Perturbation Analysis of High Level Instrumentation for SPMD Programs

Sekhar R. Sarukkai*
Dept. of Computer Science
Indiana University, Bloomington, Indiana

Allen D. Malony†
Dept. of Computer and Information Science
University of Oregon, Eugene, Oregon 97405

Abstract

The process of instrumenting a program to study its behavior can lead to perturbations in the program's execution. These perturbations can become severe for large parallel systems or problem sizes, even when one captures only high level events. In this paper, we address the important issue of eliminating execution perturbations caused by high-level instrumentation of SPMD programs. We will describe perturbation analysis techniques for common computation and communication measurements, and show examples which demonstrate the effectiveness of these techniques in practice.

1 Introduction

Any measurement of a program's execution, no matter how non-intrusive, can perturb how the program behaves (important for debugging) and performs (important for performance evaluation). In contrast to perturbations of correct execution behavior, perturbations of performance behavior, although more tractable in theory, pose difficult problems for performance analysis in practice. For example, programs run on massively parallel systems with detailed performance measurement can have subtle perturbation effects that lead to large performance evaluation errors. In general, a balance exists between measurement detail and measurement perturbations that depends on the requirements of the performance evaluation problem being addressed. When perturbations lead to performance data that is too inaccurate to adequately evaluate a performance issue, measurements must be reduced, but this occurs at the risk of lost performance detail and, again, performance evaluation accuracy. Clearly, understanding perturbation effects of performance measurement is important for determining appropriate levels of instrumentation. However, improving the tradeoff of measurement detail and evaluation accuracy will come only from the incorporation of better perturbation analysis techniques in performance data processing.

The issue of perturbation analysis has been treated extensively by Malony [7, 8, 9]. However, that work dealt exclusively with shared memory programs and assumed the presence of fine-grained, low-level instrumentation. In this paper, we focus on source level instrumentation of programs based on the Single-Program, Multiple-Data (SPMD) execution model. Our aim is to determine the types of perturbations caused by high level instrumentation of SPMD programs and to devise analysis methods for removing perturbation effects.

The assumptions made regarding SPMD instrumentation are few. We assume that the un-instrumented parallel program is sequentially consistent (i.e., satisfies Lamport's sequential consistency criteria [5]). This implies that the instrumented program will be non-interfering and safe [1]. We assume that the instrumentation detects and records events independently in each processor. As such, the instrumentation does not

*Supported by the NSF New Technologies Program under grant NSF ACS 911616 and a contract from IBM. Current address: Recom Technologies, MS 269-3, NASA Ames Research Center, Moffett Field, California 94035.

†Supported in part by a contract from Rome Labs under Air Force contract no. AF 30602-92-C-0135.

4th ACM PPOPP,5/93/CA,USA

affect the partial order of the original program. However, since the actual execution may have inherently nondeterministic behavior, a single performance measurement reflects only one possible execution scenario.

Section 2 discusses the characteristics of SPMD instrumentation, the effect of instrumentation on program performance, and the requirements for accurate perturbation analysis. Section 3 presents the perturbation analysis of several important instrumentation cases of interest in SPMD computations. In Section 4, we show experimental results that validate the use of perturbation analysis in practice.

2 Instrumentation

The SPMD execution paradigm offers the programmer a high level abstraction of parallel computation. In SPMD programs implemented using message passing, processors perform computations on local data, while executing local copies of the same code. When data from other processors are required, processors perform communication operations to obtain the data. This communication is also used for processor synchronization. Logically, there are no sequential regions in a SPMD program, since the whole program is executed in parallel by all the processors.[1]

From the perspective of the programmer, unless low level instrumentation is provided in the target system, the performance characteristics of a SPMD program execution must be determined through an analysis of high level, source code events. While it is clearly important to know if source code events alone are sufficient for performance evaluation, it is also important that the collected performance data accurately reflect the program's actual execution – perturbation effects introduced by the source instrumentation have to be eliminated or reduced to acceptable levels.

We assume that instrumentation inserted in a SPMD program generates data in the form of events that are stored in a trace file. There are a number of source level events that that might be captured during SPMD program execution. These can be broadly classified as:

- *Processor computation events:* These events occur in each processor and reflect local computation states. The events on a particular processor are dependent only on local computations. However, the timing of the events can be affected by the execution dependencies that exist between processors. The measurement of these events is necessary to capture timing information of local processor computations for reasons of execution profiling.
- *Message communication events*: Messages are used for different purposes such as data exchange, broadcast communication, and synchronization. Measurements of these events are needed to observe the performance of communication operations used to enforce data and control dependencies that exist in the SPMD computation.

The capture of processor computation or message communication events can perturb the execution of a program both directly and indirectly. Any event measurement will introduce a timing perturbation in a region of execution local to when the event occurred. However, perturbations can accumulate and can propagate through an execution, affecting the order and timing of events later captured.

As shown in Figure 1, the process of perturbation analysis takes the measured event trace and produces an event trace that is an approximation of the actual execution. It uses the execution semantics of the operations captured by the high-level events to reduce or eliminate the perturbations introduced in the measurement. The process of eliminating any perturbation effects must ensure that:

- causality is preserved (e.g., a message should not be received before it has been sent);
- partial order of the events is preserved (e.g., a new communication should not be initiated before a previous communication upon which it might depend has been completed); and
- deadlocks are not introduced.

Causality can be preserved by enforcing ordering and timing relationships between message

[1] The SPMD execution model is the basis for several programming languages targeting massively parallel systems [4, 6].

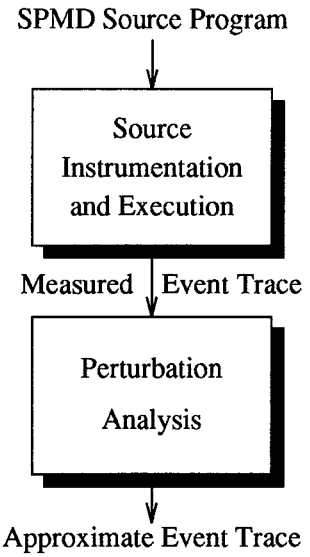

Figure 1: Perturbation Analysis Process

send and receive operations. Partial order is preserved on a single processor by guaranteeing that the time order of any two events is unchanged. By satisfying causality and partial order constraints during perturbation analysis, it follows that if no deadlock occurs in the observed execution of the program, none will be present in the approximated execution.

Our approach to perturbation analysis attempts to resolve the timing of measured events on each processor in sequence. The time of an event is approximated based on the perturbations introduced by events occurring prior to it and the timing relationships of events upon which it depends. It is our contention that perturbation effects of SPMD program instrumentation can be effectively analyzed by considering only the perturbations that result from instrumenting standard computation, communication, and synchronization primitives. Results from test studies of several applications run on different parallel systems show this approach is effective in practice; see Section 4. The following section describes the formal techniques involved in perturbation analysis for different SPMD program operations; a full description of the analysis and experimental results can be found in [10].

3 Perturbation Analysis

For the perturbation analysis discussion that follows, we consider as input a trace file consisting of a time ordered set of events. The time stamp of the ith event stored by a processor p in the trace file is represented by $t_m^p(i)$ ($p \leq P$, where P is the total number of processors that execute the program). The m indicates that the time stamp is a measured value. The perturbation analyzer generates a new trace file with the same number of events, but with recomputed time stamps. The time stamp of the events generated by the analyzer are represented as $t_a^p(i)$; the a subscript indicates that the time stamp is approximated.

If we let T represent the actual time taken to execute a SPMD program without instrumentation, T_m indicates the total time of the measured program and T_a the total approximated time after perturbation analysis. The difference between the measured and actual execution time of the program, $|T_m - T|$, gives an indication of the total amount of measurement error introduced into the program. The difference between the measured and approximated execution times, $|T_m - T_a|$, gives an indication of the amount of perturbation error that can be accounted for in the analysis. The aim of eliminating the perturbations is to ensure that $|T_a - T| \leq |T_m - T|$ and to minimize $\delta = \frac{|T_a - T|}{T}$.

3.1 Programs without Communication

Although somewhat unrealistic, a SPMD program without any inter-processor communication is the easiest to analyze. This case consists of a set of processors executing the same program on some local data, independent of each other. When instrumented, such a program produces a trace file containing processor computation events only. In this case, no communication occurs, so the perturbation analysis is all localized to the processors.

The effect of perturbation on such simple programs is an increase in the execution time on each processor. Although the perturbation effect for each processor is not dependent on the number of events captured in other processors, it is dependent on the number of events captured locally. Thus, one could see large perturbation effects for

the SPMD computation in execution time due to the differences in the local perturbations on each processor.

The elimination of perturbation effects is performed incrementally, starting from the first event collected in each processor. Given $t_m^p(i)$, the approximated time stamp of the event, $t_a^p(i)$, is given by: $t_a^p(i) = t_m^p(i) - f(i-1)$ where $f()$ is an *instrumentation overhead* function. Assume that the time taken to store an event can be approximated by a constant α, then $f(i) = \alpha i$.

3.2 Programs with Global Synchronization

SPMD programs typically involve communication between processors for purposes of data exchange and synchronization. Different communication mechanisms require different perturbation analysis. Here we consider global synchronization. Global synchronization can manifest itself in SPMD programs in one of the following ways:

- barrier synchronization operations,
- global broadcast operations, or
- global reduction/combining operations.

Since the perturbation analysis for each case is similar, we concentrate on barrier synchronization.

A barrier is loosely defined as a point where all participating processors synchronize. Here, we treat the case where all P processors perform the barrier synchronization. The high level instrumentation of barrier synchronization involves capturing two events for each processor: barrier *entry* and barrier *exit*. Unless the barrier operation is explicitly coded at the source level, capturing low level communication events may be prohibitive. Thus, the high level semantics of barrier operation are all we can use in perturbation analysis.[2]

[2]We should note that global barriers act as boundaries to perturbation propagation. Because there is a point in time when all the processors are waiting for a single event to occur (i.e., the global synchronization), perturbation analysis errors occurring before that global synchronization point do not carry forward beyond that point in the approximation – the timings of all events following the synchronization point are relative to the time of the synchronization. Once the time of the barrier synchronization event is resolved, all following event times can be approximated.

Perturbation analysis of global synchronization must determine $t_a^p(entry)$ and $t_a^p(exit)$ for each processor p. Whereas $t_a^p(entry)$ can be calculated separately for each processor, $t_a^p(exit)$ depends on the time when the last processor arrives at the barrier as well as the barrier semantics concerning how processors exit. For sake of discussion, we assume that the processors leave the barrier sequentially in the reverse order of their arrival.

Figure 2 shows the measured and approximated execution for a SPMD program involving a barrier synchronization. Notice that, after the removal of the instrumentation overhead, it is possible that a different processor is the last to arrive at the barrier; in the figure, processor $P3$ is the last to arrive at the barrier in the measured execution whereas $P2$ is the last to arrive in the approximated execution. Once the approximated entry event times, $t_a^p(entry)$, are known, the exit event times, $t_a^p(exit)$, can be approximated by $t_a^p(exit) = t_a^l(entry) + (P - i + 1) * \beta$, where l is the last processor to enter the barrier (and the first processor to exit the barrier), processor p is the ith processor to enter the barrier, and β is the synchronization overhead per processor. Now all we have to do is approximate β. If F is the first processor to exit the barrier synchronization (in the measured execution) and L the last processor to exit the barrier synchronization (in the measured execution), β can be approximated from the measured data as $\beta = \frac{(t_m^L(exit) - t_m^F(exit))}{(P-1)}$.

In the case of logarithmic or sequentially scheduled barriers, the perturbation analysis is simpler because the order in which the processors exit from the synchronization is fixed. The analysis of these barrier cases and the cases of global communication operations (e.g., broadcast) and global reduction/combining operations can be found in [10].

3.3 Programs with Processor-pair Synchronization

Inter-processor communication in SPMD programs is most commonly performed by one processor writing data to another processor which reads the data. The read-write operations can be *blocking* or *non-blocking*. The most commonly used form of communication is blocking read-write. Invoking a blocking read operation causes

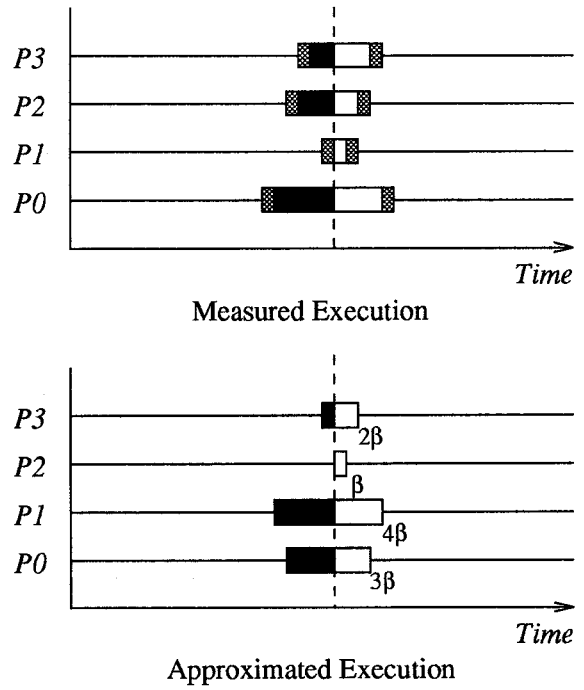

Figure 2: Perturbation analysis of barrier synchronization

the calling processor to block until the data has been received and copied into the program address space. Two mechanisms may be used for blocking writes: asynchronous and synchronous. Invoking a blocking write using some compilers, such as Fortran D [4], causes the calling process to block until the data has been copied out of the program address space into the system address space. However, this does not mean that the process must wait for the message to be actually received by another processor (i.e., the write blocks on buffer copy but is asynchronous to the receive). In other systems, blocked communications are implemented by ensuring that both the receiving and the sending processors have executed their respective communication calls before actually initiating the message transfer (i.e., blocking synchronous write).

Non-blocking messaging, supported in machines such as the iPSC/860, allow computation and message copying to be performed in parallel. The time to copy data into a system buffer is the amount of time that can be saved using non-blocking instead of blocking writes. However, in some cases, message startup time for non-blocking messages is higher than for blocking communication; hence, it is used selectively. In the blocking and non-blocking case, the high-level events captured in the SPMD program are assumed to be the begin and end of each of the communications, read or write.

The effects of perturbations in processor-pair synchronizations are more complex than in the global synchronization cases. Unlike the global synchronization cases, the approximated events must be resolved in sequence while carefully maintaining causality and partial order. Blocked synchronous communication is similar to a rendezvous between two processors followed by a message communication. The perturbation analysis for this case can be derived from that for global broadcasts where the number of processors is restricted to two. The non-blocked read-write case does not pose any problems for analysis per se, but it does includes a *busy wait* component, normally used in association with this call, that can lead to non-deterministic behavior. Although blocked asynchronous communications is more restrictive than the pure non-blocked case (since the read operation waits explicitly for the message), the perturbation analysis still must estimate message transfer times using communication timing models to improve perturbation resolution accuracy. We analyze this situation in detail below.

It is important to note that in some cases the processor that transmits the data may not be known to the receiving processor (e.g., when the receiving processor performs a "blind" receive). In such cases, the perturbation analyzer will have to determine which sending processor's data was actually read from the receiving processor's buffer. Unfortunately, it may be impossible to accurately determine this information without detailed knowledge of machine execution or access to lower level events. Instead, the perturbation analysis must assume the order in which the messages are read from the I/O buffer.

Perturbation analysis of blocked asynchronous read-write communications is complicated by the fact that the times the receiver and sender perform the read and write operations, respectively, can be reversed from the measured to the approximated execution. Consider the situation in Fig-

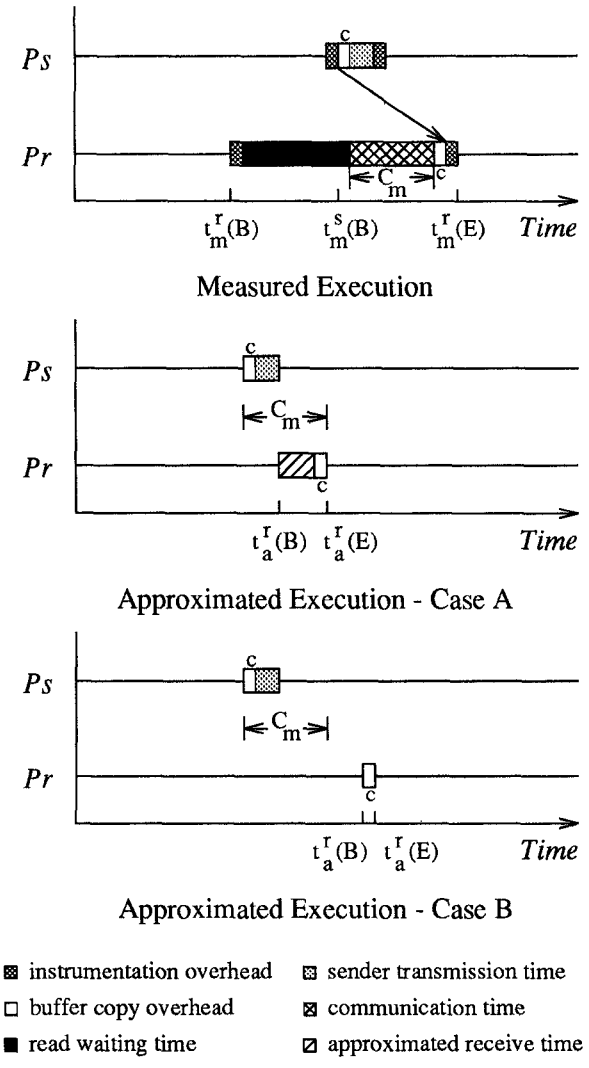

Figure 3: Perturbation analysis of blocked asynchronous communication – Scenario 1

ure 3. Here, processor Ps sends a message to processor Pr. In the measured execution, $P2$ arrives at the read earlier than the write and has to wait. Once Ps starts the write (at time $t^s_m(B)$), it begins sending the data to Pr which receives all the data by time $t^r(E)$; here, r refers to the receiver, s refers to the sender, B is the beginning event (send or receive), and E is the endinging event (send or receive). The difference in time between the initiation of the write and the completion of the read is the time taken to actually pass the data: communication time, C_m, plus the time to copy the data to and from system buffers, c. This time should be preserved in the approximated execution.

Now, let us look at the approximated execution. Suppose that perturbation analysis produces an approximation of the beginning read and write events such that $t^s_a(B) < t^r_a(B)$. There are two cases to consider. In Case A, the receiving processor performs the read after the sending processor performs the write, but before the communication has ended. In this case, the completion of the read operation will depend on when the write was initiated. The time of completion of the read is the sum of the approximated time of initiation of the send, $t^s_a(B)$, and the measured time for communicating the data, C_m, and buffer copy time, c, in the receiving processor. Case B illustrates the situation where the approximated time the read begins occurs after the communication finishes. Here, the time of completion of the read is the approximated time for read begin plus the buffer copy time.

In Figure 4, we see another scenario where, in the measured execution, the read operation is initiated after the write operation has completed. To apply the same perturbation analysis as above, the communication time must be determined. However, in this case, communication time cannot be exactly determined from source level instrumentation alone, because there is no event indicating when Pr received the data – only when the read operation completed. Rather, an estimation of the communication time must be used.

Figure 4 illustrates one of two situations that can arise in the approximated execution. Here, we see Pr beginning a read operation after the entire message has been transmitted; note, $t^r_a(B)$ does not depend on $t^s_a(E)$. However, how is $t^r_a(E)$ calculated? Since the time needed to transmit the data (represented by C_m) cannot actually be determined only from source level events, we have to approximate the communication time in one of two ways: 1) based on the length of the message in bytes n and the network bandwidth, or 2) based on lower and upper bound analysis. If the rate of transfer of a byte is γ, the start up time is S, and the per-hop delay is H, then the approximated communication time, C_a, can be estimated by: $C_a = S + n \times \gamma + h \times H$, where h is the number of hops.[3] The approximated time of the comple-

[3]The constant values can be determined by using a

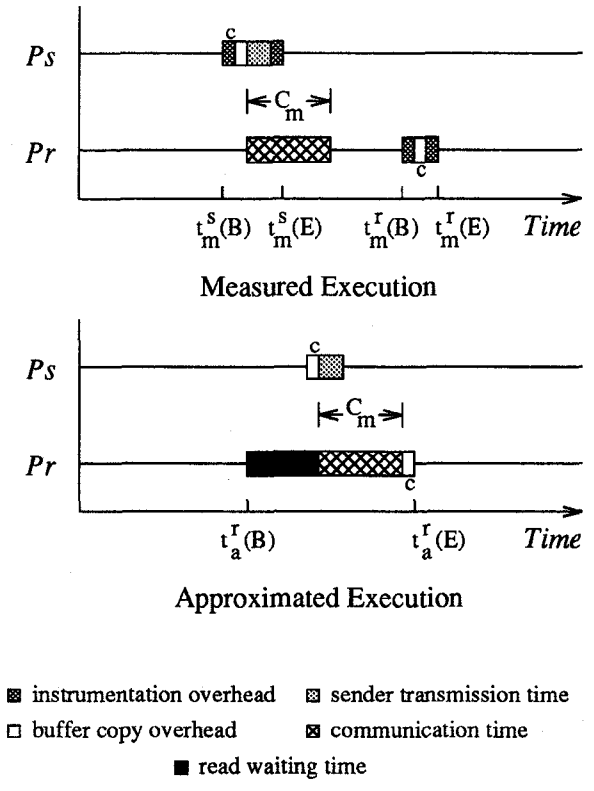

Figure 4: Perturbation analysis of blocked asynchronous communication – Scenario 2

tion of the read operation can then be expressed as:

$$t^r_a(E) = MAX \begin{cases} t^s_a(B) + C_a + 2 \times c \\ t^r_a(B)) + c \end{cases}$$

Alternatively, we can use lower and upper bound values of the communication time to calculate $t^r_a(E)$. The lower bound assumes that the communication is infinitely fast (i.e., $C^l_a = 0$). The upper bound is assumes that the measured time between the write begin and the read end is the communication time (i.e., $C_a = C'_m = t^r_m(E) - t^s_m(B)$). The actual communication time will be between these two bounds.

linear-least square fit by experimenting with a number of different message sizes and processors. In [3], Dunigan estimates the constant values for messages > 100 bytes, for the iPSC/860 cube to be: $S = 136\mu$secs, $\gamma = 0.4\mu$secs and $H = 33\mu$secs.

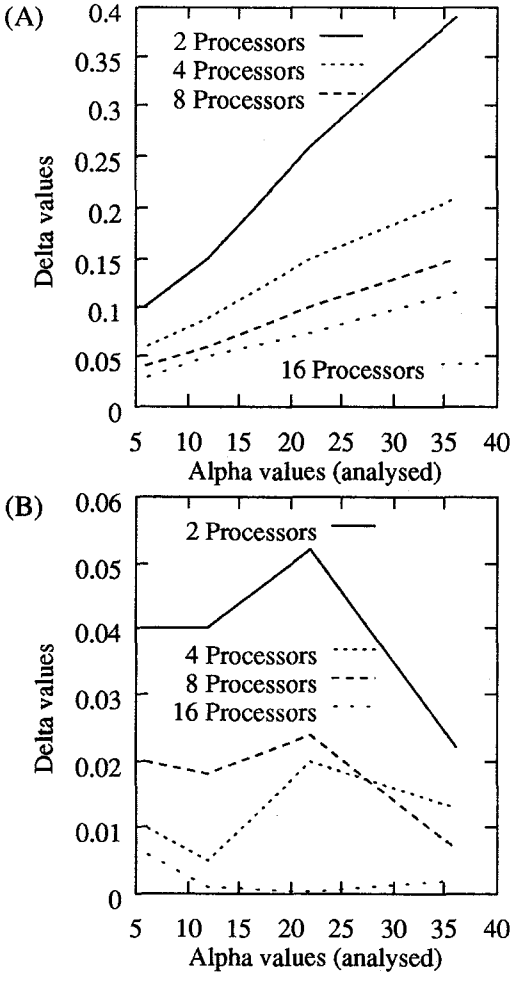

Figure 5: Perturbation analysis results from barrier synchronization experiments. (A): Measured versus Actual. (B): Approximated versus Actual.

4 Experimental Results

In this section, we present some experimental results to determine the efficacy of the perturbation analysis approach. All of the experiments were performed on an Intel iPSC/860 machine. Each experiment generated a trace file that was passed through a perturbation analyzer, producing a new trace file with approximated event orders and time stamps.

4.1 Barrier Synchronization

Our first experiment considers a program with global barriers. We use a simple test program that performs some local computation in each it-

eration of a loop and then synchronizes all the processors at the end of each iteration. A sequence of barrier executions result. The begin and end of each barrier is instrumented. We want to investigate the effect of the instrumentation overhead, α, on perturbation analysis. As α increases, the amount of intrusion introduced in the observed program execution also increases. Graph (A) in Figure 5 shows the difference in total execution time between the program's measured execution and its uninstrumented execution, δ_m, as a percentage of the uninstrumented time, for different number of processors and α values. The number of processors varies between 2 and 16, while α varies between 6 and 36 time units. The graph shows that α and δ_m are positively correlated, as expected. However, notice that the δ_m value changes are more significant for a smaller number of processors. This is due to the fact that the amount of time taken to execute the program increases as the number of processors increase (with the same number of events stored per processor), due to increase in synchronization time. This results in a reduction in percentage time spent in instrumentation overheads with increasing number of processors. For a program that scales well, however, we would tend to see the exact opposite effect, because the time to execute the program decreases with increase in number of processors, while the instrumentation time would remain constant.

Using perturbation analysis for barrier synchronization, we can determine δ_a values from the approximated execution derived from the measured trace. Graph (B) in Figure 5 we see that perturbation analysis can determine a total program execution time that is within 5 percent of the actual execution of the program, independent of α value and number of processors used.

4.2 Ocean Code

In general, measurement of an application program will involve capturing events from several different types of communication operations. For simplicity, the next experiment included events from blocked non-synchronous read-write communication only. We instrumented a parallel ocean circulation code [2] to capture communication between processors when transpose operations were performed on various matrices. In the code, each matrix is divided into slabs of columns. Each slab is divided into P blocks (for P processors), with the pth block communicated to processor p using a blocked non-synchronous read-write mechanism. We executed this program under two different scenarios: 1) the number of computation events stored by each processor decreases as the number of processors is increase, and 2) the number of computation events stored per processor is constant.

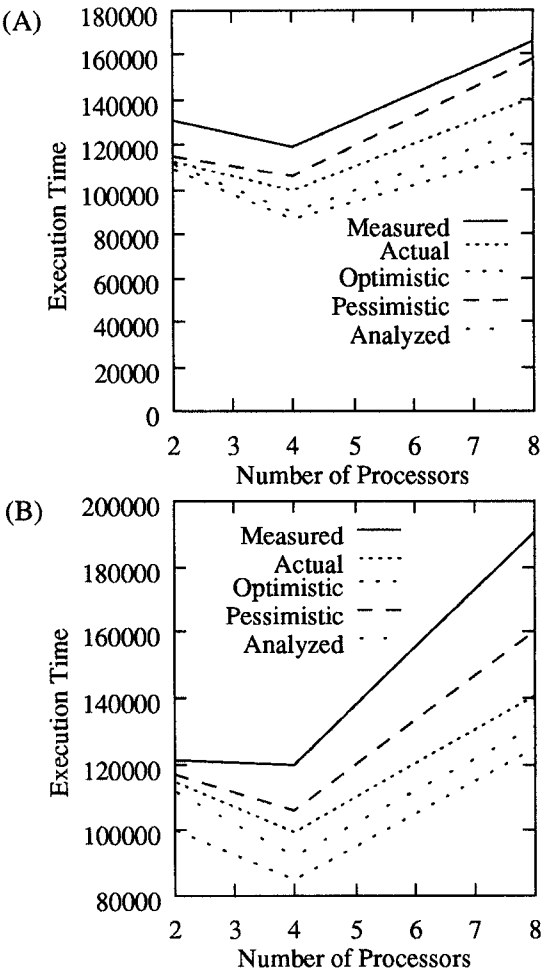

Figure 6: (A): Decreasing events per processor. (B): Equal events stored per processor.

Each case was analyzed using certain assumptions about the program execution. The cases we considered were:

- optimistic analysis
- pessimistic analysis

- analysis based on a communication model

Optimistic analysis corresponds to the lower bound analysis presented in Section 3.3 (i.e., we assume that messages are communicated infinitely fast in the approximated execution). As shown in Figure 6 (Graph (A)), the execution times from optimistic analysis are consistently less than the actual execution times. This is the expected result. Pessimistics analysis corresponds to the upper bound analysis presented in Section 3.3 (i.e., we assume that message communication times can always be determined from the time difference between the beginning of the write and the completion of the corresponding read in the measured execution). In Figure 6 (Graph (A)), the execution times from pessimistic analysis are consistently larger than the actual execution times, as expected.

To improve the accuracy of perturbation analysis, we must apply a communication model in the analysis that approximates the actual communication time. Following the model presented in [3], we assume that the communication time for a message of size n that traverses h hops is given by:

$$C_a = \begin{cases} 76 + n \times 0.4 + h \times 11 & n < 100 \\ 136 + n \times 0.4 + h \times 33 & n \geq 100 \end{cases}$$

Using this expression in the analysis results in an approximated execution more closely reflecting the actual execution for the different numbers of processors tested; see Figure 6 (Graph (A)). The fact that the analysis results in execution times that are lower than the actual execution times might be explained by the choice of constant factors in the model or by other errors in the perturbation analysis.

If the number of computation events stored is proportional only to the problem size, the number of computation events per processor reduces as the number of processors are increased, reducing per processor perturbation in turn. However, if the number of computation events collected per processor is constant, the total number of events collected increases in proportion to the number of processors. As shown in Figure 6 (Graph (B)), as the number of processors increase, the deviation between the observed and actual execution also increases. This implies that the execution time is not influenced only by the events stored in each processor independently. As in the analysis producing Graph A, we get lower and upper bound approximations on the actual execution time in Graph B using optimistic and pessimistic perturbation analysis. Similarly, the analysis based on modeled communication falls between the two bounds and consistently tracks the actual execution time of the program.

Now consider the same program but with all the receives replaced by a "blind" receive (i.e., the receiving processor can read any data that is present in its input buffer, without knowledge of the message's sending processor). Assume also that the events do not store information about the transmitting processor. In such a case, the analysis for the receiving processor will not be able to accurately match the processor whose value has been read from the receiving buffer. Rather, assumptions must be made in the perturbation analysis about the order in which the blind receives are satisfied. Possible strategies for perturbation analysis to follow include:

- Preserve communication order: The order of sends and receives in each processor is assumed to remain the same from the measured to the approximated execution. This is clearly a bad assumption for unbalanced instrumentation, since some processors may incur more perturbations than others, severely altering communication order in the measured execution.
- Last In First Out ($LIFO$): All sends in the approximated execution are stored in a stack. Each time the receiving processor performs a receive, it reads the last message written to its buffer (i.e., the most recently received message). Typically, this is not a good choice, since messages are most often written to a receiving queue.
- First In First Out ($FIFO$): All sends in the approximated execution are stored in a queue. The receiving processor reads data from its message queue, the next message read being the oldest one in the queue.

Executing the program with instrumentation that varies with the number of processors, we observed a 20% slowdown in actual execution time for 4 to 16 processors; see Figure 7. Using a

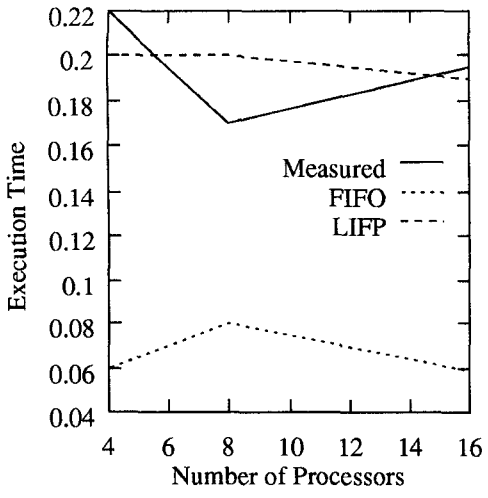

Figure 7: Perturbation analysis with communication order approximations

LIFO strategy in perturbation analysis, we notice that the error is not reduced. However, as expected, the *FIFO* strategy performs better than the *LIFO* strategy, significantly reducing the error in perturbation analysis.

The results of this experiment demonstrate that, even with simple assumptions regarding the execution characteristics at lower levels, the perturbation analysis can conservatively reduce perturbations using only high-level events.

5 Conclusion

Current and future performance analysis of SPMD programs requires the runtime measurement of program execution. Measurement issues concern the level of program instrumentation and the probe effect. In this paper, we have concentrated on high level source instrumentation of SPMD programs and have presented a practical approach to the systematic elimination of perturbation effects introduced by this instrumentation. The perturbation analysis techniques presented provide a means of generating a new set of events from a measured trace which more closely reflect the actual execution of the program. With the use of a few examples, we have shown that these techniques do help to reduce perturbation effects. Equally important, however, is the fact that the approach is based on high-level events.

References

[1] P. Bates and J. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," *Journal of Systems and Software*, April 1983.

[2] Z. Christidis, "Parallel Calculations on the Wind-Driven Oceanic Circulation Using Fourier Pseudospectral Methods," *Proceedings of Third Interrnational Conference on Supercomputing (ICS88)*, Boston, May 1988.

[3] T. Dunigan, "Performance of the Intel iPSC/860 and Ncube 6400 Hypercubes," *Technical Report ORNL/TM 11491*, Oak Ridge National Laboratory, 1991.

[4] G. Fox et al., "Fortran D Language Specification," *Technical Report 90-141*, Rice University, Dept. of Computer Science, 1990.

[5] L. Lamport, "How to make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers 28,9*, September 1979.

[6] J. Lee and D. Gannon, "Object Oriented Parallel Programming: Experiments and Results," *Supercomputing '91*, Albuquerque, pp. 273–282, November 1991.

[7] A. Malony, "Event Based Performance Perturbation: A Case Study," *Third ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 201–212, Apr. 1991.

[8] A. Malony and D. Reed, "Models for Performance Perturbation Analysis," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging,* Santa Cruz, May 1991.

[9] A. Malony and D. Reed, "Performance Measurement Intrusion and Perturbation Analysis,"*IEEE Transactions on Parallel and Distributed Systems*, Vol.3, 4, July 1992.

[10] S. Sarukkai, "Performance Debugging Environments for Parallel Programs," *PhD Thesis*, Indiana University, December 1992.