a question if balancing the workload in parallel step S_k, where $1 \leq k \leq 6$.

What-if question	Predicted time T_3 (ms)	What-if question	Predicted time T_3 (ms)
—	30.6	Q4(4)	27.2
Q4(1)	29.1	Q4(5)	29.6
Q4(2)	21.6	Q4(6)	29.9
Q4(3)	26.4	Q4(1) $\wedge \cdots \wedge$ Q4(6)	11.0

zeroing the waiting time of every communication event. Thus, the most efficient modification was to balance the workload in all parallel steps. To realize this, we developed the BSLBR method [24], which splits images into interleaved pieces rather than block pieces. BSLBR achieved an execution time of 14.8 ms.

For hint (2), we examined the sparsity of images at each stage. We then found that exploiting the sparsity of images was inadequate immediately after the first compositing step, so that it augmented load imbalance in S_2. To address this issue, we developed the BSLMBR method [24], which exploits more sparsity of images with a low overhead. BSLMBR reached a minimum execution time of 11.8 ms, which is close to the minimum predicted time, $T_3 = 11.0$ ms.

6 Related Work

Paradyn [7] performs an automatic run-time analysis for searching performance bottlenecks based on thresholds and a set of hypotheses structured in a hierarchy. For example, it locates a synchronization bottleneck if waiting time is greater than 20% of the program's execution time. Aksum [8] is also based on a similar approach that is capable of multi-experiment performance analysis for different problems and machine sizes in order to automate performance diagnosis of parallel and distributed programs. Because these diagnoses are based on dynamic instrumentation technology, sophisticated search strategies for reducing run-time overhead [9] and for quickly finding bottlenecks [10] are developed recently.

SCALEA [25] is a performance instrumentation, measurement, and analysis system that explains the performance behavior of each program by computing a variety of performance metrics regarding data movement, synchronization, control of parallelism, additional computation, loss of parallelism, and unidentified overheads.

Prophesy [26] gives insight into how system features impact application performance. It provides a performance data repository that enables the automatic generation of performance models. To develop these models, it uses coupling parameter [27], a metric that quantifies the interaction between kernels that compose an application.

TAU performance system [28] aims to enable online trace analysis in order to overcome the inherent scalability limitations of post-mortem analysis. Although its preliminary testbed is presented in [28], scalability performance tests are left as future work.

Dimemas [29] is a simulator for analyzing the influence of sharing a multiprocessor system among several parallel applications. It enables studying the influence of different scheduling policies in the global system performance.

In contrast to these previous works, the novelty of PerWiz is the assessment of performance bottlenecks to guide developers to bottlenecks with a promised improvement.

7 Conclusion

We have presented a performance prediction tool named PerWiz, which focuses on assessing the potential improvement of message passing programs. PerWiz directs developers to where a significant improvement can be achieved, as a result of applying typical tuning techniques. To enable this, PerWiz performs a post-mortem analysis based on a parallel computational model, then locates domino paths and parallel steps in the program. Furthermore, PerWiz predicts the execution time under specific assumptions such as if zeroing wait time, balancing the workload and the amount of messages.

In case studies where PerWiz played a key role in tuning MPI programs, we confirmed that focusing on a domino path effectively improves the performance of MPI programs. Therefore, we believe that PerWiz is useful for developers to investigate the reduction in execution time that will be derived from a modification.

Future work includes incorporating dynamic instrumentation technology into PerWiz to reduce instrumentation overhead for large-scale applications. Furthermore, runtime optimization techniques are required to tune irregular applications that dynamically vary program behavior according to run-time situations.

References

1. Message Passing Interface Forum: MPI: A message-passing interface standard. Int'l J. Supercomputer Applications and High Performance Computing **8** (1994) 159–416
2. Heath, M.T., Etheridge, J.A.: Visualizing the performance of parallel programs. IEEE Software **8** (1991) 29–39
3. Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K.: VAMPIR: Visualization and analysis of MPI resources. The J. Supercomputing **12** (1996) 69–80
4. Zaki, O., Lusk, E., Gropp, W., Swider, D.: Toward scalable performance visualization with Jumpshot. Int'l J. High Performance Computing Applications **13** (1999) 277–288
5. Rose, L.A.D., Reed, D.A.: SvPablo: A multi-language architecture-independent performance analysis system. In: Proc. 28th Int'l Conf. Parallel Processing (ICPP'99). (1999) 311–318
6. Yan, J., Sarukkai, S., Mehra, P.: Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. Software: Practice and Experience **25** (1995) 429–461
7. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn parallel performance measurement tool. IEEE Computer **28** (1995) 37–46
8. Fahringer, T., Seragiotto, C.: Automatic search for performance problems in parallel and distributed programs by using multi-experiment analysis. In: Proc. 9th Int'l Conf. High Performance Computing (HiPC'02). (2002) 151–162
9. Cain, H.W., Miller, B.P., Wylie, B.J.N.: A callgraph-based search strategy for automated performance diagnosis. Concurrency and Computation: Practice and Experience **14** (2002) 203–217
10. Roth, P.C., Miller, B.P.: Deep Start: a hybrid strategy for automated performance problem searches. Concurrency and Computation: Practice and Experience **15** (2003) 1027–1046

11. Block, R.J., Sarukkai, S., Mehra, P.: Automated performance prediction of message-passing parallel programs. In: Proc. High Performance Networking and Computing Conf. (SC95). (1995)
12. Hollingsworth, J.K.: Critical path profiling of message passing and shared-memory programs. IEEE Trans. Parallel and Distributed Systems **9** (1998) 1029–1040
13. Eom, H., Hollingsworth, J.K.: A tool to help tune where computation is performed. IEEE Trans. Software Engineering **27** (2001) 618–629
14. Ino, F., Fujimoto, N., Hagihara, K.: LogGPS: A parallel computational model for synchronization analysis. In: Proc. 8th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'01). (2001) 133–142
15. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM **21** (1978) 558–565
16. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: Proc. 4th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP'93). (1993) 1–12
17. Alexandrov, A., Ionescu, M., Schauser, K., Scheiman, C.: LogGP: Incorporating long messages into the LogP model for parallel computation. J. Parallel and Distributed Computing **44** (1997) 71–79
18. Herrarte, V., Lusk, E.: Studying parallel program behavior with `upshot`. Technical Report ANL–91/15, Argonne National Laboratory (1991)
19. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22** (1996) 789–828
20. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.K.: Myrinet: A gigabit-per-second local-area network. IEEE Micro **15** (1995) 29–36
21. O'Carroll, F., Tezuka, H., Hori, A., Ishikawa, Y.: 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). (1998) 243–250
22. Schnabel, J.A., Rueckert, D., Quist, M., Blackall, J.M., Castellano-Smith, A.D., Hartkens, T., Penney, G.P., Hall, W.A., Liu, H., Truwit, C.L., Gerritsen, F.A., Hill, D.L.G., Hawkes, D.J.: 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). (2001) 573–581
23. Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: a call graph execution profiler. In: Proc. SIGPLAN Symp. Compiler Construction (SCC'82). (1982) 120–126
24. Takeuchi, A., Ino, F., Hagihara, K.: An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. Parallel Computing **29** (2003) 1745–1762
25. Truong, H.L., Fahringer, T.: SCALEA: a performance analysis tool for parallel programs. Concurrency and Computation: Practice and Experience **15** (2003) 1001–1025
26. Taylor, V., Wu, X., Stevens, R.: Prophesy: An infrastructure for performance analysis and modeling of parallel and grid applications. ACM SIGMETRICS Performance Evaluation Review **30** (2003) 13–18
27. Geisler, J., Taylor, V.: Performance coupling: Case studies for improving the performance of scientific applications. J. Parallel and Distributed Computing **62** (2002) 1227–1247
28. Brunst, H., Malony, A.D., Shende, S.S., Bell, R.: Online remote trace analysis of parallel applications on high-performance clusters. In: Proc. 5th Int'l Symp. High Performance Computing (ISHPC'03). (2003) 440–449
29. Labarta, J., Girona, S., Cortes, T.: Analyzing scheduling policies using Dimemas. Parallel Computing **23** (1997) 23–34