A Performance Monitoring Environment and its Use for the Study of Paging and I/O Activity of Parallel Programs

Kuei Yu Wang

Dan C. Marinescu

Report Number:
96-014
A PERFORMANCE MONITORING ENVIRONMENT
AND ITS USE FOR THE STUDY OF
PAGING AND I/O ACTIVITY OF PARALLEL PROGRAMS

Kuei Yu Wang
Dan C. Marinescu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR-96-014
February 1996
A Performance Monitoring Environment 
and its Use for the Study of 
Paging and I/O Activity of Parallel Programs 

Kuei Yu Wang and Dan C. Marinescu 
Computer Sciences Department 
Purdue University 
West Lafayette, IN 47907-1398 

Abstract 

We present a suite of tools - the Musketeers - for monitoring and analysis of paging, I/O and communication activity of parallel programs. The tools support automatic instrumentation and allow the user to select the statements to be instrumented and to define the format of a trace record, and to include data from kernel structures and from the user’s address space. 

We discuss a methodology to correlate the activity of processes running concurrently, regardless of the size of the process group. Such a correlation is necessary because I/O and paging devices, as well as the interconnection network of an MPP, are shared resources subject to the load generated by all processes running concurrently. In addition to finding the load placed upon system resources, the methodology is useful for studying means to improve the I/O throughput - by buffering I/O requests directed to the same device within a certain time window and transferring large I/O data blocks from the I/O device - or for studying scheduling. 

Finally, we present results of measurements of paging and I/O activity for several parallel programs. 

Keywords: performance measurement, monitoring tools, performance evaluation methodologies, demand paging, I/O, gang scheduling, parallel and distributed programming, MIMD systems
1 Introduction

High performance computing requires powerful computing systems and programs capable of exploiting concurrency. Massively Parallel Processing systems, MPPs, and workstations interconnected by very high speed networks provide the hardware support for scientific and commercial applications of high performance computing. Yet, designing a program which exploits concurrency is a difficult task which might require: development of entirely new algorithms, some understanding of the architecture of the target system and of issues like load balancing, parallel communication, parallel input-output, and last but not least, a careful analysis of the performance of the concurrent program. While performance and scalability are the crucial aspects in the development of scientific and engineering applications, reliability, fault tolerance, security, and maintainability are equally important for industrial and commercial applications of parallel and distributed computing.

Probably the most difficult aspects of performance tuning of a parallel program are those related to communication and I/O. Parallel systems are typically I/O bound for most of their applications, since parallel programs process very large amounts of data and the CPU is many orders of magnitude faster than the I/O. Parallel access to the data, data buffering, and hiding the high latency of I/O operations are rather elusive tasks even for the most experienced designers of concurrent applications. Virtual memory support is a relatively new feature of MPPs and its uninhibited use may contribute significantly to the degradation of performance of a parallel system. Performance monitoring tools to study communication, I/O and paging activity of parallel programs are thus needed not only for program optimization but also for the design of new systems.

The evolution of performance monitoring and analysis tools for parallel and distributed computing reflects the status of the field as a whole, namely incremental progress and desire to provide simple and effective solutions to rather complex and difficult problems. The designers of such tools are faced with delicate decisions concerning architectures, programming languages and operating systems to be supported, in addition to generic requirements as smooth learning curve, ease of use, generality, and flexibility. Other challenges in designing performance monitoring and analysis tools for parallel and distributed system are: guiding the user in the instrumentation process, collecting data during parallel program execution with minimal intrusion [19], data reduction and analysis, and the ability to present the results in a meaningful way.
A large number of performance monitoring and analysis tools exist, yet in this paper we survey only a few: Pablo, a large scale effort undertaken by Reed and his co-workers at University of Illinois, Paradyn built by Miller and his group at University of Wisconsin, the ParaGraph and PICL systems developed at Oak Ridge by Heath and Geist, and the AIMS Toolkit built by Yan at NASA Ames Research Center. Pablo [15, 22] is a performance analysis environment supporting data capture, analysis, and presentation across a variety of platforms and programming languages. Pablo includes software performance instrumentation, graphical performance data reduction and analysis, and support for mapping performance data to both graphics and sound. The performance tools are built from individual modules that can be interconnected and reconfigured. Paradyn [14, 17] takes a different approach, a dynamic instrumentation, to build scalable tools for studying long-running parallel programs. Dynamic instrumentation allows instructions for data collection to be inserted at run time into an application. ParaGraph [12, 13] is a general-purpose graphics system for visualizing the behavior and performance of parallel programs on message-passing multi-computer architectures. ParaGraph can be combined with PICL, Portable Instrumented Communication Library [11], which supports portability and instrumentation. AIMS, discussed in [28], is a comprehensive set of performance tools for parallel message passing programs and it supports source code instrumentation, run-time monitoring, graphical profiles and automated modeling techniques.

The operating system support for demand paging in MPPs and the realization that, for many applications, existing MPPs are I/O bound created the need to gather data covering the paging and I/O behavior of parallel programs. Burger et al. in [7] report on the behavior of some shared memory parallel programs on a simulated distributed shared memory machine, arguing that demand paging has limited value on current parallel machines because of the synchronization and memory reference patterns of parallel applications and the high page-fault and context switching overheads of parallel machines.

In this paper we report on Musketeers, a suite of tools designed specifically to monitor the paging, I/O and communication activity of parallel programs running on distributed memory MIMD systems. At a time when performance monitoring tools are increasingly integrated with operating systems – e.g. the NT Performance Monitor [6] – and with the compilers, we have built an environment for performance monitoring that can be ported with relative ease to very different systems. One may argue that most of the ideas which make the Musketeers flexible and easy to port, such as the use of templates to define the statements to be instrumented and to describe
kernel data structures, the generation of dynamic libraries to support a user defined format for the trace records, the use of the Tcl and Tk [21] for creating the GUI, can be encountered in some system designed in the past.

We built the Musketeers because the learning curve of most of the existing tools is rather steep and none of them allowed us to study the paging activity of parallel programs. We developed a methodology to correlate the activity of processes running concurrently, regardless of the size of the partition. Such a correlation is necessary because I/O and paging devices, as well as the interconnection network of an MPP, are shared resources subject to the load generated by all processes running concurrently. First, for each process, we determine the event rate for the performance measure under scrutiny, then we isolate the peaks of activity. Finally, we correlate the intensity and the time of occurrence of the peaks across processes. This methodology can be used to study paging, I/O and communication activity of parallel programs. In addition to finding the load placed upon system resources, the methodology is useful for studying means to improve the I/O throughput - by buffering I/O requests directed to the same device within a certain time window and transferring large I/O data blocks from the I/O device - or for studying scheduling.

The information we gathered from the study of the paging activity of our own programs and of the NAS benchmark suite confirms that in case of gang scheduling or co-scheduling of tasks on MPPs, [1], it is not feasible to hide the high latency of a page fault by a context switch of the process group. A process group consists of all processes which need to be scheduled concurrently. The working set of a process group consists of the processes which need to communicate among themselves and need to be active at the same time. Our measurements show that the timing and the intensity of the peaks of paging events of the processes in the process group are very dissimilar even when the Same Program Multiple Data, SPMD, paradigm is used. More studies of the paging activity of the working set of a process group are needed to understand the behavior of parallel programs and to find better ways to use the CPU cycles of a parallel system.

The tools have also helped us understand how to better implement a shared virtual memory needed for our applications and how to improve the performance of a 3-D FFT program. The lesson we learned was to reduce the number of I/O requests by reading large blocks of data and distribute the data over the entire set of PEs or cache them on a few data server nodes with large internal memory.
Section 2 of the paper presents an overview of the Musketeers, discusses environment customization, monitoring intrusion and alternatives for minimizing perturbation. Then it describes data analysis modules for generic event and state information, and analysis of paging and I/O activity. Finally, it covers porting the tools to the NT environment. In Section 3, we describe a set of programs used for solving very large problems in structural biology, namely the phase refinement and extension of electron density maps of large viruses [9], and present the result of measurements. In the same section, we analyze programs developed by others and widely distributed as the NAS benchmark suite [3, 4].

2 The Musketeers

Our goal in designing the Musketeers was to create simple tools for system architects and more sophisticated users of parallel machines. To build cost effective parallel machines, system architects need to study paging, I/O, and communication behavior of a variety of parallel programs with little knowledge of how the programs are organized and operate. The sophisticated users of parallel systems wish to improve the performance of their programs without the need to learn the more subtle aspects of the architectural design of a specific system, or of the operating system. Moreover, they desire to tune the performance of their programs on machines with different architectures, operating systems, and communication libraries.

To support flexibility, we provide a tool to define the instrumentation environment. The tool uses a set of generic templates containing system-dependent information about communication and I/O libraries, the arguments for each system call, as well as a set of kernel structures used to store performance data by various operating systems. The user of the tool creates customized templates which contain only the statements to be instrumented and defines the data to be collected for every event with a specific operating system and communication library in mind. One of the tools presented supports automatic instrumentation. The end-user chooses the set of statements to be instrumented and then the tool carries out the instrumentation process with little or no intervention.

The monitoring process in Musketeers is based upon a simple execution model of a parallel program running on a distributed memory MIMD system. Such a program consists of a set of tasks running concurrently, one or possibly multiple tasks per node. Each task can be either active
or suspended. Each active task can be in one of three possible states: Compute – the task executes its own code and issues system calls other than those related to I/O and communication, I/O – the task has invoked an I/O system call and is waiting for its completion, and Communication – the task has invoked a communication system call and is waiting for its completion. Our monitoring system detects events caused by the transition from one state to another and allows the user to decide on actions to be taken for each such event.

The Musketeers suite is composed of four performance tools, Athas, which supports the definition of the instrumentation environment, Porthos for creation of program package and instrumentation, Aramis for execution of the instrumented programs and data collection, and D’Artagnan for trace data analysis. The graphical user interface (GUI) based on the X Window system with Motif-like user interface is built using the Tcl and Tk [21] software packages. To allow manipulation of data stored on different systems and to run on heterogeneous environments, the Musketeers suite uses the Bond system. Bond [24] is a parallel virtual environment which allows the execution of parallel and sequential programs on sequential machines, clusters of workstations, and massively parallel systems. Each tool of the Musketeers suite manipulates different types of data: Athas deals with system data structures and statement templates needed to create the instrumented code, Porthos with application’s source files and D’Artagnan with trace data. By using the Bond environment, Athas and Porthos could run on the front-end of the target system, Aramis on the target parallel system, and finally D’Artagnan may analyze the trace data on a fast sequential machine. The structure of the Musketeers is discussed below.

Athas. Athas is responsible for creating a customized instrumentation environment for the users of the Musketeers. It allows a user to define a set of statements to be instrumented, then to specify the format of the trace record for each of the selected statements. After the selection process, Athas generates a dynamic trace library with one entry for each selected statement. In addition to the default non-uniform sampling mode, when trace records are generated at every transition from one state to another, Athas supports uniform sampling, where trace records are generated at pre-defined intervals, as well as mixed mode sampling, where trace records are generated at state transitions and at predefined intervals.

Athas supports probing. In this mode the dynamic trace library provides modules which only accumulate counters for the selected events without generating trace records. By using probing and knowing the size of the trace record associated with each event, Athas is able to estimate the total amount of data generated by every instrumented statement. The information
collected during probing can be used to reduce the amount of data collected during the program monitoring to a minimum and to guarantee the least intrusive monitoring experiment [26] capable of collecting all relevant data needed for the performance analysis. The format of the trace files is consistent with the Self Defining Data Format (SDDF), [2]. By generating trace records in SDDF we can use data processing and visualization facilities provided by Pablo, and also be able to translate the data into other formats, e.g. PICAL, and then use ParaGraph for data visualization.

**Porthos.** This tool supports instrumentation of a variety of source code on different systems. To instrument a parallel application, the user first defines a program package — a catalog of source files of a parallel program — and then applies the instrumentation tool on the source files of the program package. The instrumentation takes place when the list of selected statements produced by Athos is applied on the source files of the program package and the statements found in the source code are substituted by their corresponding instrumentation primitives. In the *guided instrumentation mode*, Porthos allows the user to inspect the instrumented code and select/discard substitutions. In the *automatic instrumentation mode* all matched statements are substituted by their corresponding instrumentation primitives.

**Aramis.** Aramis is responsible for executing the instrumented program and collecting the trace data. The instrumented user program and the trace library are compiled and linked forming a traceable program which produces customized trace records. In a distributed memory MIMD system, a parallel program consists of a set of node programs, one for each processing element (PE). At execution time each instrumented node program generates trace records and writes them into a separate trace file.

**D'Artagnan.** D'Artagnan consists of a set of trace data analysis packages. At this time we provide one module for generic event and state analysis, two methodologies for the study of the paging activity of parallel programs — the *skyline analysis* and the *cumulative analysis* — as well as modules for the analysis of the I/O activity.

### 2.1 Environment Customization

Athos allows the user to select a set of events to be monitored and to define the data to be collected when such events occur during the program execution. Different instrumentation environments are created for different purposes, e.g. an environment for message passing studies defines only message passing events. The steps for customizing the instrumentation environment are: (1)
statement selection, (2) definition of the trace record format, and (3) generation of the dynamic trace library.

The input to the statement selection process is a Generic Statements Template, GST, containing communication and I/O primitives as well as user-defined statements. The output is a Customized Statements Template, CST, containing the list of selected statements. For example, the GST file may contain MPI, PVM, NX, and native communication and I/O primitives, while the CST file may contain only the asynchronous read/write, iread/iwrite statements.

Athos uses a Data Structure Template, DST - a list of kernel and user defined data structures - and information about the parameters of each statement or system call provided by GST, to specify the structure of performance data records associated with every statement selected. Examples of OSF/Mach kernel data structures are vm_statistics for virtual memory activity counter, tbl_diskinfo for disk drive activity, and tbl_sysinfo for process manager information.

The user is able to select specific items from kernel and user-defined data structures available on the target system, as well as parameters of the system call.

The Dynamic Trace Library, DTL is a collection of routines called by the instrumented code during the execution of the instrumented program. It consists of a Customized Trace Library, CTL, and a System Dependent Library, SDL. The first, CTL, is generated using information gathered during the trace record definition phase, consisting of instrumentation primitives and routines for handling customized trace records and their descriptors. The second, SDL, contains generic routines for manipulating trace records, retrieving kernel data and controlling the writing of trace records to the secondary storage; SDL is specific to the target system and independent of the format of the trace record. Several trace libraries may be generated, for example the first one to study the paging activities in the compute state and the second one to study the I/O activity in all three states.

2.2 Monitoring and Intrusion

Monitoring the execution of a parallel program requires some transformation of the original source code into an instrumented code. Though most operating systems provide some monitoring support, only seldom the information they provide is sufficient to understand the behavior of a program and improve its performance. Specialized tools like Aramis are used to perform such transformations of the source code into an instrumented code. Sometimes a compiler may
support the instrumentation of a program.

One of the most serious concerns when instrumenting a program is the level of intrusion of the monitoring process. Any measuring system perturbs the system being observed to some degree [19]. Hardware probes are the least intrusive but their use is limited by cost and scope of the monitoring process; usually they deliver low-level information with limited or no context. Software probes are the most intrusive but their use is widespread, sometimes in conjunction with some hardware support.

Yet another form of intrusion which affects the data obtained through monitoring is due to the interactions of the program being monitored with programs running concurrently in other partitions of the system. This type of interactions may be due to contention for the I/O and communication resources, and they always lead to an increase of the execution time. Even when a program is not being instrumented, its execution time may vary significantly for the same input data, depending upon the activities of other programs competing for system resources. While the first type of intrusion, that due to instrumentation, cannot be eliminated entirely, there are costly ways to eliminate the second type, by using a system in a single user mode.

The “probe effects” measure how the instrumentation alters the behavior of the original program. The “instrumentation mechanism” influences how the annotations alter the program and what they can deliver as raw information [5, 16, 23]. By modifying the instrumentation mechanism, we focus on two fronts for minimizing the probe effects: (a) decrease the overhead in the total execution time caused by the I/O operations of trace library, and (b) avoid significant changes on the memory reference patterns of a parallel application during the monitoring procedure.

In this section, we discuss the implementation of several variations of the Aramis parallel software profiler which runs on the Intel Paragon XP/S system. Such a distributed memory MIMD system consists of a number of compute, I/O, and service nodes connected by an interconnection network. The I/O nodes control access to the systems disks, tape drives, network connections, and other I/O devices. Processes on compute nodes access the I/O facilities using standard system calls, just as if they were directly connected to the I/O facilities, but the requests are routed to the I/O nodes. The service nodes execute users’ shells, system commands, and other non-parallel programs.

The level of intrusion of a monitoring system based on software probes (like Aramis) depends
upon the overhead for gathering the data associated with each event, as well as the overhead for storing the trace data on external storage. Trace data associated with events are collected, stored in internal buffers by the compute nodes during the execution of the instrumented code, and then written to an external storage in a trace file. A summary of the optimization strategies for minimizing the intrusion of the monitoring process in Aramis follows. The objective of the techniques described below is to overlap communication and I/O operations related to the writing of trace data with the actual execution of the original code by the compute nodes. An in depth discussion of our studies can be found in [26].

**NOPT – no optimization.** This method is based upon a naive implementation of SDL, the system dependent trace libraries. The buffer size equals the size of the largest trace record, and a buffer is written to the external storage immediately after being filled in when an event occurs. The execution time of the instrumented program increases dramatically with the increase in the number of nodes and the event rate generated by each node. The event rate is determined by the placement of the instrumented statements. Instrumenting a statement in an inner loop can be disastrous.

When an instrumented code runs in a partition with a large number of compute nodes, and each node has a high rate of write requests, each for a relatively small amount of data, the contention for the I/O nodes compounded with the inefficiency of writing short records leads to a significant degradation of performance. The ratio of compute to I/O nodes of an MPP is usually large (10:1 or larger) and the optimal I/O performance is obtained for very large blocks.

**BIO – Buffered I/O.** This method attempts to minimize the number of I/O requests by buffering the trace data in the node program memory. The larger is the size of the buffer the fewer I/O operations are needed. If the buffer size equals the size of the trace file then data can be written after the program completes its execution. If enough memory is available the execution time of the instrumented code is only slightly larger than that of the original code. One could use probing to estimate the size of the trace file, hence the size of the buffer.

Defining very large buffers affects adversely the execution time of the instrumented code because the amount of real memory available to the program decreases and the paging rate increases. Recall that the virtual address space of the instrumented code consists of the original code augmented with buffers space and access routines for the instrumentation. The BIO strategy

---

1 Recall that the size of trace records is variable; it is determined by the user defined format for each event.
used double buffering and asynchronous I/O operations.

**SNPW – Service Node Writer Process.** In this mode the compute nodes do not perform any I/O operation related to performance monitoring. The message passing mechanism for inter-node communication is used to send the trace records to a writer process running on a service node. We use synchronous communication between the client process running in the compute node and the server process running on the service node. The writer process in the service node provides services to all clients running in the compute nodes. It uses synchronous I/O operations.

Inter-node communication is considerably faster than I/O, and the amount of blocking experienced by a compute node is reduced significantly. A communication buffer in the compute node is still needed to improve the performance, although the buffer size is smaller compared with the BIO case. A drawback of SNWP is that the server process runs on heavily loaded service nodes. The service node uses standard UNIX scheduling, and the writer process competes for the CPU cycles with all other processes running on behalf of other users of the system.

**MP – Multiple Processes in the compute nodes.** In MP mode there are two processes on each compute node: one for executing the parallel application code (the *compute* process) and the other (the *writer* process) responsible for sending the performance data to the file system. In this mode we remove the bottleneck created in the SNWP mode, when all compute nodes use the same server, by migrating the writer process from the service node to the individual compute nodes. This approach has the advantage of releasing the compute process from executing profile-related I/O as in the case of NOPT and BIO modes. The disadvantage is that the compute process has to share the CPU cycles with writer process. The communication between a compute-writer pair is done through inter process communication mechanisms. Two MP alternatives have been studied, the first based on message passing mechanism (MP/MP) and the second based on shared memory and semaphores (MP/SM). This technique is well suited for the new generation of Paragon systems with two compute and one communication processors per node.

We use the NAS Integer Sort (IS) benchmark [3, 4] to compare the approaches presented above, see Figure 1 for execution time and Table 1 for load size of IS with different trace libraries. The IS benchmark is suitable for studying the overhead of trace library because of the communication pattern among the compute nodes. The algorithm requires global exchanges; each compute node must exchange data with all other nodes. The number of trace records generated per compute node increases proportionally to the number of nodes when communication
Figure 1: The effect of the optimization level of the system-dependent trace library upon the execution time of an instrumented program. Five options are considered: no optimization, buffered IO with buffer size 68KB and 600KB, writer process running in a service node or running in the compute node on a Paragon XP/S system are compared with the execution time of the uninstrumented code. The experiment uses the Integer Sort NAS benchmark program to sort arrays of $2^{19}$ (Class A) and $2^{21}$ (Class B) integers. The level of intrusion of the monitoring process increases with the number of nodes. The best results are obtained with a server writer process running in the compute node, the method used in Aramis.

events are traced. The effect of overloading the I/O sub-system is evident when IS is executed in a large number of nodes.

The best results in terms of monitoring execution time correspond to the BIO approach with buffer size about the size of the individual trace file (ranging from 40 kbytes in an 8 node execution to more than 500 kbytes per node in a 128 node execution). The drawbacks of BIO include (a) the need for predicting the size of trace and (b) large buffer size affecting the virtual memory reference pattern of the application being profiled. The optimal tradeoff between memory usage

<table>
<thead>
<tr>
<th>load size</th>
<th>IS</th>
<th>NOPT</th>
<th>BIO 68KB</th>
<th>BIO 600KB</th>
<th>SNWP 8KB msg</th>
<th>MP/MP compute</th>
<th>MP/MP writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>173920</td>
<td>182880</td>
<td>183168</td>
<td>183168</td>
<td>183264</td>
<td>200160</td>
<td>165504</td>
</tr>
<tr>
<td>.data</td>
<td>24704</td>
<td>25504</td>
<td>25536</td>
<td>25536</td>
<td>25568</td>
<td>25152</td>
<td>22560</td>
</tr>
<tr>
<td>.bss</td>
<td>161088</td>
<td>161728</td>
<td>231392</td>
<td>771040</td>
<td>169920</td>
<td>166080</td>
<td>165504</td>
</tr>
<tr>
<td>total</td>
<td>359712</td>
<td>370112</td>
<td>441236</td>
<td>975744</td>
<td>378752</td>
<td>391392</td>
<td>353568</td>
</tr>
</tbody>
</table>

Table 1: The size of the Integer Sort NAS benchmark object file with different trace libraries.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of Nodes</th>
<th>Execution Time (in seconds)</th>
<th>Trace File Size per Node (bytes)</th>
<th>Execution Time (in seconds) with profiling</th>
<th>Trace File Size per Node (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embarrassingly Parallel Class A</td>
<td>32</td>
<td>21.16</td>
<td>1.84 K</td>
<td>21.29</td>
<td>1.84 K</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>10.45</td>
<td>1.70 K</td>
<td>12.18</td>
<td>1.70 K</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>5.24</td>
<td>0.70 K</td>
<td>6.87</td>
<td>0.70 K</td>
</tr>
<tr>
<td>Class B</td>
<td>64</td>
<td>42.40 (4/32)</td>
<td>33.08 K</td>
<td>44.82</td>
<td>33.08 K</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>21.42 (4/32)</td>
<td>1.82 K</td>
<td>24.41</td>
<td>1.82 K</td>
</tr>
<tr>
<td>Multigrid</td>
<td>32</td>
<td>16.35 (16/32)</td>
<td>~ 0.5 M</td>
<td>17.99</td>
<td>~ 0.5 M</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>8.39 (16/64)</td>
<td>~ 0.5 M</td>
<td>9.89</td>
<td>~ 0.5 M</td>
</tr>
<tr>
<td>Conjugate Gradient</td>
<td>32</td>
<td>10.93 (4/32)</td>
<td>&gt; 3.0 M</td>
<td>25.47</td>
<td>&gt; 3.0 M</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>7.43 (4/64)</td>
<td>~ 3.7 M</td>
<td>11.40</td>
<td>~ 3.7 M</td>
</tr>
<tr>
<td>3-D Fast Fourier Transform</td>
<td>32</td>
<td>18.46 (16/32)</td>
<td>~ 180 K</td>
<td>18.07</td>
<td>~ 180 K</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>9.65 (16/64)</td>
<td>~ 360 K</td>
<td>(32/32)</td>
<td>~ 360 K</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(64/64)</td>
<td></td>
<td>10.11</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>32</td>
<td>8.17</td>
<td>~ 137 K</td>
<td>9.19</td>
<td>~ 137 K</td>
</tr>
<tr>
<td>Sort</td>
<td>64</td>
<td>4.88</td>
<td>~ 267 K</td>
<td>5.41</td>
<td>~ 267 K</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>2.88</td>
<td>~ 529 K</td>
<td>3.72</td>
<td>~ 529 K</td>
</tr>
<tr>
<td>Lower-Upper Diagonal</td>
<td>64</td>
<td>241.67 (4/64)</td>
<td>~ 1.8 M</td>
<td>277.61</td>
<td>~ 1.8 M</td>
</tr>
<tr>
<td>Scalar</td>
<td>64</td>
<td>285.46 (4/64)</td>
<td>~ 1.0 M</td>
<td>381.74</td>
<td>~ 1.0 M</td>
</tr>
<tr>
<td>Pentadiagonal</td>
<td></td>
<td>(16/64)</td>
<td></td>
<td>811.00</td>
<td></td>
</tr>
<tr>
<td>Block Tridiagonal</td>
<td>64</td>
<td>247.25 (4/64)</td>
<td>&gt; 8.0 M</td>
<td>260.47</td>
<td>&gt; 8.0 M</td>
</tr>
</tbody>
</table>

Table 2: The execution time of NAS Benchmark programs. The NAS programs were profiled with MP/MP.sel monitoring library. The notation \((n/m)\), in which \(n \leq m\), indicates that the application was executed on \(m\) nodes but only \(n\) nodes were profiled. When \(n \neq m\), only the first \(n\) nodes collect trace information.

and monitoring intrusion is provided by the MP/MP approach, currently implemented in Aramis. Using this approach and a small buffer size for data exchange, 8KB, we observe a small increase in the execution time, ranging from 4.9% to 29.2% for Class B, 64 nodes and Class A, 128 nodes respectively.

The effects of the instrumentation upon the execution time of the NAS Benchmark programs is summarized in Table 2. The measurements reported in Table 2 are obtained using MP/MP.sel monitoring library, a variant of MP/MP approach, which provides the user with some control over the amount of trace data collected. The user can select the number of nodes to be monitored when a profiled application generates large amounts of data, and also can apply uniform sampling mechanism when a profiled application, such as the Embarrassingly Parallel benchmark, generates small amounts of trace data. The effect of collecting large amounts of data reflects on the execution time of Conjugate Gradient, LU factorization, Scalar Pentadiagonal, and Block.
Tridiagonal benchmarks. For example, the execution time of Scalar Pentadiagonal benchmark increases 34% when collecting data on four out of 64 nodes and 184% when collecting on 16 out of 64 nodes.

2.3 Data Analysis

A monitoring tool, such as Aramis, produces a collection of trace files containing a wealth of information about the behavior of the application being monitored. To extract the relevant information necessary for a certain type of performance analysis and present it in a comprehensive form, one needs to construct a set of analysis tools. D'Artagnan consists of a set of analysis and visualization modules. The data analysis modules process and summarize the large trace record set, creating a concise model for the application under study, and the visualization modules support graphical representation of raw and processed trace data.

Though our primary concern is the study of the paging, I/O and communication activities of a parallel program, D'Artagnan includes also a generic event analysis module, in addition to specialized modules for the study of I/O and paging.

We are investigating means of integrating our modules, and eventually incorporate a rule based system capable of generating analysis modules knowing the format of the trace records and the significance of each atomic data element.

Generic analysis, event and state information

As discussed in previously, our monitoring strategy is based upon detecting transitions from one state of a process or thread to another. The user controls the format of a trace record; a trace record may contain data collected by the kernel relative to global and shared resources, and data specific to the program.

The generic analysis module provides information like: the total number of events recorded during the execution, the total fraction of time spent by a node in each state, the average lifetime of a state, etc. When we instrument only I/O and communication system calls, a program can be in one of three states: compute, communicate and I/O. Synthetic data related to this model are very useful in understanding the behavior of a parallel program; they provide the first line of defense against major flaws in the program design. For example, if a large fraction of all
events are related to I/O operations and most of program’s life time is spent in an I/O state, even without looking at the code we can establish that some form of buffering, for reading and writing the data, would be very useful.

**Analysis of the paging activity**

In the following, we describe specific data analysis modules used in D’Artagnan for correlating the paging activities of parallel programs running on DMIMD systems and we introduce a methodology for reduction, interpretation, and analysis of the paging activity called “skyline analysis”.

For most of the parallel applications studied, the profiles for different paging activity indicators show transients of high paging activity, peaks, intermixed with longer periods of low activity, backgrounds. Thus, data reduction [25] is the first step of the analysis and consists of: peak selection, background filtering, and skyline representation for each individual trace data file. The data reduction is used to expedite the identification of activity bursts and to reduce the amount of data needed for describing the program behavior. After data filtering, the paging profile is reduced to a much more compact representation, composed of peaks and background regions. Figure 2 illustrates the activity profile of a paging indicator, the background filtering and the compact skyline representation of the same activity profile. D’Artagnan provides several graphic visualization tools to depict the paging behavior, such as the time line of a parameter collected in the trace records before filtering (raw trace data), the time line of the compact paging profile of the same parameter (skyline representation), along with other graphic tools used during the data reduction procedure.

**Correlation of the paging activity of individual node programs.** The methodology used to study how similar or dissimilar are the paging activities of individual PEs is to isolate the peaks of activity and to study how their amplitude and time of occurrence are correlated.

The time correlation is performed by isolating the N highest peaks of each node and comparing the amplitude variation of each peak among nodes. Similarly, we compare the time of occurrence of each peak among nodes. Figure 3 shows the methodology of selecting and correlating peaks, and Figure 6 shows an example of the output of skyline analysis (see Section 3.2).

**Cumulative profile.** The cumulative profile is used to determine the total load caused by paging activity on the communication and on the I/O subsystem. We use an algorithm that adds
Figure 2: Paging activity profile. (A) represents a profile of a paging activity indicator – the page fault occurrence rate. (B) is a compact representation of (A), the skyline representation derived from (A) after performing data reduction by retaining peaks and averaging the background periods.

Figure 3: Skyline analysis – correlation of peak of activity among three node programs. Four peaks ($P_{1,2,3,4}$) are isolated for each node. Amplitude correlation compares the amplitude of $P_i$ across all three nodes, and time correlation looks at the earliest and the latest occurrence time of a certain peak.
Figure 4: The parallel procedure to produce the cumulative profile of a paging activity indicator. Eight skylines of individual node programs are reduced to one cumulative profile using two processors.

The individual skylines pairwise until a cumulative profile is obtained. This algorithm is easily implemented in a parallel machine by distributing individual skylines to each processing element and then combining the partial sum of the skylines, resulting in one single overall cumulative profile. Figure 4 illustrates the synthesis of the cumulative profile of eight individual skylines using two processors.

The skyline and the cumulative analysis can be applied to other performance indicators besides those describing paging activity. This skyline analysis provides a synthetic way of describing the correlation of the activities of individual nodes in time and intensity and can be applied to I/O and communication indicators, e.g. data rates. The same applies to the cumulative analysis.

Analysis of I/O activity

I/O is often the bottleneck in high performance computing. Most of the existing Massively Parallel systems are imbalanced, the CPU bandwidth being considerably larger than the memory and I/O bandwidth. To make matters even worse, the users of parallel machines are seldom
sophisticated enough to take advantage of parallel file systems and optimize their I/O access patterns. There are few tools to help them, and optimization techniques suitable for one system may not be available or productive on other systems.

Two types of data analysis are performed on the I/O trace data collected. The first, I/O statistics, calculates the overall statistics of I/O operations and the second, I/O graphics, depicts graphically the evolution of the I/O operations throughout the program execution. The I/O statistic analysis reports the I/O operation count, duration of the operations, number of bytes involved in the operations, average transfer size, and histogram of the distribution of the number of bytes accessed or manipulated for operations such as read, write, and seek.

The I/O graphics package allows the user to visualize, compare, and correlate the activities of several I/O operations among PEs. For example, the user could either select read, write and seek operations of PE 1 to be shown and see how the selected operations happen during the execution, or select the write operation of all PEs to visualize similarities and differences among them. An example of I/O analysis use is given in Section 3.3.

2.4 Porting the Musketeers to the NT environment

Porting software development tools from different flavors of UNIX to an environment based upon a very different philosophy like Windows NT is a difficult task. Windows NT, like Mach, is a multi-threaded symmetric multiprocessing operating system relying heavily on the client-server paradigm. The micro kernel provides low level operating system functions like thread scheduling, interrupt dispatching and synchronization, I/O, and handling of basic objects. The dynamic link library layer, DLL, insulates the kernel, called NT executive, from hardware dependencies [10].

A few comments covering I/O and virtual memory in Windows NT follow. In NT, I/O operations involve file objects. To read, write or to perform any I/O operation, a user thread invokes file object services under the control of a file manager, which directs these requests to actual files, pipes, or any other destinations. The virtual memory manager provides a set of services allowing processes to allocate memory, read and write virtual memory, lock pages in memory, protect pages, flush pages to the backing store, and share memory.

The NT system has a Performance Monitor [6]. This tool allows a user to collect performance data pertinent to core objects like processes, threads, processor(s), memory, cache, paging file, logical and physical disk and others. Some of the objects can have multiple instances, e.g. there
could be multiple processes, threads, disks, processors, and so on. The user can monitor remote objects and may add his/her own counters to an existing object, and may modify the sampling interval to suit specific monitoring needs. The performance monitoring tool provides a user interface to select the computers, the objects on each computer, and the specific counters for each object to be monitored. The NT Performance Monitor has a flexible format allowing a user to customize the display of performance data.

The performance data collected by the Performance Monitor is self-described and structured hierarchically: a performance data block describes the system and points to individual structures describing the performance data for each type of object. In turn this structure points to the performance data for different instances of the object. A user can write a custom performance monitor using as a model the Windows NT Performance Monitor.

Although it allows collecting system-wide performance information, the NT Performance Monitor does not record the effects of the execution of a specific program. The latter can only be achieved by instrumenting a user program. This is precisely the function performed by the Musketeers, which allow a user to define the format of a trace record to include system and user data, and to create such trace records when specific statements are encountered during the execution of the program.

A non trivial task in porting the Musketeers to the Windows NT environment is the design of the system dependent library, SDL, used by Athos to create dynamic trace libraries as discussed in Section 2.1. The function of the core SDL in the NT environment is the retrieval of performance counters from the system's registry database through the registry-queuering mechanism provided by NT [6]. The SDL extracts from the NT structures only those performance counters requested by the user.

Porting the Musketeers to the NT environment proved that our design philosophy was correct. Writing the tool in Tcl/Tk allows us to port it to a variety of environments with minimal effort. The Musketeers have very few operating system and/or compiler dependencies. Only the system dependent libraries, used to access performance data collected by the kernel and to write the trace records, must be rewritten when the tools are ported from one environment to another. The use of templates to define the statements to be instrumented and to describe the kernel structures or the structure of the performance data, in case of Windows NT, provides the flexibility expected from a portable tool.
3 Applications

We used the Musketeers tools to study the performance of programs developed by our group and by others, such as those in the NAS benchmark suite [3]. Our group is developing parallel programs for solving very large structural biology problems [8, 9], and we used the tools described in the previous sections to study the performance of these programs. We discuss below the use of the Musketeers tools to optimize two programs in the Molecular Replacement suite, envelope and fftsynth.

The envelope program computes the molecular envelope of a virus. It needs as input a 3-D lattice with up to $10^9$ grid points and produces a lattice of equal size as output. For every grid point, information about the electron density and a mask describing if the grid point is located in the protein, nucleic acid or solvent is provided. A spherical virus has icosahedral symmetry, and the program exploits this to get better estimates of the electron density at every grid point by calculating the average of the electron density of all points related by non-crystallographic symmetry.

The program implements a shared virtual memory and operates in two modes, the DD mode, where the shared virtual memory resides on the external storage device, and the DC mode, where the input data is distributed over the set of compute nodes. Reference [8] describes different data management strategies for implementing a shared virtual memory. The entire data set is partitioned into Data Allocation Units, DAUs. The working set of DAU$^i$ consists of all DAUs needed to carry out the computations associated with DAU$^i$. DAU faults occur when a compute node needs to access a DAU stored elsewhere; the penalty for a DAU fault can be significant. In the DD mode a DAU fault requires a disk access, and in the DC mode it requires access to data stored on a different compute node or on a data server node. A load balancing algorithm distributes the DAUs based upon an estimate of the amount of work associated with each of them. Both modes exploit the locality of reference and attempt to minimize the number of DAU faults by processing the DAUs assigned to each compute node to maximize the intersection of the working sets of DAUs processed in sequence.

The second program, fftsynth [18], carries out a 3-D FFT. It reads in a set of complex valued structure factors (discrete Fourier coefficients), computes the FFT, and writes out the calculated electron density. A 3-D FFT is obtained by a 2-D FFT followed by a 1-D FFT in the third dimension. The algorithm requires a global exchange between phase one and phase two.
of the algorithm. If the amount of the combined local memory of the PEs is large enough to hold all the data, then intermediate results are exchanged through message passing among PEs, otherwise the global exchange is done using an external file.

3.1 Using generic event and state information for program optimization

A performance monitoring environment should be able to uncover obvious shortcomings of an algorithm or its implementation with minimal user effort. Only after understanding the usefulness of a tool a user is likely to gain confidence in the tool and be willing to learn to use it in more sophisticated ways. We applied this principle in the design of the Musketeers. When the user selects the default instrumentation mode of the Musketeers, all transitions among compute, I/O, and communication states are instrumented, and a trace record contains only the time of the event. The default analysis provides information about the total number of events per node, the total time spent in each state by each compute node and the average time spent in each state.

We describe now the effort to optimize the envelope program using the Musketeers. The histogram of the total number of events for each event type is very revealing. For example, Figure 5(d) shows that in the DD mode each node performs a very large number of I/O operations, about 2000, regardless of the number of nodes. Clearly such a program does not scale very well. It is no surprise that the contention for the I/O nodes increases drastically. When running in 64 compute nodes each of them spends most of its time waiting for the completion of an I/O operation – see 7(f) and 7(c). On the other hand, when the program reads the entire input lattice and caches it in the local memory of the individual nodes, as it is done in the DC mode, the number of I/O events per node decreases significantly, see 7(a), the average time spent in the I/O state decreases, and we observe a linear speedup.

The very moment we examined the distribution of events per node and the time spent in each state, it became obvious that the DD mode was I/O bound, and we realized that we need to either reduce the penalty for a DAU fault or the number of faults or both. Reducing the penalty for a DAU fault by caching the data in the compute nodes avoids access to the file system. To reduce the number of faults we needed to increase the amount of memory available in each node. This example illustrates the fact that only the designer of a parallel program can use performance data effectively. Without knowing the algorithm it would be very difficult to suggest ways to
Figure 5: Summary of information about the envelope program running in the DC (a, b, c) and DD (d, e, f) modes. The total number of events per node (a and d), the time spent in each state (b and e), and the average life time of an event (c and f). Data caching in the DC mode reduces substantially the number of I/O events, increasing the average time spent by each node in compute states.

cache the data as we did in the DC mode.

3.2 Paging and scheduling of parallel programs

Demand paging is one of the common features of commodity Workstation Operating Systems, like UNIX or OSF/1. Recently MPPs like the Intel Paragon or IBM SP2 running commodity operating systems and offering demand paged virtual memory have emerged.

Gang scheduling is a concept introduced by Ousterhout [20]. He observed that a MIMD system performance degrades when a parallel application does not have all its interacting processes scheduled at the same time. Often, the number of active processes in a process group changes dynamically and the term co-scheduling relates to the concept of scheduling at the same time only the active processes in a process group. To use effectively networks of workstations for solving large problems, their operating systems need to support some form of co-scheduling [1]. At the present time MPP operating systems like the Paragon OSF support static gang scheduling with busy waiting. This means that once a page fault occurs the PE blocks until the page is brought in. This clearly affects the overall processor utilization.

Supporting both demand paging and gang scheduling is a rather difficult proposition. Demand
The cumulative profile gives also a strong indication on how correlated are the paging events occurring in different processes of a process group. Figures 7, 8 and 9 show the cumulative rate of page-ins for the Integer Sort and the LU factorization in the compute state and communication.
In OSF terminology, a page-in fault occurs whenever the page is not available locally. Figure 7 and 8 show that the Integer Sort has a very brief period of intense paging activity around the ten second mark. This is probably caused by the global exchange phase of the algorithm. Most PEs experience page faults at about the same time, and this limits the effects of the page-in upon the performance. On the other hand, in case of LU factorization—a communication intensive algorithm—the page faults of different PEs are totally independent and are distributed throughout the entire program execution. This behavior is likely to lead to a low PE utilization. A 3-D FFT has a page-in behavior similar to the Integer Sort, as shown in Figure 10, where the peaks of the page faults are concentrated within two seconds or so starting at the three second mark. The 3-D FFT requires a global exchange, and we suspect that the peak of the paging activity is related to this global exchange. The Multigrid and the Embarrassingly Parallel benchmark, Figure 11 and 12, exhibit similar behavior, namely an isolated brief peak surrounded by low activity. In summary, we have observed two types of paging behavior, one suitable for gang scheduling with busy waiting (for example, the Multigrid benchmark) and one for which busy waiting is likely to lead to a noticeable performance penalty (for example, the LU factorization).

The study of the communication events allows us to determine the dynamics of the process working sets, namely the set of processes that need to be active at the same time because they communicate among themselves. For algorithms like the Integer Sort, there is virtually no communication until the global exchange phase, when the size of the processor working group jumps to $P$ (the number of processors in the partition).

3.3 Monitoring the I/O activity of a parallel program

In the following, we describe the study of the I/O activity of the fftsynth program on a Paragon XP/S system. First, we created a customized statement template consisting of I/O statements cread, cwrite, lseek, open, close, selected from the list of all Paragon OSF/1 system calls. Then we defined the trace record format for each of the selected statements to include information as the file accessed, number of bytes transferred, and mode of the I/O operation.

All I/O operations carried out by the fftsynth program are confined to one C routine, and this was the only code we had to instrument. The input and output data files for the program reside on the Parallel File System, PFS. For very large problems an external PFS file is used for
Figure 7: The cumulative page-in rate in the compute state of a class B Integer Sort benchmark running on 64 nodes. The total execution time was 33.26 seconds and the time reported by the NAS benchmark was 19.53 seconds.

The size of the trace files is determined by the size of the input data, the number of processors and the execution mode. As pointed out earlier, the global exchange needed by 3-D FFT is carried out "in-place" if the total amount of memory available to the compute nodes is sufficient to store the input data. Otherwise intermediate results have to be written to an intermediate data set. For example, for an input data set size of about 40 MB, the average size of the trace file collected by an individual node is 80 KB for execution with 8 PEs and 50 KB for 16 PEs. The number of I/O operations and consequently the size of the trace files is drastically reduced when the same problem is solved with 32 PEs and the global exchange is done through message passing. Then the average trace file size is 6 KB.

The I/O analysis modules provide generic statistics for I/O operations. A first observation from the summary of statistics of the fftsynth program (running on 16 nodes and using a scratch file for storing intermediate results) is that the total time spent on lseek operations exceeds the total time spent in read or write operations. An examination of the code revealed that each read or write was preceded by an lseek. Since each node program
Figure 8: The cumulative page-in rate in the communication state of a class B Integer Sort benchmark running on 64 nodes. The total execution time was 33.26 seconds and the time reported by the NAS benchmark was 19.53 seconds.

happens to access a file sequentially, we eliminated all but the first lseek carried out by a node program. Then we ran again the instrumented program only to observe that the time required by the read and write operations have increased. At that time it became obvious that a global synchronization occurred for every single I/O operation because the system was using the default semantics for file access, the so called M_UNIX mode.

Among several PFS file access modes that the Paragon OSF/1 supports, the default is the M_UNIX mode. This mode supports the standard UNIX file sharing semantics by allowing single reader, single writer access, although each node has its own file pointer. The MASYNC mode supports multiple readers and multiple writers, but this mode does not guarantee that I/O operations are atomic, and thus does not support standard UNIX file sharing semantics for different processes accessing the same file. In this case, the application must control parallel access to the file. Further investigations revealed that by changing the I/O mode of the PFS files manipulated to MASYNC, we obtained a significant performance improvement as shown in Figure 13.

As expected, the performance of the fftsynth program is affected by the way the global
Figure 9: The cumulative page-in rate in the communication state for 4 nodes of a class A Lower-Upper diagonal benchmark running on 64 nodes. The total execution time was 279.97 seconds and the time reported by the NAS benchmark was 277.61 seconds.

exchange is performed; if done "in place" the number of I/O operations is reduced drastically. If the global exchange must be done using an external file then letting the nodes access the file asynchronously reduces substantially the execution time of I/O operations. But this is only possible because each node accesses a consecutive portion of the data sequentially. Figure 14 illustrates the evolution of lseek operations of PE 1. This I/O activity analysis package has shown the usefulness of the Musketeers to debug and optimize the parallel program execution.

4 Conclusions

There is little doubt that to build better parallel machines we need to understand the behavior of parallel programs. The difficulties in understanding the behavior of parallel programs begin with the instrumentation phase. The tools we developed allow the user to select the statements to be instrumented, and then to define the format of the trace record generated by each traceable event. The tools, written in Tcl/Tk, are portable and easy to use. In this paper we also present our efforts to minimize intrusion in the design of the System Dependent Trace libraries for the
Figure 10: The cumulative fault rate in the compute state for 64 nodes of a class A 3-D FFT PDE benchmark running on 64 nodes. The total execution time was 22.64 seconds and the time reported by the NAS benchmark was 10.14 seconds.

Intel Paragon.

We have measured the paging and I/O activity for a suite of programs used in Structural Biology, and for programs in the NAS Benchmark suite. We have developed two techniques for studying the correlation of the activities of individual node programs. Both techniques are more useful for drawing qualitative rather than quantitative conclusions concerning the activity of parallel programs. The skyline analysis studies the rate of occurrence of a certain event. We isolate peaks of activity and then correlate the amplitude and the time of occurrence of individual peaks. The cumulative analysis is carried out by adding individual skylines to determine global event rates.

We have observed two types of paging behavior. Programs like the Integer Sort, 3-D FFT have only short bursts of paging activity, and we expect that such programs do not pay a hefty performance penalty when running under operating systems which support gang scheduling with busy waiting. Other programs, like LU factorization, exhibit chaotic paging behavior and perform poorly under busy waiting. Clearly, the paging behavior depends upon the relationship between the working set of a program and the amount of memory available in each node. Details about
Further measurements are necessary to establish a correlation between the paging activity, the process working set of a process group, and the communication patterns of the application. From our measurements we have observed that programs with regular communication patterns, e.g., global communication, exhibit short bursts of paging, while those with irregular communication patterns exhibit an erratic paging.

Our measurements have confirmed what was already known: an application should perform as few I/O operations as possible and transfer blocks as large as possible. Whenever possible cache the data either by distributing the data across nodes or by using a few data server nodes with large internal memory.

We believe that future operating systems for massively parallel systems need to provide some form of Virtual I/O, VIO. We cannot hide the high latency of an I/O operation by context switching, therefore we have to find means to reduce the latency. We cannot force a user to transfer large blocks of data to/from the external storage, therefore we have to decouple the physical I/O operations from the user requests. In a VIO system, a number of I/O managers
Figure 12: The cumulative fault rate in the compute state for 64 nodes of a class B Embarrassingly Parallel benchmark. The total execution time was 48.90 seconds and the time corresponding to the NAS benchmark measurement was 44.82 seconds.

Figure 13: The execution time of fftsynth using an external file for global exchange for two I/O modes, M_UNIX and M_ASYNC with an input of size 40 MB.
Figure 14: Average number of I/O operations per second in intervals of 10 seconds of PE 1 during the execution of fftsynth.

maintain I/O mapping tables, I/O staging areas and transfer data to/from external devices in large blocks chosen to optimize the I/O transfer rates. In case of an I/O request, an I/O read operation takes place only when the data element is not already cached in the staging area as result of a previous read operation.

Acknowledgments

The authors want to express their thanks to Jerry Baugh for helpful suggestions and to John Jackson for sharing with us his insight into the intricacies of the Paragon OSF. This research is supported in part by NSF grants BIR-9301210 and MCR-9527131, by the Scalable I/O initiative, by a grant from Intel Corporation and by CNPq Brazil.
References


