-- --

Multi-Application Support in a Parallel Program Performance Tool

R. Bruce Irvin
rbi@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

Program performance measurement tools have proven to be useful for tuning single, isolated, parallel and distributed applications. However, large-scale parallel machines and heterogeneous networks often do not allow for such isolated execution, much less isolated measurement. Performance measurement tools should allow users to study workload scheduling policies, resource competition among application programs, client/server interactions in distributed systems, and comparisons of application programs running on multiple hardware platforms. To enable and encourage such studies, we have extended the IPS-2 parallel program measurement tools to support the analysis of multiple applications (and multiple runs of the same application) in a single measurement session. This multi-application support allows the user to study each application as a logically separate entity, study groupings of the applications based on their physical location, or study the entire collection of applications.

We used the new multi-application support in three case studies. In these studies we examined (1) the effects of clock precision on the quality of performance data, (2) the effects of gang scheduling on competing parallel applications, and (3) the performance interaction of client processes and server in a database system. The multi-application support allowed quick comparison of different versions of a program, with a concrete visual and numeric comparison. Also, it directly showed performance dependences between parallel applications.

1. Introduction

During performance debugging, a programmer usually studies an isolated program. Isolation eliminates much of the complicated background interference that can make program performance characteristics irreproducible. However, programs are actually run in more complex environments. Large-scale parallel systems are often timeshared among a workload of application programs; in heterogeneous distributed environments, individual applications communicate with servers and contend with other clients for server access. Scheduling and contention can significantly affect the performance of individual application programs, but a programmer often cannot determine whether a program's performance is affected by such interactions or to what extent.

We have implemented a new mechanism for analyzing multiple application programs using the IPS-2 parallel program performance tools [12]. Multi-application support in the IPS-2 system has enabled

This work was supported in part by National Science Foundation grants ASC-9015462 and CCR-9100968, Office of Naval Research grant N00014-89-J-1222, and grants from Sequent Computer Systems and Sun Microsystems.

the study of application programs running in complex environments, and has opened the door to a wide range of new measurement possibilities. This paper discusses several uses of multi-application support, describes our implementation, and describes how we have used the new feature in three studies. The first study explores the problem of imprecise clocks in measurement systems, the second measures the effects of barrier synchronizations in timeshared workloads, and the third examines the performance of a client/server database system.

IPS-2 is an interactive, trace-based, post mortem performance measurement system that operates in parallel and heterogeneous distributed environments. To support multiple application programs in IPS-2, we have made some simple changes to the IPS-2 graphical user interface and have developed novel extensions to Critical Path Analysis [15]. Although the changes to our system were small, the benefit has been great. We have used the multi-application feature for studying such diverse problems as workload scheduling strategies, the effects of heavy loads on programs, operating systems, and hardware, the analysis of client/server database systems, the effects of imprecise clocks on performance measurement, the performance of programs on competing operating systems and hardware platforms, and the performance effects of best case vs. worst case application input sets.

From our experiences with multiple application analysis in IPS-2, we conclude that program performance tools can support complex test environments. IPS-2 allows programmers to run and analyze multiple programs simultaneously, enables comparison studies, and supports the study of workloads. Programmers are able to combine the performance displays and metrics of multiple applications or multiple versions of the same application to directly compare performance results. Finally, performance analysis techniques enable the study of interactions between cooperating programs or the contention of competing programs.

Section 2 lists many uses of a performance monitoring system that supports multiple applications. Section 3 gives an overview of the IPS-2 system and provides context for Section 4, which describes the changes to IPS-2 for multiple application support. Section 5 presents our performance case studies and Section 6 draws conclusions from our experiences.

2. Uses of Multiple Application Support

Although multi-application support is a simple feature, its importance becomes apparent when you consider its many uses. We have been surprised at the wide range of performance problems for which this feature has been used. This section describes the use of multi-application measurement for the analysis of multiple cooperating applications, multiple competing applications, multiple versions of the same application, and the operating system and networks underlying these applications.

IPS-2 with multiple application support allows users to study the performance of a group of applications running as a workload. Users can study the aggregate behavior of a workload, the interactions among the applications, or the performance of individual applications in the presence of other applications. The user can also study the effect of the workload on various parts of the system. IPS-2 includes a simple, open interface for the incorporation of external data from hardware, network, or operating system monitors [8]. If the computing environment already includes such performance monitors then their output can be directed into IPS-2 through the external data interface. Data gathered by external monitors may be used in the same displays and analyses as data gathered with IPS-2 program tracing, and the user can correlate system performance with workload performance. For example, procedure-level CPU metrics can be plotted alongside bus utilization, paging rates, and network traffic.

If the applications in a workload communicate using messages, semaphores, or other methods, then IPS-2 can be used to analyze individual applications and interactions between the applications. Critical Path Analysis, which analyzes process interactions, may be applied to applications in isolation, to a single application and the applications with which it interacts, or to an entire workload.

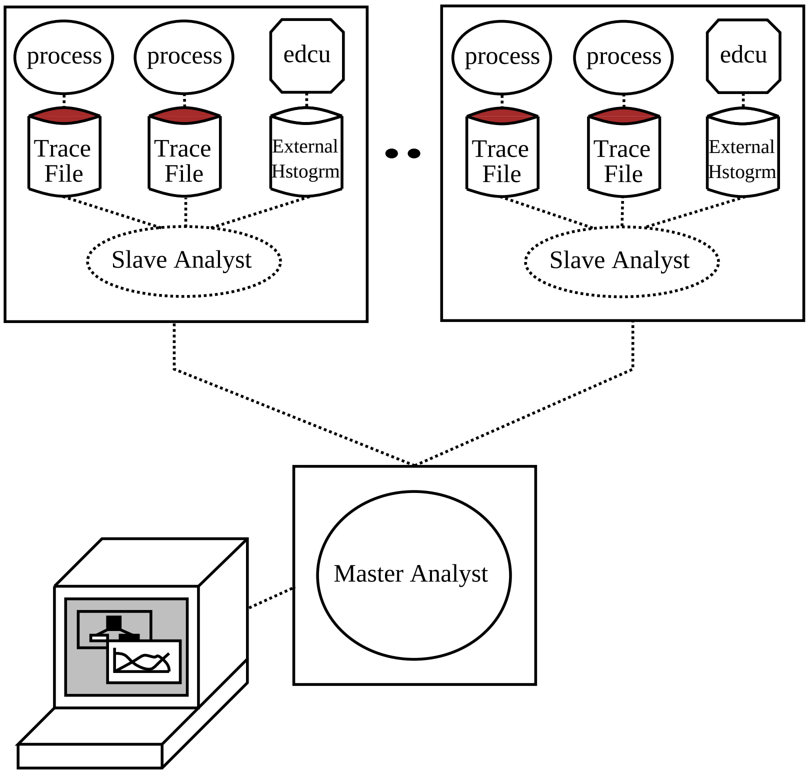
Playback of old measurement sessions is a standard feature in most performance tools. With multiple application support, IPS-2 allows multiple old sessions to be replayed in the same session, or an old session replay may be combined with a new active session. This comparison feature has many uses including the study of the evolution of a program through several versions, the changes in an application when running on new hardware platforms or operating systems, the performance of a server under various client loads, or the comparison of an algorithm running with best case vs. worst case input sets.

A user can also compare the measured performance of a program with simulations or analytical predictions. If an external simulation or analysis tool can produce IPS-2 style traces or use the external data interface, then its results can be incorporated into a session for comparison.

3. Overview of IPS-2

IPS-2 is an interactive, trace-based, post mortem performance measurement system that operates in parallel and heterogeneous distributed environments. We changed none of the basic structure of IPS-2 to support multiple applications in a single session. However, we modified the user interface to handle the new multiple application model, we added a mode to allow the comparison of old measurements with new measurements, and we extended Critical Path Analysis to enable analysis of individual programs within larger workloads. This section provides an overview of the basic structure of IPS-2 and provides context for readers who are not familiar with the system.

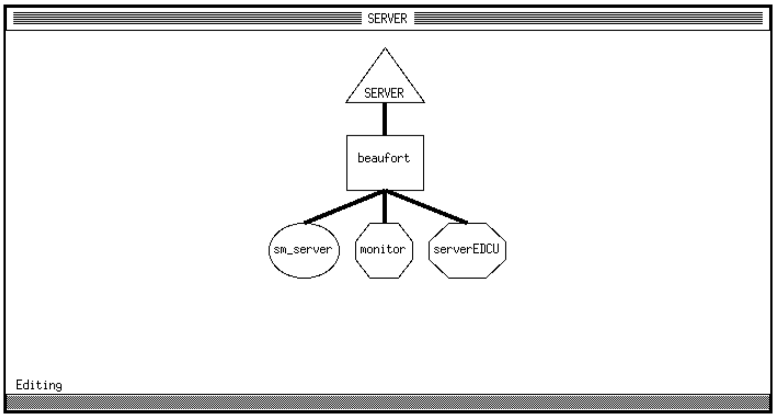
Figure 1 shows the system structure of IPS-2. IPS-2 consists of an instrumentation library that collects traces from application programs, an external data collection interface that is used by external performance monitors, Slave Analysts that collect and process trace data and external performance data, and a Master Analyst that provides a graphical interface to the user. In addition, IPS-2 provides an open interface for external visualization systems. To instrument applications for use with IPS-2, a user specifies an additional switch during program compilation. The compiler switch causes instrumentation code to be inserted automatically into the program and causes an instrumentation library to be linked with the executable.



Structural Overview of the IPS-2 System
Figure 1.

After compilation is complete, an IPS-2 user runs the Master Analyst and describes what application processes and external data collectors to run, where to run them, and what command lines to use to run them. If users wish to replay old traces, then they may give the process identifier in place of the command line. After describing the processes, the user asks the Master to run the processes or replay the old traces. The Master then starts Slave Analysts on all of the machines used by the application, and tells the Slaves which processes to run and which old traces to replay. If a particular Slave must run processes, then it waits for them to complete before processing traces. During trace processing, each Slave reduces the traces to performance metrics and reports the metrics back to the Master for display.

The IPS-2 graphical representation of parallel and distributed application programs is a tree, and the IPS-2 program tree is used both for describing the application to be run and for querying about performance information. The four levels of the tree include the program, machine, process, and procedure levels. Users select nodes in the tree to obtain performance information about a particular node. Figure 2 shows a sample program tree before execution. The triangular node at the program level represents the entire application. The rectangular node at the machine level represents a single machine. Elliptical and octagonal nodes at the process level represent application processes and external data collectors respectively. Procedures are represented with rectangular nodes, but they appear in the tree only after the application traces have been processed.



Logical Program Tree
Figure 2.

IPS-2 provides several analysis techniques, each of which can be applied to any node or level of the program tree. Critical path profiles display the elements (machines, processes, and procedures) that determine the elapsed time of a parallel or distributed program [15]. Metric tables display performance metrics for individual tree nodes, and profile tables display a metric for each node at a given level of the program tree. NPT profiles display a process time metric that is normalized by the number of concurrently executing processes [1]. Gprof tables display process and procedure performance data in the style of the Unix utility **gprof** [7]. IPS-2 allows the user to define time periods, called *phases*, and any of the metrics and analyses may be constrained to any phase of execution.

The primary method of program visualization in IPS-2 is the time histogram, which plots performance metrics over the duration of a program's execution. Time histograms are used for displaying performance curves, for defining program phases, and for guiding trace browsing displays, which provide a very low level view of program events. The system also provides a visualization interface, which allows external graphical display tools to use IPS-2 performance data.

4. Modifications to IPS-2 for the Support of Multiple Applications

The support of multiple application programs in a performance tool is a simple idea. A tool need only allow the user to open multiple views of performance information and incorporate data from a collection of measured applications. Users of traditional, single program tools can approximate this effect by simply running multiple analysis sessions at the same time.

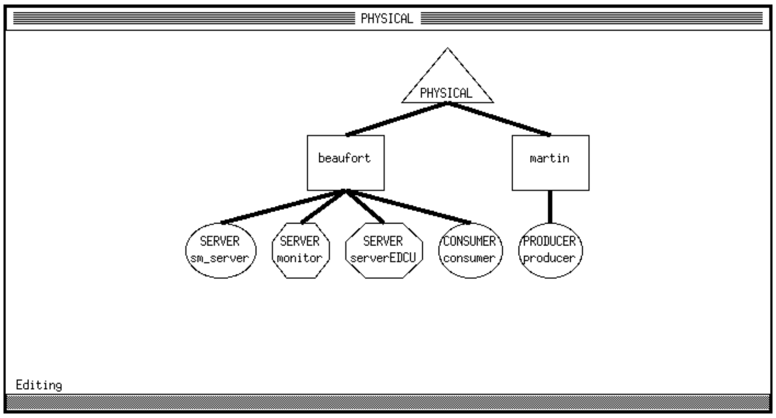
However, just as you cannot easily use a performance tool designed for single process programs to analyze a parallel program, you cannot easily use a single program performance tool to analyze multiple programs. Without specific support for multiple application programs, a tool cannot help the user make direct comparisons between applications or understand the causes or effects of contention for shared resources. Running multiple versions of a single program tool also increases demand for workstation resources such as screen space and memory.

This section explains how we have enhanced the IPS-2 performance tools to support multiple applications. The modifications include an expanded user interface, a comparison mode of operation, and extensions to Critical Path Analysis.

4.1. Multiple Applications in the IPS-2 User Interface

We have changed the IPS-2 user interface to allow the user to create multiple program trees. Each tree created by the user, called a *logical* tree, contains one application. Each logical tree is a distinct view into the performance of a workload, and nodes may be selected to obtain performance information for the corresponding application.

The IPS-2 Master automatically creates one additional tree, the *physical* tree. The physical tree duplicates all nodes in the logical trees and groups them according to their physical layout (all processes from all applications that use a given machine are grouped under the same machine node). The physical tree is used primarily for selecting performance information about the entire workload. For example, a procedure-level profile of I/O operations using the physical tree profiles all procedures in all of the trees whereas the same analysis performed in a logical tree only profiles procedures in that logical view. The physical tree is also used to isolate the performance of a particular machine. Figure 3 shows an example physical tree.



Physical Tree
Figure 3.

4.2. Comparison Mode

Previous versions of IPS-2 have allowed two modes of operation: one that actively runs the applications and one that replays old traces. We have added a new feature, called *comparison mode*, that allows old traces to be replayed and compared with either new traces or other old traces. The IPS-2 user selects comparison mode for one or more of the application trees, and then selects either active or replay mode for the other trees. Any trace that is replayed in comparison mode will have its time base shifted to match the time base of the active or replay mode programs.

Because most of the IPS-2 analyses depend on relative timestamps, the shift in time reference is unnoticeable to the user. However, the difference is apparent in time based visualization displays such as time histograms (See section 5.1) where it appears as though the comparison mode applications ran at the same time as the active and replay mode applications.

4.3. Multiple Application Critical Path Analysis

Critical Path is an analysis technique that guides the user to the sections of code in a parallel program that caused the program to run slowly [15]. Interactions between processes (e.g. messages, semaphores, barriers, and locks) form a set of dependences between the processes. Critical path analysis constructs a directed acyclic graph, called the Program Activity Graph or PAG, of these dependences (see Figure 4). Each arc in the PAG is assigned a weight proportional to the amount of time consumed between the two points on the arc. For CPU time arcs, the length of the arc is the process time consumed. For message arcs it is the time required to send the message between processes. For unproductive time such as spin time at barriers, the weight is zero. The longest time-weighted path through the PAG is the critical path. A Critical Path Profile is a profile of the procedures, processes, and machines along the critical path. At each level in the program tree, IPS-2 can profile the items that form the critical path by sorting the items by their cumulative contributions to the critical path.

This section describes how we have extended critical path analysis to support multiple application programs in a single measurement session. In particular, we discuss Intra-Application Critical Path, which allows the user to examine the critical path of one application in isolation, and Inter-Application Critical Path, which allows the user to examine a single application and its interactions with other programs. A third type of critical path analysis, Global Critical Path, computes the critical path of all of the application programs combined. The Global Critical Path is simply the Intra-Application Critical Path for the Physical Tree, and it will not be discussed further in this section.

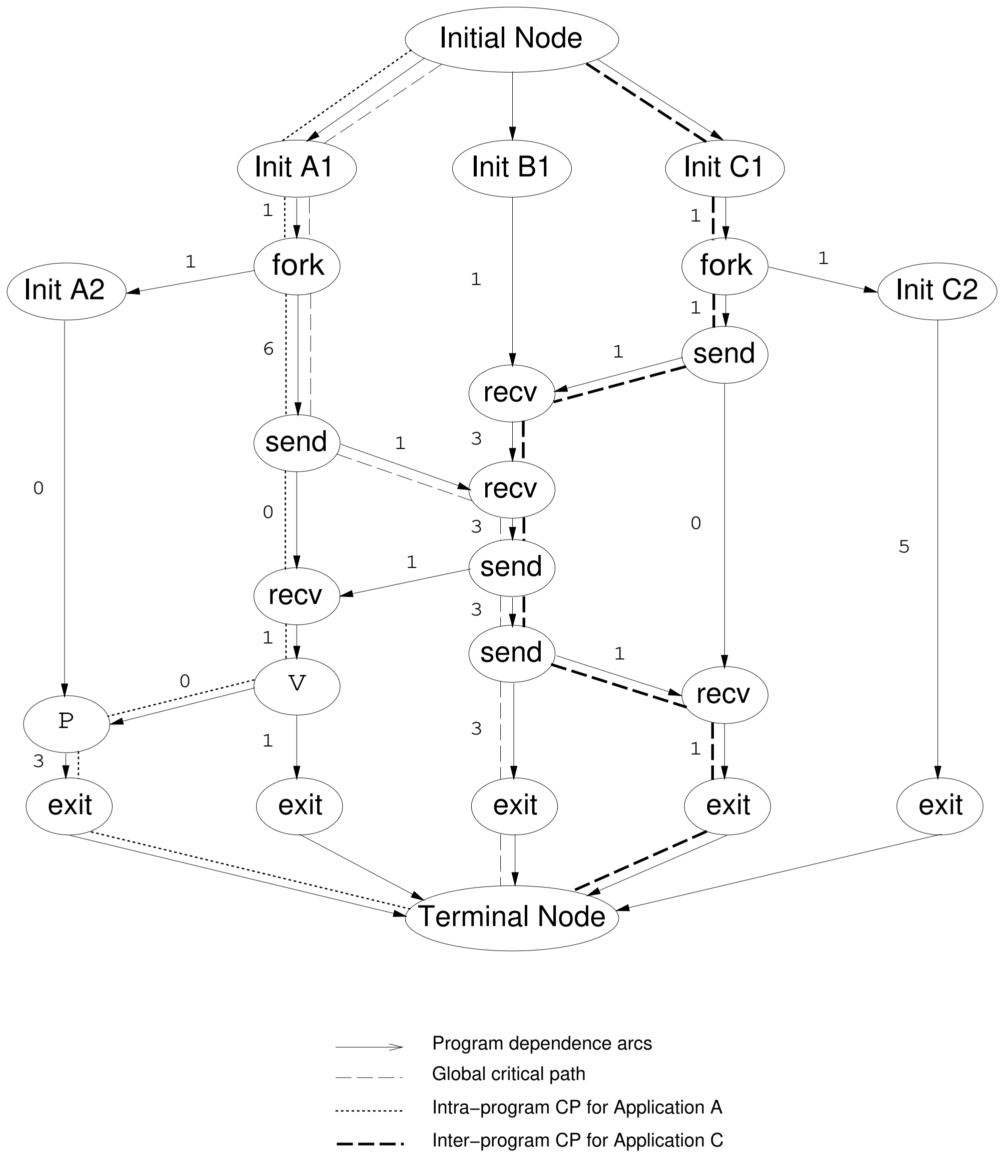
4.3.1. Intra-Application Critical Path

The Intra-Application Critical Path is the longest time-weighted path through the PAG of a single application. All inter-process dependence arcs in the application's PAG are used in calculating the critical path, but dependence arcs that lead to or from other applications are not considered. In this way, a single application can be analyzed in isolation even if it was run with other applications. Figure 4 shows a PAG of three programs. The Intra-Application Critical Path of Program A crosses process boundaries, but never crosses the application boundary.

Our implementation takes advantage of the isolation characteristics of the Intra-Application Critical Path. Since only one application is examined, we only construct the part of the overall PAG that includes the selected application. If the entire PAG has already been built for Inter-Application or Global Critical Path, then Intra-Application Critical Path simply ignores existing inter-application arcs.

4.3.2. Inter-Application Critical Path

The Inter-Application Critical Path is the longest time weighted path that begins and ends in a particular application. PAG arcs that lead to other applications may be included only if they lead to PAG arcs that return to the selected application. The Inter-Application Critical Path allows the user to determine if other applications have limited the performance of the selected application. For example, in a client/server programming model, the Inter-Application Critical Path of the client will indicate elements in both the client and the server that limit the performance of the client. If the server is on the client's Inter-Application Critical Path, then we can study the procedures in the server that executed on the client's behalf.



Sample PAG showing three types of critical path

Figure 4.

Our implementation of Inter-Application Critical Path is a modified version of the original critical path algorithm. In the original algorithm, each PAG node other than the initial and terminal nodes consists of either one or two inbound arcs and one or two outbound arcs. The initial node may have several outbound arcs and the terminal node may have several inbound arcs. The algorithm starts a forward pass from the initial node by sending a zero path length message to each of its outbound successors. Whenever a PAG node has received path length messages from each of its predecessors, it records the longest such path length, and then sends new path lengths to each of its successors. Each path length sent to a successor is the sum of the longest inbound path length and the arc length from the current node to the successor. This forward *diffusion* pass continues until the terminal node has received messages from each of its predecessors. After the forward pass is complete, each PAG node has recorded the length of the longest path to itself from the initial node and the predecessor that is the immediate neighbor along that path. The critical path is the longest path from the initial node to the terminal node. The actual nodes and arcs of the critical path are found by traversing from the terminal node backward through the PAG, always taking the predecessor arc with the greatest total path length.

The algorithm for the Inter-Application Critical Path starts two types of diffusion from the initial node: an active diffusion which starts in the application of interest and a passive diffusion which starts in all the other applications. The active diffusion always dominates the passive diffusion at nodes where the two meet, and all arcs reached by the passive diffusion are marked with zero length. The backward traversal from the terminal node will only begin with an exit node from a process in the application of interest, but otherwise the backward traversal is the same as in the original version.

Figure 4 shows the Inter-Application Critical Path of program C. We can see that part of program B is on program C's critical path, but that program A is not. However, we could argue that program A does have an effect on program C because it contends for access to program B. Inter-Application Critical Path does not take this effect into account directly.

5. Experience

This section describes the results of three performance studies that utilized the multi-application features of IPS-2. Section 5.1 presents a comparison mode analysis of a single application that was measured with process time clocks of varying precision, Section 5.2 presents a workload analysis of a program that was tuned in isolation, and Section 5.3 presents an analysis of clients and servers in a client/server data storage manager. With each study, we have included IPS-2 displays used during the analysis of the applications.

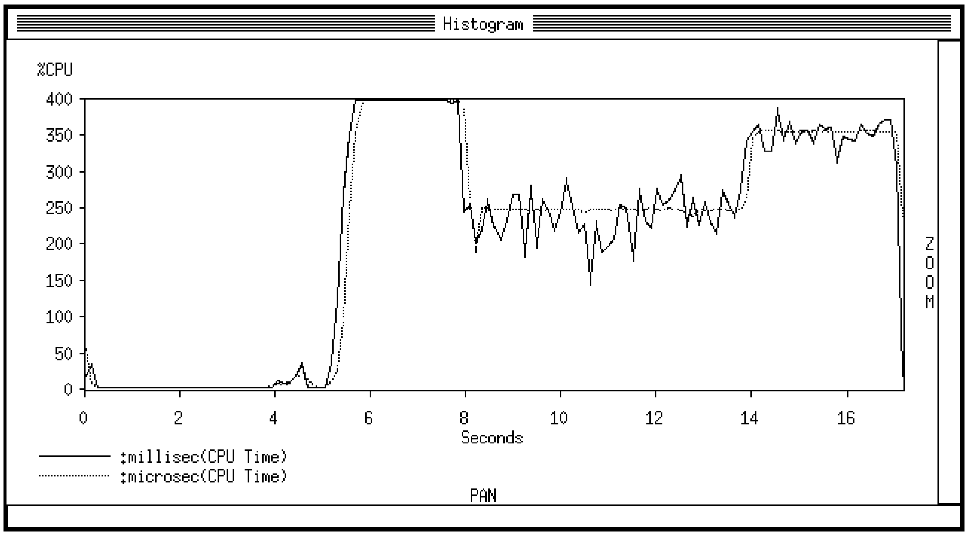
5.1. The Effect of Clock Precision on Performance Analysis

Precise measurement of time is crucial to the success of event-based performance analysis. If time measurements are not precise, then several events may have identical timestamps and event analysis can only approximate the relative costs of the activities that caused the events. If such approximations are adequate, then there is no reason to require systems to support high resolution clocks, but if we find that imprecise clocks yield errors in our analysis then we must find ways to improve our measurements. This study explores the effects of imprecise clocks on time histograms and critical path analysis.

Providing precise clocks *should not* be a problem since computers are generally synchronous devices controlled by system clocks running at very high frequencies [4]. The system clock defines the highest frequency at which events can occur, and therefore it should be possible to provide a register that is incremented each time the system clock ticks. However, most systems do not provide such high resolution clocks, and they almost never use precise clocks for process time measurements (virtual time for a process that only advances while the process is running). Process time clocks typically advance at frequencies that are 3 or more orders of magnitude slower than the system clock.

Sequent Symmetry systems supported microsecond precision clocks for wall time measurements, but only 10 millisecond precision clocks for process time measurements. We have enhanced our Sequent Symmetry's Dynix Operating System kernel to use microsecond precision counters for process time measurements. To study the effect of this enhancement on IPS-2 analyses, we used the new multi-application comparison mode to analyze measurements of a shared memory database join application made before and after the change to the process clock.

Figure 5 displays a comparison mode time histogram display of total CPU time for each run of the application. This display provides a concrete visual illustration of the effects of clock precision. Imprecise clocks introduce roundoff noise into the curves that make detailed features of the curves difficult to identify. The gross features of the two curves are roughly the same, and if gross features are sufficient for analysis, then imprecise clocks may be adequate. However, if the details are important then precise clocks are required. Otherwise, we cannot determine whether a feature appears because of a program behavior or because of measurement error.



Comparison Mode Time Histogram Display

Figure 5.

Figures 6 and 7 show procedure-level intra-application critical path profiles for the shared memory join application, one for each run of the application. For the run with the millisecond clock, the critical path profile shows procedure `partition` as the most important procedure, while the critical path profile for the run with the microsecond clock lists procedure `effect_join` at the top and procedure `partition` as second. The qualitative results in this comparison study are reproducible, so we can conclude that the clock resolution is the source of the different results.†

This type of error occurs when imprecise clocks cause critical path analysis to attribute too much weight to the program elements that are active at the moment when the process time clock advances. The procedure that is active when the clock advances will be assigned all of the time since the previous clock tick even though its actual cost may be small. To reduce such error, we need clocks that advance at a frequency closer to the maximum rate at which program events occur.

Procedure-level Intra-Application Critical Path (millisec)

machine:process	func name	time	%time
TOTAL LENGTH		11.78	
ALL:ALL	partition	3.04	25.81
ALL:ALL	effect_join	2.69	22.84
ips2:shm_join.new[2]	random_shuffle	1.69	14.35
ALL:ALL	exchange_buffers	1.45	12.31
ALL:ALL	readblk	0.62	5.26
ips2:shm_join.new[2]	init_relation	0.51	4.33
ALL:ALL	writeblk	0.39	3.31
ALL:ALL	return_buffer	0.34	2.89
ALL:ALL	get_output_buffers	0.33	2.80

Figure 6.
Critical Path with 10 msec clock.

Procedure-level Intra-Application Critical Path (microsec)

machine:process	func name	time	%time
TOTAL LENGTH		9.35	
ALL:ALL	effect_join	2.65	28.37
ALL:ALL	partition	1.89	20.24
ips2:shm_join.new[6]	random_shuffle	1.71	18.32
ALL:ALL	exchange_buffers	0.59	6.30
ips2:shm_join.new[6]	init_relation	0.48	5.15
ALL:ALL	writeblk	0.40	4.27
ALL:ALL	readblk	0.38	4.11
ALL:ALL	get_output_buffers	0.30	3.24
ALL:ALL	return_buffer	0.28	3.02

Figure 7.
Critical Path with 1 usec clock.

5.2. Scheduling, Synchronization Policies, and Workload Performance

This case study examines a widely studied application [14] that was previously tuned in isolation using IPS-2 [8]. The application, called *psim*, simulates an indirect *k*-ary, *n*-cube processor-memory interconnection network. Over the course of a simulation several memory request packets are issued from each simulated cpu. The packets travel over the request half of the network, are serviced by the memories, and then carry results over the result half of the network back to the issuing cpu. The simulator computes the

† In a different study, we verified that procedure `effect_join` is actually more important to the runtime of the application [9].

state of each network device (processor, switch, or memory) in parallel for one clock cycle and then performs a barrier synchronization before beginning the next clock cycle.

The psim program statically assigns processes to compute the states of network elements and achieves nearly linear speedup for up to about 10 processors. The greatest cost of parallelization is the time spent at barriers after each simulated clock cycle. This barrier waiting cost is highest at the beginning of the simulation when the first request packets are filling the simulated network and at the end when the last result packets are draining from the network. The simulator uses spin locks to enforce mutual exclusion on queues at each simulated device, but these locks are accessed by at most two processes and are held for very short periods of time. Therefore, lock waiting is not a significant factor in the performance of psim. The first column of the Metric Table in Figure 8 shows a summary of the performance of a single psim running in isolation.

To study the performance of psim outside of an isolated environment we used the new multi-application facility of IPS-2 to run two four-process copies of the psim application concurrently on a 4 processor Sequent Symmetry. The second copy of the application was given the same input values as the first and the two ran concurrently, competing for shared resources. The second and third columns of the Metric Table in Figure 8 summarize the performance of the two psims running concurrently. The table columns show that the elapsed time of the two concurrent psims is more than thirty times greater than the elapsed time for the run of one psim on the same machine.†

The increase in elapsed time is best explained by the difference in barrier synchronization between the isolated psim and the two concurrent psims. The average time per barrier is approximately 40 times greater when psim is run with other competing processes. The enormous increase is caused by the implementation of the barriers − each process spins until all other processes reach the barrier. In a workload environment, there is only a small probability that all of the processes in a particular application are scheduled at the same time, and a process that is busy waiting will use its entire time quantum before releasing its processor. Therefore, other processes remain blocked until the end of the time quantum before running.

The problems with this type of *always-spin* barrier are well understood [16], and several solutions have been proposed to fix them. One solution is to use barriers that block after only a small amount of spinning [3], and the other is to co-schedule the processes of each application [11]. We implemented the latter alternative and the results are summarized in the fourth and fifth columns of the Metric Table in

† The slowdown was so severe that we initially suspected a bug in the program or a crash of the system.

Metric Table	ORIGINAL	NOGANG1	NOGANG2	GANG1	GANG2
Barrier Wait Time	9.79	413.67	410.25	9.56	8.10
Barriers	1,004.00	1,004.00	1,004.00	1,004.00	1,004.00
CPU Time	9.55	9.68	9.63	9.99	10.06
Elapsed Time	8.54	213.31	213.31	14.09	13.67
Spin Locks	45,483.00	45,483.00	45,483.00	45,483.00	45,483.00
Spin Time	2.32	2.35	2.33	2.46	3.03
Time Per Barrier	0.01	0.41	0.41	0.01	0.01

A Multi-Application Metric Table
Figure 8.

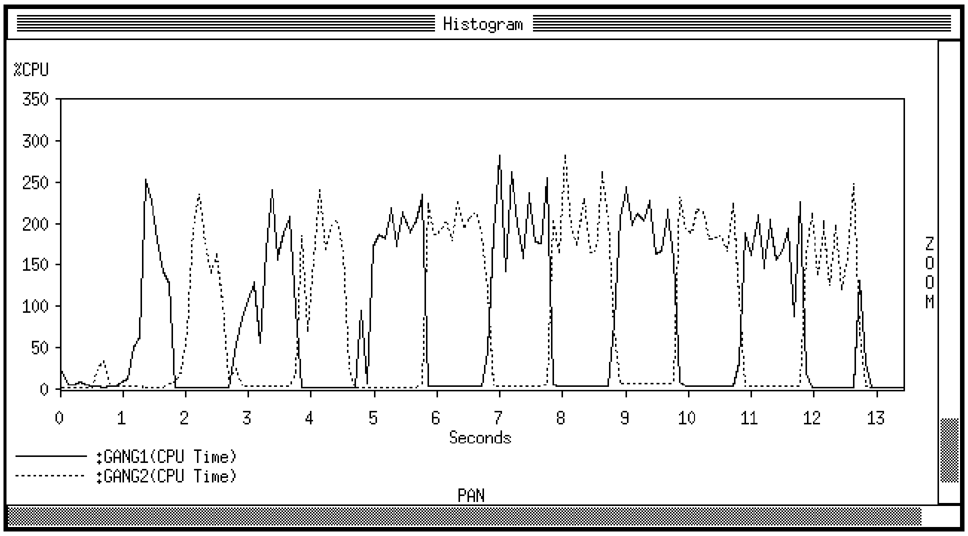
Figure 8. The results show that when all of the processes of an application are scheduled together, the cost of always-spin barriers is reduced and the elapsed time of each application is reduced substantially. The table's data also confirm the prediction [16] that waiting time at spin locks is not significantly affected by competing processes.†

Our co-scheduler is a simple server that allows processes to register themselves with an application identifier. The server then uses UNIX signals to schedule all processes with identical application identifiers at regular intervals. Figure 9 shows a Time Histogram display with CPU utilization curves of two psims co-scheduled at intervals of one second. The alternating periods of high CPU utilization indicate that excessive barrier synchronization is no longer a significant factor in the performance of the two concurrently running applications.

5.3. Client/Server Database Storage Manager

Our third case study examines the performance of the EXODUS Storage Manager, a system that supports the storage of persistent objects, files, and indices for use by database systems [5, 6]. EXODUS uses a client/server model to allow simultaneous access to objects by multiple applications in a distributed environment. The server is the main repository for objects and provides support for lock management, transaction logging, page allocation and deallocation, and recovery/rollback. The server uses a single

† Small changes in values for CPU time, Elapsed Time, Barrier time, and Spin Time due to variations in machine load are not considered significant.



Two Psims Running With a Co-Scheduler

Figure 9.

multi-threaded process to handle requests from multiple clients and uses separate disk processes to perform asynchronous I/O. A client library that is linked with each application communicates with the server, performs data and index manipulation, and manages a memory buffer pool for the client application. Application programs are either written in the E programming language [13] or call client library routines directly.

For our experiment, we used the new multi-application features of IPS-2 to analyze the server and a set of sample client applications. The client applications produce and consume objects in a database. We ran a single producer and consumer pair concurrently until they had each handled 100 4-kilobyte data objects. The server and the consumer ran on a single DECstation 3100, and the producer ran on a separate DECstation 3100 (see Figure 3). The server used several threads and spawned 2 disk processes, one for the transaction log volume, and one for the data volume used by the client applications.

We began our analysis by examining the overall performance of the EXODUS server. The process-level Intra-Application Critical Path profile in Figure 10 shows the cumulative time that each thread contributed to the isolated critical path of the server. The profile indicates that thread `sm_server[2]` was responsible for over 30% of the server's critical path with the remainder divided among the other threads, each accounting for a small percent.

After the critical path profile identified `sm_server[2]` as an important thread, we refined our analysis to the procedure level. IPS-2 gprof profiles organize the procedures of a particular process or thread into a hierarchical dynamic call graph format. We used IPS-2 gprof to analyze `sm_server[2]` from the main procedure down to the procedures that accounted for most of the thread's CPU time. IPS-2

Process-level Intra-Application Critical Path (SERVER)

statistic	time	%time
TOTAL LENGTH	18.65	
CPU: beaufort:sm_server[2]	6.29	33.72
CPU: beaufort:sm_server[13]	1.25	6.70
CPU: beaufort:sm_server[3]	0.93	4.96
CPU: beaufort:sm_server[17]	0.75	4.00
CPU: beaufort:sm_server[19]	0.66	3.54
CPU: beaufort:sm_server[10]	0.65	3.48
CPU: beaufort:sm_server[7]	0.63	3.37
CPU: beaufort:sm_server[11]	0.61	3.29
CPU: beaufort:sm_server[18]	0.59	3.16
CPU: beaufort:sm_server[15]	0.57	3.08
CPU: beaufort:sm_server[0]	0.54	2.87

Figure 10.
Critical Path of Server

Gprof Table - SERVER:beaufort:sm_server[2]:openLogDisk

parents <cycle> / name / children <cycle>	%time	self	descendents	called/total / called+self / called/total
main		0.00	5.20	1/1
openLogDisk	82.53	0.00	5.20	1
regenLog		1.80	3.39	1/1
io_MountVolume		0.00	0.01	1/2
openLogFile		0.00	0.00	1/1
linkVolume		0.00	0.00	1/2

Figure 12.
Gprof Profile of Top Server Thread

Process-level Inter-Application Critical Path (PRODUCER)

statistic	time	%time
TOTAL LENGTH	23.81	
CPU: martin:producer	7.75	32.57
CPU: beaufort:consumer	5.34	22.41
Msg: martin:producer->beaufort:sm_server[2]	1.42	5.95
CPU: beaufort:sm_server[2]	0.89	3.72
CPU: beaufort:sm_server[13]	0.77	3.25
CPU: beaufort:sm_server[19]	0.44	1.85
Fork: beaufort:sm_server[2]->beaufort:diskrw[1]	0.41	1.74
CPU: beaufort:sm_server[7]	0.38	1.61
CPU: beaufort:sm_server[14]	0.37	1.54
CPU: beaufort:sm_server[11]	0.36	1.53
CPU: beaufort:sm_server[10]	0.32	1.36

Figure 11.
Critical Path of Producer Client

Process-level Inter-Application Critical Path (CONSUMER)

statistic	time	%time
TOTAL LENGTH	22.68	
CPU: beaufort:consumer	8.21	36.19
CPU: martin:producer	5.12	22.57
Msg: martin:producer->beaufort:sm_server[2]	0.87	3.82
CPU: beaufort:sm_server[13]	0.83	3.69
CPU: beaufort:sm_server[2]	0.69	3.03
CPU: beaufort:sm_server[7]	0.43	1.91
CPU: beaufort:sm_server[14]	0.39	1.74
CPU: beaufort:sm_server[15]	0.35	1.53
CPU: beaufort:sm_server[20]	0.32	1.43
CPU: beaufort:sm_server[11]	0.32	1.41
CPU: beaufort:sm_server[17]	0.32	1.41

Figure 13.
Critical Path of Consumer Client

gprof also lists the total CPU time for a thread, and in this case, the thread's total CPU time was equal to the thread's contribution to the Intra-Application Critical Path. Therefore, all of the thread's CPU activity was on the server's critical path. Figure 12 shows the gprof entry for `openLogDisk`, the descendant of `main` whose CPU time (along with its descendants' CPU time) accounted for over 80% of `sm_server[2]`'s CPU time. The `openLogDisk` procedure is an initialization routine that normally accounts for only a small amount of time, but since our experiment was run on a uninitialized EXODUS server, a significant amount of processing is spent regenerating the transaction log. This cost is listed in Figure 12 as CPU time for the procedure `regenLog`.

Given the initial understanding of the server's performance, it is useful to understand the client activity that caused this performance. With multi-application IPS-2 we can shift our view to other parts of the system. In particular, we can examine the client applications and their interactions with the server. Figure 11 shows a process-level Inter-Application Critical Path profile of the producer client. The profile shows that the producer is responsible for only about one third of its own critical path, with the remainder distributed among the consumer client, server threads, and message delays. Even though the producer and consumer do not communicate with each other directly, they still can appear on each other's Inter-Application Critical Path because server threads doing work on behalf of clients must wait for one another inside of the server. The profile also lists the time spent forking the disk process that handled the client data volume.

It is interesting to see that message delays from the producer client to the server are responsible for a noticeable portion of the producer's Inter-Application Critical Path, but that message delays back to the producer are not listed. The imbalance appears because a single server thread (`sm_server[2]`) handled all message receives from the producer, while replies to the producer were performed by many threads. Critical Path Analysis considers inter-thread queuing dependences while calculating the critical path, so if one thread receives a request from a client and assigns another thread to service the request, the critical path may follow this dependence. Therefore, the critical path time for message receives is concentrated in one thread while the time for reply arcs is spread among several threads, and no single thread has enough to reach the top of the profile. We verified this analysis with a machine level Inter-Application Critical Path profile of the producer (not shown). The critical path profile at the machine level lists message delays from the producer's machine to the server's machine that are equal to the message delays from the server's machine to the producer's machine.

The consumer client's Inter-Application Critical Path profile, shown in Figure 13, was similar to that of the producer. Again, the consumer itself was responsible for only about one third of its own critical path. The profile also shows that message delays from the producer to the server were significant, indicating that the consumer's performance was limited by the performance of the producer. This is consistent with observations made during execution: even though the consumer started later than the producer, it eventually caught up and waited for the producer to produce new data objects. The consumer's critical path does not include the fork of any disk processes because the consumer accessed the same data volume as the producer, and only one disk process is forked per open volume.

Process-level analysis has given us a structured view of the relationships between the client applications and the server. To continue our study of the system we could refine our view of specific threads and processes with procedure-level critical path and profile analyses. Procedure-level analyses identify specific procedures to be tuned and have led to performance improvements during past studies [8, 9]. However, for the present study we were primarily interested in identifying which applications affected each other, and process-level analysis was sufficient.

6. Summary

The support of multiple applications in a parallel program performance tool is a simple idea that is easy to implement, yet is not supported in current parallel performance tools. We have found that this feature has opened the door to a wide range of interesting performance studies, and we have demonstrated its use in a variety of measurement experiments.

Our experiences with multi-application IPS-2 have allowed us to make a number of discoveries about parallel and distributed programs. In our first case study, comparison mode allowed us to demonstrate that imprecise clocks can lead to noisy visualizations as well as misleading performance analyses. In our second case study, we used logical views to analyze two parallel programs competing for CPU resources. The study showed that co-scheduling can have an enormous effect on the performance of competing parallel programs that perform barrier synchronizations. In our third study, we used Intra-Application Critical Path Analysis to isolate a server's performance in a client/server database storage manager. We then used Inter-Application Critical Path Analysis to demonstrate how unrelated client applications can affect each other and the server in the database system. In each case, the ability to analyze multiple applications in a single session allowed us to examine the applications in ways that were not previously possible.

To accommodate multiple application programs in a single IPS-2 session, we have added logical and physical program views. Logical views allow the user to isolate analysis to a particular application without running the application in isolation. The physical view allows the user to study the performance of an entire workload. We feel that this organization is well suited to the types of analysis supported in IPS-2 and other tools that encourage a hierarchical top-down performance analysis methodology [2, 10]. Most of the IPS-2 displays and analyses have remained unchanged, but we have made two novel extensions to critical path analysis. Intra-Application Critical Path allows the user to isolate the performance of a single application, and Inter-Application Critical Path allows the user to study a single application and other applications that may have limited its performance.

When measuring programs in complex environments, there are other important keys to success besides supporting multiple applications in a single measurement session. For example, the applications measured in our studies used complex programming facilities such as signals, threads, shared file descriptors, asynchronous I/O, dedicated I/O processes, and connectionless inter-process communication. Correctly handling such facilities is worth the effort if we can measure interesting applications and learn more about the true nature of parallel and distributed program performance.

7. Acknowledgements

We wish to thank Mike Zwilling and Nancy Hall for their help with the Exodus Storage Manager, Joann Ordille for authoring the shared memory database join application, and Eugene Brooks for authoring the psim simulator. We also thank Jon Cargille, Jeff Hollingsworth, Krishna Kunchithapadam, and Christopher Maguire for their comments and suggestions on improving this paper.

References

1. T. E. Anderson and E. D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance", *SIGMETRICS 1990*, Boston, May 1990, pp. 115-125.
2. T. Bemmerl, "The TOPSYS Architecture", *CONPAR 1990*, 1990, pp. 732-743.
3. B. Bershad, E. D. Lazowska and H. Levy, "Presto: A System for Object-Oriented Parallel Programming", *Software: Practice and Experience 18*, 8 (August 1988), pp. 713-732.
4. D. Black, "The Mach Timing Facility: An Implementation of Accurate Low-Overhead Usage Timing", *USENIX Mach Workshop*, October 4-5, 1990, pp. 53-71.
5. M. Carey, D. DeWitt, J. Richardson and E. Shekita, "Object and File Management in the EXODUS Extensible Database System", *Proc. of the 1986 VLDB Conference* , Kyoto, Japan, August 1986.
6. M. Carey, D. DeWitt and E. Shekita, Storage Management for Objects in EXODUS, in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky (ed.), Addison-Wesley, 1989.
7. S. L. Graham, P. B. Kessler and M. K. McKusick, "gprof: a Call Graph Execution Profiler", *SIGPLAN '82 Symposium on Compiler Construction*, Boston, June 1982, pp. 120-126.
8. J. K. Hollingsworth, R. B. Irvin and B. P. Miller, "The Integration of Application and System Based Metrics in A Parallel Program Performance Tool", *1991 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, April 1991, pp. 189-200.
9. J. K. Hollingsworth and B. P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation", *Supercomputing '92*, November 1992.
10. T. J. LeBlanc, J. M. Mellor-Crummey and R. J. Fowler, *Analyzing Parallel Program Executions Using Multiple Views*, Journal of Parallel and Distributed Computing, 1990.
11. S. T. Leutenegger and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", *Proc. 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1990, pp. 226-236.
12. B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), pp. 206-217.
13. J. Richardson and M. Carey, "Persistence in the E Language: Issues and Implementation", *Software Practice and Experience 19*(December 1989), .
14. S. S. Thakkar, Performance of Parallel Applications on a Shared-Memory Multiprocessor System, in *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela (ed.), ACM Press, 1990, 233-256.
15. C. Yang and B. P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs", *8th Int'l Conf. on Distributed Computing Systems*, San Jose, Calif., June 1988, pp. 366-375.
16. J. Zahorjan, E. D. Lazowska and D. L. Eager, *The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors*, University of Washington Technical Report 89-07-03, July 1989.