A Performance Analysis Tool for Performance Debugging of Message Passing Parallel Programs

Fumihiko Ino and Kenichi Hagihara

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

Abstract

This paper describes the design and implementation of Gordini, a performance analysis tool that is capable of automatically locating places in the source code where a communication optimization technique can be applied for performance debugging of message passing parallel programs. Our automatic search approach is based on data dependence analysis on trace files. It currently supports three techniques: communication–computation overlap, message aggregation, and collective communication. In case studies, Gordini assists us in improving the performance of sophisticated programs coded by experts, the two first prize winners of software contests. Therefore, we believe that our automatic approach is useful for application developers to locate communication bottlenecks in programs.

1 Introduction

Developing efficient parallel applications is not easy compared to sequential applications. For example, message passing parallel applications with inappropriate data/workload distribution can easily result in poor performance due to frequent communication and load imbalance among processors. Therefore, to create faster applications, it is important to locate and eliminate performance bottlenecks that limit application performance.

In order to assist developers in performance analysis of message passing parallel applications, many performance analysis tools [5] have been proposed in the past. Such tools offer useful capabilities based on postmortem analysis of a trace file, namely a file of time-stamped events recorded during program execution. For example, Paragraph [8], VAMPIR [13], Jumpshot [23], and TAU [18] provide the timeline view of events and messages, which enables developers to intuitively understand the status and behavior of processors along the time axis. They also are capable of displaying several diagrams with statistical analysis of program execution and that of communication primitives.

In contrast to these postmortem tools, some tools employ dynamic instrumentation techniques to analyze large-scale applications running for hours or days on hundreds of processors. AIMS [22] supports source code instrumentation, runtime monitoring, graphical execution profiles, performance indices, and automated modeling techniques in order to support performance visualization, profiling, and modeling. Paradyn [12] performs automatic runtime analysis for performance bottleneck search 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.

Thus, earlier tools offer useful capabilities for performance analysis. However, only Paradyn guides developers directly to performance bottlenecks in applications. Most of the tools need developers to (1) locate performance bottlenecks in graphical views, (2) identify the causal reasons for their occurrences, and (3) determine the methods for their elimination. All of the above performance debugging processes are due to developers. Therefore, even if developers notice that a communication routine spends most of the overall execution time, they must guess the causal reasons. For example, it may be due to the long waiting time for synchronization between processors, or simply be due to the accumulated time for iterative execution of the routine inside a loop body. Accordingly, some developers fail to identify the causal reasons, resulting in poor performance.

Furthermore, finding performance bottlenecks in large-scale applications is not easy for developers, because such applications usually generate large trace files, which make graphical views complicated. These complicated views lose the visual advantage of intuitive understanding, so that even experienced developers may miss performance bottlenecks hidden in graphical views.

Proceedings of the 2004 International Conference on Parallel Processing Workshops (ICPPW'04)
1530-2016/04 $20.00 © 2004 **IEEE**

Paradyn supports this search process by locating performance bottlenecks with their causal reasons. However, developers need to consider how to eliminate the bottlenecks. Furthermore, because its search strategy is structured in a hierarchy like as modules, routines, and statements, some bottlenecks can be missed in a higher level of the hierarchy. Therefore, detailed analysis may be required to expose the bottlenecks hidden in a lower level of the hierarchy.

To overcome issues (1)–(3), our tool aims at satisfying the following two requirements.

R1: The tool is capable of locating performance bottlenecks automatically.

R2: The tool is capable of guiding developers how to eliminate the located performance bottlenecks.

Satisfying the above requirements frees developers from performance bottleneck search and from determining the methods for bottleneck elimination, allowing them to focus on program modification.

In this paper, we describe the design and implementation of Gordini, a performance analysis tool that aims at satisfying requirements R1 and R2 for message passing parallel programs. The key capability of Gordini is automatic bottleneck search based on data dependence analysis on trace files. Gordini directs developers to places in the source code where communication optimization techniques can be applied. It currently supports the following three techniques: communication–computation overlap [17], message aggregation [2], and collective communication. We also present some case studies in which Gordini successfully improves the performance of sophisticated programs.

2 Overview of Gordini

2.1 Architecture

Gordini currently supports parallel programs written using the C language and the Message Passing Interface (MPI) standard [11]. It runs on computing platforms where the C++ language and the Tcl/Tk toolkit, a graphical user interface (GUI) toolkit, are available.

Gordini consists of three components: performance bottleneck search, performance analysis, and performance visualization modules, as shown in Figure 1. The performance visualization module offers typical views including line/circle diagram and timeline views for presenting performance analysis results. It also provides a source code view for presenting bottleneck search results. Visualizing the bottlenecks directly on the source code is essential to assist developers in performance debugging. If they are not presented on the source code, developers have to locate the places that trigger the bottlenecks. Therefore, developers can fail to eliminate them, if the search results are not associated with the source code.

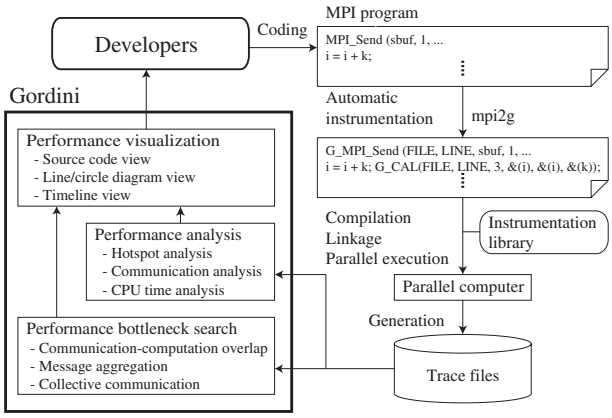

Figure 1. Performance debugging process with Gordini.

Figure 1 shows the performance debugging process with Gordini. First of all, developers must instrument the target MPI programs to generate trace files. This instrumentation can automatically be done by mpi2g, an instrumentation tool provided by Gordini. It performs pattern matching to replace all of the MPI routines and assignments in the program with instrumented routines and assignments, respectively. Next, they have to execute the instrumented code on a parallel computer so that generate trace files. Giving the trace files to Gordini provides developers helpful insights and hints for performance improvement. According to this information, developers are required to investigate whether the located places can easily be modified, and if possible, they can apply optimization techniques to the places.

Figure 2 shows an example of the source code view, which present search results obtained by data dependence analysis (described later in Section 3). The right-side window presents a performance bottleneck located on the source code. On the other hand, the left-side window shows the same bottleneck located for every event, aiming to present more detailed results. This event-level view is motivated by a situation where optimization techniques are conditionally applicable to a statement. For example, suppose that an assignment is placed inside a loop body. Then, this statement generates many computation events, and only the first ten of them may be allowed to overlap with a communication routine, as presented in Figure 2. To deal with this conditional case, Gordini presents the results on the event-level view in addition to the source code view.

In Figure 2, there is a triple-nested iteration at lines 78, 84, and 85. In this case, developers can overlap communication with computation in three steps. First, they should separate the middle loop into two loops, a loop for the

Proceedings of the 2004 International Conference on Parallel Processing Workshops (ICPPW'04)
1530-2016/04 $20.00 © 2004 **IEEE**

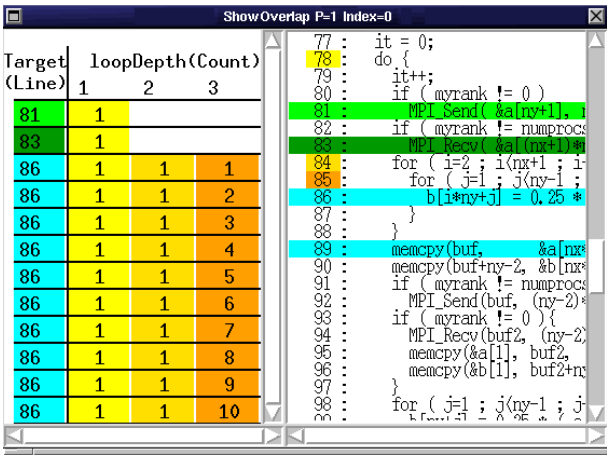

Figure 2. Source code view of Gordini. The right- and left-side windows present a search result for statements and events, respectively. The right-side window indicates that send and receive routines at lines 81 and 83 can overlap with computations at lines 86 and 89. The left-side window shows this in more detail and enumerates overlappable events with their corresponding lines in the source code and their execution number associated with each loop. It indicates that communication events generated from lines 81 and 83 can overlap with computation events generated from line 86 during from the first to the tenth iterations of the most inner loop (line 85) under the first iterations of the remaining two loops (lines 78 and 84).

first ten iterations and a loop for the remaining iterations. Then, they should replace blocking routines (`MPI_Send` and `MPI_Recv`) with nonblocking routines (`MPI_Isend` and `MPI_Irecv`), and finally, placing waiting routines (`MPI_Wait`) between the separated loops realizes overlap.

2.2 Trace File Generation

Trace files are generated for each process participating in parallel execution. The processes record all events occurred during the execution. Table 1 shows the recorded events with their information classified into two groups: common information recorded for every event and unique information recorded for specific events. This information is the basis for the bottleneck search module of Gordini.

A send event and a receive event correspond to an execution of a send routine and that of a receive routine. A computation event corresponds to an execution of other statements such as an assignment to a variable and a copy operation between buffers.

Table 1. Events and their information recorded in trace files.

Event type	Recorded information
All	Event ID, corresponding source code, and occurrence time
Computation	Read/write address
Send/receive	All arguments including source/destination process, send/receive address, and its size

3 Performance Bottleneck Search Based on Data Dependence Analysis

This section describes the key capability of Gordini, an automatic bottleneck search based on data dependence analysis. Due to the space limitation, we present algorithms for communication–computation overlap and message aggregation.

3.1 Motivation for Data Dependence Analysis

All the three techniques mentioned in Section 1 have a common characteristic. That is, they try to eliminate communication bottlenecks by optimizing the execution order of statements. Therefore, when applying these techniques to a program, it is necessary to take into consideration data dependence among executed statements, namely events. Otherwise, the modified program may show buggy behaviors due to broken dependences. Thus, data dependence analysis is required to search places where the three techniques can be applied.

Let $\rightarrow$ denote a relation that represents data dependence among events. Data dependence from event a to event b, $a \rightarrow b$, can be classified into the following three types: output, flow, and anti dependences [2]. For events a and b such that $a \rightarrow b$, the occurrence of a must be prior to that of b.

We also define relation $\rightarrow$ from a set of events to an event. $\mathcal{A} \rightarrow b$ represents that $\exists a \in \mathcal{A}, a \rightarrow b$, where $\mathcal{A}$ is a set of events and b is an event.

3.2 Search Algorithm for Communication–Computation Overlap

In order to assist developers in communication–computation overlap, Gordini computes a set of computation events for each communication event in a trace file.

Figure 3 describes our search algorithm for communication–computation overlap. The algorithm requires trace file $\mathcal{T} = \{e_1, e_2, \ldots, e_N\}$, where e_i represents the i-th event occurred on a process and N represents

Proceedings of the 2004 International Conference on Parallel Processing Workshops (ICPPW'04)
1530-2016/04 $20.00 © 2004 **IEEE**

Algorithm for communication–computation overlap
Input: $\mathcal{T} = \{e_1, e_2, \ldots, e_N\}$, a trace file with N events
Outputs: $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$ and $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n$, where $\mathcal{O}_m$ represents a set of computation events that can overlap with communication event c_m $(1 \leq m \leq n)$

```
1.  n = 0;
2.  for i = N downto 1 do begin
3.      if (e_i is not communication event) then continue;
4.      n = n + 1;
5.      c_n = e_i; O_n = φ;
6.      for j = i + 1 to N do begin          // Forward search
7.          if (c_n → e_j) then break;
8.          Add event e_j to set O_n;
9.      end
10.     for k = i − 1 downto 1 do begin      // Backward search
11.         if (e_k → c_n) then break;
12.         Add event e_k to set O_n;
13.     end
14. end
```

Figure 3. Search algorithm for communication–computation overlap.

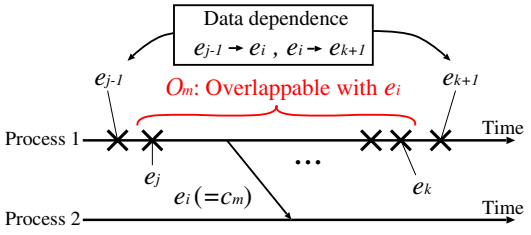

Figure 4. Search process for communication–computation overlap.

the number of events. Then, it returns a set of communication events, $\mathcal{C} = \{c_1, c_2, \ldots, c_n\}$, where $c_m \in \mathcal{T}$ is a communication event and n $(\leq N)$ is the number of communication events in $\mathcal{T}$, with its overlappable computation events $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_n$, where $\mathcal{O}_m$ is a set of computation events overlappable with c_m $(1 \leq m \leq n)$.

Figure 4 shows a search process for communication–computation overlap. Because any computation event a that overlaps with communication event b must satisfy $\neg(a \rightarrow b) \wedge \neg(b \rightarrow a)$, for communication event e_i $(= c_m)$, the algorithm computes a sequence of events $e_j, e_{j+1}, \ldots, e_k$ such that $(j < i < k) \wedge (e_{j-1} \rightarrow e_i) \wedge (\forall l\ (j \leq l < i), \neg(e_l \rightarrow e_i)) \wedge (e_i \rightarrow e_{k+1}) \wedge (\forall l\ (i < l \leq k), \neg(e_i \rightarrow e_l))$. To compute this, the algorithm performs forward and backward search from c_m until finding event e such that $c_m \rightarrow e$ and $e \rightarrow c_m$, respectively.

Thus, the algorithm searches computation events that can overlap with a communication event. Another approach is to search communication events that can overlap with a computation event. However, this opposite approach possibly results in less overlappable events because programs usually use temporary working variables, which cause many data dependences among computation events due to repeated assignments to the variables. Therefore, searching from communication events is necessary to avoid such many data dependences among computation events.

Note here that our algorithm independently analyzes each of trace files generated on processes. Because the algorithm is based on data dependence analysis, it does not require clock synchronization of processors during trace file generation. We discuss on the timing gap among processors' clocks later in Section 5.3.

Algorithm for message aggregation
Input: $\mathcal{T} = \{e_1, e_2, \ldots, e_N\}$, a trace file with N events
Outputs: $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, n sets of aggregatable send events

```
1.  n = 0;
2.  for i = N downto 1 do begin        // Pre-processing
3.      if (e_i is not send event) then continue;
4.      for j = i − 1 downto 1 do begin
5.          if (e_j → e_i) then break;
6.          Swap event e_j with event e_{j+1};
7.      end
8.  end
9.  i = 1;
10. while (i < N) do begin             // Event classification
11.     if (e_i is not send event) then continue;
12.     n = n + 1;
13.     c_n = e_i; A_n = φ;
14.     while (i < N) do begin
15.         i = i + 1;
16.         if (A_n → e_i) then break;
17.         if (e_i is not send event) then continue;
18.         Add event e_i to set A_n;
19.     end
20. end
```

Figure 5. Search algorithm for message aggregation.

3.3 Search Algorithm for Message Aggregation

Our algorithm for message aggregation classifies the send events in a trace file into groups of send events such that all events in the same group can be aggregated into one message send.

Figure 5 describes the algorithm, which requires trace file $\mathcal{T} = \{e_1, e_2, \ldots, e_N\}$, and then returns n sets of send events $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n$, where $\mathcal{A}_m$ $(1 \leq m \leq n)$ is a group of send events that can be aggregated with each other and n is the number of classified groups.

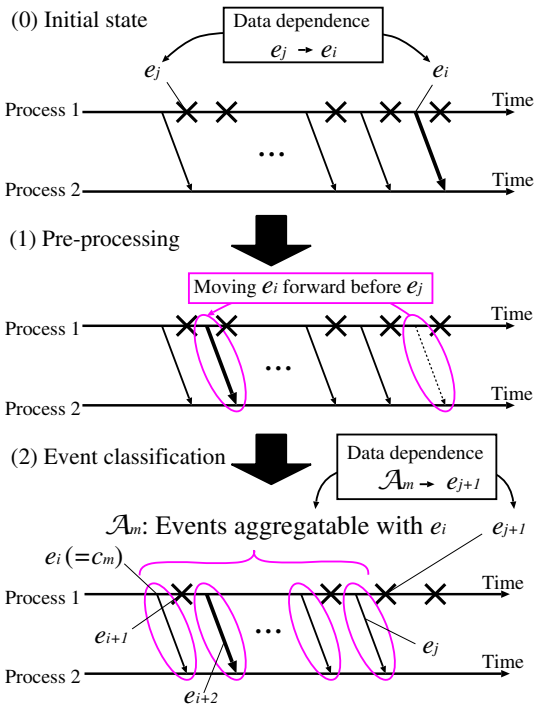

Figure 6. Search process for message aggregation.

The algorithm mainly consists of two processing stages. In the first stage, it alters the execution order of send events in order to aggregate more events. After this alternation, the algorithm generates a pre-processed trace file in which send events are moved forward as long as data dependences are kept. In the succeeding stage, the algorithm classifies aggregatable send events from the head to the tail of the pre-processed trace file.

Figure 6 shows a search process for message aggregation. For send event e_i $(= c_m)$, the algorithm computes a sequence of events $e_{i+1}, e_{i+2}, \ldots, e_j$ such that $(i < j) \wedge (\mathcal{A}_m \rightarrow e_{j+1}) \wedge (\forall e_s \in \mathcal{A}_m, \forall k\ (s < k \leq j), \neg(e_s \rightarrow e_k))$, where $\mathcal{A}_m$ $(1 \leq m \leq n)$ is all of the events sending to the same destination in $e_{i+1}, e_{i+2}, \ldots, e_j$. To compute this, it performs forward search from c_m until finding event e such that $\mathcal{A}_m \rightarrow e$.

During the event classification stage, the algorithm investigates whether the occurrence of send events can be delayed. Therefore, all events in $\mathcal{A}_m$ must be aggregated with the last occurred event $t \in \mathcal{A}_m$, where $1 \leq m \leq n$. For example, all events in $\mathcal{A}_m$ presented in Figure 6 must be aggregated with event e_j. Note here that the pre-processing stage enables event e_{i+2} to be grouped into $\mathcal{A}_m$. Without the pre-processing stage, $\mathcal{A}_m \rightarrow e_{j+1}$ prevents adding event e_{i+2} to $\mathcal{A}_m$, because e_{i+2} is originally occurred after e_{j+1}.

4 Case Studies

In order to demonstrate the usefulness of Gordini, we applied Gordini to two sophisticated MPI programs [1, 20] that won the first prize in NEC Cenju-3 category of Parallel Software Contest (PSC) held in 1995 [14] and in 1996 [15].

We tried to improve their performance on an NEC Cenju-3 [10] and an NEC Cenju-4 [9] using trace files generated on a cluster of PCs. The reason why we use our cluster for trace file generation is simply due to the limitation on usage of the storage available on the Cenju-3 and Cenju-4.

Note here that using trace files for different platforms is an uncritical problem for regular programs, which show the same behavior for different inputs. This benefit comes from our data dependence analysis approach that requires only the order of occurred events.

Our cluster has a total of 16 nodes with Pentium II processors running at 450MHz. These nodes are interconnected by a Myrinet switch [3] yielding a full-bandwidth of 1.28 Gb/s. The MPI implementation used for experiments is MPICH [6], a scalable implementation available on most of distributed memory multiprocessor systems.

4.1 Case 1: Performance Improvement by Communication–Computation Overlap

The first program [20] solves a system of linear equations, $Ax = b$, by using Gaussian elimination, where A represents a known dense matrix of size $n \times n$, x represents the required solution vector, and b represents a known vector of size n. The source code is approximately in 5000 lines. In the contest, it solves several equations ranging from $n = 1000$ to 2000 on a 64-node Cenju-3.

We instrumented all statements in the kernel of the code, including MPI routines, assignments, and memory copy primitives, but excluded uncritical statements, for example, assignments for array initialization. Then, we executed the instrumented program with $n = 256$ on $p = 4$, where p is the number of processors. The trace files are 70 MB in size and contains approximately 1.2 million events.

Table 2 summarizes search results obtained by Gordini after about 20 minutes search on a Pentium II 450MHz computer. Gordini indicates that communication–computation overlap and message aggregation can be applied to this program, however, collective communication is inapplicable.

According to these results, we investigated if the located places can easily be modified, and then decided to apply communication–computation overlap to two places. The remaining places are left as they are, because applying the techniques to these places is not easy due to the structure of the source code. For example, Gordini points out that several communication routines can be aggregated, however,

Table 2. Search results obtained by Gordini. Trace files are generated on 4 processors. E, S, and T represent the number of located events in trace files, that of located MPI routines in source code, and the time in minutes required for search, respectively.

Optimization technique	Gaussian $n = 256$			DFT $m = 25\ 600$		
	Results		Time	Results		Time
	E	S	T	E	S	T
Comm.–Comput. overlap	922	14	22	109	22	5
Message aggregation	244	6	20	12	2	2
Collective comm.	0	0	20	0	0	1

Table 3. Execution time for Gaussian elimination program on Cenju-4, where p represents the number of processors.

Problem size n	$p = 4$			$p = 16$		
	Execution time (s)		Speedup (%)	Execution time (s)		Speedup (%)
	Before t_1	After t_2	σ	Before t_1	After t_2	σ
128	0.014	0.014	0.0	0.023	0.023	0.0
256	0.054	0.047	12.9	0.052	0.052	0.0
512	0.351	0.240	31.6	0.152	0.135	11.2
1024	3.367	1.918	43.0	0.846	0.577	31.8
2048	29.454	15.783	46.4	7.149	4.084	42.9
4096	234.418	121.515	48.2	60.114	31.748	47.2

they are called from different loops placed in different files. We also prevent modifying the places with short execution time, because such places are small bottlenecks, which possibly give little improvement on the overall performance.

Table 3 shows t_1, t_2, and $\sigma = (t_1 - t_2)/t_1$, where t_1 and t_2 are the execution times before and after program modification, respectively, and σ is the speedup on execution time. Here, the execution time is measured for Gaussian elimination and backward substitution on the Cenju-4. Because the original program is modified after the software contest, we failed to measure its execution time on the Cenju-3 due to runtime error.

Although there is no improvement when $n = 128$, speedup σ increases with problem size n, so that t_1 reduces approximately in half when $n = 4096$. Thus, we obtain better improvement as n increases. This better improvement is derived from the modified places that emerge as larger bottlenecks with the increase of n. Actually, by measuring ρ, the accounting rate of the time for the modified places to the overall time t_1 on $p = 16$, we found that ρ increases from 7.6% to 99.5% when increasing n from 256 to 4096. Thus, this program has a performance characteristic that enables reducing more execution time as problem size increases.

4.2 Case 2: Performance Improvement by Message Aggregation

We also applied Gordini to a complex DFT program [1] that won the first prize in PSC'96. In this contest, it solves a problem with size $m = 3\ 276\ 800$ on a 128-node Cenju-3. The program is approximately in 1000 lines.

As we did in Section 4.1, we instrumented only the kernel of this program by using mpi2g. We then generated a trace file for $m = 25\ 600$ and $p = 4$.

As shown in Table 2, Gordini points out that communication–computation overlap and message aggregation can be applied to the program. According to these results, we modified all the located places: 16 places for communication–computation overlap and two places for message aggregation. The remaining places are small bottlenecks, so that we disregarded them. Note here that message aggregation was applied to collective (gather and broadcast) communication routines.

Table 4 shows t_1, t_{ovl}, t_{agg}, t_2, and speedup $\sigma = (t_1 - t_2)/t_1$, where t_1 is the execution time before program modification, t_{ovl}, t_{agg}, and t_2 are the execution time after applying communication–computation overlap, after applying message aggregation, after applying both, respectively. Note here that the problem size of $m = 3\,276\,800$ on $p = 16$ is unmeasured due to physical memory exhaustion.

When $p = 128$ and $m = 819\ 200$, message aggregation gives shorter execution time than communication–computation overlap. In contrast, the most effective technique when $p = 32$ and $m = 3\,276\,800$ is communication–computation overlap rather than message aggregation.

This can be explained as follows. The cost for collective communications increases with p, because at least $\log p$ communication steps are required to gather (or broadcast) data. At each step, receiver processors need to wait for sender processors. Therefore, applying message aggregation to collective communications reduces the occurrence of this waiting operation, realizing shorter waiting time. Thus, message aggregation further reduces the execution time as p increases. Furthermore, increasing p with fixed m decreases the amount of computations per processor, so that the accounting rate of communication time to overall time becomes larger while that of computation time becomes smaller. In this situation, reducing communication time is more effective than masking it with computation time, so that message aggregation gives better improvement than communication–computation overlap.

On the other hand, decreasing p with fixed m increases the amount of computations per processor, so that communication–computation overlap is expected to yield better improvement. Actually, t_1 and t_{ovl} when $m = 819\,200$ indicate that it reduces the execution time by 0.022 s on $p = 16$, however, it fails to reduce it on $p = 128$.

Proceedings of the 2004 International Conference on Parallel Processing Workshops (ICPPW'04)
1530-2016/04 $20.00 © 2004 **IEEE**

Table 4. Execution time for complex discrete Fourier transform (DFT) program on Cenju-3, where p represents the number of processors. t_{ovl}, t_{agg}, and t_2 are the execution time after applying communication–computation overlap, after applying message aggregation, after applying both, respectively.

p	$m = 819\,200$					$m = 3\,276\,800$				
	Execution time (s)				Speedup (%)	Execution time (s)				Speedup (%)
	Before	After				Before	After			
	t_1	t_{ovl}	t_{agg}	t_2	σ	t_1	t_{ovl}	t_{agg}	t_2	σ
16	1.565	1.543	1.560	1.538	1.73	—	—	—	—	—
32	0.877	0.865	0.865	0.855	2.51	3.990	3.878	3.972	3.874	2.91
64	0.450	0.447	0.440	0.438	2.67	2.021	1.972	1.974	1.927	4.65
128	0.213	0.213	0.193	0.193	9.39	1.030	1.011	0.972	0.965	6.31

5 Discussion

5.1 Usefulness of Gordini

Gordini focuses on the fact that communication, as well as load imbalance, probably tends to be sources of performance bottlenecks on current distributed memory multiprocessor systems. Therefore, the performance of programs written by MPI beginners can probably be improved by the three search functions of Gordini, because such programs usually take little care to performance.

In contrast, MPI experts carefully write programs to achieve higher performance. However, our case studies show that Gordini also can improve the performance of sophisticated programs. This improvement indicates that, even MPI experts can leave performance bottlenecks unaddressed, because earlier tools focus only on statistical performance data such as the execution time and the amount of messages. Such statistical data never makes developers be certain whether all bottlenecks are already eliminated. On the other hand, Gordini focuses on data dependence so that locates every place where the three optimization techniques can be applied. Thus, the advantage of Gordini is to enable developers to prevent leaving performance bottlenecks from the viewpoint of data dependence analysis.

When using earlier tools, which perform statistical analysis on performance data, it is a time consuming task to locate the places where communication–computation overlap is applicable, because this technique masks communication time with computation time as long as data dependence is kept. Such statistical tools do not consider data dependence, so that developers have to search overlappable statements in the source code. In contrast, Gordini directly locates such places in the source code, so that it frees developers from searching them. However, developers must investigate whether they actually can overlap the located places, as described later in Section 6.

The statistical analysis approach does not effectively work for programs where all of statements spend uniformly the same execution time, because it probably present uniform graphical views for such programs, so that developers are unable to locate performance bottlenecks. In the worst case, they may believe that there is no performance bottleneck. Gordini can deal with such programs because it focuses on data dependence rather than statistical data.

However, our data dependence analysis locates performance bottlenecks without regarding the execution time. Therefore, it can direct developers to the places with short execution time, where only little improvement can be achieved. Actually, in Section 4.1, we found that there is almost no improvement on execution time when ρ is a small value ($\rho = 7.6\%$). A similar situation occurs in Section 4.2, which shows less improvement due to short communication time. Therefore, we think that the combined use of data dependence and statistical data analyses is essential to efficiently improve the performance of parallel programs. For example, developers can locate performance bottlenecks by data dependence analysis, and then they can give priority to the located bottlenecks by sorting their execution time through the use of statistical data analysis.

5.2 Automatic Search Approach

Gordini frees developers from performance bottleneck search. This capability classifies performance data into two groups, useful data relating to performance bottlenecks and the other remaining data, so that enables presenting only useful data to developers. For example, for the Gaussian elimination program presented in Section 4.1, Gordini shows only 20 lines of the source code, 14 lines for communication–computation overlap, and 6 lines for message aggregation, from trace files containing 1.2 million events. Thus, automatic search approach is necessary to present less but useful performance information to developers.

Because the current implementation of Gordini holds the

Proceedings of the 2004 International Conference on Parallel Processing Workshops (ICPPW'04)
1530-2016/04 $20.00 © 2004 **IEEE**

entire of trace files in order to search performance bottlenecks, it has a limitation on the size of trace files. For example, a computer with 2 GB of main memory can deal with the maximum of 22 million events. However, this limitation will be eliminated because the algorithms presented in Section 3 requires a portion of trace files so that allows the on-the-fly computation of bottleneck search.

Generally, the size of trace files increases with that of problems and the number of processes. Therefore, satisfying requirement R1 presented in Section 1 is essential to ensure the performance improvement of parallel programs.

5.3 Clock Synchronization

Any tool based on trace files should consider the timing gap among processors' clocks because this gap causes discrepancies, for example, the occurrence time of a receive event is prior to that of a matching send event. Although MPI provides a wall clock mechanism (`MPI_Wtime`), it is not precise enough for event based tools where an event corresponds to an execution of a statement. This timing gap is a trivial issue for Gordini, because its data dependence analysis requires only the occurrence order of events.

However, if a wall clock mechanism removes this gap in the future, our analysis can be augmented by exploiting the occurrence time of events. For example, defining a wait event from the occurrence of a send event and that of a matching receive event may realize more useful analysis based on wait events.

6 Related Work

Gordini currently needs trace files generated by program execution. In contrast to this postmortem approach, Watanabe and Yuasa [21] employ static approach for communication optimization in a data parallel compiler. They present a compilation technique that realizes communication–computation overlap without program execution. Their technique realizes communication–computation overlap by moving assignments, however, it assumes that overlappable assignments and communications are placed in the same block, so that it is unable to overlap communication with computation across different blocks. On the other hand, Gordini is based on postmortem analysis, which is independent of the structure of the source code. Overlapping communication with more assignments is essential to yield higher performance, because communication routines generally require more execution time than assignments do.

Other data parallel compilers [4, 7, 16, 19] also perform communication optimization including message aggregation and collective communication. However, most of them focus on a single nested loop because optimization across different loops is a complex and expensive problem. To the best of our knowledge, only Hall et al. [7] presents a strategy for interprocedural optimization but they manually applies optimizations according to the strategy, because they have not implemented interprocedural optimizations in their compiler. Thus, most of static approaches are based on local optimizations, where optimizations are restricted to a single block/loop in the source code, mainly due to the complexity and expense of global optimizations such as interprocedural optimizations.

Gordini currently considers data dependence but no control dependence. Therefore, developers have to investigate the located places whether they are actually overlappable. For example, suppose that Gordini points out that statement S1 is overlappable with communication routine C. If the program is written as if-S1-else-S2, then developers have to check if S2 is also overlappable with routine C. If there is data dependence between S2 and C, developers are required further to check if the branch always executes S1. Watanabe and Yuasa address this problem by restricting the search space into a block, where no control dependence exists.

Summarizing the above discussions, postmortem approach locates more overlappable computations than static approach does, so that yields higher performance. However, it requires developers to investigate if the located bottlenecks have control dependence, while static approach guarantees valid optimizations by means of rigorous analysis. Therefore, postmortem approach is effective for developers to give a helpful advice for deterministic applications that show regular behavior in every program execution.

7 Conclusions

We have presented the design and implementation of Gordini, which is capable of locating places in the source code where typical optimization techniques can be applied. Gordini performs data dependence analysis on trace files and supports three techniques for communication bottlenecks: communication–computation overlap, message aggregation, and collective communication. Gordini guides developers to performance bottlenecks in the source code and presents a specific technique for their elimination.

In case studies, Gordini successfully assists us in improving the performance of two sophisticated MPI programs that won the first prize of parallel software contests. Therefore, we believe that our automatic search approach based on data dependence analysis is useful to improve the performance of message passing parallel programs.

In particular, our approach is effective for MPI programs with the following three characteristics.

Regular behavior: Postmortem approach is effective for deterministic applications that show regular behavior in every program execution.

Proceedings of the 2004 International Conference on Parallel Processing Workshops (ICPPW'04)
1530-2016/04 $20.00 © 2004 **IEEE**

Communication bottlenecks: Because we currently focus on communication bottlenecks, our approach can improve applications with communication bottlenecks but needs other techniques to improve applications with computation or input/output bottlenecks.

Global bottlenecks: In contrast to the above two requirements, this clarifies the difference between the effectiveness of postmortem approach and that of static approach. Postmortem approach is effective for global bottlenecks, namely performance bottlenecks across different blocks, as well as local bottlenecks, namely performance bottlenecks within a single block.

Future work includes the development of performance prediction method for guiding developers to places only with significant improvement. This method is essential to prevent misleading developers into wrong modification with performance loss. Furthermore, coupling Gordini with static techniques could provide more rigorous search. For example, control dependence can be addressed by performing rigorous data flow analysis at the instrumentation time.

Acknowledgments

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

References

[1] T. Araki. Algorithm for complex discrete Fourier transform (DFT), June 1996. `http://phase.hpcc.jp/phase/sacsis/PSC96/archives/winners/araki.html` (In Japanese).

[2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[3] 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*, 15(1):29–36, Feb. 1995.

[4] Z. Bozkus, A. N. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *J. Parallel and Distributed Computing*, 21(1):15–26, Apr. 1994.

[5] 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/`.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[7] M. W. Hall, S. Hiranandani, K. Kennedy, and C.-W. Tseng. Interprocedural compilation of Fortran D. *J. Parallel and Distributed Computing*, 38(2):114–129, Nov. 1996.

[8] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, Sept. 1991.

[9] T. Hosomi, Y. Kanoh, M. Nakamura, and T. Hirose. A DSM architecture for a parallel computer Cenju-4. In *Proc. 6th Int'l Symp. High-Performance Computer Architecture (HPCA'00)*, pages 287–300, Jan. 2000.

[10] N. Koike. NEC Cenju-3: A microprocessor-based parallel computer. In *Proc. 8th Int'l Symp. Parallel Processing Conf. (IPPS'94)*, pages 396–401, Apr. 1994.

[11] Message Passing Interface Forum. MPI: A message-passing interface standard. *Int'l J. Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, 1994.

[12] 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*, 28(11):37–46, Nov. 1995.

[13] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *J. Supercomputing*, 12(1):69–80, Jan. 1996.

[14] Parallel Software Contest (PSC'95), 1995. `http://phase.hpcc.jp/phase/sacsis/PSC95/`.

[15] Parallel Software Contest (PSC'96), 1996. `http://phase.hpcc.jp/phase/sacsis/PSC96/`.

[16] W. Pugh and E. Rosser. Iteration space slicing and its application to communication optimization. In *Proc. 11th ACM Int'l Conf. Supercomputing (ICS'97)*, pages 221–228, July 1997.

[17] M. J. Quinn and P. J. Hatcher. On the utility of communication–computation overlap in data-parallel programs. *J. Parallel and Distributed Computing*, 33(2):197–204, Mar. 1996.

[18] S. Shende and A. D. Malony. Integration and application of TAU in parallel Java environments. *Concurrency and Computation: Practice and Experience*, 15(3/5):29–39, Mar. 2003.

[19] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *Proc. 9th ACM Int'l Conf. Supercomputing (ICS'95)*, pages 424–433, July 1995.

[20] O. Tatebe. LU decomposition on distributed memory machines. In *IPSJ SIG Notes*, 95-HPC-57, pages 55–60, Aug. 1995. `http://phase.hpcc.jp/people/tatebe/software/` (In Japanese).

[21] N. Watanabe and T. Yuasa. A code motion technique for communication optimization of data-parallel languages. *Trans. IPSJ*, 40(3):1257–1266, Mar. 1999. (In Japanese).

[22] 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*, 25(4):429–461, Apr. 1995.

[23] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *Int'l J. High Performance Computing Applications*, 13(2):277–288, June 1999.

Proceedings of the 2004 International Conference on Parallel Processing Workshops (ICPPW'04)
1530-2016/04 $20.00 © 2004 **IEEE**