Getting I/O Performance on the ASCI Red Platform

Ben Cole, Pat Fay, Bob Godley, Greg Henry, David Robboy, Paul Work
Intel Corporation

1. INTRODUCTION

In this paper, we describe the I/O model on the ASCI Red platform. We first describe the different I/O models available. We then provide charts to demonstrate the performance that may be expected under various scenarios. Our intention is also to describe how the various I/O facilities work and how external factors can impact performance, and to give insight into which model to use in a given situation.

Parallel I/O is a viable area of research in computer science: This alone is sufficient to indicate that getting good disk I/O performance from an application on an MPP system is nontrivial. In addition to code portability, readability, and maintenance issues, there is also the question of how many of the systems resources an application requires to achieve fast I/O. The ASCI Red system, hereafter referred to as Janus, is in a sustaining mode at the current time: Fundamentally, the total amount of system resources is now fixed, which makes this a zero-sum game. For these reasons this report will not have a single bottom line, because there will not be a single "right" answer for all applications.

On the other hand, there are some universally "wrong" answers. These are I/O configurations that result in disk I/O performance that is so bad that the application cannot run, or at least cannot run on a machine-sized problem, which is the focus of the design of the system. We will point out these undesirable configurations, explain why they do not work, and suggest alternatives. We will restrict our focus to I/O performance from parallel applications, which will exclude: 1) TCP/IP I/O to the network (NFS), 2) I/O from jobs running on the service nodes.

This paper is organized as follows: we start with a brief overview of I/O from both hardware and software perspectives. The hardware and software features which support the UFS file systems are described first, followed by the same for the PFS file systems. We then briefly describe how these components affect performance. We then describe some of the various API choices and how to choose between PFS and UFS. We then hit the benchmarking portion of this paper, which is designed both to set expectations and to increase perspective. We do this for applications which use both single nodes and multiple nodes

2. Description of I/O on Janus

2.1. Hardware

Two alternatives were considered when designing the I/O system for Janus. One was to attach a disk to each compute node, while the other was to concentrate the file system on a small set of specialized nodes that process I/O requests. For a number of reasons Intel chose the latter option, using Redundant Array of Independent Disks (RAIDs) for secondary storage. A RAID device achieves a higher effective transfer bandwidth over a single disk drive by the RAIDs controller striping file data across multiple spindles in parallel. However, a RAID does not approach its maximum transfer rate until the request is large. For the RAID devices on Janus, transfer sizes of 512 KB or larger are necessary in order to approach 32 MB/sec, the maximum bandwidth for these RAIDs on Janus. These RAIDs have a 512 KB cache to help mitigate the effects of small transfer sizes. This 512 KB cache on the RAID is the determining factor of the PFS 512KB block size. RAIDs on Janus are attached to eagle nodes running TOS. Each RAID is attached to a separate SCSI controller, and there are two SCSI controllers (and thus,

two RAIDs) per I/O node. Each I/O node also enforces a limit on the number of I/O requests that can be outstanding at any one time.

2.2.Software

When an application starts, it is assigned one or more I/O service processes, called fyods. Fyods are application programs running on TOS, on the service nodes. By default, the yod allocates one fyod for each 256 compute nodes in a parallel application. The user may specify the number of fyods to allocate on the **`yod`** command line. All communications between an application process and the fyods are conducted via Remote Procedure Calls (RPCs) over the high-speed communications network. Cougar, the OS which runs on the compute nodes, cannot directly access any of the file systems mounted on janus. All application I/O is initiated with an RPC from the compute node to an fyod, which performs the I/O on behalf of the Cougar application by making UNIX system calls.

2.3.UFS

UFS is the default UNIX file system for TOS. The files local to the RAIDS of Janus are UFS files, except for those under the directory /pfs. The UFS file system was derived from what used to be called the Berkeley Fast File System. There is a size limitation for UFS files: a single file cannot exceed 2GB -1 (2,147,483,647 bytes).

A UNIX file system traditionally has a buffer cache that helps increase I/O performance of small requests. The TFLOPS operating system uses memory mapped files rather than a static buffer pool. When a process opens a file, the file is mapped into the virtual memory space of that process, and when the process reads from the file, the data is paged in using the demand paging mechanism.

The *file system block size* is the unit that a file system allocates for files. For example, if the file

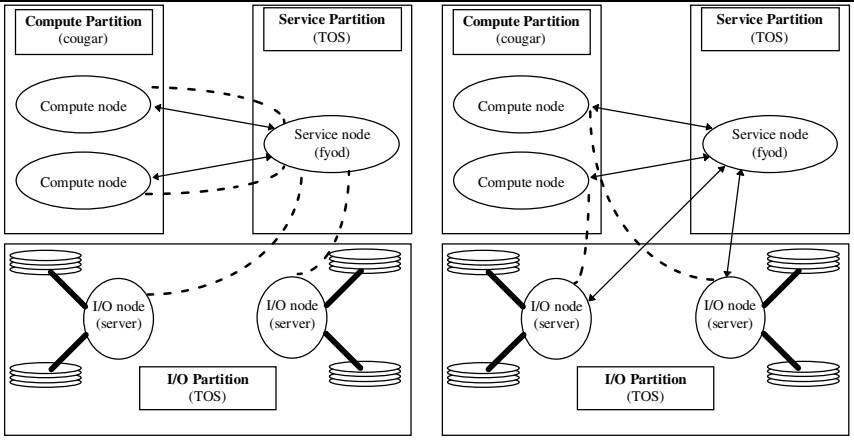

Figure 1: Data and Control Flow for UFS (left) and PFS (right). The straight lines represent control flow, while the arcs represent data flow

system block size is 8KB, then a 9 KB file causes the file system to allocate two 8 KB blocks. The 7 KB of unused space in the second block cannot be allocated to another file. For Janus, the file system block size for UFS files is 8 KB. Note that the size of the TOS page is also 8 KB.

When an application process on a compute node writes a block of data to a UFS file, an RPC containing the address of the buffer and the buffer's length is sent to the fyod assigned to that node. The data itself is not sent with the RPC. The fyod allocates a buffer and uses the Portals interface to copy the user file data from the compute node to its buffer in the Service Node. The fyod then uses TOS system calls to write the file data to the UFS file. The buffer allocated by the fyod is up to 128KB in size.

To clarify a point, we have described two levels of buffering. The fyod allocates a buffer in order to transfer the data from the compute node. This is not a cache, but just a place to hold the data in order to write it to the file. In addition, the operating system uses memory mapped files in order to cache file data on the service node. Both of these levels of buffering exist for UFS files only, not for PFS, as we shall see.

The fyod processes an application read request from a UFS file analogously: the fyod allocates a buffer and uses TOS system calls to read the requested file data into its buffer. The fyod then uses the Portals interface to copy the file data into the compute node's memory.

The data travels the following path when reading from a UFS file. The data goes from the disk to a TOS server on an I/O node, which sends it to a memory-mapped buffer on the service node where the fyod resides. On the service node, TOS transfers the data from the memory-mapped file to the fyod's application buffer in user space, which is more likely to be a page mapping operation than an actual copy of data in memory. Finally, the fyod sends the data to the user's buffer on the compute node. Thus the data makes three hops, or possibly four if it must be copied to the fyod's buffer. On the other hand, when reading from a PFS file, there are only two hops: From the disk to the I/O node and from there directly to user memory on the compute node.

Because the block size of the UFS file system is 8KB and the operating system buffers file system blocks, UFS can be preferable to PFS when making small or unaligned file requests, especially if the data will be re-used.

2.4.PFS

The design goal for PFS was to optimize the transfer of large blocks of data. When looking for the I/O bottlenecks on an Intel MPP system, one realizes that the communications fabric is faster than the bandwidth to a single disk. For example, on Janus the backplane can sustain bandwidths of ~370 MB/sec (in one direction), while the disk subsystem on an I/O node can sustain at best 64 MB/sec. Thus in principle a single application node with a large amount of data to move can get it done more quickly moving it in chunks (known as stripes) to different RAID arrays.

A PFS file is striped over multiple UFS files in a round-robin fashion. Each of these files is referred to as a *stripe file*. For example, Figure 2 represents a PFS file system striped across four UFS file systems which are physically located on two RAIDs. The amount of file data that is stored in one stripe file before moving to the next stripe file is called the *stripe unit size*.

Each PFS file system has a certain stripe size and stripe factor, which is the maximum number of RAIDS associated with that PFS file system. The stripe size is required to be an integer multiple of the disk block size. Intel studied the performance of the disks as a function of block size, and determined that 512K bytes was the optimal block size for performance. The default stripe size was set at 1 MB to minimize overhead for large block transfers at the RAID level. . The "/pfs/tmp_*" file systems are each striped 2 ways across the RAIDs attached to a single I/O node. The "/pfs/multi" file system is striped 28 ways across all the RAIDs that are available for PFS stripes. (A single *stripe directory*, or section of disk set aside for PFS data, may service more than one PFS.)

The underlying UFS file systems for the stripe files is initialized differently for PFS: the file system block size is 512 KB to match the cache size of the RAID controller. Also, stripe files do not have the 2GB limitation: it is possible for a single stripe file to consume all the disk space in its UFS file systems.

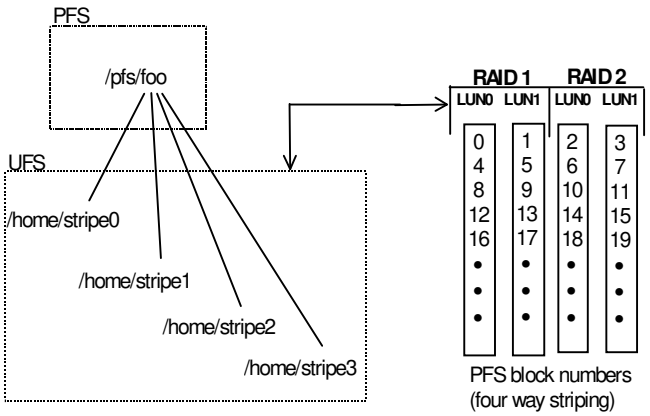

Figure 2: PFS stripe file mapping

When an application process on a compute node writes a block of data to a PFS file, an RPC containing the address of the buffer and the length of the data is sent its fyod. The data itself is not sent with the RPC, only the control information is sent. The fyod makes a UNIX system call, just as it does for UFS files. TOS internally dispatches the request to the correct I/O node, and if necessary breaks up the request and dispatches parts of it to multiple I/O nodes, where the TOS servers handle the requests. The I/O nodes transfer the data directly from user memory on the compute node to the RAID hardware, bypassing the service node, as illustrated in Figure 1. There is no file system buffering. Data from the application buffer fans out across all affected I/O nodes in parallel thus achieving a high aggregate data transfer rate. For reading PFS files, the processing and parallelism are analogous to writing. I/O nodes read the data from the stripe files and send it directly to the user's buffer(s) on the compute node(s), via the portals interface.

The operating system internally throttles very large I/O requests by breaking them into packets of no more than 4MB per stripe file at a time. We believe this to be invisible at the application level.

2.5. API CHOICES

Cougar, like UNIX, provides the user with a wide variety of options for performing I/O. There are several different programming interfaces, two file systems, and two I/O modes, resulting in a large array of possible alternatives. The programming interfaces include the following:

- For C or C++, the Unix operating system calls (open, read, write, ...).
- For C or C++, the Unix Standard I/O Library (fopen, fread, fwrite,...), also known as the stream I/O library, or for short, the *stdio* library. This library provides buffering at the application level.
- For C++, the *fstream* objects, including the << and >> operators.
- For FORTRAN, the FORTRAN runtime library (OPEN, READ, WRITE,...) , whose procedure names are similar to the names of the Unix system calls but are distinct.
- For all languages, the asynchronous interfaces (iread, iwrite,...) which are unique to the Intel Supercomputers, and not portable.

The two file systems are UFS and PFS. The two I/O modes are MUNIX and MASYNC. The various combinations and their advantages and disadvantages are considered in the next few subsections.

2.5.1. Unix system calls: **open, read, write**, and friends

These are the standard Unix system calls. They are portable to all Unix systems. From a performance point of view, the significant feature of these interfaces is non-scalability. For example, when a process on a compute node does a *read()*, that request is served by an fyod, which does a *read()* system call on its own service node. Until that *read()* call returns, the fyod is blocked and will not serve other requests from any other node. Therefore, these calls are sequential per fyod and not scalable.

For single-node I/O, these interfaces are slightly faster than the asynchronous interfaces because there is less overhead at the fyod level. A single system call sometimes involves less latency for the fyod than managing an asynchronous event.

2.5.2. Unix standard I/O library: **fopen, fread, fwrite**, and friends

These library interfaces are at least as portable as the Unix system calls, being specified by the C language. From a performance point of view, the standard I/O library is preferable for a single node doing small or unaligned I/O requests. The requests are buffered locally by the library on the compute node in a buffer called a *stream buffer*. This results in actual file I/O activity only when the buffer is flushed, for a write, or in the case of a *read*, only when a new buffer is needed. The default buffer size is 8KB but it can be increased with **setbuffer()** and related functions.

The **fwrite()** function invokes the asynchronous **cwrite()** system call. An **fwrite()** call with a size greater than the buffer will bypass the buffer and do a **cwrite** without copying the data (after flushing whatever is in the buffer). Therefore, if you set the buffer size to a multiple of the PFS block size and do **fwrites** of that size, the **fwrite** function is equivalent to **cwrite** and is scalable. If the **fwrite** requests are smaller than the buffer size, then there is the additional overhead of copying the data to the buffer, but when the buffer is flushed it will still get the I/O performance of **cwrite** to a PFS file, if the buffer size has been set to a multiple of the PFS block size.

The **fread()** function eventually invokes the **read()** system call, so it is not scalable. In addition, **fread** does not bypass the buffer for large requests, so the data is always copied, resulting in lower bandwidth than the system calls.

2.5.3. The C++ *fstream* objects

These objects are part of C++ and their implementation is opaque to us. They probably do buffering analogous to the standard I/O library, and eventually make the Unix system calls **read** and **write**. We do not know whether it is possible to control the buffer size. This I/O interface can be expected to be non-scalable, and to use non-optimal request sizes for PFS files.

2.5.4. FORTRAN I/O runtime library

The FORTRAN runtime library ultimately makes Unix standard I/O library calls (**fread**, **fwrite**, etc.). This library can be expected to be non-scalable, and to use non-optimal request sizes for PFS files.

The asynchronous I/O interfaces (**iread**, **cread**, etc.), which are scalable, are also available for FORTRAN programs, in addition to the standard runtime library. Asynchronous I/O in FORTRAN is not portable, but has the performance and scalability advantages described below.

2.5.5. Asynchronous I/O functions for the ASCI Red system

The asynchronous calls are **iread, iwrite, cread, cwrite, iodone,** and **iowait.** These interfaces are scalable; that is, multiple compute nodes doing I/O concurrently can attain a bandwidth in proportion to the maximum aggregate I/O bandwidth of the I/O nodes of the system, rather than being serialized. From the point of view of functionality and semantics, **write()** does the same thing as **cwrite()**, and in fact both are equivalent to the sequence:

iwrite();
iowait();

The asynchronous calls are non-blocking at the fyod level, which means that an fyod can initiate multiple requests on behalf of multiple compute nodes, whether to a single file or multiple files (see the I/O Modes Section below for multiple nodes accessing a single file). Provided that an application's requests span multiple PFS stripes, each fyod can keep multiple I/O nodes busy and has the potential to exploit most of the aggregate I/O bandwidth of the system.

In the operating system, asynchronous I/O is implemented by spawning a Mach thread for each open file to which an fyod does an asynchronous operation. For example, if an fyod supports 256 compute nodes, and if each compute node opens 10 files and does asynchronous I/O to all of them, then the fyod will spawn 2560 Mach threads to manage that I/O. These threads do not terminate for the life of the fyod, but if a file is closed, then its threads depress their priority and, the threads are re-used when other files are opened.

Although asynchronous I/O can give the highest performance on the TFLOPS system, the programmer must be aware of an issue with UFS file systems. An fyod can serve up to 256 compute nodes, and each compute node process can open many files. So it is possible for an fyod to initiate many hundreds or even thousands of asynchronous I/O requests in parallel. As we have seen, the UFS file system buffers these requests on the fyod's service node using demand paging. This can result in thrashing due to excessive paging. To avoid this situation, it is best to use asynchronous I/O on the PFS file system, or else to carefully throttle the number of active requests at the application level.

Another point about asynchronous I/O to UFS files is that a UFS file system exists on a single I/O node. Although one or more fyods can make concurrent asynchronous requests to a UFS file, those requests will queue up at the OSF server on the I/O node, so the benefit of asynchronous I/O to a single UFS file system is limited. However, if the compute nodes are making requests to files on multiple UFS file systems, then a single fyod can attain more parallelism with asynchronous requests than with synchronous requests.

Warning:
In the R2.3 release (August 1998) of the TFLOPS operating system, there are some anomalies if an application terminates abnormally by an Control-C signal or a 'kill -9' command and the application has an asynchronous read or write operation pending. Asynchronous read and write operations are the iwrite(), iread(), cwrite(), and cread() functions. Although the command prompt may return quickly to the user's console, the system has not completed the operation. The system terminates the yod but leaves the fyod running. The system may take an excessive amount of time (e.g. more than five minutes) to complete the processing of the Control-C or 'kill -9' event, or the fyod may not go away until a reboot (this locks the files that the fyod was processing).

A user or system administrator can detect this condition by the presence of a fyod with a Parent Process ID (PPID) of 1 in the display of the 'ps -ef' command. If the fyod is then hit with a 'kill -9', it will be marked "<exiting>" in the display of 'ps -ef'. Wait for the process to exit (it may take a reboot to clear it.) Until the <exiting> process is gone, do not execute an application that attempts to open the files to which the asynchronous I/O was directed. Otherwise the file system containing that file becomes locked-up, which will require a reboot of the system to clear. Intel expects that this problem will be fixed by the R2.4 (December 1998?) release.

2.6. I/O Modes

There are two I/O modes which specify the semantics of I/O calls. The modes are the standard Unix file access mode, dubbed **MUNIX**, and **MASYNC,** not to be confused with the asynchronous function interfaces listed above. The current default is **MUNIX**, which provides the standard semantics of the Unix operating system. In **MUNIX** mode, reads and writes are atomic. That means, in summary, that a process can overwrite data written by another process, but they can not partially overwrite each other.

For this discussion, when we say *processes* we mean TOS processes, which is to say fyods. This is a little confusing, but first let's just talk about how Unix processes act, and then afterwards we'll consider the implications for massively parallel jobs. For example, suppose two Unix processes write to the same offset of the same file. In **MUNIX** mode, the operating system maintains a token which only one process at a time can own, so that one process will write to the file, then the other process will get the

token and overwrite the same offset in the file. This is all handled by the operating system, the user process never sees the token. In **MASYNC** mode, there is no token and a record in a file might end up containing partial data from each process. Or, one process may read a record that has been partially updated by another process.

Now let's look at how that affects Cougar applications. **MASYNC** mode permits concurrent access to a shared file between TOS processes, which in this case are fyods. Therefore, you should consider **MASYNC** mode for applications that have more than one fyod, and whose nodes share a file. By default, applications with over 256 compute nodes have more than one fyod. If the compute nodes share a file, then in **MUNIX** mode, I/O to the file between fyods is serialized. Even in cases where multiple nodes do not share a file, **MASYNC** mode can still provide a 5 - 20% improvement over **MUNIX** mode because relaxing the constraints reduces system overhead.

The mode used by an application is specified by using the "-munix" or "-masync" arguments to yod. There is no way to specify **MASYNC** mode on a per-file basis.

2.6.1. Warning: Using MASYNC mode with sparse files

There is currently a bug in TOS such that if a file is written in MASYNC mode and if the file is sparse and has a hole at the end of a PFS stripe unit, then the inode for the file will have an incorrect file size and data will be lost. For example, if multiple nodes open a PFS file whose stripe unit size is 1MB, and if each node seeks to a file offset of 1MB * mynode() and then each node writes less than 1MB of data, then there will be a hole in the file at the end of each stripe unit, resulting in data being lost. This only applies to MASYNC mode; writing in MUNIX mode works correctly.

Until this bug is fixed, suggested workarounds are:

- When using MASYNC mode, avoid writing sparse files. If a file is partitioned among multiple nodes, make sure each node writes its entire partition.
- If you are writing sparse files in MASYNC mode and it is not practical to fill in the entire file, then write something to the end of each PFS stripe unit of the file, so a hole does not extend to the end of a PFS stripe unit.
- If the above workarounds are not practical, then avoid writing sparse files by opening a separate file per compute node.
- Use MUNIX mode.

2.7. PFS I/O Performance in Theory

To summarize some of the implications of the paper thus far:

- MASYNC mode permits parallel I/O for applications that have more than one fyod and that share a file, by allowing multiple fyods to access a file in parallel.
- Asynchronous I/O requests permit parallel I/O for multiple compute nodes per fyod, whether files are shared between nodes or not.
- For optimal performance, requests to PFS file systems should be a multiple of the PFS stripe size, which is 1MB.
- If they are less than 1MB, then requests to PFS file systems should at least be a multiple of the file system block size, which is 512KB, because the operating system does not buffer PFS files.

Optimal single-node performance should result when request sizes are a multiple of the number of stripes times the stripe size in the PFS file system being accessed. This allows the operating system to generate one RPC to the I/O node(s) for each stripe in the file system. Each I/O node in the PFS file system then receives an even number of I/O requests and can access both attached RAIDs simultaneously.

When a single node reads from or writes to a file in one of the "small" PFS file systems (only two stripes, each on the same I/O node), in order to achieve efficient I/O rates, the request size should be a multiple of 2 MB (1 MB stripe size times two stripes per file system). The upper limit on transfer rates will be around 64 MB per second to/from these file systems. If an application explicitly spreads its data across multiple file systems of this type, one can reasonably expect the I/O performance to scale up to the number of I/O nodes in the system (14). In fact, total transfer rates approaching 1 GB per second have been achieved using this technique on a dedicated system.

Using the above formula, the optimal request size (from a single node) for accessing a file in the /pfs/multi file system would be a multiple of 28 MB (1 MB stripe size times 28 stripes)! Because it is rare that an application would have this much data (per node) to read or write, we will examine ways to achieve efficient I/O with smaller request sizes. The /pfs/multi file system can be most effectively used when multiple nodes are accessing the same file. Under MASYNC mode, this can occur without blocking at the fyod level, and automatically sets up an access pattern similar to that of multiple nodes accessing separate files on different file systems.

In detail, then, to effectively realize good I/O transfer rates, an application should use MASYNC mode and have each node open the file to be accessed. The data for each node should be set up in blocks that are a multiple of 1 MB in size, and prior to performing an I/O operation, each node should then perform a seek operation to position its file pointer to a position in the file which corresponds to the start of a stripe file. The easiest way to do this is to use the node number as a multiplier. Node number range from 0 to n - 1 (where n is the total number of nodes assigned to the application). Multiplying the node number by the request size will then give a value for the seek operation. The controlling fyod can then assign I/O operations in the most efficient manner. If the data for a particular node is not equal to a multiple of 1 MB, then I/O performance will be less than the maximum possible, but will still be more efficient than other access strategies.

Warning: In view of the bug described in Sec. 2.6, the above paragraph can be a recipe for losing data if each node does not completely fill up its partition of the file. Until that bug is fixed, note the workarounds listed in Sec. 2.6.1.

3. Observations of I/O Performance

We now turn from a discussion of how I/O is designed for this system to measurements of how it performs under various conditions.

3.1.Service Partition Load Effects

The first observation made by anyone testing I/O on this system under normal operating conditions is that repeated I/O experiments produce different results. The key to understanding this is the phrase “normal operating conditions.” I/O performance on this system is sensitive to the level of activity in the service partition. Figure 3 shows normalized bandwidth as a function of service partition load, which is defined as the ratio of the time required for a known task running in the service partition divided by the time required for the same task when it is the only job in the service partition. We see clearly that I/O performance degrades sharply with the onset of service partition load. Further, it was discovered (and recently fixed) that fyods of other jobs that are in an “idle” state (waiting for the application to make an I/O request) place a heavier burden upon the service nodes where it resides. Performance documented in the remainder of this report are on an otherwise idle machine, and should be considered as theoretical maxima by users on Janus under normal operating conditions.

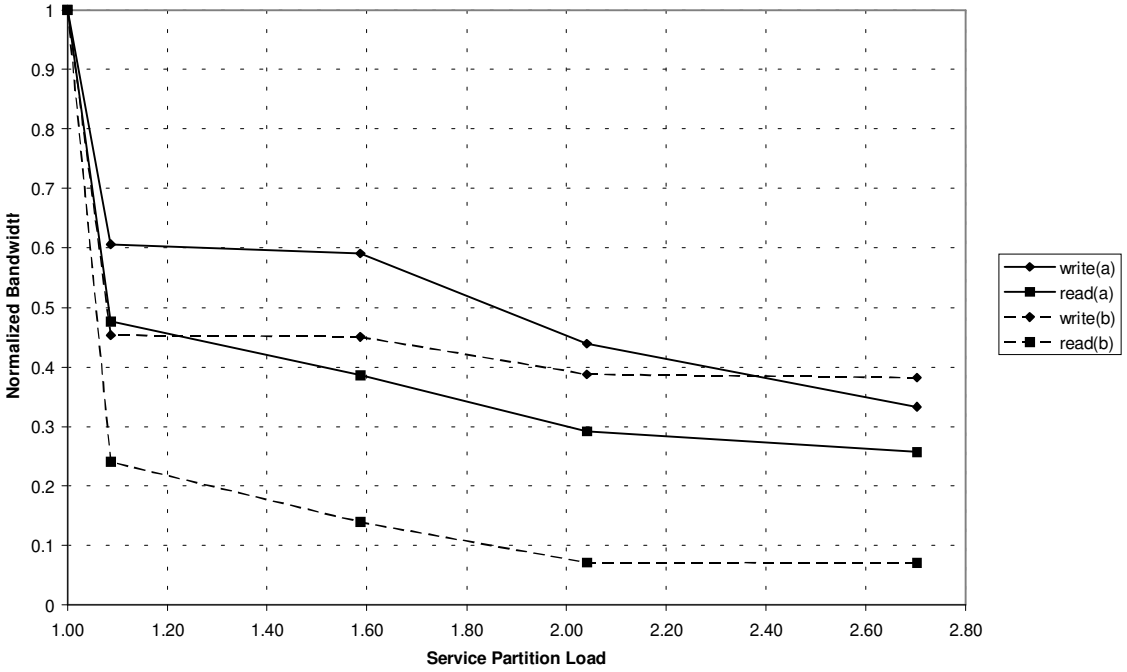

Figure 3: Effect of service partition load on I/O performance

3.2.SINGLE-NODE STUDIES

The remainder of this report is structured in a narrative fashion retracing the steps one goes through in characterizing I/O performance on ASCI Red. While single-node studies are not sufficient (and may in fact be misleading) to characterize large-scale I/O performance, they are a necessary first step in understanding I/O on this system. For these studies we employed the "rw" benchmark used by Intel for I/O studies on both Paragons and Janus (see source in appendix A). The API's that were studied were fwrite/fread with buffering (rw_fbuf), cwrite/cread (rw_c), read/write from C (rw), and read/write from Fortran (rwf).

Single node and small-node studies, particularly of UFS files, are susceptible to caching effects which can produce misleading results. Figure 4 shows the results of runs varying the request size for a single node reading/writing a 16 MB file, which is typical of a restart file generated by a single node for applications at Sandia. These results would indicate that UFS files provide reasonable performance for reads using the C interface with request sizes above 256KB, and that for writes using the streams interface with buffering would provide performance between 5 and 7 MB/sec. The other obvious conclusion from this data would be that I/O rates were erratic. However, in Figure 5, for which we see the same plots for 300 MB files, a different story emerges. Figure 5 indicates that for UFS, regardless of API or request size, a read bandwidth of ~5 MB/sec and a write bandwidth of ~3 MB/sec.

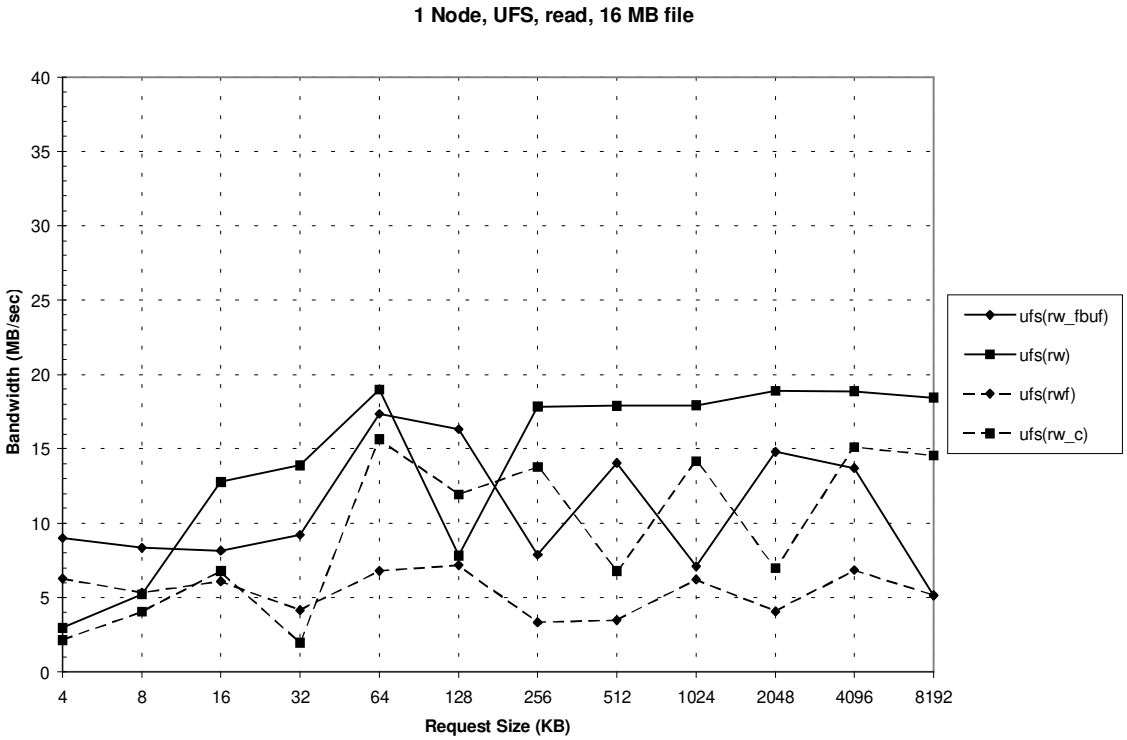

Figure 4a

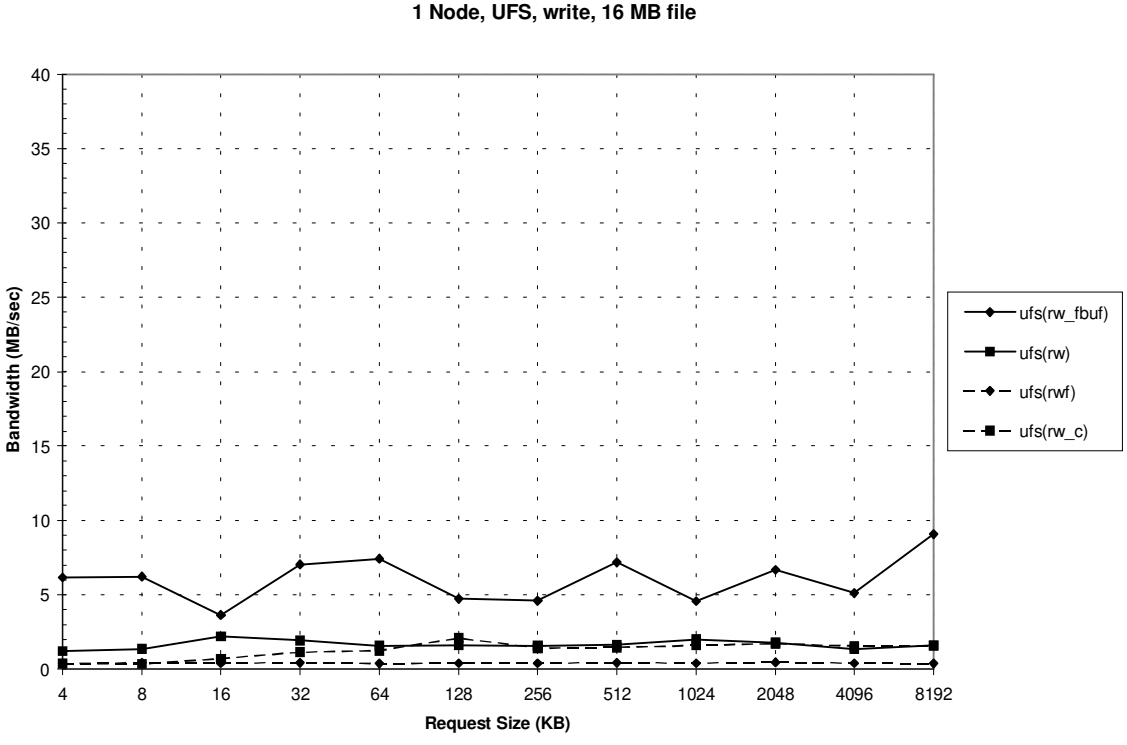

Figure 4b

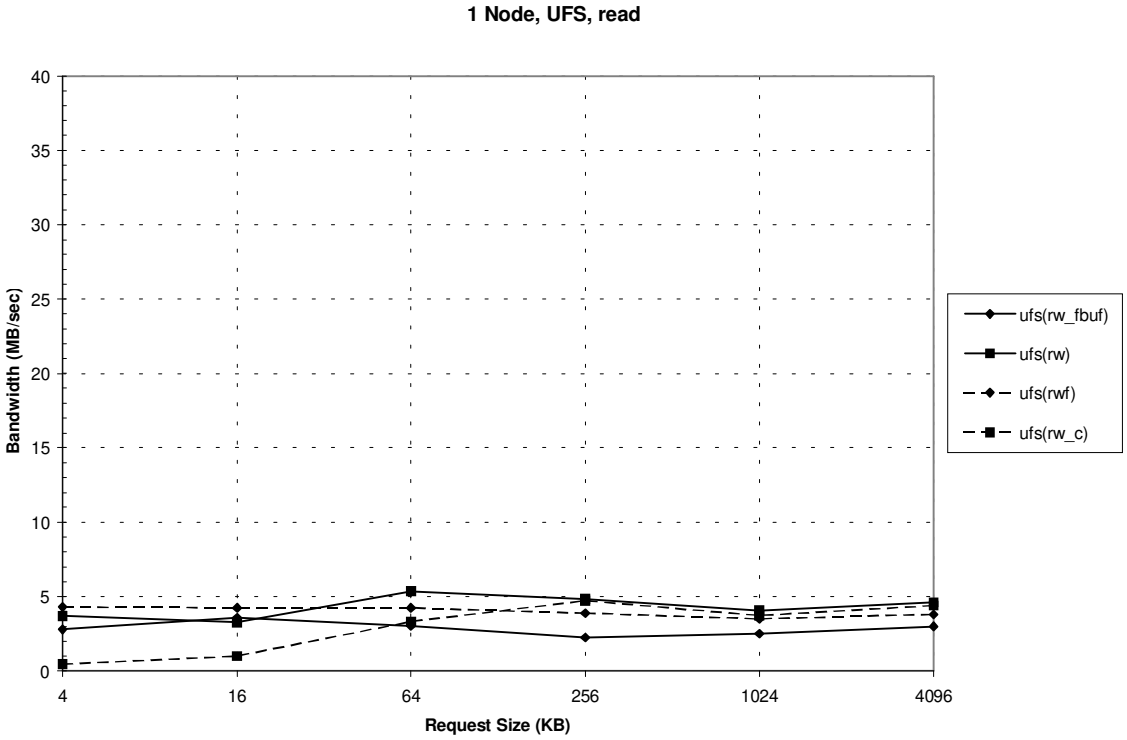

Figure 5a: Single node reading from UFS

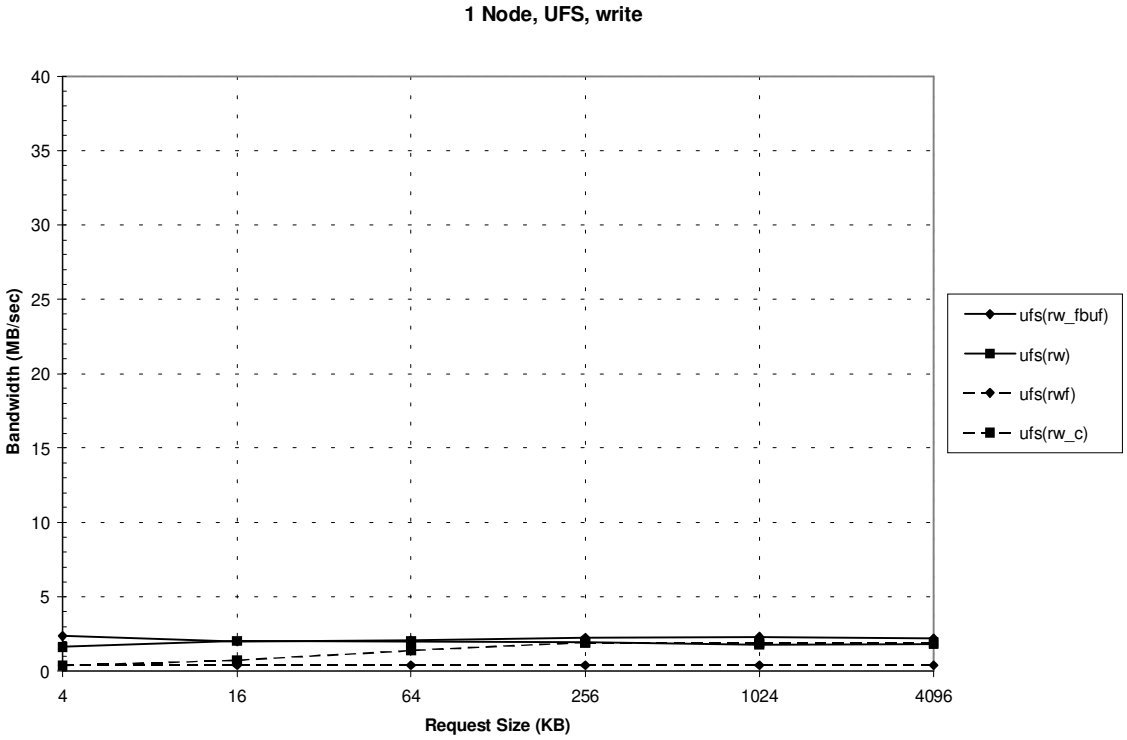

Figure 5b: Single node writing to UFS

The reason that the 16 MB data shows higher, if erratic, performance is because UFS I/O is cached on the service nodes using virtual memory techniques. This process, which means that pages of memory corresponding to sections of a UFS file can remain accessible to service nodes even after the termination of the yod/fyod that spawned them and hence be accessible without actually contacting the disk, makes the observed performance of UFS for small files a function of the recent history of the entire system. Further, for large-scale applications there will not be sufficient virtual memory space to satisfy the requests from a large number of compute nodes. This means that the performance for a large number of nodes attempting to simultaneously access UFS files will resemble the 300 MB file performance, even if the individual files are smaller. For this reason, all remaining single-node results are presented for 300 MB files.

The next study is a comparison of the API's discussed for the 2-way striped PFS and UFS file systems, the PFS results for which results are shown in Figure 6. For PFS, unlike UFS, performance is a strong function of request size, and varies between the various interfaces. For reads, we see that for small request sizes the streams interface with buffering is best, owing to the effect of having the data buffered on the compute node. (i.e. for small request sizes, most requests are satisfied by memcpy's.) Above 64 MB, the standard C read/write interface is preferable for single-node I/O. For writes, below a crossover point at about 512 KB streams I/O dominates, while above this the read/write interface is the clear winner.

Because there are 2 RAIDS participating in the PFS file system whose performance is shown in Figure 6, we cannot legitimately compare Figures 5 and 6. We now introduce the concept of a scaled, or "per-RAID," performance. However, by dividing the PFS bandwidth by the numbers of RAIDS participating in the PFS file system one obtains a scaled performance which can be compared with UFS. Figure 7, then, is the scaled version of Figure 6. The degree to which that performance approaches 30 MB/sec, then, is an estimate of how close we are to the design target of the machine. In particular, for Figure 6 we see that the performance per SCSI is in the 10-15 MB/sec range for large requests, and falls dramatically as the I/O request size decreases. Comparing Figures 5 and 7 provides an "apples to apples" comparison of PFS I/O performance vs UFS I/O performance, showing that even on a per-RAID basis for large-scale I/O PFS is preferred.

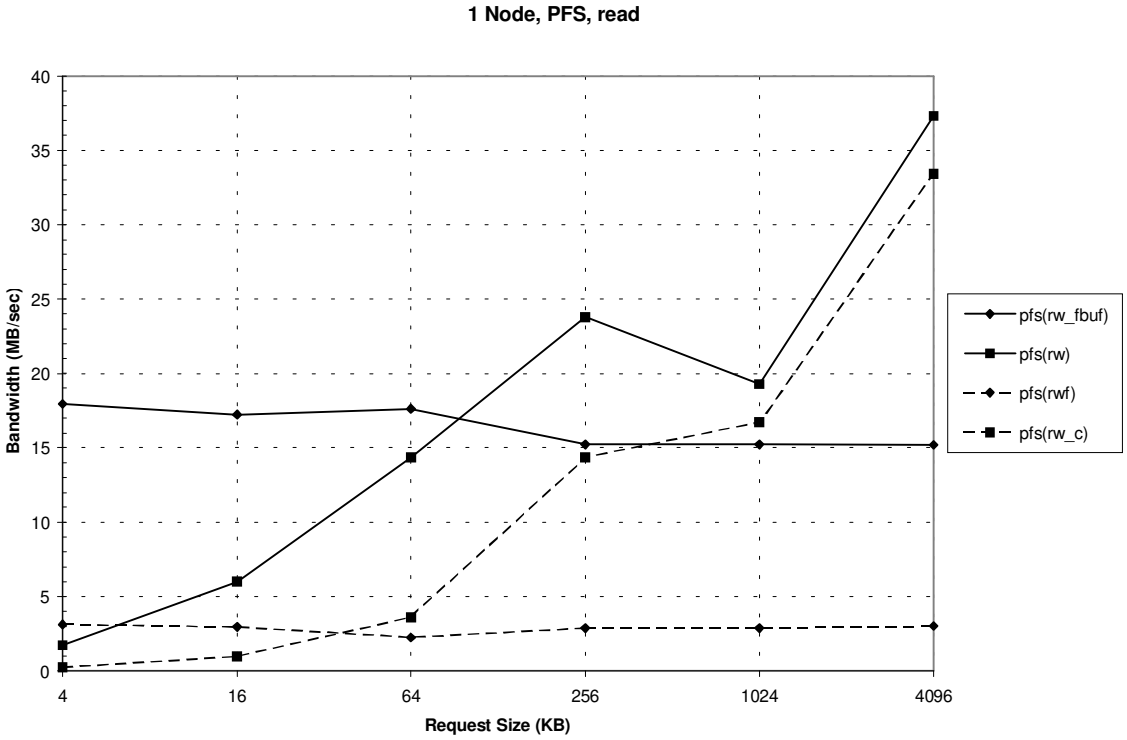

Figure 6a: Single node reading from a 2-way striped PFS

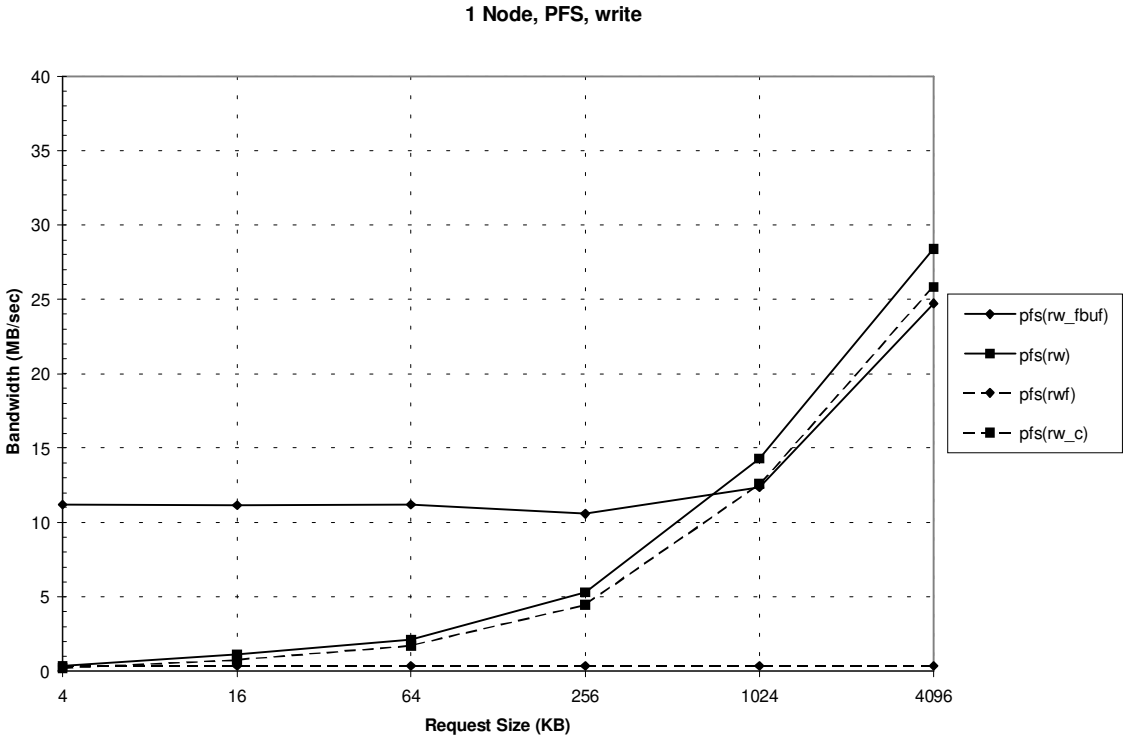

Figure 6b: Single node writing to a 2-way striped PFS

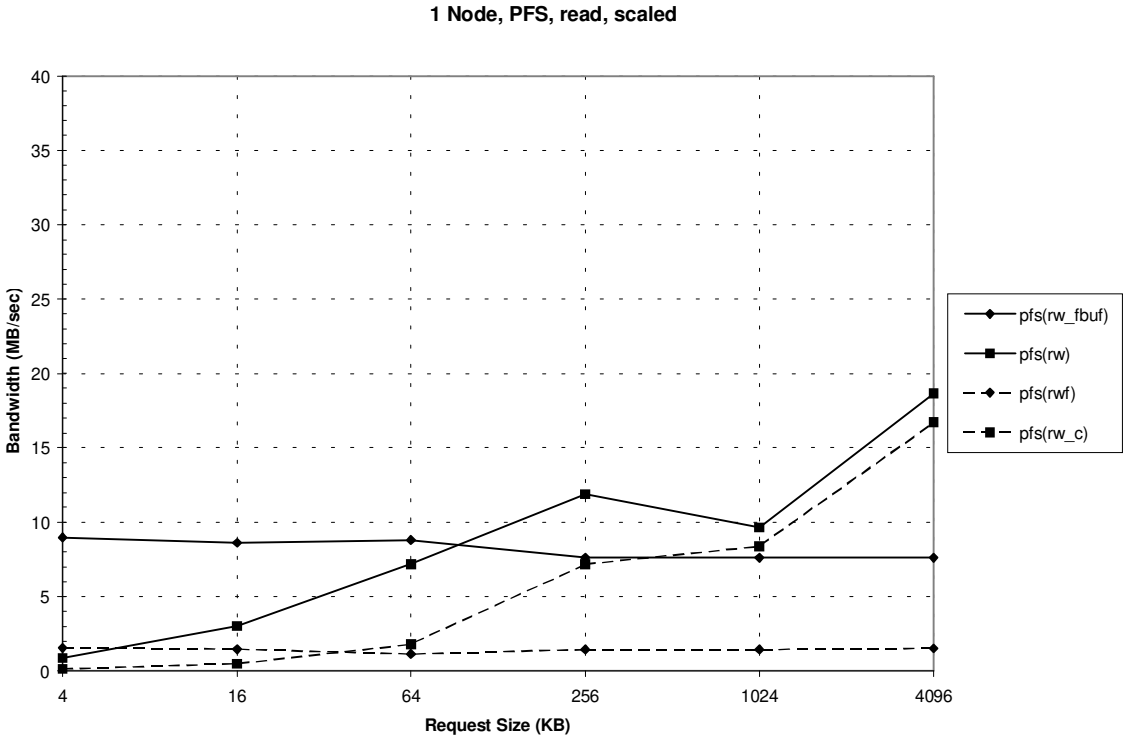

Figure 7a: Single node reading from 2-way striped PFS, showing per-raid performance

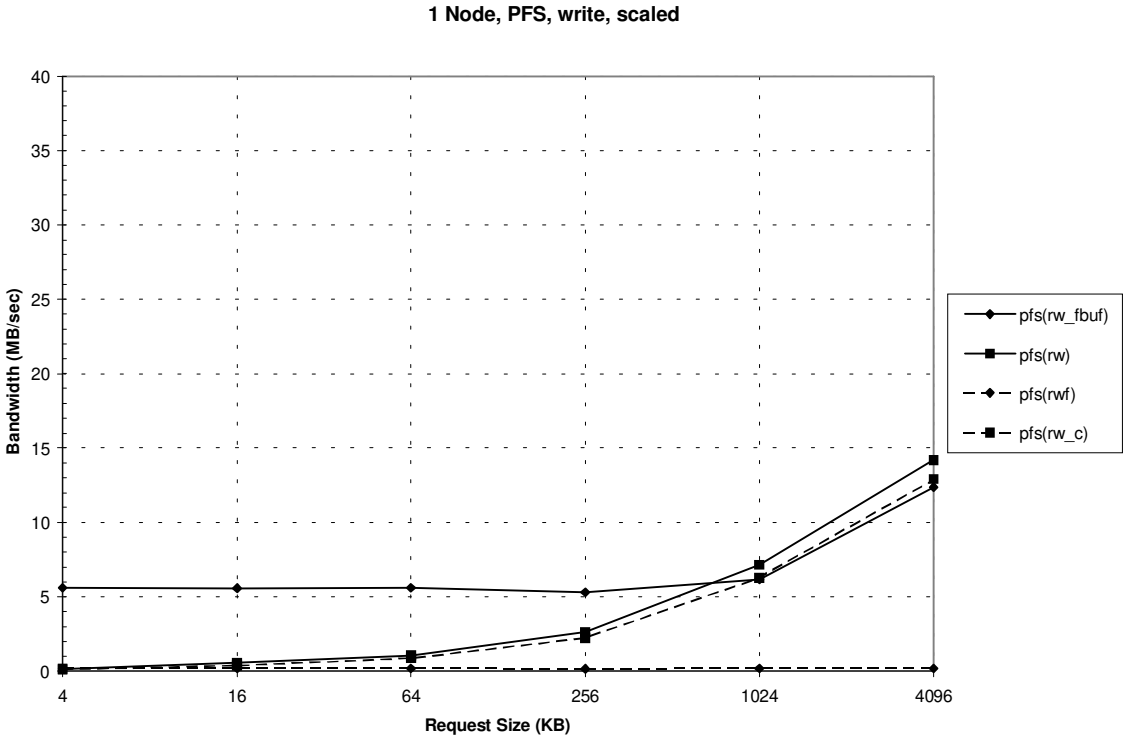

Figure 7b: Single node writing to 2-way striped PFS, showing per-raid performance.

3.3.MULTI-NODE MULTI-FILES STUDIES I

We now look at some simple cases of multi-node applications performing I/O. For these studies, the bandwidth shown in the plots is the aggregate over all compute nodes; each node sees 1/N of this bandwidth (where N is the number of compute nodes performing I/O). There are two many-to-one conditions that can occur in these instances: between I/O requests and the fyod daemons, and between I/O requests and the I/O nodes themselves. An additional potential bottleneck is the number of service nodes on the system. If there are multiple fyods on a single service node they will compete for resources. Our test system, known as Nighten, has a 4-way striped PFS file system with a block size of 512K and a stripe factor of 1 MB. During these tests Nighten was configured to have 5 service nodes (excluding the boot node).

3.3.1. Effect of the number of fyods on Performance

For UFS I/O, data moves through the fyods as shown in Figure 1. Hence in the absence of other considerations, one might expect to see better UFS performance as one increases the number of fyods. This only happens for files that have been read and are still in the fyod 'file cache'. Increasing the number of fyods for ufs increases the amount of memory available to your application for 'file cacheing', however, several limiting factors come into play:

- If more than one fyod exists on a service node, then the two fyods will compete for CPU cycles and memory on this service node, *regardless of whether the two fyods belong to the same yod, or even the same user.*
- For read()/write() file requests to UFS, the fyod allocates a data buffer of up to 128KB and breaks up I/O requests into 128KB chunks. In addition, the operating system buffers file data on the service node. For asynchronous UFS I/O requests, the fyod allocates a buffer equal in size to the total size of the request. Many simultaneous cwrite()s for example, cause the fyod to consume lots of memory on the service node. Combined with the memory-mapped buffer pool, this can cause paging, and system performance can slow down. Multiple fyods on a service node will aggravate this. The easiest way to avoid this is to use PFS. Once paging sets in, UFS I/O performance drops into the 0.2 MB/sec range. Given that the paging sets in due to trying to access lots of UFS data simultaneously (with the async I/O APIs), accessing that data at 0.2 MB/sec can have a very severe impact on an applications performance.
- Since a UFS file system resides on a single I/O node, requests from multiple fyods are likely to queue up on the I/O node and be processed sequentially.
- Increasing the number of fyods increases the likelihood that you will land on a busy service node, which will slow down I/O.

Even when one has dedicated access to the entire system, diminishing returns set in when the number of fyods exceed the number of service nodes. In a more typical user environment, it is observed that UFS I/O performance is sensitive to the activity on the particular service nodes that one's application is using for its fyods. As we can see from Figures 8, 9, and 10, UFS doesn't particularly benefit from increasing the number of fyods.

For PFS I/O, the fyods play a much smaller role. The fyods do not buffer data, they just pass the requests along to the I/O nodes, and the operating system does not buffer data. PFS I/O is less susceptible to other activity in the service partition and needs fewer fyods to achieve peak performance (further reducing service partition load). In general, not much I/O performance improvement has been seen for more than one fyod per 128 nodes for PFS if one is using the asynchronous APIs (cread/cwrite/iread/iwrite/fwrite). For the synchronous APIs (read/write/fread) performance increases with the number of fyods (see Figures 10a,b) and approaches the asynchronous API performance. This is because the asynchronous API starts one thread per file and by increasing the number of fyods, the synchronous 'one thread per fyod' PFS API moves closer to the 'one thread per file' asynchronous PFS API.

We demonstrate the impact of varying the number of fyods by considering a 4-node application that uses 4 fyods, for which each fyod has its own service node. Figure 8 shows the results of this experiment. We wrote large files so the impact of UFS caching is minimized, and for the streams interface used a 1 MB buffer, which is why the results all match the 1 MB value in Figure 8a. The other PFS files show performance strongly dependent on request size. Figure 9 shows the results of forcing the 4 nodes to share a single fyod. The UFS performance is relatively unchanged, as regardless of how many fyods are used all data flows to a single RAID. The PFS I/O performance drops for the synchronous APIs (fread/read/write) from that seen in Figure 8 because now their are fewer processes (1 fyod vs 4) to handle synchronous requests. For the asynchronous APIs (cread/cwrite/fwrite) then you don't need to have more than one fyod per 128-256 nodes as you can see from figure 10. Using the asynchronous API with PFS is preferred versus using the synchronous APIs and increasing the number of fyods.

Figure 10 shows a further illustration of this concept. In this experiment, a series of 64-node runs were performed with varying numbers of fyods. As the size of the files was sufficiently large to avoid caching effects, the performance for UFS files did not show a correlation, holding in the 5-10 MB/sec range. For PFS files with asynchronous APIs, there is also little correlation, as the fyods perform their portion of handling the I/O requests quickly, without serialization. However, for PFS with synchronous APIs, there is serialization at the fyod level. This is why there is a correlation between the performance for these APIs and the number of fyods, and why this performance stops rising once it reaches the performance for the asynchronous APIs.

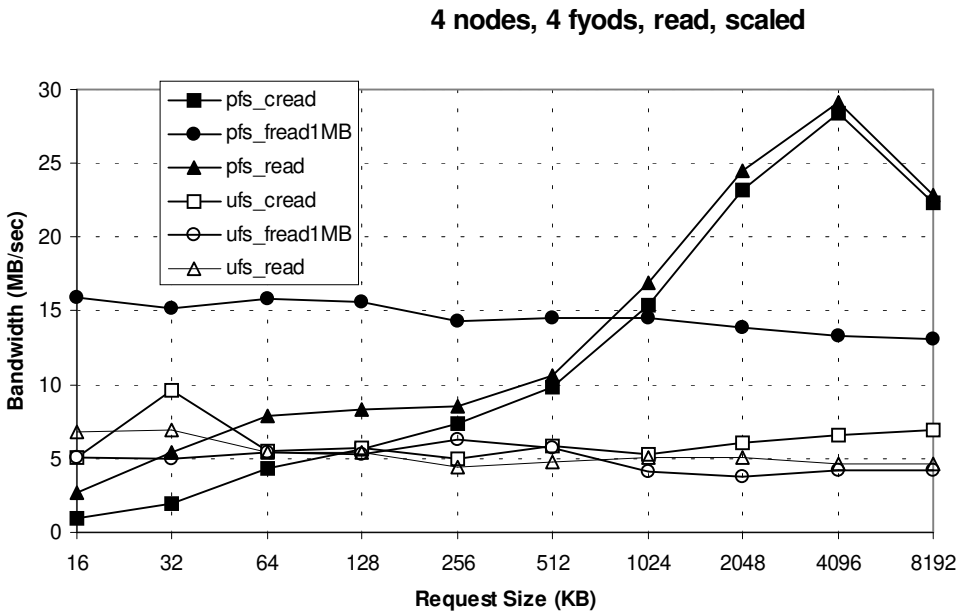

Figure 8a - Read performance as a function of request size for 4 nodes with 4 fyods.

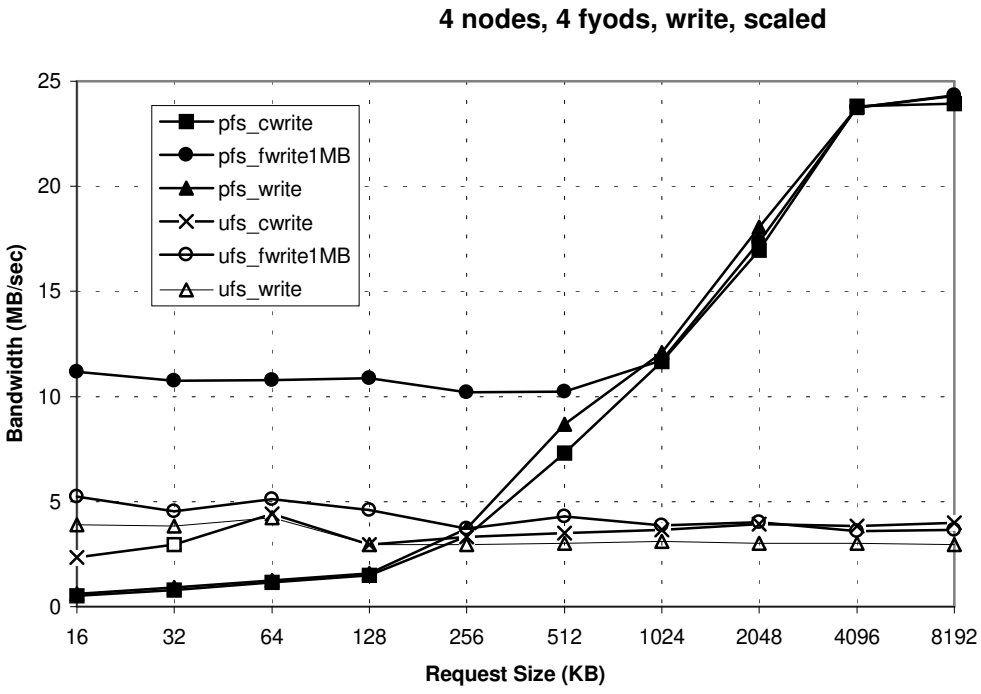

Figure 8b - Write I/O performance as a function of request size. Compare this figure with figure 9b to see the impact of reducing the number of fyods.

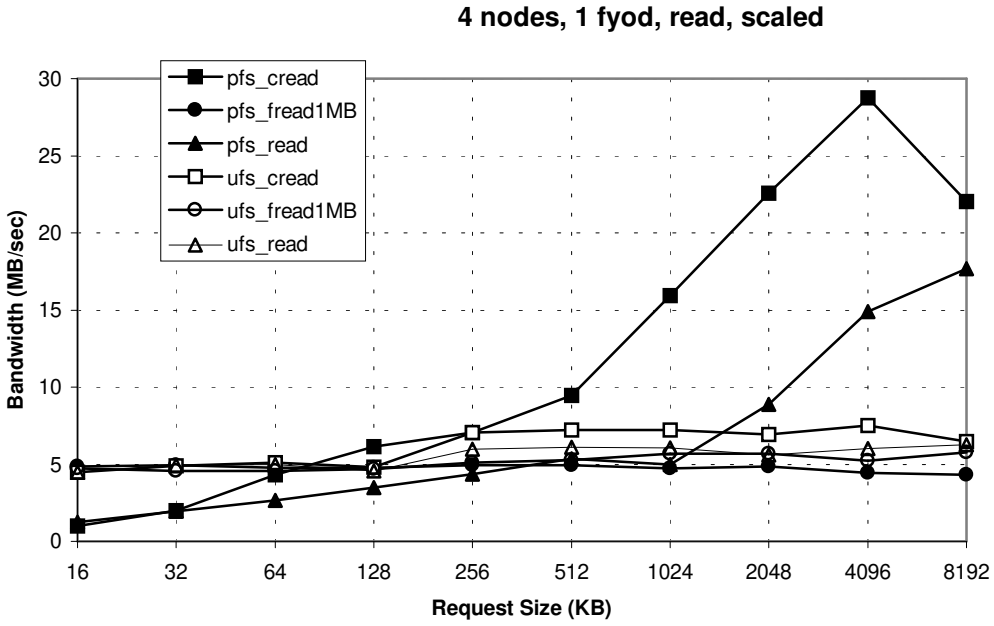

Figure 9a - Read I/O performance as a function of request size. pfs_fread1MB has dropped due to only having 1 fyod to process I/O requests. Pfs_cread drops for request size 8192KB due to the pfs raid being 4-way striped. Optimal I/O is achieved for multiples of the 4-way stripe size.

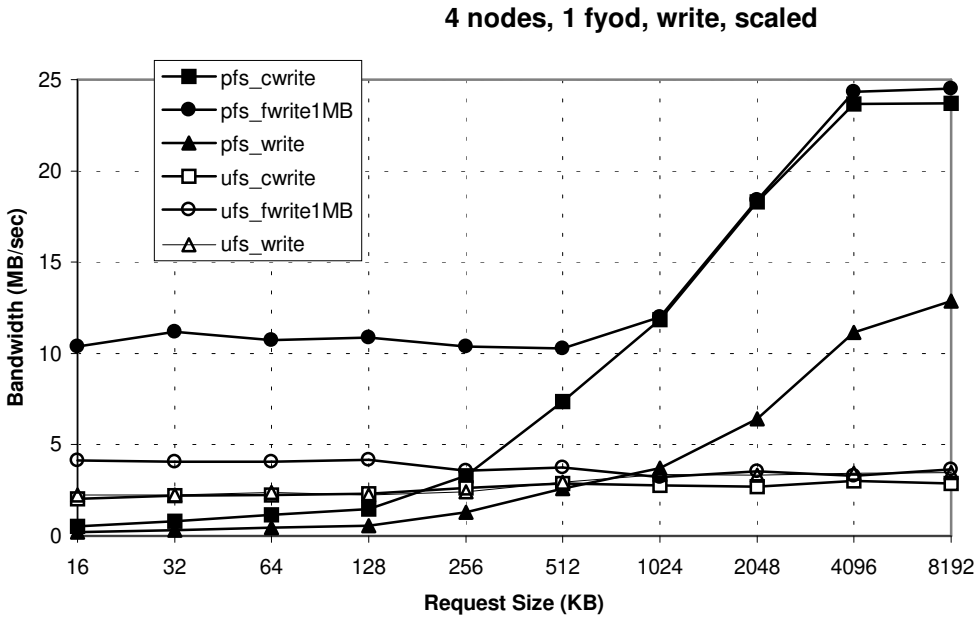

Figure 9b - write performance as a function of request size. Note that pfs_fwrite1MB is asynchronous so it should (and does) perform as well as pfs_write with a 1MB request size.

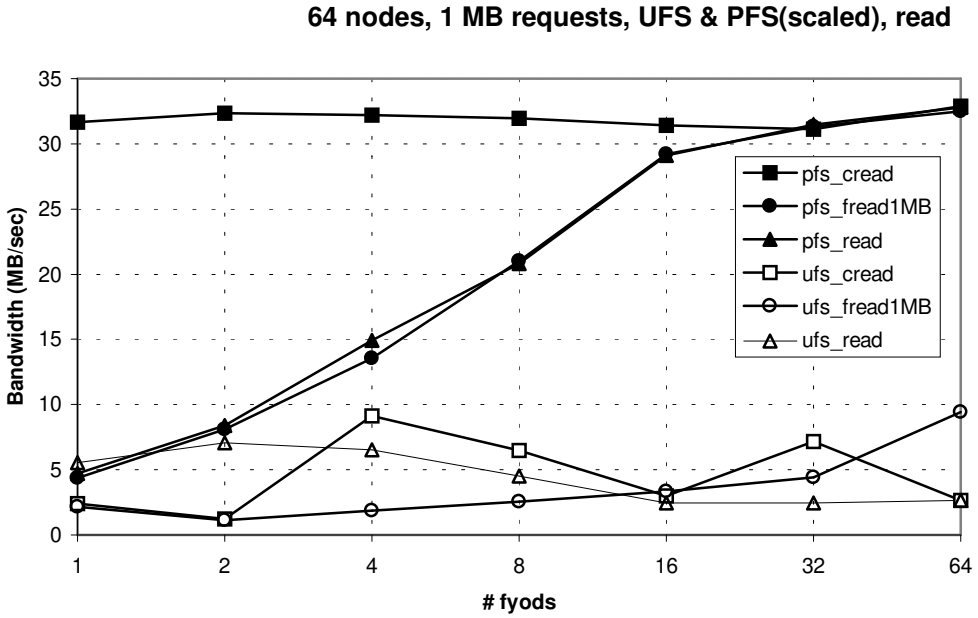

Figure 10a - Read performance as a function of the number of fyods. Pfs_cread (asynchronous API) starts 64 threads to handle writing to the 64 files. Pfs_cread performance is relatively independent of the number of fyods. This test was run on a system with only 5 service nodes so multiple yods and fyods were started on each service node. Pfs_fread1MB and pfs_read perform as well as pfs_cread once the number of fyods approaches 1 fyod per file.

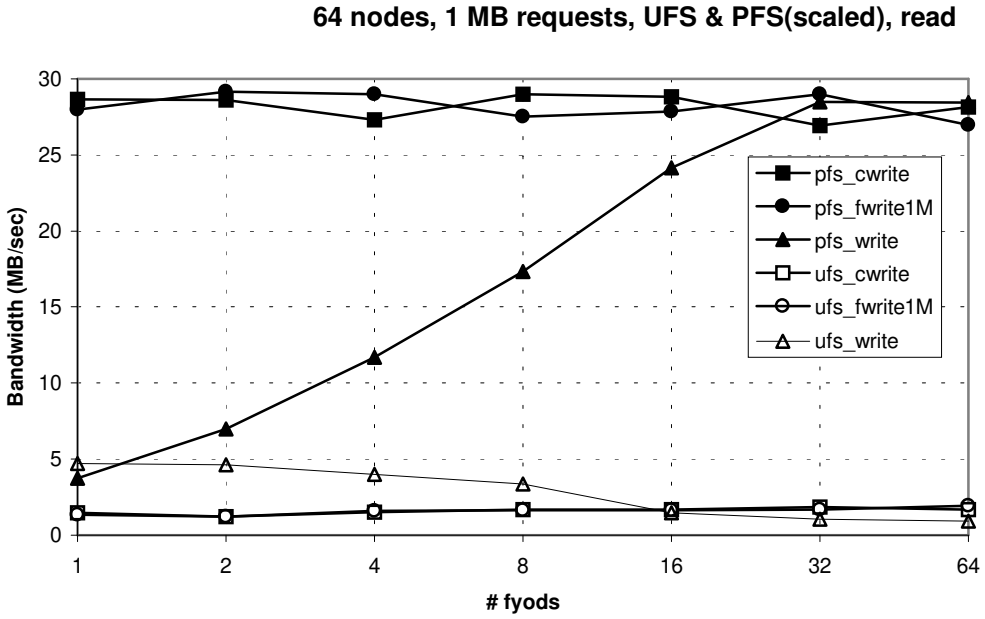

Figure 10b - Write performance as a function of #fyods. Only synchronous pfs_write is greatly impacted by increasing the #fyods.

3.4.MULTI-NODE SHARED FILE STUDIES

We now examine the impact of multiple nodes accessing the same file, both for reading and writing. In all our tests for this section, we wrote or read a minimum of 1 Gb total to avoid caching. During the first of these tests we fixed the request size at 2 Mb and varied the number of nodes, the results of which are shown in Figure 11. We see not only that PFS outperforms UFS, but also that the use of -masync mode yields a dramatic performance improvement over unix mode. The choice of mode for UFS files had negligible impact.

For the second study we chose two sizes for the application (16 and 64 nodes) and varied the request sizes from 64 kilobytes to 2048 kilobytes. During these tests we varied both the file system type (PFS/UFS) and the I/O mode (-masync/-munix) Figure 12 shows the results from these studies. For both 16 and 64 nodes, we see again that using the -masync mode provides significant performance improvement. Further, we also note that making the requests (and hence the seeks) a multiple of the stripe factor has an impact on performance.

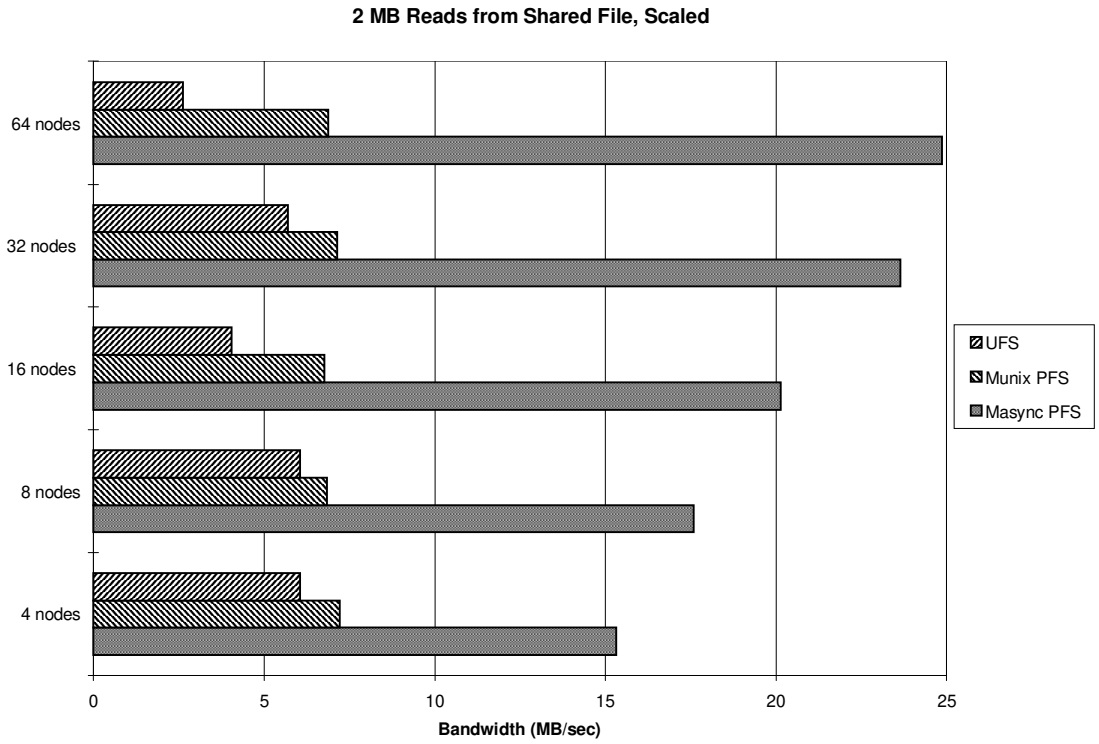

Figure 11a

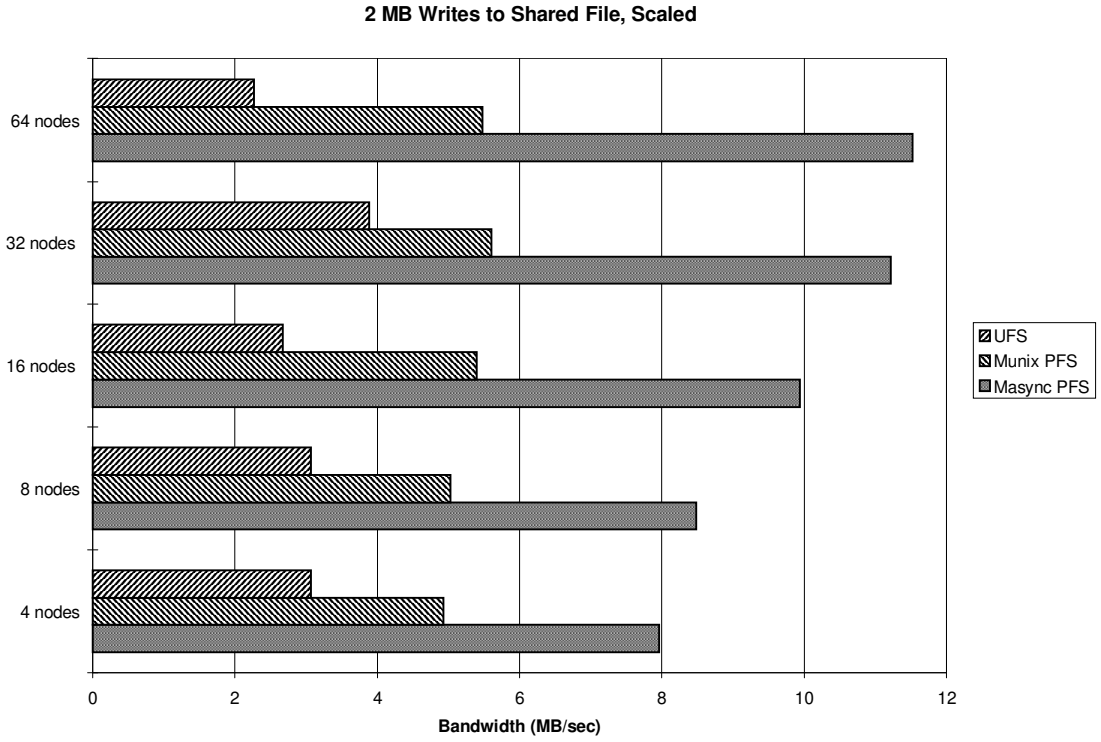

Figure 11b

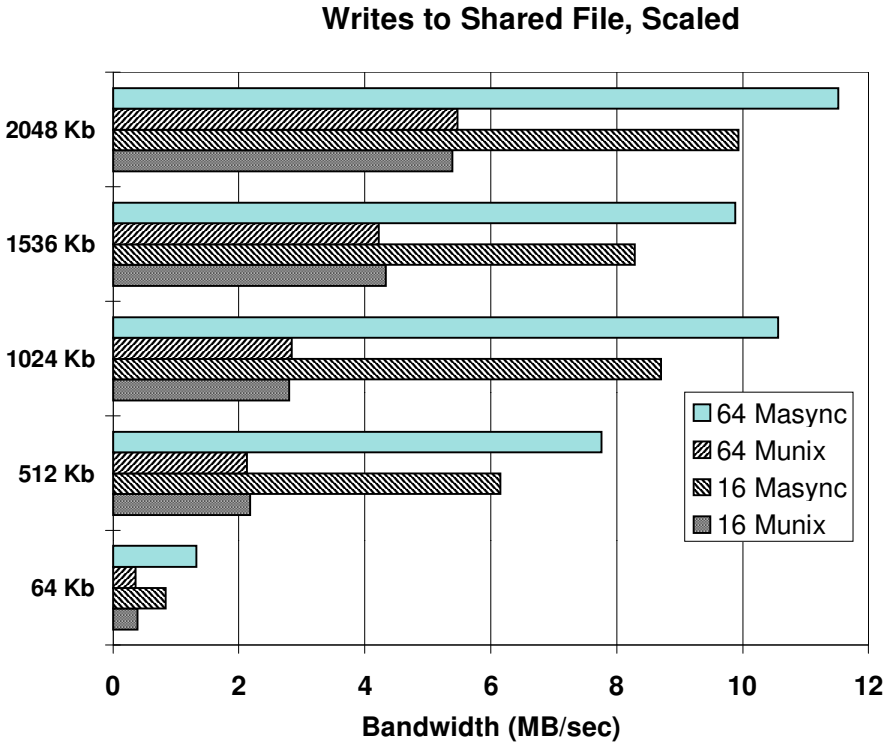

Figure 12a: Performance Results from writing to a shared file

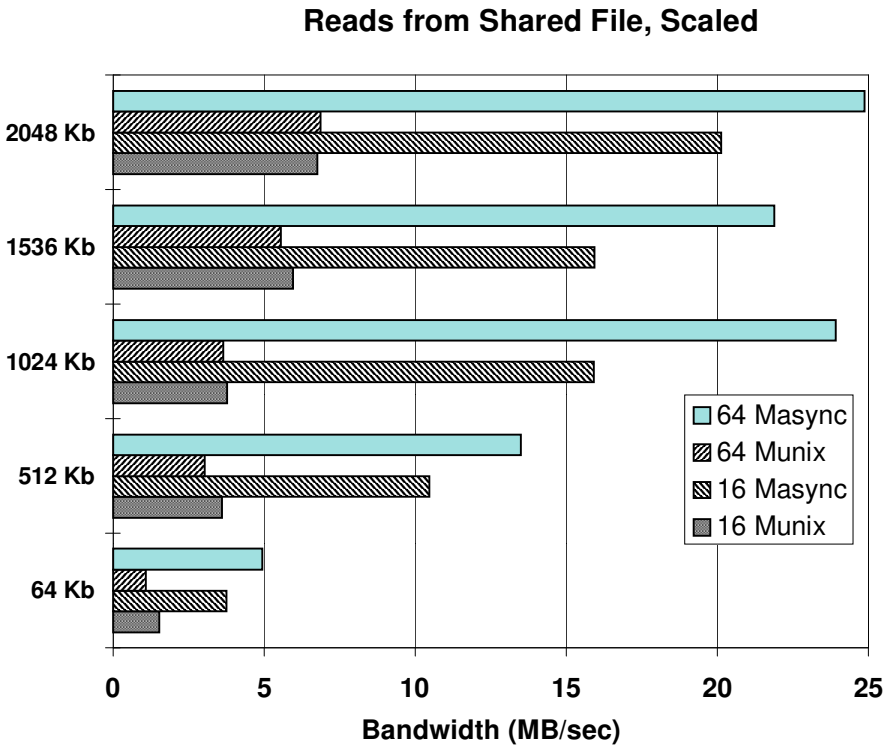

Figure 12b: Performance results from reading from a shared file

3.5.MULTI-NODE MULTI-FILES STUDIES II

Given these simple examples, we are now in a position to investigate the I/O performance of Janus. Because of the sensitivity of I/O performance to the activity level of the service partition, I/O tests must be done in a dedicated mode to be reproducible. As Janus is a heavily subscribed machine, this time is in short supply. For this reason, the results presented in this section are sketchy, and are presented as tables rather than as plots.

Currently, Janus has 14 I/O nodes dedicated to providing PFS access, and 6 RAIDS dedicated to providing UFS access. Janus in its large configuration has 3072 nodes, and in its small configuration has 1192 nodes. Attempting to compute a representative ratio of compute nodes to I/O nodes (or file systems, in the case of UFS) yields a range of numbers, most of which are O(10^2). Rather than looking at machine-sized runs, we chose instead to consider runs of size 128 nodes as a starting point for this reason.

Table 1 shows the results of such runs on Janus under OS Release 2.1 while operating in a single-user mode, using the (cwrite/cread) and (fwrite/fread) interfaces. In this experiment we used 2MB requests, and each node accessed a 16 MB file. This data indicates that the behavior on Janus for reasonable applications tracks the performance seen on the small test system, and demonstrates that with proper care good I/O performance is obtainable on this system. In a related experiment, a run on 1192 nodes using PFS-multi yielded bandwidths of 782.3 write, 710.3 read.

Request size	cwrite	cread	fwrite	fread
Multi				
512	563 (20.1)	314 (11.2)	616(22.0)	211(7.54)
1024	650 (23.2)	536 (19.1)	671(24.0)	208(7.42)
2048	655 (23.4)	628 (22.4)	680(24.3)	210(7.50)
4096	646 (23.1)	670 (23.9)	638(22.8)	214(7.64)
PFS				
512	28.7 (14.4)	56.0 (28.0)	37.1(18.6)	54.4(27.2)
1024	34.9 (17.4)	55.1 (27.6)	34.2(17.1)	53.9(27.0)
2048	34.8 (17.4)	53.9 (27.0)	(no data)	(no data)
4096	38.0 (19.0)	54.6 (27.3)	(no data)	(no data)

Table 1: Performance comparison on Janus in dedicated mode using cwrite/cread and fwrite/fread interfaces from 128 nodes. The numbers in parentheses for PFS are the scaled ("per-RAID") performance.

In a different experiment, a varying number of fyods and compute nodes were run on Janus to see how the scaling studies presented in preceding sections mapped over onto the full system. The results for these runs are presented in Tables 2 and 3. These results indicate that in the limit of small applications (at least) a ratio of 64 compute nodes to 1 fyod seems about optimal. We did not have the opportunity to press this study into the regime where the number of fyods exceeded the size of the service partition; we expect this scaling would break down at that point.

	# Cougar Nodes					
# fyods	16	32	64	128	256	512
1	174	243	220	138	62	
2	174	328	480	442	279	104
4	172	345	520	681	718	724
8	179	313	529	675	800	690
16	176	326	488	594	607	744

Table 2: Aggregate Write bandwidth in MB/sec as a function of number of compute nodes and number of fyods for asynchronous API on Janus under 2.1 in dedicated mode.

	# Cougar Nodes					
# fyods	16	32	64	128	256	512
1	224	331	373	184	68	
2	210	370	463	446	374	113
4	226	379	488	565	618	635
8	229	354	436	552	719	605
16	229	294	445	557	640	653

Table 3: Aggregate Read bandwidth in MB/sec as a function of number of compute nodes and number of fyods for asynchronous API on Janus under 2.1 in dedicated mode.

4. CONCLUSIONS and COMMENTS

The purpose of this section is to summarize the results of this report and to provide recommendations to the user community. One important but often-overlooked recommendation is simply to minimize the amount of I/O that is performed. It is worthwhile to do a few calculations on paper to estimate what resolution is necessary rather than just taking the highest resolution that can be obtained. Or, to use the vernacular, "Do as much I/O as you need, but no more." For applications programmers, whenever possible use a good I/O library. Your Intel Computational Scientists will be helpful in identifying existing libraries and providing you with contact information (e-mail janus-help@sandia.gov.)

When doing I/O programming, first know your I/O parameters. How much data are you moving per request? Are you reading one value per read statement? Are you using ASCII files? Is your average single I/O request less than 256 KB? If so, and if you are not using the streams interface from C, then it probably makes sense to use UFS, and you should realize that you will have an upper bound of a few MB/sec per file system. The read()/write() interface is probably the preferred interface to use here, as it performs about as well as any and has the "Good Citizen" feature of not potentially causing paging in the service partition.

If you are using streams but one or more of the rest of the questions in the preceding paragraph apply to you, you should consider using setvbuf () to increase your on-node buffer size. This is done at the expense of memory on the node, but a 1 Mb buffer is less than 1% of the memory on a node, so the tax really isn't that severe. Once this is done, you should be able to get good performance from PFS. If you're doing big I/O, then use cread/cwrite. These perform best and are less sensitive to service-node load.

Don't play "fyod games." The only case where it helped was for synchronous APIs to PFS files. Again, if you really want performance, change APIs. The default is 1 fyod per 256 compute nodes: This works reasonably, and you certainly don't need more than 1 fyod per 64 compute nodes. Further, the upper limit on the number of fyods should be the size of the service partition (count the "S" letters in a showmesh output). Increasing the number of fyods increases the likelihood of landing on a busy service node.

If you're using a shared-file model, always use -masync. Don't leave holes in your files.

Consider others. If your I/O is "ugly" and takes a long time, it's slowing I/O down for others as well.

Finally, keep your Intel staff informed of what you are up to when you start changing how your application does I/O. The system may have changed since this report was generated, or your change may trigger some undesirable feature in the system in a way that is not obvious. In either event, everyone wins if the Intel folks know what is changed.

5. Appendix A: Sample RW Test Code used

For the sake of brevity, only the version used for the cwrite/cread tests is reproduced here; the others only differ in the appropriate file "handle" used (stream pointers or file descriptors). The two pieces rw_c.c and timer.c are compiled using "pgcc -cougar -DPARAGON -DNX -o rw_c rw_c.c timer.c". Copies of all sources, along with a Makefile, are on Janus in /Net/projects/intel_cs/rw_source.

rw_c.c

```
/*
 * Name: rw
 * Usage: rw [-nqw] [-m mode] <file> <request size in KB> [<file size in MB>]
 * Read or write a file of a specified size, using a specified I/O request size,
 * and display performance statistics.
 * Author: Brad Rullman, Intel Supercomputer Systems Division, 1994. Send
 * suggestions, comments, bug reports to brad@ssd.intel.com. You are free to
 * distribute this software, as long as this notice is not removed.
 * Note: This is Ben Cole's hacked-up version for Cougar using cwrite/cread
 * -q   Quiet.  Only two lines of terse output are generated which include
 * real time spent performing the I/O, CPU time, bandwidth, and number of I/O
 * calls/sec.  Without this switch, a more verbose summary of input data and
 * output data in long format is displayed.
 * -w   Write the file specified by <file>.  If this switch is not specified,
 * the default action is to read the file.
 * Description: This program reads or writes a file of a specified size, using a
 * specified I/O request size, and displays benchmark statistics describing
 * the resulting I/O performance.
 * The <file> argument specifies a path to the target file.
 * <request size in KB> specifies the size of the buffer used in the read() or
 * write() system call.
 * When writing, the <file size in MB> argument is required and specifies the
 * size of the file to create.  Only full requests are written, so if <file
 * size in MB> * 1024 is not a multiple of <request size in KB> the resulting
 * file size may be larger than what was requested.  When reading, the <file
 * size in MB> argument is optional and specifies how much of the file is to
 * be read;  if not specified, the entire file is read.  When executing rw as
 * a parallel application on multiple nodes, this argument specifies the
 * amount of data written or read by each node, thus the total file size may
 * vary depending on the file sharing mode used.  For example:
 *
 * If the file is being written, the performance results include the time
 * required to sync all file data to disk via fsync() to avoid any gains from
 * possible data caching done by the OS.  If the file is being written and it
 * already exists, it will be truncated and overwritten (if the user has the
 * appropriate file permissions).
 *
 * When executing rw as a parallel application on multiple nodes, bandwidth
 * times displayed reflect the time required for all nodes to completely
 * finish reading or writing.  In other words, all nodes synchronize before
 * and after the I/O is performed.
 *
 * Depending on the file system being exercised, caution should be used when
 * interpreting benchmark results, due to possible data caching by the OS.
 * To defeat caching, it's best to use large file sizes (64 MB or more), and
 * alternate files being read.  For example, write one large file, write
 * another (filling any data cache with different data), then read the first,
 * and then read the second.
 */
#include <fcntl.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/limits.h>
#include <sys/dir.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <mach/mach_types.h>
#include <nx.h>
```

```
#define VM_PAGE_SIZE   8192
#define INFINITY       INT_MAX

#define MAXNAMLEN NAME_MAX
#ifdef  PARAGON
#ifndef MAXNAMLEN
#define MAXNAMLEN      NAME_MAX
#endif
#endif

#define perror         nx_perror

/* These globals are input from the command line.*/
char           *CmdName;
boolean_t       SeparateFiles = TRUE;
boolean_t       Verbose = TRUE;
boolean_t       Reading = TRUE;
char           *FileName;
int             ReqSize,FileSize;

boolean_t       Parallel = FALSE;
boolean_t       NXApplication = FALSE;
long            MyNode = 0;
long            NumNodes = 1;

extern double   RealTime;

extern void     Start_Timer();
extern void     Stop_Timer();
extern void     Print_Times();


main(argc, argv)
   int             argc;
   char          **argv;
{
   extern int      optind;
   extern char    *optarg;
   int             numloops, oflags;
   int             fd, numio, datacount = 0;
   char           *buf, *buf1, *p;
   register int    c;
   int             iomode = M_UNIX;
   char           *opstring, *modestring = NULL;
   int             error;

   /*
    * Set MyNode variable, used to determine which nodes display results.
    * Note MyNode is always 0 on non-parallel architectures.
    */
   MyNode = _mynode();
   if (MyNode < 0) {          /* not built with -nx */
      MyNode = 0;
      NumNodes = 1;
   } else {
      NXApplication = TRUE;
      if ((NumNodes = _numnodes()) < 1) {
         perror("numnodes");
         exit(1);
      }
      if (NumNodes > 1) Parallel = TRUE;
   }

   /* Parse command line arguments.*/
   CmdName = (p = rindex(*argv, '/')) ? ++p : *argv;
   while ((c = getopt(argc, argv, "nqw")) != EOF) {
      switch ((char) c) {
      case 'n':
         SeparateFiles = TRUE;
         if (!NXApplication) {
            perr0("%s: %s must be built with -nx to use -n\n",
                  CmdName, CmdName);
            exit(1);
         }
```

```
      if (modestring) {
         perr0("%s: Invalid to mix -m and -n switches\n",
               CmdName);
         exit(1);
      }
      break;
   case 'q':
      Verbose = FALSE;
      break;
   case 'w':
      Reading = FALSE;
      break;
   default:
      usage();
      break;
   }
}

argc -= optind;
argv += optind;

if (Reading) {              /* -w flag not used */
   if ((argc < 2) || (argc > 3)) usage();
} else {
   if (argc != 3) usage();
}

if (SeparateFiles) {               /* -n flag used */
   if ((FileName = malloc(MAXNAMLEN)) == NULL) {
      perror("malloc");
      exit(1);
   }
   sprintf(FileName, "%s###", *argv++);
   argc--;
} else {
   FileName = *argv++;
   argc--;
}

ReqSize = atoi(*argv++) * 1024;
argc--;

if (argc > 0) {            /* file size specified */
   FileSize = atoi(*argv++) * 1024 * 1024;
   numloops = (FileSize % ReqSize) ?
      ((FileSize / ReqSize) + 1) : (FileSize / ReqSize);
} else {
   FileSize = 0;
   numloops = INFINITY;
}

/*
 * Allocate a page-aligned buffer.
 */
buf = buf1 = (char *) malloc(ReqSize + VM_PAGE_SIZE);
if (buf == NULL) {
   perror("malloc");
   exit(1);
}
buf = (char *) (((int) buf + (VM_PAGE_SIZE - 1)) & ~(VM_PAGE_SIZE - 1));

/*
 * Open the target file, creating if necessary.
 */
if (Reading) {
   oflags = O_RDONLY;
   opstring = "read";
} else {
   int             i;
   oflags = O_CREAT | O_RDWR | O_TRUNC;
   for (i = 0; i < ReqSize; i++) {
      buf[i] = 'w';
   }
   opstring = "write";
}
```

```
   Start_Timer();
   fd = open(FileName, oflags, 0640);
   if (fd < 0) {
         perror("open");
         exit(1);
      }
   Stop_Timer();

   gsync();
   fd = open(FileName, oflags, 0640);
   /* Perform the I/O, time it, and display output.*/
   if (Verbose) {
      pmsg("\tfile name  = %s", FileName);
      if (SeparateFiles) pmsg("  (%d files)\n", NumNodes);
      else if (modestring) pmsg("  (%s mode)\n", modestring);
      else pmsg("\n");
      pmsg("\topen time  = %.2f sec\n", RealTime);
      pmsg("\tI/O size   = %d KB\n", ReqSize / 1024);
   }
   if (Reading)
      numio = doread(fd, buf, ReqSize, numloops, &datacount);
   else
      numio = dowrite(fd, buf, ReqSize, numloops, &datacount);
   if (Verbose) {
      if (Parallel) {
         pmsg("\tnum I/O    = %d %ss  (%d X %d nodes)\n",
              numio * NumNodes, opstring, numio, NumNodes);
         pmsg("\ttotal data = %.2f MB  (%.2f MB X %d nodes)\n",
              (double) datacount * NumNodes / 1024 / 1024,
              (double) datacount / 1024 / 1024, NumNodes);
         pmsg("\tbandwidth  = %.2f MB/sec  (aggregate)\n",
              ((double) datacount) / RealTime / 1024 / 1024)
              * NumNodes);
      } else {
         pmsg("\tnum I/O    = %d %ss\n", numio, opstring);
         pmsg("\tdata count = %.2f MB\n",
              (double) datacount / 1024 / 1024);
         pmsg("\tbandwidth  = %.2f MB/sec\n",
              ((double) datacount) / RealTime / 1024 / 1024);
      }
   } else
      Print_Times((u_long) datacount * NumNodes,
                  (u_long) numio * NumNodes);

   Start_Timer();
   close(fd);
   Stop_Timer();
   if (Verbose) pmsg("\tclose time = %.2f sec\n", RealTime);
   free(buf1);
   exit(0);
}

int doread(fd, buf, buflen, numreads, datacount)
   int              fd,buflen,numreads;
   char            *buf;
   int             *datacount;
{
   int              numpages = (buflen % VM_PAGE_SIZE) ?
   ((buflen / VM_PAGE_SIZE) + 1) :
   (buflen / VM_PAGE_SIZE);
   char             touch_page;
   int              i, j, actual, numio = 0, count = 0;

   if (Parallel) gsync();
   Start_Timer();
   for (i = 0; i < numreads; i++) {
      actual = cread(fd, buf, buflen);
      if (actual < 0) {
         perror("read");
         exit(1);
      }
      if (actual == 0) {      /* end of file */
         break;
      }
```

```
      /*
       * Ensure that data is fully mapped in to user space.
       */
      for (j = 0; j < numpages; j++) {
        touch_page = buf[j * VM_PAGE_SIZE];
      }

      numio++;
      count += actual;
   }
   if (Parallel) gsync();
   Stop_Timer();

   /* If a specific amount of data was requested to be read (i.e. the <file
    * size in MB> argument was specified) but the file being read wasn't that
    * big, then return with an error.*/
   if ((FileSize) && (count < FileSize)) {
      perr("%s: Error: EOF reached before specified file size\n",
           CmdName);
      exit(1);
   }
   /* Print a warning if the last read was not for the full request size.*/
   if (count < (numio * buflen)) {
      perr("%s: Warning: EOF reached, last read a partial\n",
           CmdName);
   }
   *datacount = count;
   return (numio);
}


int dowrite(fd, buf, buflen, numwrites, datacount)
   int             fd,buflen,numwrites;
   char           *buf;
   int            *datacount;
{
   int             i, actual;

   if (Parallel) gsync();
   Start_Timer();
   for (i = 0; i < numwrites; i++) {
      actual = cwrite(fd, buf, buflen);
      if (actual < 0) {
        perror("write");
        exit(1);
      }
      if (actual < buflen) {
        perr("%d: Error: requested %d bytes, actual = %d\n",
             CmdName, buflen, actual);
        exit(1);
      }
   }
   if (Parallel) gsync();
   if (MyNode == 0) fsync(fd);
   Stop_Timer();
   *datacount = numwrites * buflen;
   return (numwrites);
}

/* VARARGS1 */
pmsg(s, a1, a2, a3, a4, a5, a6)
   char           *s;
{
   if (MyNode == 0)
      (void) printf(s, a1, a2, a3, a4, a5, a6);
}


/* VARARGS1 */
perr0(s, a1, a2, a3, a4, a5, a6)
   char           *s;
{
   if (MyNode == 0)
      (void) fprintf(stderr, s, a1, a2, a3, a4, a5, a6);
}
```

```
/* VARARGS1 */
perr(s, a1, a2, a3, a4, a5, a6)
   char           *s;
{
   char            s2[80];
   if (Parallel) {
      (void) sprintf(s2, s, a1, a2, a3, a4, a5, a6);
      (void) fprintf(stderr, "Node %d: %s", MyNode, s2);
   } else {(void) fprintf(stderr, s, a1, a2, a3, a4, a5, a6);}
}


usage()
{
   if (Parallel) gsync();
   perr0("usage: %s [-nqw]  <file> <request size in KB> [<file size in MB>]\n",
         CmdName);
   exit(1);
}
```

timer.c

```
#include <stdio.h>
#include <nx.h>
#include <sys/resource.h>
double         RealTime;              /* real time */
double         CPUTime;               /* CPU time (sys + user) */
static double time0;          /* time at which timing started */
extern int     MyNode;
/* Prepare and start the timer.*/
void Start_Timer(){time0 = dclock();}

/*Stop and read the timer.  Returns real time, and CPU time (sys + user).*/
void Stop_Timer()
{
       double time1;
       time1 = dclock();
       RealTime = (time1 - time0);
       CPUTime = RealTime;
}
void Print_Times(nbytes, ncalls)
unsigned long  nbytes;
unsigned long  ncalls;
{
       if (MyNode != 0) return;
       printf("%ld bytes @ %d per IO in %.2f real sec = %.2f KB/sec (%.2f MB/sec)\n",
              nbytes, nbytes / ncalls, RealTime,
              ((double)nbytes) / RealTime / 1024,
              ((double)nbytes) / RealTime / 1024 / 1024);

       printf("%d I/O calls, %.2f msec/call, %.2f calls/sec, %.2f CPU sec\n",
              ncalls, 1000.0 * RealTime / ((double)ncalls),
              ((double)ncalls) / RealTime, CPUTime);
}
```