Timestamp Consistency and Trace-Driven Analysis for Distributed Parallel Systems

C. Eric Wu, Yew-Huey Liu, and Yarsun Hsu
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

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 an integrated approach for timestamp consistency, and performance analysis techniques for distributed parallel applications on IBM SP1/SP2 systems. The trace generation facility requires no modification on application source code, and is able to generate message passing and system events with minimal trace overhead. The techniques for timestamp consistency preserve the logical ordering of events, and are the basis in our tools for trace-driven analysis. 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. Analysis results for NAS kernel benchmarks are reported.

Keywords: timestamp consistency, trace generation, distributed parallel systems, performance tuning, message passing time, local wait time

1 Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. The logical order or event sequence may be quite obvious in a sequential program running on a single-processor computer system, however, it is sometimes hard to say that one of two events in a distributed system occurred first due to multiple local clocks.

One possible way of monitoring the behavior of a program is to generate trace records while executing the program. Trace generation is essential for trace-driven analysis [3, 9, 11], timing simulation [16, 20], interactive debugging [13, 15], and program visualization [5, 10]. In a distributed system, however, each processor (or node) has its own local memory as well as local clock, and processors communicate with one another by exchanging messages. In such a system, trace records are generated by multiple processors, and it is often the case that separate streams are produced independently in multiple nodes. There is a serious problem associated with performance analysis, i.e., the clock synchronization problem [8]. The logical order of events cannot be guaranteed in the trace due to discrepancy among local clocks. Most distributed systems, such as a cluster of workstations connected via a local area network, do not have a global clock. The event sequence in each stream therefore provides only a partial ordering of events in the system. As a result, many trace generation facilities in distributed systems are forced to do additional work to ensure consistent timestamps at the expense of increased trace generation 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 such a trace generation facility may have adverse impact on the total elapsed time and other timing-sensitive program behavior.

A causality graph was proposed in [21] to handle partial ordering events when events from different streams are gathered, and it does not require barrier synchronization during trace generation. A causality graph of partial ordering is built while reading the trace events, and events which are no longer required for ensuring logical order of subsequent events are processed and discarded from the graph. This approach assumes that point-to-point communication events are not available in the stream for group (or collective) communication routines such as broadcast. Thus, an additional rule, i.e. the begin event of a group operation must precede the end event of the operation on all other group members, are imposed to ensure a consistent total ordering of all events. The causality graph approach is not very efficient if trace streams include system activities such as process dispatch and I/O events, which is the case we are dealing with. This is because that group communication routines may take a long time to execute, creating a big causality graph in which most events have nothing to do with the group communication routine, thus resulting in a very slow merge process. Another side effect of the causality graph is that the merge process needs to keep track of all group membership information, so as to check if a node which does not belong to the group also calls the group communication routine.

There are at least a couple distributed parallel systems which have a global clock as well as local clocks. The IBM SP1/SP2 systems provide 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/SP2 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 [18].

The existence of the global clock in the IBM SP1/SP2 system is very useful for performance analysis, especially for solving the partial ordering problem when dealing with multiple trace streams. 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. If every trace record was generated with a global timestamp, it would be necessary to access the global clock for every trace event. This implies much bigger trace overhead, which is not tolerable for the operating system to generate system events.

In this paper, we discuss our integrated solution to deal with the partial ordering problem and provide workload statistics for distributed parallel applications. The paper is organized as follows. Our trace generation facility for IBM SP1/SP2 systems is described in Section 2. To our knowledge, it is the first trace facility which is able to generate not only message passing events, but also system events such as process dispatch and I/O with minimal overhead. In Section 3 we discuss our techniques to ensure a consistent total ordering of all events. Trace-driven analysis tools are described in Section 4. Using the techniques for timestamp consistency as the basis, we are able to accurately extract useful information, such as the time stolen by other processes in each node, the observed message passing time and local wait time for each message, and histograms of various metrics for analyzing the behavior of a given distributed parallel application. Concluding remarks are given in Section 5.

2 Trace Generation and Timestamps

2.1 Trace Generation and Overhead

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.

In addition, the trace facility should impose minimal overhead and provide user-controllable events. If trace overhead is large, the timestamp associated with each event may have been altered significantly, and the workload statistics and run-time data obtained in post analysis may be hardly meaningful. For user-controllable events, the trace facility should trace not only message passing events, but also system activities such as process dispatch and I/O events, at user's request.

Many trace systems such as those in [2, 4, 16] require source code modification to generate trace events. Thanks to the dynamic linking nature in the message passing layer, our AIX trace generation facility requires no modification on 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 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. To reduce the sizes of trace files, the trace facility uses a 32-bit timestamp to record a fraction of a second, and generates heart-beat events at the rate of once every 0.9 second if no other events are recorded. This reduces the timestamp size from two words to one word in every trace record, while the timestamp accuracy remains unchanged.

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, system calls, and I/O events such as read and write, etc. For message passing events, we capture the begin and end events for each basic (point-to-point) communication routine such as send and receive. For group (collective) communication routines we not only capture the begin 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 a one-word event header (so-called a hookword which identifies the event type and record length) and a one-word 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.

Table 1 shows the trace overhead for generating message passing events, including all ponit-to-point and collective communication events, for 7 distributed parallel programs on an IBM SP1 system.

Table 1: Trace generation overhead for message passing events

Program	Elapsed Time (sec)	No. of Messages	Overhead (%)
cholesky	29.55	2772	0.034
hpc	6.32	15675	1.241
is	17.27	1134	0.001
fft	45.22	406	0.001
mg	19.16	3334	0.044
ep	55.32	14	0.000
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 the LU decomposition of a 1000×1000 matrix. The rest 5 programs, Integer Sort (is), 3-D FFT PDE (fft), Multigrid (mg), Embarrassingly Parallel (ep), and Conjugate Gradient (cg), are Class A kernels in the NAS 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 Global and Local Timestamps

Due to the existence of a global clock in IBM SP1/SP2 systems, we choose to access the global clock periodically in each processor if trace is activated. A global clock event consists of both the global and local timestamps, while any other event has only a local timestamp to minimize trace generation overhead. Each processor generates global clock events periodically at its own pace. The periodic global clock events are then used in analysis tools to adjust local timestamps for local clock skews.

Fig. 1 shows the local clock adjustments using global clock events in an eight-node trace generated on a free-run IBM SP1.5 system, in which all nodes are RS/6000 model 370 nodes (same as in IBM SP1) and the adapters are TB2 adapters (same as in IBM SP2). The total elapsed time of the application is about 23.8 seconds. In this example we access the global clock at the rate of once a second.

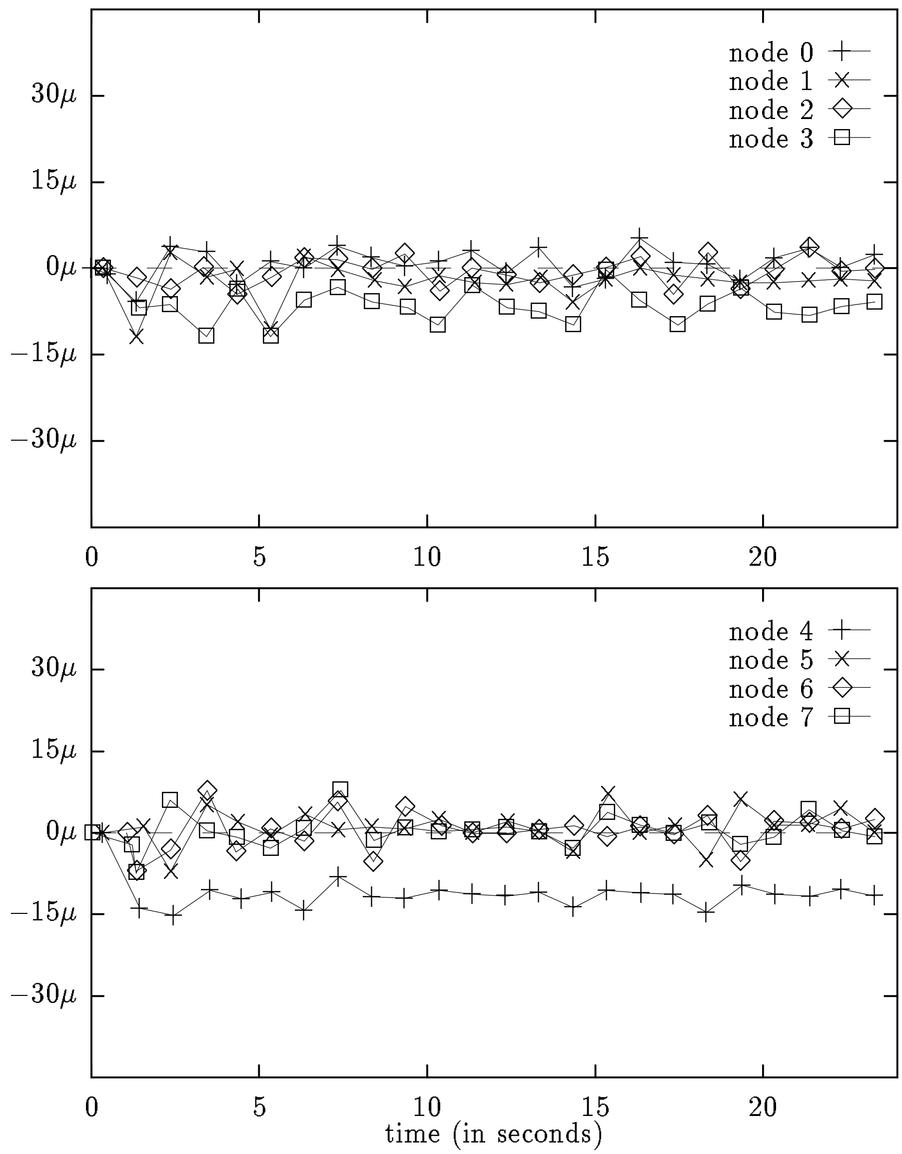

Figure 1: Local clock adjustments over time. Access frequency: once every second

There are in total 194 global clock events, eight of which were generated right before the application started up on each node. These eight global timestamps show the starting sequence of the trace streams. The global clock is actually accessed twice before the application starts to minimize the impact of temporal system activities such as cache misses or page faults. The difference between the global and local timestamps in each of the remaining global clock events indicates how much local clock adjustment is needed. Fig. 1 shows that both positive and negative adjustments are possible, although some nodes (e.g. node 7) could be always running ahead and need only negative adjustments. Almost all adjustments in Fig. 1 are less than (positive or negative) 15 microseconds. Message latency is the time required for a zero-byte message to reach its destination node. If adjustments are always less than half of the message latency, we can guarantee that no receive-before-send events could be observed.

We increase the access frequency to once every 400 milliseconds in each node, hoping to solve the partial ordering problem caused by mismatched local clocks. Fig. 2 shows the required clock adjustments in a period of about 11 seconds.

Most clock adjustments in Fig. 2 are indeed much smaller than those shown in Fig. 1, due to the higher access frequency of the global clock. However, a big adjustment occurred shortly after 1.65 seconds at node 6, where the difference between the local and global timestamps is -44.1 microseconds. The next global clock event at node 6 required a positive 41.8 microseconds, indicating that the previous discrepancy between two timestamps comes from temporal system activities, such as page faults, cache misses, or context switch. Such a phenomenon caused by temporal system activities results in a large adjustment followed by another large adjustment in the opposite direction. Because the global clock is accessed before the local clock and the access to the global clock is time-consuming, any temporal system activity between the two accesses will delay the access to the local clock, creating a larger-than-expected local timestamp and resulting in a large negative potential adjustment. To reduce the discrepancy between the global and local timestamps, cutting a global clock record could be done in the non-preemptive mode inside the operating system kernel, at the expense of additional trace overhead

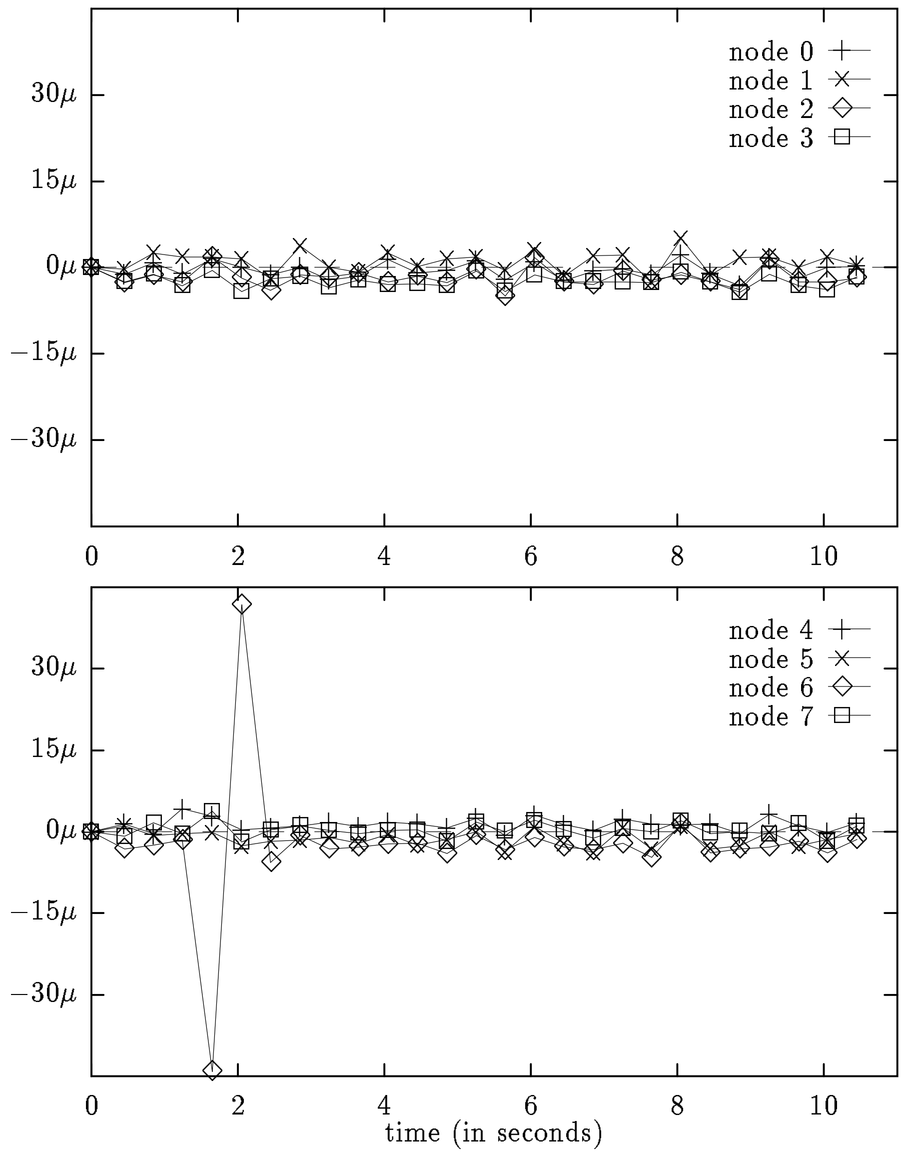

Figure 2: Local clock adjustments over time. Access frequency: once every 400 milliseconds

for switching from the user space to kernel space.

Note that the clock skew between the global clock and a local clock may be larger than the skew between two local clocks, especially if the global clock is driven by a crystal with a different frequency. In an IBM SP1 system, for example, the global clock is running at 40 MHz and all local clocks are running at 62.5 MHz. The difference between the global and local timestamps may be even larger if time daemons, such as *timed* and *xntpd* [14] which adjust local clocks periodically based on other sources, exist in the distributed parallel system. Unfortunately, most massively parallel processors (including IBM SP1/SP2 systems) do have time daemons running to ensure reliable transactions, file creations, and file modifications. Unless all nodes run freely with respect to their local clocks and global clock events are all generated in the non-preemptive mode, periodic global clock events alone are not enough to guarantee a consistent ordering of all events.

The specification of clock crystals indicates how much two clocks can drift apart under an operational environment. If two clocks may drift at most 0.01% and global clock events are generated at the rate of once every 400 milliseconds, any global clock event whose timestamps differ by more than 40 microseconds have to be a result from time daemons or temporal system activities. If it was caused by time daemons, there would be a system event such as a *settimer()* event before the global clock event in the trace (as long as it is enabled), or the adjustment is usually known (such as 1 millisecond in *xntpd* version 1). Otherwise it is caused by temporal system activities with the distinct feature of two large opposite potential adjustments, and should be ignored.

3 Consistent Total Ordering of All Events

It is obvious that using local timestamps does not ensure a consistent ordering of all events. To provide an effective basis for trace-driven analysis and program visualization of distributed parallel programs, a consistent total ordering of events is required. 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 an *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 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. 3 provides an overview of the basic matching algorithm for a consistent total ordering of events.

The `Tree_Initialization()` routine gets one event from each trace stream and sorts them based on timestamps. After the initialization phase the event with the smallest timestamp in the tree 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

```
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);
                        continue;
                }
                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);
                }
        }
        Process(event);
        Get_Next_Event_From_Stream_To_Tree(Node_ID);
}
```

Figure 3: Basic Matching Algorithm

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.

4 Timestamp Consistency as the Basis for Analysis Tools

The techniques for timestamp consistency are used as the basis to develop tools for analyzing trace events. Without a consistent total ordering of all events, the run-time data and statistics extracted from trace streams may be hardly meaningful.

In addition to common statistics such as the number of messages sent and received, our tools are capable to show the time "stolen" by other processes in each node, the observed message passing time and the observed local wait time for each message. In the following sub-sections we analyze trace files generated for NAS kernels to illustrate our trace-driven analysis techniques for distributed parallel systems.

4.1 Computation Time versus Stolen Time

The computation time on a given node is the sum of all computation intervals, defined as periods of time between the end event of a communication routine and the begin event of the next communication routine, less 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 only 50 % of the total elapsed time is used for computation in the trace, there must be room for improvement in data partition, algorithm, or implementation.

An IBM SP1/SP2 system has a built-in flexibility for running sequential and parallel tasks. Each node in the system is a fully functional, time-sharing processor with a connection to the high-performance switch network, so that the nodes can be used individually or collectively to run both sequential and parallel tasks. Therefore, the time stolen by other processes such as system daemons and other user processes in each node may help a user understand why one particular run is much slower than other identical runs.

Table 2 shows the trace elapsed time, computation time, and stolen time, along with the total elapsed time for each of the five NAS kernels on an IBM SP1.5. All numbers are in seconds, and those in parentheses are the total elapsed times. Note that the total elapsed times are different from those in Table 1, because the adapters are different (i.e., TB2 adapters in SP1.5 and TB0 adapters in SP1). The trace time of an application in each node is slightly smaller than the total elapsed time of the application, because the begin and end of the trace streams on each node are staggered.

The communication time in a node is the trace time minus the computation time and stolen time. It can be seen in Table 2 that the time spent for communication varies from a fraction of 1 % (for Embarrassingly parallel) to over 25 % (for Integer sort and 3-D FFT PDE) of the total elapsed time. The time stolen by other processes is relatively small, thanks to all colleagues for their timely cooperations. It is important to generate trace records for both message passing and system events for performance analysis. For example, it would be impossible to obtain stolen times without process dispatch events. Without these system events, a user will not have enough information to determine why certain events or routines were slower than expected. Note that the time stolen by other processes 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.

Table 2: Trace time, computation time, and stolen time

is (11.575)	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7
Trace time	11.575	11.572	11.572	11.570	11.570	11.571	11.571	11.571
Computation time	7.945	8.088	7.921	7.942	7.966	8.012	8.042	8.000
Stolen time	0.112	0.246	0.247	0.106	0.144	0.027	0.316	0.217
fft (27.048)	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7
Trace time	27.048	27.045	27.043	27.043	27.044	27.043	27.044	27.043
Computation time	19.546	20.020	19.564	19.545	19.559	20.015	20.031	20.022
Stolen time	0.152	0.295	0.196	0.314	0.280	0.168	0.205	0.288
mg (15.823)	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7
Trace time	15.807	15.810	15.800	15.804	15.814	15.810	15.813	15.798
Computation time	13.781	13.868	13.667	13.677	13.311	13.729	13.731	13.689
Stolen time	0.077	0.351	0.087	0.288	0.189	0.162	0.062	0.137
ep (55.302)	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7
Trace time	55.294	55.293	55.294	55.296	55.296	55.293	55.295	55.297
Computation time	54.818	54.821	54.816	54.807	54.805	54.815	54.814	54.804
Stolen time	0.444	0.453	0.476	0.370	0.378	0.454	0.342	0.360
cg (44.771)	node 0	node 1	node 2	node 3	node 4	node 5	node 6	node 7
Trace time	44.768	44.764	44.766	44.767	44.765	44.766	44.764	44.766
Computation time	37.396	37.579	38.169	36.720	37.338	37.414	36.794	34.713
Stolen time	0.520	0.544	0.617	0.437	0.532	0.329	0.417	0.318

4.2 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 detail histogram table for identifying dominant communication routines will be helpful. A communication interval is defined as a period between the begin and the end events of a communication routine, including any time period stolen by other processes. A communication interval could be large or small, depending on the type (blocking vs non-blocking) of a communication routine and the size of the message. Fig. 4 shows the total times spent in communication routines during the execution of the Integer Sort and 3-D FFT PDE benchmarks.

It can be seen from Fig. 4 (a) that both the Integer Sort and 3-D FFT PDE use only two group communication routines, *MP_INDEX()* and *MP_COMBINE()* [6], and almost all the communication time was spent in *MP_INDEX()* in either benchmark. The *MP_INDEX()* routine performs an all-to-all personalized communication operation for all nodes in a specified group, and the *MP_COMBINE()* routine performs a reduction on data distributed across a group. Although the Integer Sort benchmark calls the *MP_INDEX()* routine 20 times in each node, the total time spent in the routine is less than that for the 3-D FFT PDE benchmark. In fact the average time spent in *MP_INDEX()* for the Integer Sort is about 1/6 of that for the 3-D FFT PDE benchmark, mainly because the difference in message sizes: 270 Kbytes per message versus 2 Mbytes per message.

Fig. 4 (b) shows the times spent in communication routines during the execution of the Conjugate Gradient (cg) benchmark. Unlike the Integer Sort and 3-D FFT PDE which use group communication routines exclusively, the Conjugate Gradient was written using point-to-point communication routines *MP_SEND()*, *MP_RECV()*, and *MP_WAIT()* [6]. The non-blocking *MP_SEND()* routine identifies an area in memory as a buffer from which a message is sent and returns (possibly before the message is completely sent), and the non-blocking *MP_RECV()* routine identifies a buffer for an incoming message

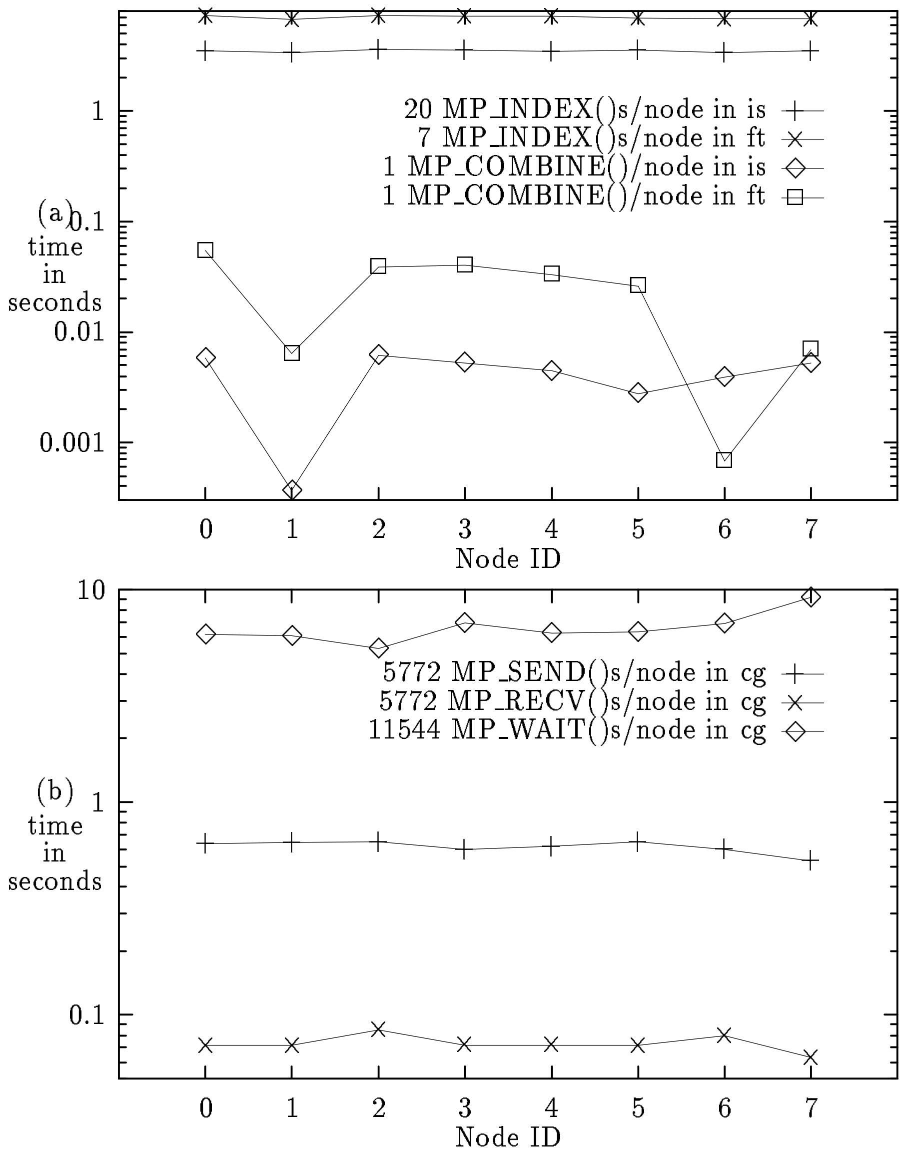

Figure 4: Total times spent in communication routines. (a) Benchmarks is and fft, (b) benchmark cg

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()*. As expected, Fig. 4 (b) shows that the blocking *MP_WAIT()* is the dominant communication routine in cg. Similar information for other 2 NAS kernels are not presented here due to limited space in this paper. These observations provide a user with some hints as to where one might focus to optimize the code.

4.3 Observed Message Passing Time and Local Wait Time

A user may want run-time information about the communication portion of his/her 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 among the most important information for message passing applications.

Fig. 5 illustrates the definition of the observed message passing time and local wait time of a message. The observed message passing time of a message is defined as the time between the begin event of the send at the source node and the end event of the receive at the destination node. It is actually an upper bound for the actual message passing time, which is defined as the time between the begin event of the send and the point at which the message 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 check it.

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. The observed message passing time of each message in such a program should be close to its estimated lower bound, which is defined as the message size over effective point-to-point bandwidth

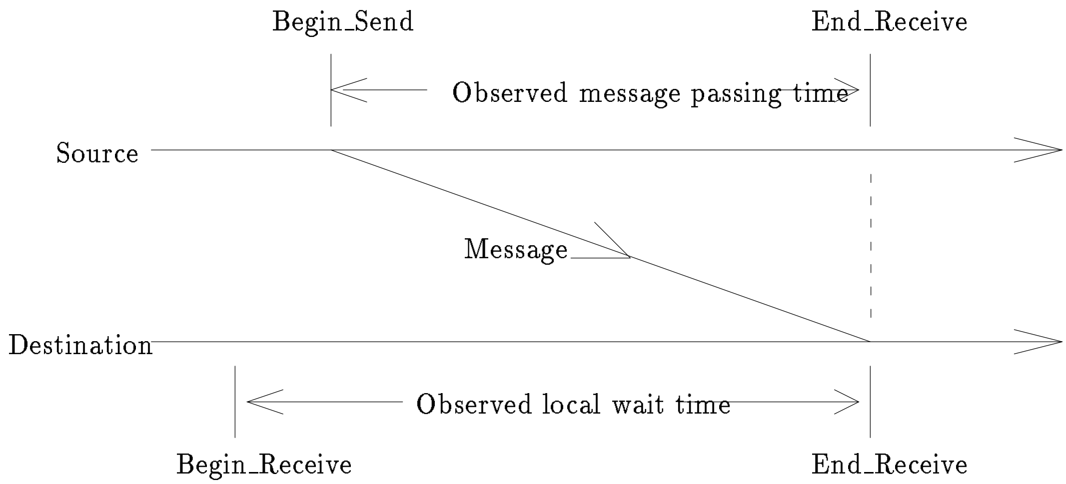

Figure 5: Observed message passing time and local wait time

plus the message latency.

Fig. 6 shows the observed message passing time and local wait time, along with source node ID, destination node ID, message type, message size, the time since activating the trace when the message is received, and the name of the receive routine, for the first 25 messages received in the 3-D FFT PDE benchmark. Note that the message type -10 is used in group communication routines for point-to-point messages. Since the message sizes are the same, one would expect the observed message passing times and local wait times for these messages to be similar. As can be seen in Fig. 6, messages of the same size have similar observed message passing times and local wait times, although they do vary due to the fact that each node is running at its own pace (i.e. in SPMD mode).

It is important to provide users with run-time data such as the observed message passing time and local wait time for each message, especially if performance tuning is one of our concerns. If a message is received with larger_than_expected observed message passing time using a basic point-to-point message passing routine, a closer look may help the programmer modify the source code to improve the total elapsed time. For example, if the receiving node has other active processes during a run, process dispatch

Src	Dest	M_Type	M_Size	Msg_Pass_Time	Local_Wait_Time	Recv_Time	Using
3	4	-10	2097152	0.121626016	0.117552896	1.751954073	MP_INDEX
4	3	-10	2097152	0.117710176	0.121783296	1.752111353	MP_INDEX
0	7	-10	2097152	0.157892242	0.122282240	1.786707858	MP_INDEX
7	0	-10	2097152	0.122457454	0.158067456	1.786883072	MP_INDEX
1	6	-10	2097152	0.129640735	0.124191232	1.793841666	MP_INDEX
6	1	-10	2097152	0.124355297	0.129804800	1.794005731	MP_INDEX
2	5	-10	2097152	0.158219760	0.124342016	1.794164518	MP_INDEX
5	2	-10	2097152	0.124509456	0.158387200	1.794331958	MP_INDEX
2	0	-10	2097152	0.117228490	0.124678656	1.911571200	MP_INDEX
0	2	-10	2097152	0.124860982	0.117410816	1.911753526	MP_INDEX
3	6	-10	2097152	0.163888137	0.122156544	1.916007938	MP_INDEX
7	1	-10	2097152	0.129316433	0.122018304	1.916033251	MP_INDEX
1	7	-10	2097152	0.122025135	0.129323264	1.916040082	MP_INDEX
4	5	-10	2097152	0.164322701	0.122113792	1.916287782	MP_INDEX
6	3	-10	2097152	0.122453239	0.164184832	1.916304633	MP_INDEX
5	4	-10	2097152	0.122359411	0.164568320	1.916533401	MP_INDEX
0	4	-10	2097152	0.122831769	0.117871872	2.034414233	MP_INDEX
4	0	-10	2097152	0.118021223	0.122981120	2.034563584	MP_INDEX
2	7	-10	2097152	0.126694748	0.122407936	2.038457490	MP_INDEX
7	2	-10	2097152	0.122561444	0.126848256	2.038610998	MP_INDEX
3	1	-10	2097152	0.122305770	0.122575872	2.038618851	MP_INDEX
1	3	-10	2097152	0.122731286	0.122461184	2.038774265	MP_INDEX
6	5	-10	2097152	0.122847012	0.122564864	2.038864166	MP_INDEX
5	6	-10	2097152	0.122699996	0.122982144	2.038999298	MP_INDEX
2	4	-10	2097152	0.118044515	0.122241280	2.156665497	MP_INDEX
.....							

Figure 6: Observed message passing time and local wait time

events will show up in the trace and by simply re-running the application, the user may obtain very different results. Another possibility for a large observed message passing time is that the receiving node issued the receive routine 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. If a message is received with a large local wait time, the node may have called the receive routine too early.

Similar observations can be applied to messages received by group communication routines, as seen in Fig. 6, becuase point-to-point message passing events within each group communication routine can be captured as well. Thus, the observations on message passing times and local wait times provide a systematic way for developers of the group communication routines and application programmers to check the run-time behavior of their programs.

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 Fig. 6, the average of the observed message passing time is around 139 milliseconds, which corresponds to an effective point-to-point bandwidth 14.4 Mbytes per second. Since the maximal effective point-to-point bandwidth is around 30 Mbytes/sec for large messages, there may be some room for possible performance improvement in this version of the group communication routine. The observed local wait times are about the same as the observed message times, indicating a balance in the implementation of the routine.

4.4 A Tool for Performance Tuning

Given the message latency A in seconds and effective point-to-point bandwidth B in bytes/second, the estimated 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. Note that the effective point-to-point bandwidth varies for messages of different sizes due to the overhead associated with message passing routines. Large messages tends to have larger effective point-to-point bandwidth than small messages. A user may obtain the parameters A and B from the manufacturer of the distributed parallel system or traces generated by a carefully written program using the analysis tools. The observed message passing time obtained from trace files can then be compared to this estimated lower bound.

The information about message passing time and local wait time is very useful for performance tuning, especially for synchronous applications in which each node starts 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 the 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. 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 $b \geq m$.

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 theoretical papers discussing message consolidation under this condition [17, 19], and it is very promising to be able to develop analysis tools for performance tuning. 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 would completely overlapped, local wait times are eliminated, and the performance would be improved.

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, it provides a systematic approach to do performance tuning for distributed parallel applications.

4.5 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 times at which each message was sent and 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.

Fig. 7 shows the observed point-to-point bandwidth in the hpc trace for matrix LU decomposition. An observed bandwidth is obtained for a point-to-point message, using its message size divided by its observed message passing time. If several messages are of the same size, the average bandwidth is used in Fig. 7. It can be seen that the effective point-to-point bandwidth is close to 30 MBytes per second (1 M = 2^{20}). Note that Fig. 7 is not a figure for showing the characteristics of the switch network, in which the point-to-point bandwidth should be measured without network congestion and interference from other system or user processes. A dip of bandwidth in Fig. 7 merely reflects temporary system activites when the messages are in transit, and has nothing to do with the characteristics of the switch network. Therefore, the figure gives an estimate about the performance of the switch network for the hpc program at the time when the program was executed.

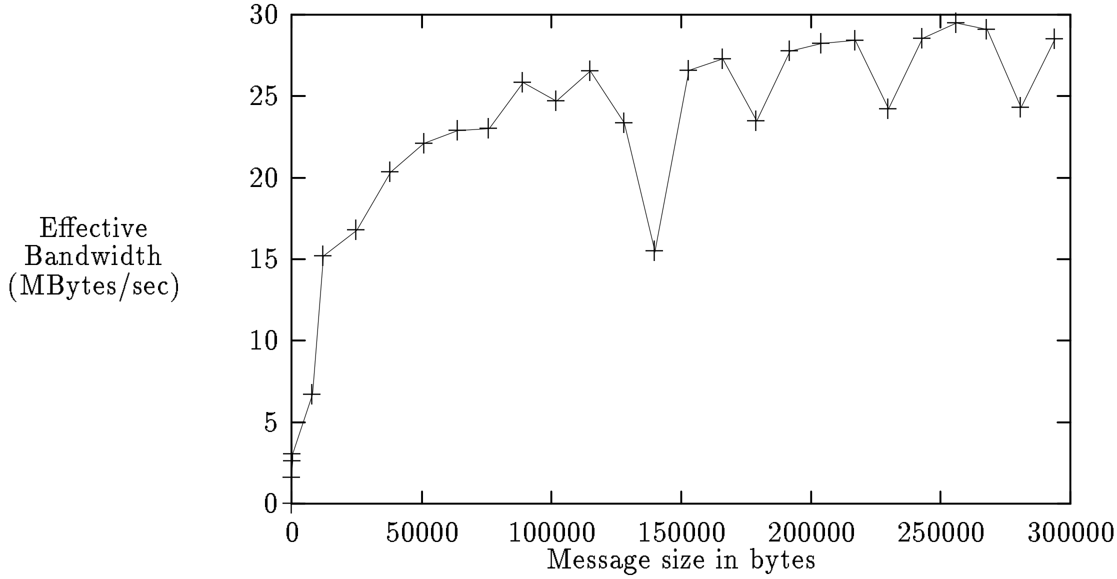

Figure 7: Observed effective point-to-point bandwidth in hpc trace

5 Summary

In this paper we have described the techniques for trace generation and ensuring a consistent total ordering of events in distributed parallel systems. Our trace facility is able to generate both system and message passing events with minimal overhead, and users do not have to modify application source code for trace generation.

The techniques for ensuring a consistent total ordering of all events are then used as the basis for trace-driven analysis. Due to the ability to generate 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 monitoring and 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 routines, to the Parallel Applications Group for developing the NAS benchmark suite on IBM SPn systems, to Ron Mraz for discussions on time daemons, and to M. Snir for his 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] M. Pongami, W. Hseush, and G. Kaiser, Debugging Multi-threaded Programs with MPD. *IEEE Software*, vol. 8, no. 3, pp. 37 - 43, May 1991.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] D. Zernik, M. Snir, and D. Malki, Using Visualization Tools To Understand Concurrency. *IEEE Software*, pp. 87 - 92, May 1992.