# LIMITED DISTRIBUTION NOTICE
# (NOT TO BE REMOVED FROM DOCUMENT):

**This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communication and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).**

# Trace-Based Performance Analysis and Tuning for Distributed Parallel Applications

C. Eric Wu, Yew-Huey Liu, Caroline Benveniste, Chuen-Liang Chen*
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Wei-Hwan Chiang
IBM Power Parallel Systems
Neighborhood Rd.
Kingston, NY 12401

**Abstract**

A continuous stream of event data describing the progress of parallel program execution is required for trace-driven analysis, such as workload characterization, interactive debugging, or program visualization. Unfortunately, it is often the case that separate streams are produced independently by multiple processors in the system, and the logical order of events cannot be guaranteed due to discrepancy among local clocks. In this paper, we present our approach for timestamp consistency, and our techniques on performance analysis and tuning for distributed parallel applications. Our trace generation facility requires no modification on application source code, and is able to generate message passing and system events with minimal trace overhead. Trace-driven analysis tools are developed to extract useful information, such as the time stolen by other processes in each node, and the observed message passing time and local wait time for each message. We then present our optimization technique whose goal is to reduce the total elapsed time of the distributed parallel application.

**Keywords:** trace generation, distributed parallel systems, performance analysis and tuning, message passing time, local wait time

*On leave from Department of Computer Science and Information Engineering, National Taiwan University

# 1 Introduction

Distributed parallel systems have become pervasive in recent years. Unfortunately, programming distributed parallel systems can be a very involved and intricate task. This is because that each processor (or node) has its own local memory, and processors communicate with one another by exchanging messages. Non-local data references in such a system need to be converted into explicit and expensive send/receive routines. For a poorly written program, the communication overhead along with the serialization effects of the send and receive operations could destroy any benefits of distributed parallel execution.

One possible way of monitoring a distributed parallel application is to generate trace records while executing the program. The collected trace events are then used to analyze workload characterization and calculate statistics. With the help of analysis tools to obtain run-time information such as the observed message passing time and local wait time, the user will better understand the execution of the application program, and from this the user may be able to modify the program to improve performance. Trace driven techniques have been widely used for performance analysis. For parallel and distributed systems, trace event generation is essential for trace-driven analysis [3, 9, 11], timing simulation [17, 21], interactive debugging [13, 16], and program visualization [5, 10]. There are two basic requirements for trace generation: minimal trace overhead and controllable trace events. Trace overhead should be kept as small as possible. If trace overhead is large, the timestamp associated with each event may have been altered significantly, and the statistics obtained in post-processing may be meaningless. For example, the trace overhead for message passing events in [15] could be as high as 500+ % of the total elapsed time. Statistics generated from trace events using such a trace facility is hardly meaningful and may have little value for performance analysis.

In addition to efficiency, users may want to see message passing and system events in order to monitor

system behavior and/or tune program performance. Events could be enabled or disabled at the time of activating trace generation to provide controllable trace events. Therefore, our goal for trace generation is to generate controllable message passing and system events with minimal trace overhead.

There is one serious problem associated with performance analysis for distributed parallel applications, i.e., the clock synchronization problem [8]. The problem comes from the fact that trace events are produced independently by multiple processors in the distributed parallel system, and the logical order of events cannot be guaranteed due to discrepancy among local clocks.

An IBM SP1 system is used in our project. The IBM SP1 system provides a globally synchronized clock with three crystals, in which one is used as the master. This technique of triple modular redundancy (TMR) is used to guarantee a good global clock for driving the switch network. Each processor in the IBM SP1 system has an adapter which connects the processor to the switch network. The adapter includes a clock register driven by the master clock crystal. During system initialization the number of cycles required to set each clock register is calculated, and clock registers are set one by one so that their values match one another. Since all clock registers are incremented by the same crystal, they are synchronized as one global clock [19]. The existence of a global clock in IBM SP1 systems is very useful for performance analysis in general. We will describe how the global clock is used to avoid the well-known clock synchronization problem.

In this paper, we discuss our approach for timestamp consistency and our techniques on performance analysis and tuning for distributed parallel applications. The paper is organized as follows. Our trace generation facility for IBM SP1 systems is described in Section 2, along with the approach for timestamp consistency to avoid the well-known clock synchronization problem. Trace-driven analysis tools are described in Section 3. Using the statistics obtained from the analysis tools, we derive new techniques in Section 4 on performance tuning for distributed parallel applications. Concluding remarks are given

in Section 5.

# 2 Trace Generation and Timestamp Consistency

## 2.1 Trace Generation

An ideal trace generation facility should be able to generate trace events with minimum directives from the user. In other words, a user should not have to modify his/her code for generating trace events. Unlike many trace systems such as those in [2, 17], our AIX trace generation facility requires no modification of application source code for trace event generation. If the application source code is available, additional user markers provided by the trace facility can be inserted into the source code. This allows the user to mark specific portions of the program for performance analysis.

The AIX trace facility is able to capture a sequential flow of time-stamped events, providing a fine or coarse level of detail on message passing and system activities. The AIX operating system is instrumented to provide general visibility to system execution, and the same is true for message passing events, since message passing routines are also instrumented to provide detailed trace information. Care was taken in the design and implementation of the trace generation facility to make the collection of trace data efficient, so that system performance and flow would be minimally altered by activating trace generation. For example, the trace facility pins the data collection buffer in the main memory to reduce trace generation overhead, and the size of the data collection buffer can be specified by the user at the time of activating trace generation.

Trace generation is controlled by an environment variable TRACEOPT, which defines trace options such as the system and message passing events the user is interested in, the size of the data collection buffer pinned in the main memory, and the name for trace files. If the environment variable is not defined, the application will run without generating trace events. Possible system events include process dispatch,

page fault, I/O and file-oriented system calls such as read and write, etc. For message passing routines, we capture the beginning and end events for each basic communication routine such as send and receive. For group (collective) communication routines we not only capture the beginning and end events of the collective communication routine, but also the underlying point-to-point communication events. Thus, users and system programmers who are interested in the implementation and performance of collective communication routines have a systematic way to compare the performance of different algorithms or various implementations of the same algorithm for performance analysis and tuning.

The trace facility provides a way for users to turn on specific events for trace generation. For example, a user may be interested in message passing and process dispatch events. Other system events, such as page fault and I/O events, will not be generated if the user does not explicitly ask for them in the environment variable TRACEOPT. Thus, a user may choose which system and message passing events he/she wishes to see in the trace.

The cost of cutting a trace record is broken into two parts: the cost of testing whether the event is enabled and then calling the trace buffer insertion routine, and the cost of the trace buffer insertion routine. If a typical trace record has 3 words of data in addition to an event header (i.e. a hookword which identifies the event type and its record length) and a timestamp, the average cost of cutting a trace record is around 110 machine instructions. Thus, the trace generation facility is very efficient and adds only a few microseconds to the elapsed time for each trace event.

Due to the existence of a global clock in the IBM SP1 system, we decide to take advantage of it to avoid the clock synchronization problem. However, our experience shows that it is much more expensive to access the global clock than the local ones. This is because a local clock register is inside the processor and can be accessed in tens of nanoseconds (one machine instruction), while the global clock register is on the adapter and several microseconds are required, including software overhead, to access it. As the

result, we choose to access the the global clock periodically in each processor. Since message passing routines are all executed in user's space, message passing and periodic global clock events are generated in user's space.

A global clock event consists of both the global and local timestamps, and any other event has only a local timestamp to minimize trace overhead. Each processor generates global events periodically at its own pace during a period of trace generation, and the periodic global clock events are eventually used by analysis tools to adjust local timestamps in post analysis.

Table 1 shows the trace overhead for generating message passing events, including all point-to-point and collective communication events, for 7 distributed parallel programs.

Table 1: Trace generation overhead for message passing events

| Program | Elapsed Time (sec) | No. of Messages | Overhead (%) |
|---|---|---|---|
| cholesky | 28.77 | 2772 | 0.034 |
| hpc | 6.32 | 15675 | 1.241 |
| isort | 17.27 | 1134 | 0.001 |
| fft | 45.22 | 406 | 0.001 |
| ep | 55.32 | 14 | 0.000 |
| mg | 19.16 | 3334 | 0.044 |
| cg | 50.73 | 46176 | 0.292 |

The first program in Table 1 is an eight-node program for dense matrix column-Cholesky factorization, and the second is a four-node program for LU matrix decomposition. The rest 5 programs, Integer sort (isort), 3-D FFT (fft), Embarrassingly parallel (ep), Multigrid (mg), and Conjugate gradient (cg), are kernels in the NAS (Numerical Aerodynamic Simulation) benchmark suite [1], all with 8 nodes. The more messages a program has, the more message passing events the trace files contain. It can be seen in Table 1 that the trace overhead is less than 0.3 % in most cases. If low-level point-to-point message passing events are not traced within each collective communication routine, the trace overhead could be even smaller. On the other hand, if system events such as process dispatch are also collected, the

trace overhead will increase, depending on how many system events are enabled.

## 2.2 Timestamp Consistency

It is obvious that local timestamps alone do not ensure a consistent ordering of all events. Thus, many trace generation facilities are forced to do additional work to ensure consistent timestamps at the expense of increased trace overhead. For example, the *tracenode()* routine in PICL [4] synchronizes local clocks using a barrier synchronization. Lamport [8] developed a distributed algorithm for resource-sharing systems to extend the partial ordering to a consistent total ordering of all events by creating additional messages among processors: another form of barrier synchronization. Since barrier synchronization may take a long time, activating trace generation in such systems may have significant impact on the total elapsed time and other timing-sensitive program behavior.

We have periodic global clock events in the trace for post-analysis tools to adjust local timestamps. While periodic clock adjustments are usually smaller than the message latency, the difference between the global and local timestamps may be quite large if time daemons, such as *timed* and *xntpd* [14] which adjust local clocks periodically based on external sources, exist in the distributed parallel system. Unless all nodes are free-run with respect to their local clocks, periodic global clock events are not enough to guarantee a consistent ordering of all events, no matter how frequent the global clock is accessed by each node. Therefore, we need a safeguard to guarantee a consistent total ordering of all events.

A consistent total ordering of events is required to provide a basis for trace-driven analysis and program visualization of distributed parallel programs. Such a consistent ordering and correlation of events can be achieved by topological sorting that preserves the partial ordering of events.

Our trace generation facility generates all point-to-point message passing events under the control of TRACEOPT, even for collective communication routines. The relationship between a send and its corresponding receive is part of the fundamental ordering of all events. If a receive such as a blocking

*MP_WAIT()* in [6] is waiting for an unspecified message, the trace facility will generate information about the message eventually received, such as its source node ID, message type, and message size, so that the sends and receives can be easily matched by analysis tools.

A height-balanced AVL tree [7] is used to maintain the partial ordering of events. Each trace stream at most has one event in the tree, so that the number of elements in the tree is less than or equal to the number of trace streams at all times. Events in the tree are sorted based on timestamps, and the first global timestamp from each trace stream is used to determine the starting sequence of the trace streams. Fig. 1 provides an overview of the basic matching algorithm for a consistent total ordering of events.

```
Tree_Initialization();
while ( Get_Smallest_Timestamp_Event_From_Tree(&Node_ID, event) != NO_MORE_EVENT) {
        switch(Event_Type) {
        case End_Receive:
                if ( No_Msg_Info(Event_Dest, Event_Src, Msg_Type) )
                        Delete_From_Tree(Event_Dest);
                break;
        case Begin_Send:
                Attach_Msg_Info(Event_Dest, Event_Src, Msg_Type);
                if ( Is_Blocked(Event_Dest) ) {
                        if ( Waiting_For(Event_Dest, event) )
                                Insert_Into_Tree(Event_Dest);
                }
        default:
                Process(event);
                Get_Next_Event_From_Stream_To_Tree(Node_ID);
                break;
        }
}
```

Figure 1: Basic Matching Algorithm

The `Tree_Initialization()` routine gets one event from each trace stream and sort events based on timestamps. After the initialization phase the event with the smallest timestamp is deleted from the tree, and then processed before the next event from the same trace stream is inserted (using the

`Get_Next_Event_From_Stream_To_Tree()` routine). This preserves the partial ordering of events within each trace stream.

For message passing events, we preserve the send_before_receive logical ordering by matching events on the fly. The `Attach_Msg_Info()` routine attaches message information such as source node ID and message type to a list at the receiving node, and the `No_Msg_Info()` routine is used to see if there is an entry in the list for the message when an End_Receive event is processed. If an End_Receive event has been reached but the message is not yet sent by its source node, the End_Receive event at the receiving node is temporarily deleted from the tree by the `Delete_From_Tree()` routine. The End_Receive and all subsequent events at the receiving node are blocked until the node is waken up by the corresponding Begin_Send event.

When the corresponding Begin_Send event is processed, the status of receiving node is checked using the `Is_Blocked()` routine. If the receiving node is indeed waiting for this message (checked by the `Waiting_For()` routine), the End_Receive event is inserted back into the height-balanced tree with a new timestamp. The new timestamp is calculated by adding a delay to the timestamp of the Begin_Send event. This delay may be estimated by a simple network delay equation or a sophisticated network model. Since partial ordering of events and send_before_receive relationships are preserved in the basic matching algorithm, a consistent total ordering of events is achieved.

The algorithm works well regardless of whether or not a global clock exists. Therefore, the basic matching algorithm is used as the safeguard to ensure a consistent total ordering of all events. When periodic global clock events are used to adjust local timestamps, a low global-clock access rate minimizes trace overhead but increases the possibility for out_of_order message passing events. Our experience shows that an access rate of once every 400 milliseconds per node is good enough when combined with the basic matching algorithm in analysis tools.

# 3 Performance Analysis

In addition to common statistics such as the number of messages sent and received, our tools can tell us the time "stolen" by other processes in each node, the observed message passing time for each message, and the observed local wait time for each message received. In the following sub-sections we analyze trace files generated for a dense matrix column-Cholesky factorization program to illustrate our trace-driven analysis techniques for distributed parallel applications. Matrix (2400 by 2400) columns are partitioned into groups of 6 contiguous columns each in this distributed parallel program, and groups are wrap-mapped to 8 processors.

## 3.1 Computation Time versus Stolen Time

A computation interval is defined as a period between the end event of a communication routine and the beginning event of the next communication routine in a node, minus any time stolen by other processes. An application programmer can determine how well his/her code is written by checking the total computation time versus the total elapsed time. For example, if one knows the application is parallel in nature but the tool reports that only 50 % of the total elapsed time is used for computation in the trace, there must be room for further improvement in data partition, algorithm, or implementation. In addition, the time stolen by other processes such as system daemons may help the user understand why one particular run is much slower than other identical runs.

Fig. 2 shows the trace length, total computation time, and total computation time plus stolen time for each node, along with the total elapsed time for the Cholesky factorization program, which is only partially optimized. The trace elapsed time of the application on each node is slightly smaller than the total elapsed time, because the beginning and end of the traces on each node are staggered.

Fig. 2 shows that less than 40 % of the time in each node is actually spent on computation. The time

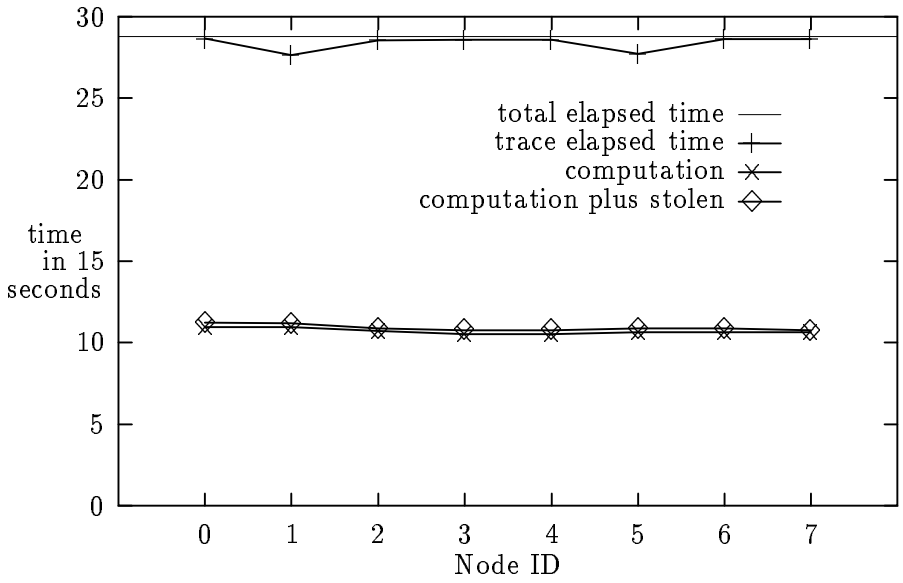


Figure 2: Computation time vs stolen time vs trace length

stolen by other processes is relatively small (about 0.2 seconds on average). The rest of the time is spent in communication routines for sending, receiving, or waiting for messages.

It is important to generate trace records for both message passing and system events. For example, the time stolen by other time-sharing processes such as system daemons in each node may have an additive effect on the total elapsed time. If a node is delayed by certain amount of time due to time-sharing activities in the node, all subsequent events in the node are delayed by that amount of time, and the ripple effect through message passing will delay events in other nodes as well. For the purpose of performance analysis and tuning, process dispatch events are critical for analyzing system and user program behavior. Without these system events, a user will not have enough information to determine why certain events or routines were slower than expected.

### 3.2 Communication Time and Histogram of Message Passing Events

Once the user knows how much time is spent in local computation as well as communication on each node, a more detailed histogram table for identifying dominant communication routines will be helpful. A communication interval is defined as a period between the beginning and the end events of a communication routine. Depending on the type (blocking vs non-blocking) of a communication routine and message size, a communication interval could be large or small. Our tool is capable of generating a histogram of message passing routines to show the total communication time as well as the average time spent in each communication routine.

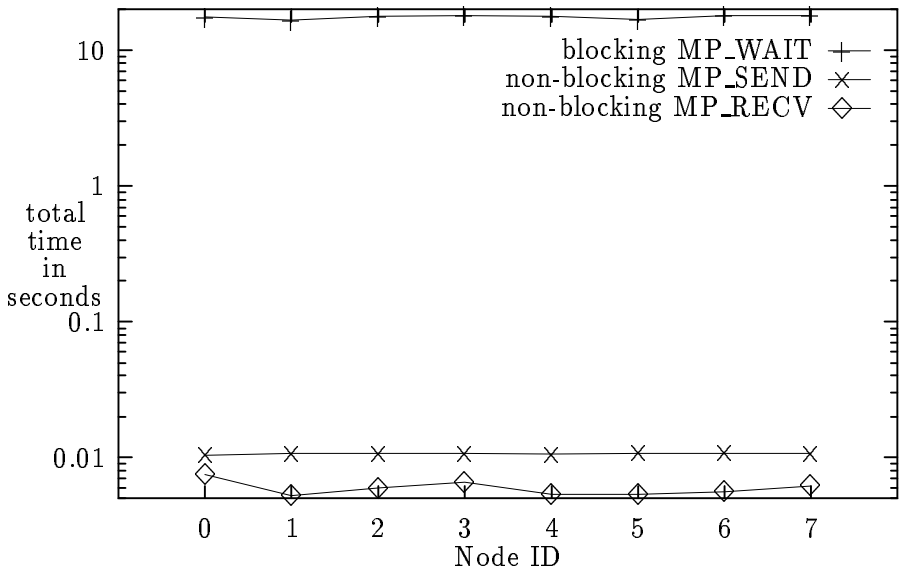


Figure 3: Three most time-consuming communication routines in the trace

Fig. 3 shows three of the most time-consuming communication routines, the blocking *MP_WAIT()*, non-blocking *MP_SEND()*, and non-blocking *MP_RECV()*, in the Cholesky factorization application. The non-blocking *MP_SEND()* routine calls the adapter to send out a message and returns, and the non-blocking *MP_RECV()* routine prepares a user buffer for the incoming message and returns immediately. The blocking *MP_WAIT()* routine is then used to make sure the buffer is cleared in the case of a non-

blocking *MP_SEND()*, or that the message is received in the case of a non-blocking *MP_RECV()*. Fig. 3 shows that processors spent most of their time in the blocking *MP_WAIT()* routine. The amount of time spent by each node in the blocking *MP_WAIT()* routine is about 17.5 seconds, while the time for computation and the time stolen by other processes are are much smaller, roughly 10.7 and 0.2 seconds, respectively. These observations provide some hints as to how one might optimize the code.

### 3.3 Observed Message Passing Time and Local Wait Time

A user may want to have additional information about the communication portion of an application program, such as how long a node waited for a specific message and what was the observed message passing time for that message. These two pieces of run-time data are perhaps the most important metrics for optimizing message passing applications.

Fig. 4 shows the observed message passing time and local wait time for a message. The observed message passing time of a message in a program is defined as the elapsed time from the beginning of the send at the source node to the end of the receive at the destination node. It is actually an upper bound for the actual message passing time, which is defined as the elapsed time from the beginning of the send to the point at which it arrives completely at the destination node. This difference exists because the message may have arrived at the destination node, but the application at the node may not yet have called the communication routine to receive the message.

The local wait time for a message is defined as the interval between the time the destination node starts waiting for the message and the time the message is received. Therefore, an "efficient" message passing program running on a machine with infinite buffers should have a local wait time close to zero for each message, and the observed message passing time of each message should be close to its theoretical lower bound, which will be addressed in Section 4.

Fig. 5 shows both the observed message passing time and local wait time for the first 25 messages

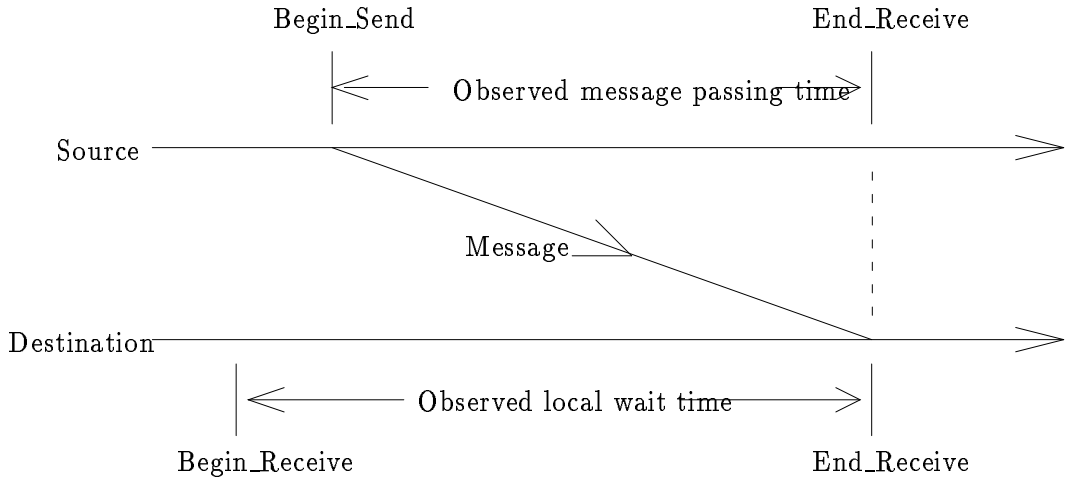


Figure 4: Observed message passing time and local wait time

received in the trace. Since the message sizes are the same, one might expect the observed message passing times and local wait times for these messages to be similar. As can be seen in Fig. 5, the first message (from node 0 to node 1) has a much larger observed message passing time than all other messages. This is caused by a much slower start of the second node (node 1). The local wait time for most of the messages is similar, with some exceptions for messages sent by node 0, also at the beginning of the trace.

Fig. 6 shows the space-time diagram for messages received in the first 1.7 seconds of the trace, along with the starting time of the application in each node. End events for non-blocking sends are not shown. It becomes apparent from Fig. 6 that the application started in node 0, and node 1 and node 5 did not begin until over 1 second had elapsed. This is caused by an overloaded file server. Thus, the observed message passing time of the first message (from node 0 to node 1) is almost 1 second. Due to the implementation of the application, node 0 does not start sending messages to other nodes until its message buffer is cleared. That in turn causes large local wait times in other nodes. Several other re-runs with and without trace generation did not show such delays.

| Src | Dest | Msg_Size | Msg_Pass_Time | Local_Wait_Time | Recv'd_At |
|---|---|---|---|---|---|
| 0 | 1 | 115248 | 0.995851019 | 0.014240000 | 1.204534795 |
| 0 | 2 | 115248 | 0.015033037 | 0.905424640 | 1.219457485 |
| 0 | 3 | 115248 | 0.014847159 | 0.931277312 | 1.234130871 |
| 0 | 4 | 115248 | 0.014910048 | 0.944049408 | 1.248937312 |
| 0 | 5 | 115248 | 0.014840524 | 0.070093824 | 1.263599564 |
| 0 | 6 | 115248 | 0.015006559 | 0.968198912 | 1.278438239 |
| 0 | 7 | 115248 | 0.014924963 | 0.989429504 | 1.293256355 |
| 1 | 2 | 115248 | 0.013546690 | 0.057693184 | 1.345374669 |
| 1 | 3 | 115248 | 0.013445036 | 0.057644288 | 1.358727863 |
| 1 | 4 | 115248 | 0.013628245 | 0.056960768 | 1.372261728 |
| 1 | 5 | 115248 | 0.013504449 | 0.054974464 | 1.385669324 |
| 1 | 6 | 115248 | 0.013458516 | 0.052996608 | 1.398970463 |
| 1 | 7 | 115248 | 0.013497240 | 0.052764416 | 1.412414115 |
| 1 | 0 | 115248 | 0.013772277 | 0.068950528 | 1.426084864 |
| 2 | 3 | 115248 | 0.013635306 | 0.057462784 | 1.482414007 |
| 2 | 4 | 115248 | 0.014172563 | 0.058785280 | 1.496499296 |
| 2 | 5 | 115248 | 0.013423615 | 0.058065920 | 1.509765068 |
| 2 | 6 | 115248 | 0.013453202 | 0.057829632 | 1.523136351 |
| 2 | 7 | 115248 | 0.013514710 | 0.058752000 | 1.536564131 |
| 2 | 0 | 115248 | 0.013446707 | 0.055913984 | 1.549916416 |
| 2 | 1 | 115248 | 0.013843518 | 0.073859584 | 1.563675403 |
| 3 | 4 | 115248 | 0.013609129 | 0.056101376 | 1.618568800 |
| 3 | 5 | 115248 | 0.013453333 | 0.055659520 | 1.631935692 |
| 3 | 6 | 115248 | 0.013488040 | 0.055734528 | 1.645331039 |
| 3 | 7 | 115248 | 0.013722604 | 0.057058560 | 1.658954659 |
| ..... | | | | | |

Figure 5: Observed message passing time and local wait time in the trace

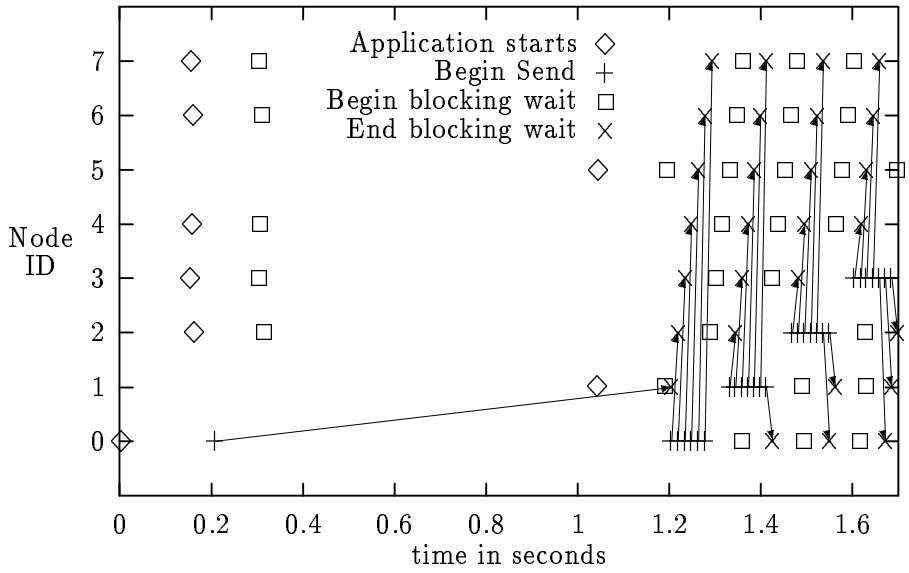


Figure 6: Spacetime diagram for message passing events

It is important to provide users with the observed message passing time and local wait time for each message, especially if performance tuning is one of the objectives. If a message is received with larger-than-expected observed message passing time, it suggests that repositioning message passing routines in the source code might improve the performance. If the receiving node has other active processes during a run, process dispatch events will show up in the trace (if it is enabled), and by simply re-running the application, the user may obtain very different results. Another possibility that may explain the large observed message passing time is that the receiving node issued the receive too late, possibly because it was tied up with local computation. If this is the case, the user may come up with new data partitioning scheme or implementation to minimize the observed message passing time. Similarly, if a message is received with a large local wait time, the node may have called the receive routine too early.

With the help of analysis tools to obtain the observed message passing time and local wait time, the user will better understand the execution of the application program, and from this he/she may be able to modify it to improve performance. In the above example, the observed message passing time is not

far from the theoretical message passing time for messages of that size. However, the local wait time is consistently several times larger than the observed message passing time except during the initial stage. In addition, Fig. 6 shows that the amount of computation associated with each message is about 0.07 seconds, which is much larger than the observed message passing time. The large local wait time, therefore, is actually caused by the delay of sending multiple messages. It can be seen in Fig. 6 that it takes about 0.09 second to send a group of 7 messages. Although the messages were sent using non-blocking routines, the limit on the adapter bandwidth and the sequential implementation of the code forces the node to send messages one at a time. Therefore, this portion of the code is the performance bottleneck and deserves more attention for possible improvement.

### 3.4 Other Useful Observations

There are many other interesting things a user may want to know about his/her message passing applications. For example, the communication patterns may be obtained using the tool to generate summaries for messages sent and received. The observed network traffic over time in bytes or number of messages may be obtained by examing the time at which each message was sent or received. In addition, the histogram tables of message sizes along with observed network bandwidth, observed message passing time, local wait time, and network traffic in bytes or number of messages can all help a user understand his/her application better.

## 4 Performance Tuning

The theoretical lower bound for the message passing time of a message is a function of the message size. To quantify the relationship between the message passing time and message size, a histogram for messages of various sizes may be generated using our analysis tool and the average observed message

passing time for each message group is also reported. The histogram table obtained from a carefully written program can be used as a simple network delay model for estimating the latency and point-to-point network bandwidth.

Given the message latency $A$ in seconds and point-to-point bandwidth $B$ in bytes/second, a simple theoretical lower bound for message passing time is given by $f(x) = A + b \times x$, where $b = 1/B$ and $x$ is the message size in bytes. A user may obtain the parameters $A$ and $B$ from the manufacturer of the distributed parallel system or a carefully written program. The observed message passing time obtained from trace files can then be compared to this theoretical lower bound.

The information about message passing time and local wait time is very useful for performance tuning, especially for synchronous applications in which all nodes start communicating at about the same time [12]. Given the amount of computation $m$ in seconds per byte associated with an incoming message, the total amount of computation required for the message is $g(x) = m \times x$. If the computation and communication intervals are completely overlapped, one will be able to improve the performance of an application. Note that $m$ may vary from one application to another, and it may also vary from one message to another in the same application.

To simplify the analysis model for distributed parallel systems, one may assume that $m$ is a constant for all messages in a given distributed parallel application. One can also assume that it takes very little time for a transmitting node to send a message, perhaps due to the existence of a communication co-processor, and that there is no network congestion. In this case, the best performance is obtained if $g(x_i) = f(x_j)$ for every pair of consecutive messages received in each node, where $x_i$ and $x_j$ are the message sizes. For two consecutive messages of the same size in applications with $m \leq b$, the local wait time will never approach zero since $f(x) > g(x)$ if $b \geq m$.

Fig. 7(a) shows that a large local wait time would be observed in the case of $b \geq m$. This is because the

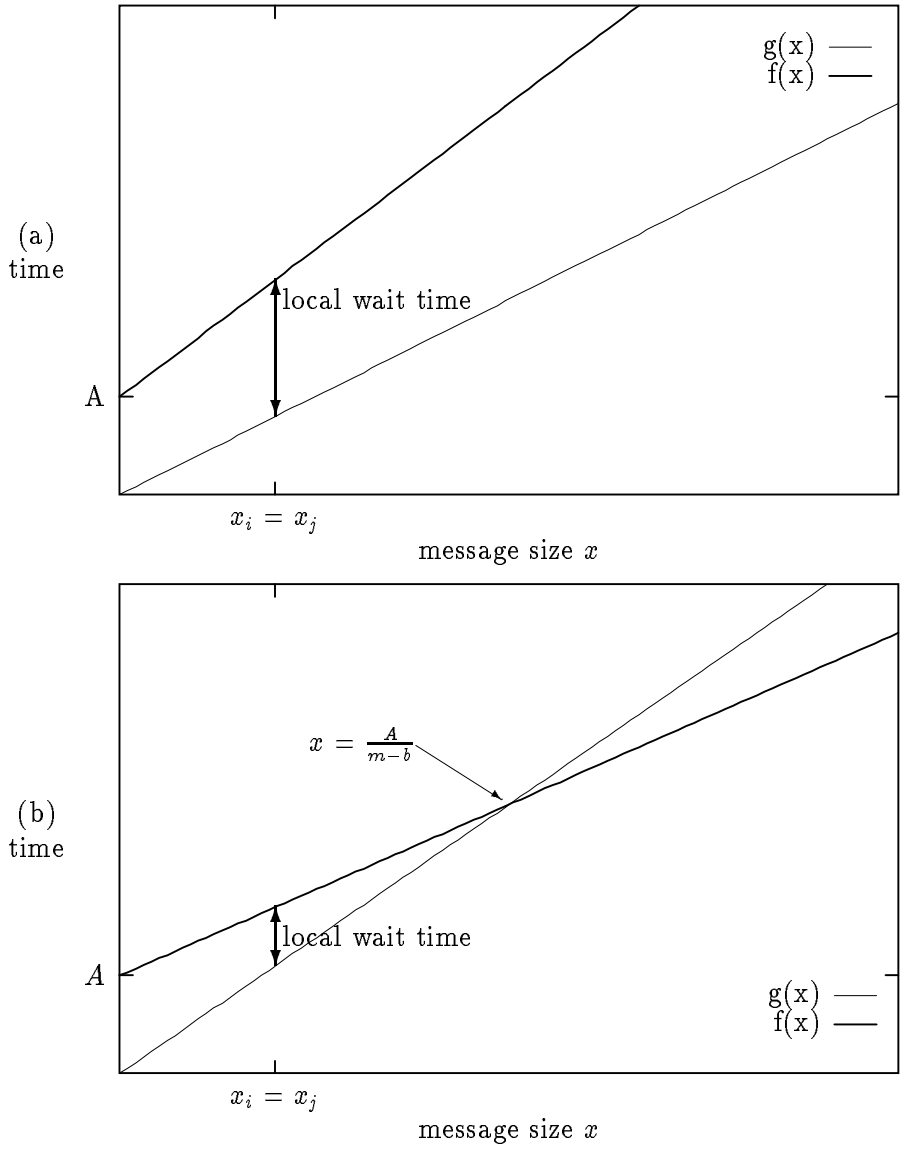


Figure 7: Message passing time $f(x)$ versus local computation $g(x)$: (a) $b \geq m$; and (b) $b < m$

amount of computation associated with each message is little compared with the message passing time. Thus, each processor has to wait for the next incoming message after local computation associated with the previous message is finished.

If $m > b$ and the message size $x < A/(m-b)$, we expect to see a large local wait time as well. There has been several research papers discussing message consolidation under this condition [18, 20], but none provides users with a tool for performance tuning. Fig. 7(b) shows that the local wait time is reduced as the message size $x$ approaches $A/(m-b)$. If $x = A/(m-b)$, the computation and communication intervals are completely overlapped, local wait times are eliminated, and the best performance will be obtained.

The parameter $m$ for a given message can be obtained from the trace, since the message size (in bytes) and the length of the associated computation (in seconds) are both known in the trace. Therefore, our tool provides a systematic approach to do performance tuning for distributed parallel applications.

# 5 Summary

Timestamp consistency is the key on performance analysis for distributed parallel applications. In this paper we have described our approach for timestamp consistency and our highly efficient trace facility as the basis for performance analysis and tuning of distributed parallel applications. Our trace facility is able to generate message passing and system events with minimal overhead, and users do not have to modify application source code to generate trace events.

We have also presented our techniques on performance analysis and tuning for distributed parallel applications. Due to the ability of generating system events, we are able to report the time stolen by other processes in each node. Thus, a user can see how the total elapsed time is partitioned, not only for each communication routine used in the application, but also for the message passing application in

the whole distributed parallel system.

The observed message passing time and local wait time are two important pieces of run-time data associated with each message. We are able to obtain these and other useful information for performance tuning.

Other useful tools for trace-driven analysis include tools for event visualization and identifying deadlocks, interactive debuggers with breakpoints possibly in the trace files, and timing simulators for predicting application performance on future distributed parallel systems.

**Acknowledgements**

We would like to thank P. Hochschild for developing the EUI-H communication layer. We are grateful to the Advanced Algorithm Group at IBM Almaden Research Center for developing the collective communication library, and to Y. Hsu and M. Snir for their constant support. In addition, we wish to thank all members of the Parallel System Organization Group at IBM T. J. Watson Research Center, for many fruitful discussions and suggestions.

## References


[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon, The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, vol. 5, pp. 63 - 73, 1991.

[2] H. Davis, S. Goldschmidt, and J. Hennessy, Multiprocessor Simulation And Tracing Using Tango. *Proc. 1991 Int'l Conf. on Parallel Processing*, pp. II-99 - II-107, Aug. 1991.

[3] M. Dubois, F. Briggs, I. Patil, and M. Balakrishnan, Trace-Driven Simulations of Parallel and Distributed Algorithms in Multiprocessors. *Proc. 1986 Int'l Conf. on Parallel Processing*, pp. 909 - 915, Aug. 1986.

[4] G. Geist, M. Heath, B. Peyton, and P. Worley, A Users' Guide to PICL: A Portable Instrumented Communication Library. *Technical Report ORNL/TM-11616*, Oak Ridge National Laboratory, October 1990.

[5] M. Heath, Visualizing the Performance of Parallel Programs. *IEEE Software*, vol 8, no. 5, pp. 29 - 39, Sep. 1991.

[6] International Business Machines, IBM AIX Parallel Environment: Parallel Programming Reference. *IBM Publication SH26-7228-00*, Sep. 1993.

[7] R. Kruse, B. Leung, and C. Tondo, Data Structures and Program Design in C. *Prentice-Hall Inc.*, Chapter 9, 1991.

[8] L. Lamport, Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of ACM*, July 1978, vol. 21, no. 7, pp. 558 - 565.

[9] T. Kerola and H. Schwetman, Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs. *Proc. of the ACM SIGMETRICS*, May 1987.

[10] D. Kimelman and T. Ngo, The RP3 Program Visualization Environment. *IBM Journal of Research and Development*, vol. 35, no. 5/6, pp. 635 - 651, September/November 1991.

[11] M. Kumar and K. So, Trace Driven Simulation for Studying MIMD Parallel Computers. *Proc. 1989 Int'l Conf. on Parallel Processing*, pp. I-68 - I-72, Aug. 1989.

[12] S. Lee and J. Aggarwal, A Mapping Strategy for Parallel Processing. *IEEE Trans. on Computers*, vol. C-36(4), pp. 433 - 442, April 1987.

[13] B. Miller and J. Choi, Breakpoints and Halting in Distributed Programs. *Proc. 8th Conf. on Distributed Computing Systems*, pp. 316 - 323, March 1991.

[14] D. Mills, On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System, *ACM Computer Communication Review*, vol. 20, no. 1, pp. 65 - 75, January 1990.

[15] R. Netzer and B. Miller, Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. *Proc. 1992 Supercomputing*, November 1992.

[16] M. Pongami, W. Hseush, and G. Kaiser, Debugging Multi-threaded Programs with MPD. *IEEE Software*, vol. 8, no. 3, pp. 37 - 43, May 1991.

[17] K. So, A. Bolmarcich, F. Darema, and V. Norton, A Speedup Analyzer for Parallel Programs. *Proc 1987 Int'l Conf. on Parallel Processing*, pp. 653 - 662, Aug. 1987.

[18] S. Shafer and K. Ghose, Improving Parallel Program Execution Time with Message Consolidation. *Proc 8th Int'l Parallel Processing Symposium*, pp. 736 - 742, April 1994.

[19] C. Stunkel et. al. Architecture and Implementation of Vulcan. *Proc 8th Int'l Parallel Processing Symposium*, pp. 268 - 274, April 1994.

[20] K. Wang and P. Mehrotra, Optimizing Data Synchronizations on Distributed Memory Architectures. *Proc. 1991 Int'l Conf. on Parallel Processing*, pp. II-76 - II-82, August 1991.

[21] C. Wu, J.-P. Prost, and C. Benveniste, The Design of a Timing Simulator for Distributed Applications. *Proc. 1992 Int'l Conf. on Parallel and Distributed Systems*, pp. 50 - 57, Dec. 1992.