SANDIA REPORT
SAND2010-1915
Unlimited Release
Printed March 2010

A Brief Parallel I/O Tutorial

Lee Ward

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-1915
Unlimited Release
Printed March 2010

A Brief Parallel I/O Tutorial

Lee Ward
Department 01423 - Scalable System Software
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS1319

Abstract

This document provides common best practices for the efficient utilization of parallel file systems for analysts and application developers.

INTRODUCTION

A multi-program, parallel supercomputer is able to provide effective compute power by aggregating a host of lower-power processors using a network. The idea, in general, is that one either constructs the application to distribute parts to the different nodes and processors available and then collects the result (a parallel application), or one launches a large number of small jobs, each doing similar work on different subsets (a campaign).

The I/O system on these machines is usually implemented as a tightly-coupled, parallel application itself. It is providing the concept of a "file" to the host applications. The "file" is an addressable store of bytes and that address space is global in nature. In essence, it is providing a *global* address space. Beyond the simple reality that the I/O system is normally composed of a small, less capable, collection of hardware, that concept of a global address space will cause problems if not very carefully utilized. How much of a problem and the ways in which those problems manifest will be different, but that it is problem prone has been well established.

Worse, the file system is a shared resource on the machine -- a system service. What an application does when it uses the file system impacts all users. It is not the case that some portion of the available resource is reserved. Instead, the I/O system responds to requests by scheduling and queuing based on instantaneous demand. Using the system well contributes to the overall throughput on the machine. From a solely self-centered perspective, using it well reduces the time that the application or campaign is subject to impact by others. The developer's goal should be to accomplish I/O in a way that minimizes interaction with the I/O system, maximizes the amount of data moved per call, and provides the I/O system the most information about the I/O transfer per request.

WORKING WITH THE NAME SPACE - WHAT YOU NEED TO KNOW

All parallel file systems provide two major sets of functions. First, a "name space" that provides the ability to create, delete, and organize files. Second, a parallel I/O mechanism used to store and retrieve data.

The name space, too, is a shared, global, coherent system. If an application creates a file, that file is instantly visible and available to all nodes in the machine. The only reason this isn't some sort of broadcast disaster on the machine is that most nodes and other applications do not care. The developer needs to allow this assumption by the file system designers to be true.

Rule NS1: Don't watch the pot. If you like to use `ls' to gauge the progress of your application, this means you! While your application is running you must avoid doing anything in the directory or directories in which the application is working. The acquisition of those file attributes can cause the locks for the related files to be withdrawn from all of your running nodes. Once acquired, the locks will have to be returned.

The name space function is implemented in different ways by different file systems. In some, it can be a single server, and in others a small collection of server. Even when implemented by multiple servers, some file systems partition the duties by directory, so that, in essence, a single directory can only be handled by a single server. The safest approach is to always think of it as a single server. Those servers can only process a limited number of interactions per unit time.

Moreover, that rate is limited by the size of the directory. The data structures used to implement the directory concept are not linearly scalable. The more files there are in a directory, the more overhead to process any operation involving that directory - *any* operation, including finding a file by name.

Rule NS2: Create and manipulate the fewest number of files in a given directory. Be reasonable, though. Accessing a file or its content from different nodes simultaneously is almost always an issue.

If you absolutely must create a large number of files, consider using multiple directories. For those file systems that have some sort of parallel metadata service, this approach will allow the file system the opportunity to spread the load between multiple servers if possible.

PARALLEL I/O

This is why we are here. Leveraging the parallel I/O capabilities of the supplied file system is intricate and complex. Understanding it is necessary, and adapting your application to work with it can pay off with orders of magnitude performance gains. Similarly, ignoring its capabilities or composition can result in very poor performance or render it unusable. We will begin by describing striping, the functional building block by which the system provides parallelism and continue with a discussion about having your application match its I/O demands to the I/O system layout policies.

Striping

Leveraging the parallel I/O capabilities of the system is the most difficult task to accomplish well. In order to understand the discussion, it is important to know a few key concepts.

The file system is accomplished by aggregating the storage resource and the servers that this resource is attached to. The user file is often split up so that pieces of it lie on different servers and on different pieces of media within each server.

What the file system is doing to accomplish this, is mapping the sequential address space of the file to the physical storage servers and media. The diagram in Figure 1 at the top shows the file address space broken up into collections of bytes; 0 to n - 1, n to 2n - 1, and so on. Below that, two I/O servers are depicted, each storing segments, the individual collections of bytes. The segments are distributed between the two servers in a round-robin fashion. For instance, in this example, the bytes that would appear in the file from 0 to n, and from 2n to 3n -1 are stored on "I/O server 1". The segments have been interleaved between the two servers.

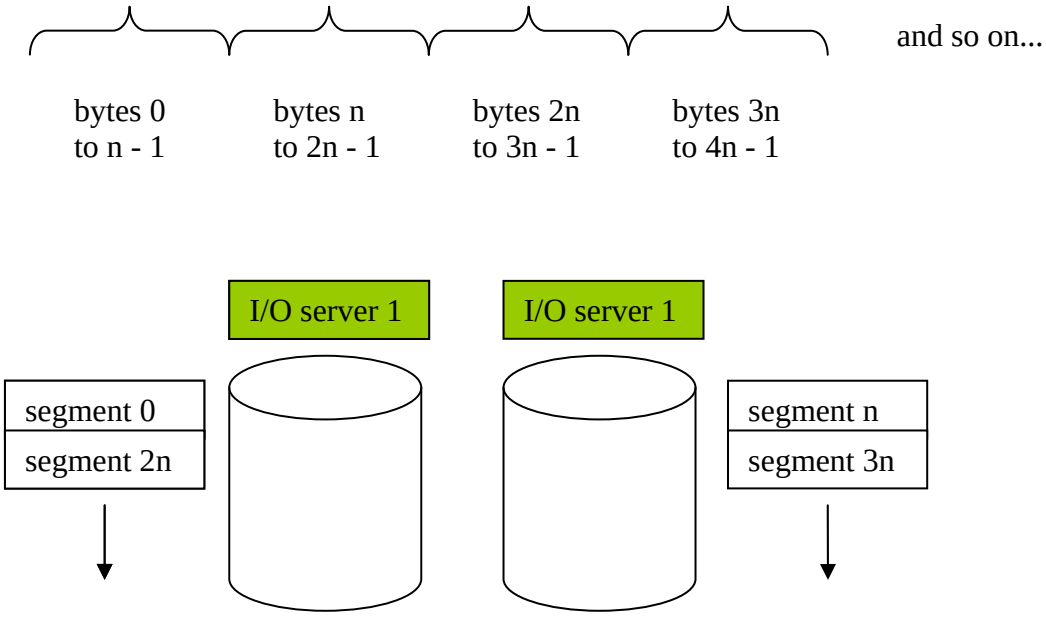

Figure 1

A collection of segments that can be moved in parallel to the I/O servers is called a **stripe**. The number of bytes per segment in the stripe is called the **stripe depth** and the number of segments involved in a stripe is the **stripe width**.

Often, the user encounters the confusing term **stripe size**. It is confusing because the formal definition is the **stripe width** multiplied by the **stripe depth**, or the total number of bytes in a given stripe. Unfortunately, it is often misused so as to be synonymous with **stripe depth** by much product documentation and by many vendors. What's done is done. Just be careful to understand what the particular document you might be looking at means when using this term. In this document, we will be clear when forced to use the term **stripe size**.

The largest implication of this striping scheme is that the highest degree of parallelism that can be achieved is limited by the **stripe width** set for your file. For your machine, there is probably a default setting for this value that is less than or equal to the number of I/O servers on the machine. This value may not be appropriate for you, and, on many file systems, you can change it. For instance, if your machine uses Lustre then the *lfs* command takes a *setstripe* argument allowing you to do this. Other file systems have similar capabilities, but we'll use Lustre as the example here. For others, please consult your system documentation. For (a Lustre) example, though, the command

```
lfs setstripe /scratch1/lee/project 65536 -1 8
```

tells Lustre that files created in the `/scratch1/lee/project` directory should use a 64KB **stripe depth**, that the stripe may begin anywhere in the set of storage servers (the -1 value), and

that the **stripe width** should be 8. Note that Lustre is one of those products that mis-uses **stripe size** as synonymous with **stripe depth**.

Please, always use -1 for the layout policy in Lustre. That value tells Lustre that it can use any of the set of storage servers for the stripe. If any other number is chosen, then Lustre will fix the layout of the file and the file system will fill without balance. This will eventually lead to some storage servers filling prematurely a consequent out of space condition, even though the file system appears to have plenty of space left. Using -1 here allows Lustre to balance the capacity on its servers. This ability to direct the layout policy is considered dangerous by many system administrators, and, as such, the *lfs* command is sometimes restricted. If that is the case, you will either have to have the policy changed, obtain special permission to use the command, or make do with the system defaults.

The command example above was applied to the directory and not to a particular file. All files created in, but not moved to, this directory will inherit the given striping parameters. The parameters may be altered on a directory that already has files, in which case all new files will inherit the parameters. Once a file is created and has data the striping parameters become fixed. It is possible to create a file and reset the stripe parameters to different values, but this capability is different on all systems, and a rare use case, so we won't discuss it further here. Consult your system documentation if this functionality is required.

It might be that the default system settings are "good enough" (perhaps your application is capable of adapting instead of relying on specifying policy?) or that you have changed the striping parameters from the default. The above *lfs* command, or similar for your file system, provides a way to query the parameters on a directory or file. For Lustre, one can determine these parameters for the directory given in the example above by issuing:

```
lfs getstripe /scratch1/lee/project
```

If the *lfs* command has been restricted then your system administrator should have documented the defaults. It is almost certainly well worth your time to find and read that documentation so that you can know what you are working with.

The **stripe depth** you will choose, or that was set for a default, is not arbitrary. While the **stripe width** governs the amount of parallelism that may be attained, the **stripe depth** determines the overhead involved. It must not be too small, since transferring each segment incurs quite a large overhead. The overhead cost is fixed per segment, so it follows that the larger the **stripe depth**, the less the impact of that overhead - within reason of course.

Stripe depths are not silly numbers. They are always powers of two in today's file systems. They are usually multiples of 4KB. Because the media attached to the IO servers is usually via a "RAID controller", which aggregates the attached media for the server, they are typically relatively large. As well, since the file system may be shared by heterogeneous client architectures, it is possible to create a file on one architecture that cannot be read from another. The 64KB stripe size used in the example above is probably a nominal minimum.

Impedance Matching

If you have written I/O routines for your application on a desktop or traditional server, you are probably aware of the "block size" attribute. This attribute lets you know what the basic unit is (in number of bytes) that provides the most efficient transfer of data to and from your file. On a parallel I/O system, the equivalent value is derived from the **stripe size** and **stripe depth** parameters discussed above. Basically, the "block size" you will want to use is an entire stripe. In bytes, that would be the **stripe width** times the **stripe depth**. This may be found in the file attributes, by a special "IOCTL" (IO control) query, or even via some other method. Different file systems supply the necessary information in different ways, or perhaps not at all. What always works, though, is if the user is allowed to specify what it should be. If that is done, the user can discover the efficient transfer size and supply it to the application. Thus, the application is universally portable.

Unlike a desktop or traditional server, the parallel I/O system involves multiple machines, a network, and a large amount of overhead coordinating the dance. If the application will constrain itself to work with the striping parameters, performance can be maximized. Failing to do so will cause the opposite. How bad is only a matter of degree - it *will* happen.

If the application requests a transfer smaller than a full stripe, then less parallelism is employed, or worse, the full parallelism is employed, and redundant data is transferred. This is an example of an "impedance mismatch", where the request parameters don't match the fixed capabilities of the underlying system.

Much of this impedance matching problem can be dealt with simply:

Rule IO1: Bigger is better. The application should try to use the largest possible transfer sizes, well in excess of full stripes if possible - the larger, the better. This suggestion is applied without limit. No matter what other rule the developer might choose to ignore, adhering to this one will always pay off.

Rule IO2: Give the file system as much information as possible. If multiple transfers from separate, non-contiguous, memory regions are to be done, and the memory content is otherwise ready for the transfer, then use the scatter-gather I/O API routines. In POSIX (see the IEEE POSIX Std 1003.1 2004 Edition) these routines are called `readv` and `writev`. Remember, within each memory region, the larger the region the better. See **Rule IO1**, above.

Supplying the file system with this kind of bulk information about a number of transfers allows it to schedule the transfers. Only the file system will know what order and what constraints are best considered given the instantaneous load. Giving it information about multiple transfers allows it to re-order and work with those constraints.

Alignment and Complete Transfers

Partial transfers are always problematic:

Rule IO3: Align the application access within the file address space to a stripe boundary or, at least, a **stripe depth**, or segment boundary. If we use the striping parameters from our Lustre example of 64KB depth and a width of 8, then a stripe is 64KB * 8 = 512KB, and we would want our application to begin all transfers on this 512 KB boundary whenever possible.

Different parallel I/O file systems are sensitive to this in varying degrees. Some can employ the buffer cache on the client to delay writes, allowing enough content to be aggregated locally to make efficient aligned transfers. All the operating system is doing is applying **Rule IO1** and **Rule IO2**, above, itself. Some can't do this, or don't do it very well. It's hard to tell. In any case, it is always safe for the application to try to align to the stripe boundary. Really, it can never hurt.

If the stripe boundaries just cannot be accommodated then, at least, align to the **stripe depth** boundaries. In our Lustre example above, these would occur at every multiple of 64KB. If this alignment cannot be satisfied, the application has moved into a whole new world of hurt. The file system will usually respond by imposing even more overhead. If the application is reading data, the file system may be forced to read in the entire segment anyway. It may, in fact, have to transfer that entire segment over the network to the client node where the application is running. If the application is sequentially sweeping the file address space, and the operating system employs a read cache, this may not be too much of an issue. If it is making non-sequential access then, at least, redundant data has been transferred and, so, bandwidth is correspondingly wasted. If doing what appears to the file system to be random accesses, then the unused data might be transferred many times, filling and refilling the buffer caches with useless data. If the application was requesting a write, then things are even worse. To accomplish this transfer, the I/O system must read the segment, alter the appropriate bytes, and write the entire segment back out. The unused bytes, in this case, are transferred twice. In other words, it must incur at least twice the overhead of a read.

Similarly, the amount of data transferred should be complete. It should be the whole stripe, or at least end on a **stripe depth**, or segment boundary.

Rule IO4: Transfers should be in multiples of a stripe or, at least, multiples of the **stripe depth**.

Violating this rule results in the same issues as violating the alignment rule in **Rule IO3**. The only difference is that the issues occur at the tail of the transfer instead of the head.

If your application can honor these rules, and you can set the stripe width, doing so can very much enable performance and system health. You can, for instance, reset the stripe width to be a multiple of what you actually need. You would be using **Rule IO4** instead of **Rule IO3**, but to help instead of work around something. This approach can leverage the various caches in the I/O servers so that the application more effectively perceives memory to memory transfer rates instead of the serious limitations involved by a disk drive. What you would be doing is increasing the size of the involved caches with every I/O server you can add. Think of it this way: the amount of parallelism desired is fully achieved, but the application is allowed to leverage a second, or more, subset of the I/O nodes to buffer or pre-fetch data.

MIDDLEWARE ASSISTS

All of the discussion to this point has assumed that you are developing an application that works directly with the file system. Adhering to those rules religiously will achieve the absolute maximum bandwidth the file system can supply.

Unfortunately, adhering to those rules is typically more than problematic for most applications. To do so often requires re-forming data access in ways that are unnatural to the application or crafting intermediate layers that copy and re-form underneath.

Not infrequently, there are system-supplied libraries that can help with this task or completely assume it. They are designed to work with the naturally application-abstracted view of the data while optimizing the placement of that data within the file, or files, to enhance performance.

One example of this is the MPI (Message Passing Interface) libraries found on all of these machines. The MPI de facto standard provides a concept of file "views". These "views" allow the application developer to group and reorder record elements so that they may be managed within the file in an orderly, optimal way. Underlying file system specific drivers within the library use the information from these "views" to do the dirty work. These have detailed intimate knowledge of the machine the application is running on and the particular file system in use, though you might have to specify which driver you want. They will do the dirty work of religiously applying all of the above applicable rules for your particular machine.

Moreover, if your application can perform it's I/O in a collective fashion, the MPI I/O routines can go much further in helping you. They can leverage extremely complex algorithms that will use your nodes to reorganize transfers and aggregate large, contiguous chunks of memory for the transfers. Some can even create distributed peer caches using very little memory per node to avoid going to the file system at inappropriate times. These are only a couple of examples of how they might help. There are a plethora of technologies in these MPI libraries that enhance performance in small and large ways.

Most importantly, these assistive technologies are transparent. If the developer will leverage the "views" capabilities provided by the interface, these MPI libraries will go to immense lengths to make sure the actual I/O against the file system occurs in the best, highest-performing fashion. Then, they will do so in a portable manner. You won't have to remake your architecture or approach for another machine. You might have to tweak hints and the like, but a disastrous rewrite will never be required.

Beware confusing data abstraction libraries with an assistive library like MPI. For instance, it is somewhat popular to use HDF (Hierarchical Data Format) and CDF (Common Data From) libraries. These libraries allow very sophisticated and powerful abstractions for problem and domain-specific solutions as applied to persistence storage of the data. They are data *abstraction* libraries. To do what they do, they effectively implement file systems within the host supplied file system. The overhead of doing this is relatively large. Certainly, they go to some length to do what they do in an efficient and orderly way, but their purpose is not performance. It is

abstraction. Not to say they are not useful or do not have their place, but if you are worried about performance, they can be very problematic.

CONTENTION

Your application will almost certainly not run in a vacuum; there are other unrelated applications using the machine. This can take many forms: a user on a head node moving data on or off the machine, working within directories, and other parallel jobs leveraging the compute cycles.

Other user's jobs are by far your greatest concern. Recall, the machine file system is a shared resource. If your application is performing I/O concurrent with another application, then you will contend for that shared resource. The file system will engage in a complex dance trying to make sure that all users get some access, and this extra gyration involves extra overhead. In short, your I/O performance will fall even more than is naively predictable.

Typically applications don't spend much of their time doing I/O. What usually happens is that an application will generate or load a large amount of memory and then work with it for awhile, then do their I/O and begin again. In other words, applications tend to perform compute and I/O in phases. Given the nature and purpose of these machines, typically, the compute phase tends to dominate.

This means that the I/O phase for most applications occurs within a relatively short window. You can leverage that fact. No, it does not mean that you can schedule it. It is impossible for one application to know when another will begin or end its I/O phase. What can be done though is to minimize your application's I/O phase and window. It should follow that the shorter your application's window, the less chance another application's window will overlap with it.

You do this simply by using all the knowledge we've given you here. Make your I/O as efficient as you can. Reduce what you need to move between the application and the file system, and, for what you do have to move, use all of the bandwidth you can get, as efficiently as you can.

In short, get in and then get out as fast as you can. Don't give the other applications a chance to impact yours.

CONCLUSION

Whether you are a user or developer of these applications, please do at least be familiar with how these parallel file systems function. As a user of a particular machine, be familiar with the specifics of the machine enough so as to be able to judge whether the system striping parameters are correct for your application or what you might need to do to change them. As a developer, being familiar with this ubiquitous striping concept and how well your application adheres to the implied constraints can mean orders of magnitude difference with respect to performance.

It is well worth your time to understand and apply this knowledge to your application. The benefits can be enormous.

DISTRIBUTION

1	MS0380	Gregory D. Sjaardema	01543
1	MS0807	John P. Noe	09328
1	MS0807	Robert A. Ballance	09328
1	MS0807	Judith E. Sturtevant	09328
1	MS1316	Steven J. Plimpton	01416
1	MS1319	Ronald B. Brightwell	01423
1	MS1319	Lee Ward	01423
1	MS0899	Technical Library	9536 (electronic copy)


Sandia National Laboratories