1994

Study of Intrusion of a Software Trace System in the Execution of Parallel Programs

Kuei Yu Wang

Report Number:

94-078

STUDY OF INTRUSION OF A SOFTWARE TRACE SYSTEM IN THE EXECUTION OF PARALLEL PROGRAMS

Kuei Yu Wang

Department of Computer Science
Purdue University
West Lafayette, IN 47907

CSD-TR-94-078
November 1994

Study of intrusion of a software trace system in the execution of parallel programs

Kuei Yu Wang

November 17, 1994

Contents

1 Statement of the problem

Any measurement disturbs the physical process being observed. This is a fundamental law of physics (see Heesenberg's uncertainty principle) and monitoring the execution of a parallel program cannot be made totally unobtrusive even when specialized hardware support is available.

In this report, we describe a monitoring system based on software probes inserted into a parallel program. During the execution of the instrumented program trace data associated with every event is collected, stored in some internal buffers and written to an external storage device in a trace file when these buffers are filled. The level of intrusion of this monitoring system depends upon the overhead of gathering the data associated with each event and the amount of data. An intrusive system will increase drastically the execution tme of the parallel program and will alter the behavior of the parallel programs we want to observe.

In our system as in many other cases, a significant level of intrusion is caused by the act of storing the trace buffers on some external device, because (a) the limited I/O bandwidth of parallel systems, (b) the high latency of I/O operations, and (c) the contention for I/O devices and the interconnects. Several choice to reduce the level of the intrusion, by selecting the size for the trace buffers, by choosing the mechanism for transporting the trace buffers to the I/O devices are discussed in this paper.

2 The trace system

The trace system is an event-driven *parallel profiling library* [2] which monitors and profiles the execution of parallel programs on the Paragon^{TM} XP/S system running the OSF/1 Mach Operating System. The parallel profiling library is linked to the program to be profiled. During the execution of the program, trace records are generated by each node program according to the dynamics of the program. A trace record is produced at each state switching of a node program (e.g. from compute state to communication state). The profiling library is also able to generate trace records at regular intervals – the regular sampling approach. The amount of trace records generated during a parallel program profiling depends on the number of nodes running the program and on the pattern of state changes of each node program.

In the initial design of the profiling library, each node program creates its own trace file and writes to it whenever a trace record is generated. The trace files are created in the current directory where the parallel program is running. Several performance degradation factors have been detected.

1. The virtually simultaneous and concurrent "opens" of trace files (one for each node) are serialized by the operating system. The overhead increases as the number of nodes running the program increases.

2. Similar problem happens to the concurrent writes. The I/O node is overloaded with simultaneous requests from every node. Although the programs are mostly running on SPMD mode, the state switchings are expected to happen at distinct but similar instants.

   The effect of I/O node contention is noticeable (increasing the execution time by more than 100%) for parallel programs running on 64 nodes or more.

3 The environment

A distributed memory MIMD system like the Intel Paragon [1] consists of a number of *compute*, *I/O* and *service* nodes connected by interconnection network, the *interconnect.*

The *I/O* nodes manage the system's disk and tape drives, network connections, and other I/O facilities. Processes on *compute* nodes access the I/O facilities using standard OSF/1 system calls, just as if they were directly connected to the I/O facilities.

The Paragon OSF/1 operating system supports three file system types:

UFS UNIX File System, the standard file system type for OSF/1.

NFS Network File System, a file system type that represents a file system on another computer on the network.

PFS Parallel File System, a file system that is optimized for access by parallel processes. PFS file systems provide file services at high transfer rates to parallel applications by striping file data across multiple I/O nodes.

The performance of a disk I/O operation depends on the type of file system used. The UFS and NFS provide standard Unix interface suitable for applications with moderate demands for I/O operations. Accessing UFS is quicker than accessing NFS, as the former corresponds to the local file system and the latter to a file system on another computer. The Parallel File System, PFS, is tuned to favor parallel and simultaneous accesses of large amount of data from disks (large files usually bigger than 2 GBytes and disk transfer blocks in the range of MBytes).

4 Implementations of the Parallel Profiling Library

Several approaches for optimizing the performance of a parallel profiler have been implemented. The goal is to decrease the intrusion by the parallel library so the data collected during the profiling reflect accurately the real execution of the program as it was executed without profiling. Two fronts of optimization are done: a) to decrease the additional time introduced by profiling library I/O operations in the execution time, b) to avoid significant changes on the memory reference patterns of a parallel application during the profiling process.

The naive implementation is a profiling library without any I/O optimization (*NOPT*), in which the node programs write performance data into their own trace file as the data are produced. The second approach, *BF*, attempts to minimize the number of request for disk I/O operation by buffering trace records and them sending to the disk as the buffers become full.

The third approach, *HN*, adopts the host/node programming paradigm in which there is a *host process* in the service node responsible for doing all the I/O operations. The host and node programs communicate with each other using the message passing mechanism.

The fourth approach, *MP*, uses multiple processes per node. In each compute node there exist two processes: the *compute* process is responsible for doing the real application job and the *trace-writer* process is responsible for doing the I/O operations produced by the execution of profiling on that node. The trace records generated by a *compute* process are sent, using message passing routines, to the *trace-writer* process in the same node and then the *trace-writer* writes the data into disk. A variant for the multiple processes approach uses *shared memory*, instead of message passing, to exchange information between the *compute* process and the *trace-writer* process.

Also the Parallel I/O (*PIO*) approach, a variant of *BF*, writes trace data to the Parallel File System (*PFS*) using double buffering mechanism and asynchronous I/O operations.

In the following subsections, we discuss the issues involved in each approach.

4.1 Profiling without optimization – NOPT

No attempt to decrease the overhead of disk I/O is done in this approach (Figure 1.) The standard Unix file system library routines, such as *fopen* and *fwrite* (buffered output), are used in this version.

This approach imposes less alteration in the memory access behavior of a parallel application because besides the memory allocated for the application itself, only some additional

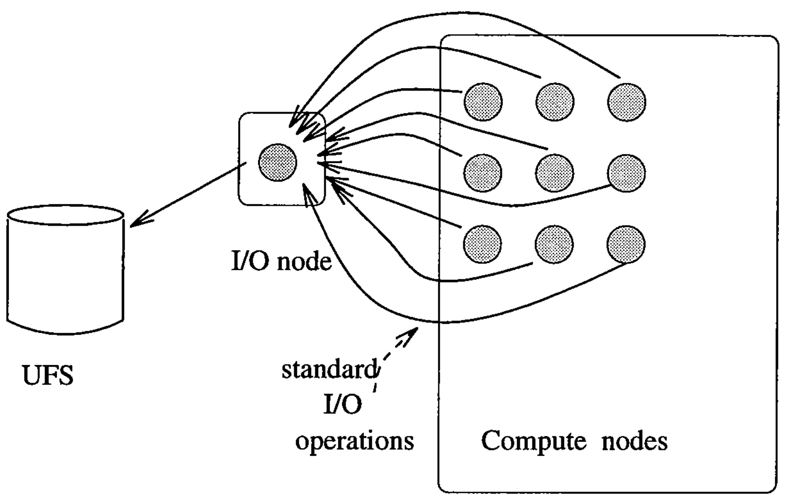

Figure 1: Profiling parallel programs without library optimization

memory is used for the profiling. The overhead in execution time of a parallel program profiling is mainly due to the large number of small I/O requests issued by all nodes at the same time. The concurrent accesses to the I/O node slow down the performance of I/O operations. This is a *"many-to-one"* (Compute nodes to I/O node) problem.

4.2 Profiling using I/O buffers – BF

In this approach, we attempt to minimize the number of requests to disk I/O operations by buffering the trace data in the node program memory (Figure 2.) The larger is the size of buffer the smaller is the number of I/O operations needed for a certain profiling process.

Significant improvements in the performance are observed when the number of I/O requests is drastically reduced (probably to one request per node at the end of profiling, for large number of parallel compute nodes). But, when we increase the buffer size to decrease the number of I/O requests, we increase the perturbation on the memory access pattern of the application, which is highly undesirable for tracing the paging behavior of parallel programs.

4.3 Profiling using the host/node programming paradigm – HN

In this approach, the compute nodes do not perform any I/O operation related to the profiling. Instead, the compute nodes send trace records to a *host* process in the service

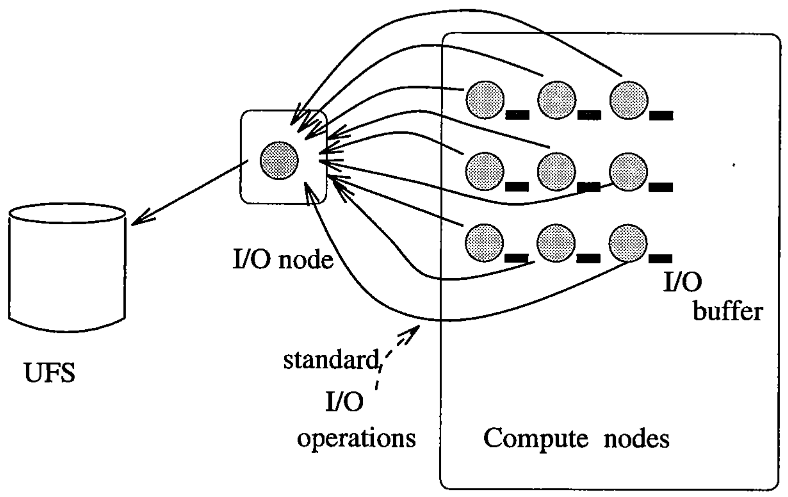

Figure 2: Profiling parallel programs using I/O buffers at each compute node.

node using the synchronous message passing mechanism[1]. The *host* process is responsible for I/O's caused by profiling, and one global trace output file is produced (Figure 3.) The message buffer in each node program consists of 120 trace records (8160 bytes).

The "many-to-one" problem now depends on the message passing protocol and latency between *compute* processes and *host* process. This approach will be better when the message passing overhead among nodes is smaller than the I/O overhead between compute nodes and I/O nodes.

A message buffer is still needed to improve the performance of the profiling, although the buffer size is smaller compared with the I/O buffers. This approach presents the advantage over the previous ones (*NOPT* and *BF*) by not blocking the application because of the I/O operations at a smaller cost of memory used for communication buffers. Since the host and node programs communicate with synchronous message passing mechanism, there are still blockings caused by the message passing mechanism. The disadvantage of a *host* program on service node is the load in the service nodes is usually heavy. Programs in service node time slice via standard UNIX scheduling; the *host* program competes for CPU cycles with all other service users and processes and this may cause delay/blocking in communicating the *compute* programs and the *host* program.

[1]To use asynchronous message passing mechanism, we need some scheme for double buffering asynchronous messages, which is not implemented in this version.

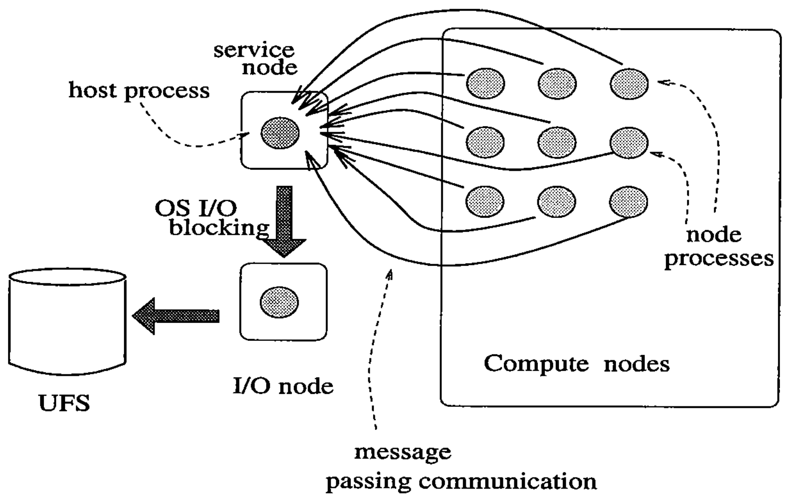

Figure 3: Profiling using the host/node programming paradigm.

4.4 Profiling using multiple processes per node – MP

On each compute node there exist two processes: one for executing the parallel application code (the *compute* process) and the other (the *trace-writer* process) responsible for sending the performance data to the file system. The *trace-writer* process is responsible for all I/O operations due to the profiling (e.g. open the trace file, write the trace data and close the trace file).

This approach has the advantage of releasing the compute process from executing profile-related I/O (*NOPT*, *BF*) and a central server is not needed, in contrast to the host/node programming paradigm (*HN*.) The communication between a *compute-writer* pair is done through IPC (Inter Process Communication) mechanisms. One problem with multiprocessing on compute nodes is the possible load unbalancing and also the introduction of synchronization problems to parallel applications. For a parallel application with fine grain synchronization, one *compute* process participating the synchronization procedure may not be scheduled because of the context switching between *compute* process and *trace-writer* process on the same node.

Two alternatives have been studied, the first is based on message passing mechanism (*MP/MP*) and the second is based on shared memory and semaphores (*MP/SM*).

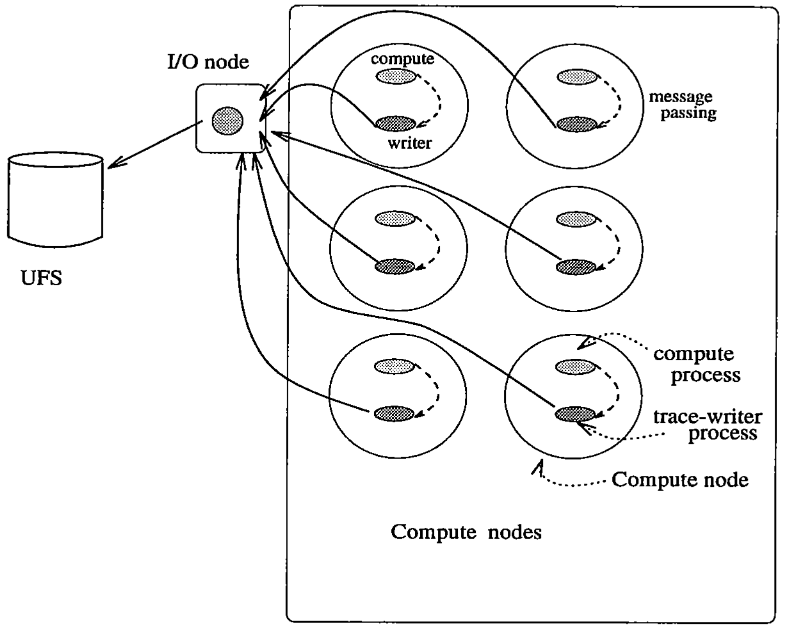

Figure 4: Profiling using multiple processes per node: message passing mechanism.

4.4.1 IPC - message passing (MP/MP)

Here the *compute* and *trace-writer* processes exchange data using a 8 KBytes message buffer (Figure 4.)

The problem appears when trying to exchange messages between processes in the same node. All the compute nodes are running a "normal" Unix scheduler, in which processes time share the CPU with time slice of 100 milliseconds (100 Hz). The library routines seem to spin wait on message activity, so they do not relinquish control to the other processes on the node often. Tests have shown that exchanging messages among processes in different nodes is faster than exchanging messages among processes in the same node. Some stratagems, such as adding *flick()* and or *swtch()* to relinquish the control of the CPU to the next scheduled process to the same node, have shown some performance improvements when compared with the pure message exchange model between *compute* and *trace-writer* processes in the same node.

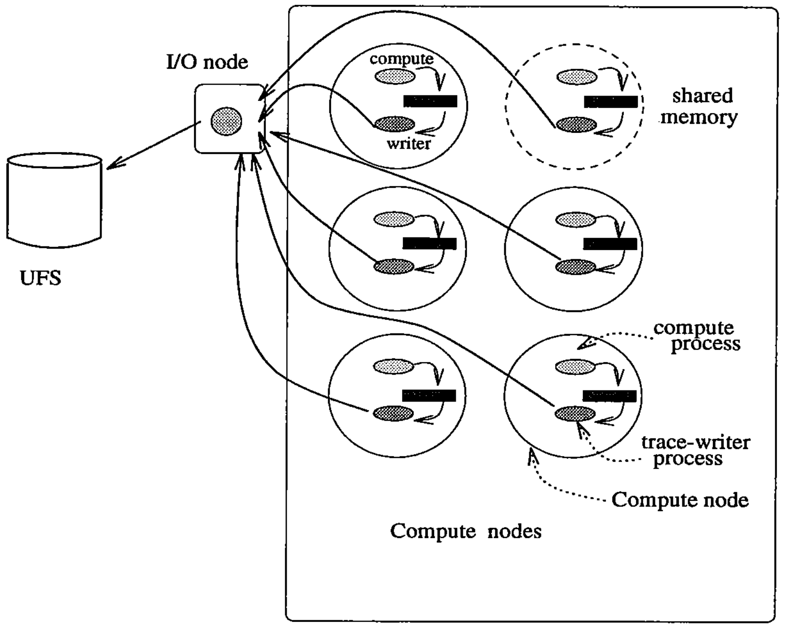

Figure 5: Profiling using multiple processes per node: shared memory mechanism.

4.4.2 IPC - shared memory (MP/SM)

Another way for exchanging data between *compute* process and *trace-writer* process is the communication through a *shared memory* area. A *share memory* area is defined for each *compute-writer* pair and *semaphores* are used to protect the concurrent accesses to the shared area (Figure 5.)

This version has been designed and implemented but the performance results are not available since *semaphore* and *shared memory* were not fully supported on the Paragon at the time of experiments.

4.5 Writing in PFS using double buffering and asynchronous I/O – PIO

The Parallel File System optimizes the access to large amount of data. In this approach, we used asynchronous parallel I/O operations and double I/O buffers of size 64 KBytes each and 128 KBytes each (Figure 6.) The objective of double buffering is to overlap compute and asynchronous I/O operations.

There are several modes to access a parallel file in PFS. We used *M_LOG* mode: the

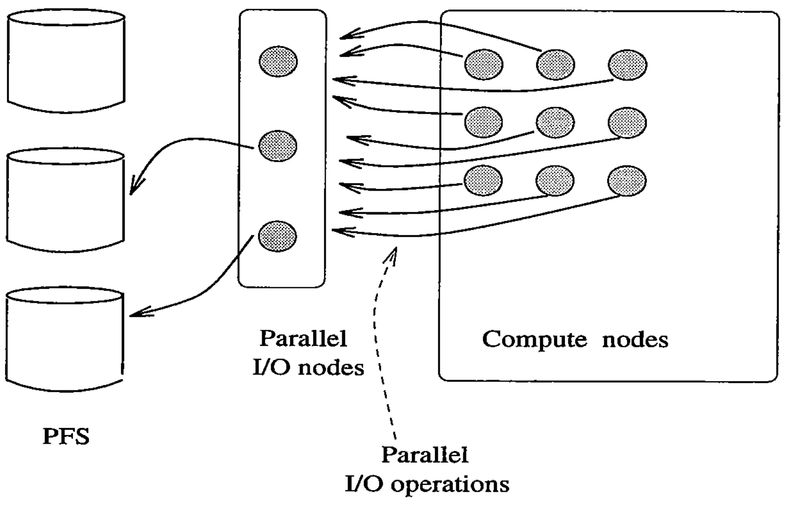

Figure 6: Writing trace files in PFS.

nodes share the file pointer, the accesses are serialized to prevent nodes from overwriting each other; there is no coordination of nodes and the operations are asynchronous. One trace file is used to store all trace data from the nodes. Two types of open operation have been used, in the first approach each process opens the file asynchronously and in the second a global open, *gopen*[2], is performed by all nodes at the same time.

The benefits of PFS are not fully exploited in our implementation because of the size of the trace files produced. The use of double buffers for asynchronous I/O's improves the performance of access to the disk, but the same problem of memory perturbation occurs as the size of buffers increases. Thus the trade-off is between the size of the buffers and the efficiency of the I/O operations. To achieve the best performance of PFS each request for write should be in the range of MBytes. Since in most of the profiling cases the trace files for each node have size about MBytes, and not GBytes, the writing of profiling data cannot benefit from parallel file operations.

4.6 General issues

The following have to be considered:

- Trace record size - the amount of information needed for analysis. It is preferable to be power of 2 which facilitates, so we could pack a number of trace records in buffers

[2]A global synchronization operation which all nodes in the application have to participate.

with the same size as the system I/O buffer size.

- I/O buffer size - should be tuned to the size (or to the multiple of it) used by the Operating System to improve the performance of I/O operations. Non-aligned requests cause fragmentation of the same into several I/O operations.

- Message buffer size - it also has to be tuned and properly aligned.

- Message passing synchronization - the sender and receiver should be "synchronized"; the *recv* should be posted before a message is sent to avoid performance degradation, as explained below.

  The Paragon message passing mechanism uses a two-level buffering scheme at destination process: a user's receive buffer and a system's temporary buffer *MBF*. When a message is sent, if the destination process has called irecv, crecv or hrecv, the data goes straight into the user's receive buffer, which is fast; otherwise, the data goes into the temporary buffer and is copied when the destination process calls receive.

- Double buffering - it allows proceeding with the computation while performing an asynchronous operation such as asynchronous message passing routines and asynchronous I/O operations (compute-I/O overlap).

5 Execution time of NAS Integer Sort (IS) Benchmark: Class A and Class B

The NAS Integer Sort (IS) benchmark tests a sorting operation that is important in "particle method" codes [3]. This type of application is similar to "particle in cell" applications of physics, wherein particles are assigned to cells and may drift out. The sorting operation is used to reassign particles to the appropriate cells.

This benchmark tests both integer computation speed and communication performance. Although floating point arithmetic is not involved, significant data communication is required.

5.1 Experiments for both class A and class B

The class A and class B experiments differ from each other in the size of elements (integers) sorted. The class A experiments sort an array of 524288 (2^{19}) integers and the class B experiments sort an array of 2097152 (2^{21}) integers.

5.2 Output trace files in UNIX File System (UFS)

1. Benchmark without profiling – base line for the execution time;
2. Profiling without I/O buffering;
3. Profiling using I/O buffers - variants:
    - Profiling using a buffer of size 69632 bytes (1024 trace records) for I/O operations related to profiling;
    - Profiling using a buffer of size 139264 bytes (2048 trace records) for I/O operations;
    - Profiling using a buffer of size 348160 bytes (5120 trace records) for I/O operations;
    - Profiling using a buffer of size 609200 bytes (8960 trace records) for I/O operations;
4. Profiling using the host/node programming paradigm
5. Profiling using multiple processes per node
    (a) in-node communication mechanism: synchronous message passing
    (b) in-node communication mechanism: shared memory

5.3 Output trace files in Parallel File System (PFS)

1. Output trace files (one for each node) in PFS.

    Use asynchronous writes and two write buffers (double buffers) of size 65280 bytes each.
2. Output trace files (one for each node) in PFS.

    Use asynchronous writes and two write buffers (double buffers) of size 130560 bytes each.
3. Output one global trace file to PFS.

    Use *"gopen()"* (synchronized open) to open the output file in M_LOG mode and double buffers of 130560 bytes each.

num proc trace file size (bytes/node)	IS	NOPT	BF - buffer size				HN
			69k	139k	348k	609k	(8k msg buffer)
CLASS A							
8 (39508)	29.1	29.29 (0.7%)					
16 (72148)	14.80	15.23 (2.9%)	15.37 (3.9%)	14.97 (1.1%)	14.95 (1.0%)	14.92 (0.8%)	
32 (137428)	8.17	12.04 (47.4%)	9.15 (12.0%)	8.24 (0.9%)	8.22 (0.6%)	8.23 (0.7%)	9.38 (14.6%)
64 (267988)	4.88	46.68 (956%)	32.20 (660%)	12.80 (262%)	4.91 (0.6%)	4.93 (1.0%)	11.30 (232%)
128 (529108)	2.88	211.87 (7356%)	195.92 (6802%)	151.89 (5274%)	106.68 (3704%)	3.40 (18.0%)	110.57 (3839%)
CLASS B							
32 (137428)	32.32	35.70 (10.5%)	33.48 (3.6%)	32.51 (0.6%)	32.25 (-0.2%)	34.45 (6.6%)	32.82 (-0.2%)
64 (267988)	18.62	60.94 (327%)	48.27 (259%)	25.73 (138.2%)	18.57 (-0.3%)	18.67 (0.3%)	35.09 (88.5%)
128 (529108)	10.46	222.60 (2128%)	196.07 (1874%)	153.44 (1467%)	115.37 (1103%)	10.98 (5.0%)	127.36 (1218%)

Table 1: Unix File System - Execution time (in seconds) of Class A and Class B experiments and the percentage of increase compared with the baseline execution time. Each node program writes to its own trace files. Trace files are in the Unix File System. The *IS* column reports the execution time of the Integer Sort without any instrumentation (the baseline execution time). There is no buffering of the trace data in the *NOPT* mode. The message size in the *HN* mode is 8kbytes.

load size	IS	NOPT	BF - buffer size				HN
			69k	139k	348k	609k	(8k msg buffer)
.text	173920	182880	183168	183168	183168	183168	183264
.data	24704	25504	25536	25536	25536	25536	25568
.bss	161088	161728	231392	301024	509920	771040	169920
total	359712	370112	440096	509728	718624	979744	378752

Table 2: Load size - Class A and Class B experiment's load sizes: (.text) + (.data) + (.bss). Section sizes of Paragon OSF/1 operating system object files.

num proc trace file size (bytes/node)	IS	**PIO** - double buffers		
		$2 * 65k$	$2 * 130k$	gopen $2 * 130k$
CLASS A				
32 (137428)	8.17	8.51 (4.2%)	8.53 (4.4%)	9.05 (10.8%)
64 (267988)	4.88	9.46 (93.8%)	5.83 (19.5%)	5.87 (20.3%)
128 (529108)	2.88	38.20 (1336%)	19.23 (667.7%)	17.99 (624.7%)
CLASS B				
32 (137428)	32.32	32.61 (0.9%)	32.60 (0.9%)	38.41 (18.8%)
64 (267988)	18.62	22.13 (18.8%)	19.42 (4.3%)	19.51 (4.8%)
128 (529108)	10.46	42.51 (40.6%)	20.92 (2.0%)	23.01 (220%)

Table 3: Parallel I/O approach - Execution time (in seconds) of Class A and Class B experiments and the percentage of increase compared with the baseline execution time. In the buffered versions ("$2 * 65k$" and "$2 * 130k$") each node program writes to its own trace file in PFS and in the *gopen* version all the node programs share the same trace file in PFS.

num proc (trace file size)	IS	MP multiple processes
CLASS A		
32	8.17	9.19 (12.5%)
64	4.88	5.41 (10.9%)
128	2.88	3.72 (29.2%)
CLASS B		
32	32.32	35.18 (8.8%)
64	18.62	19.53 (4.9%)
128	10.46	11.37 (8.7%)

Table 4: Mutiple processes per node approach. Execution time (in seconds) of Class A and Class B experiments and the percentage of increase compared with the baseline execution time. There is one *"trace-writer"* process per node which is responsible for handling all I/O requests caused by the trace system generated in that node. Each node has it own trace file.

6 Analysis of results

The *IS* benchmark is suitable for studying the overhead of profiling library because of the communication pattern among node programs; during the sorting computation, each node exchanges data with all other nodes. The increase in number of nodes executing *IS* program causes more message traffics to be generated (an *n-to-n communication* pattern) and therefore, the number of trace records generated per node increases proportionally to the number of nodes. The effect of congestion in I/O system is evidenced and worsen when *IS* is executed in large number of nodes.

Among all the profiling approaches we implemented the best result, in term of profiling execution time, was the *BF* approach with buffer size about the size of individual trace file (Table 1.)

The *BF* approach with a large enough buffer means that no trace data is written to the permanent storage (disk) during the program profiling step. This approach presents two major drawbacks:

1. The size of trace output has to be predicted before doing the "best profiling", then a buffer which is at least as large as the largest trace file must be used.
2. A large buffer may affect the virtual memory reference pattern of the application being profiled.

The purpose of *HN* approach is to release the compute processes from doing I/O related to profiling and thus, the need of buffering at each compute nodes as in *BF* approach. Although the *HN* approach would have solved the intrusion in memory behavior present in *BF*, it did not show much gain in the execution time of program profilings (see Table 1.) The overhead in the execution time is mainly because of the heavy load of service nodes in a Paragon system.

Output data into Parallel File System (*PIO* approach) using a double buffering scheme is slightly better than *BF* approach using the same size of buffers (compare Table 1 and Table 3), but the execution time is still dilated because of the I/O node contention. In the first variant of *PIO*; each node program outputs trace data to it own trace file, each node produces traces at approximately same pace (thus, the sizes of trace files are about the same) and all I/O requests fall into the same I/O node to be handled (it happens because of the implementation of disk stripping polity on the current version of Paragon.) In the other variant, all node programs output data to a global trace file, the I/O requests are serialized by the operating system.

The *MP/MP* approach presented good results despite the time sharing and scheduling scheme at each compute node level. In this approach, we have achieved a small increase in the execution time, ranging from 4.9% to 29.2% (Table 4: Class B, 64 nodes and Class A, 128 nodes), using a small buffer size (8 KBytes).

The most important issue on parallel program profiling is the accuracy of collected trace data. Every software profiling library introduces perturbation in some extent. Although the *BF* profiling approach with large I/O buffer did not slow down the program execution, it is extremely intrusive because it may change the memory reference behavior of the program under examination. The best approach so far, is the *MP/MP* profiling approach which has shown a small overhead introduced in the execution time and little perturbation in the memory access patterns (because of the small buffer used by profiling library) in each node program.

References

[1] Intel Corporation, Paragon^{TM} OSF/1 User's Guide, Inter Supercomputer Systems Division, Beaverton, Oregon, 1993.

[2] K.Y. Wang and D.C.Marinescu, "Correlation of the paging activity of the individual node programs in the SPMD execution mode". Proceedings of the Hawaii International Conference on System Sciences, Jan. 1995 (in press)

[3] D. Bailey, E.Barszcz, J.Barton, D.Browning, R.Carter, L.Dagum, R.Fatoohi, S.Fineberg, P.Frederickson, T.Lasinski, R.Schreiber, H.Simon, V.Venkatakrishnan and S.Weeratunga "The NAS Parallel Benchmarks". RNR Technical Report RNR-94-007, March 1994.