

# A Business Case for Extensions to the POSIX I/O API for High End, Clustered, and Highly Concurrent Computing

Gary Grider, Los Alamos National Laboratory  
Lee Ward, Sandia National Laboratories  
Rob Ross, Argonne National Laboratories  
Garth Gibson, Panasas Inc. and Carnegie Mellon University

05/2006

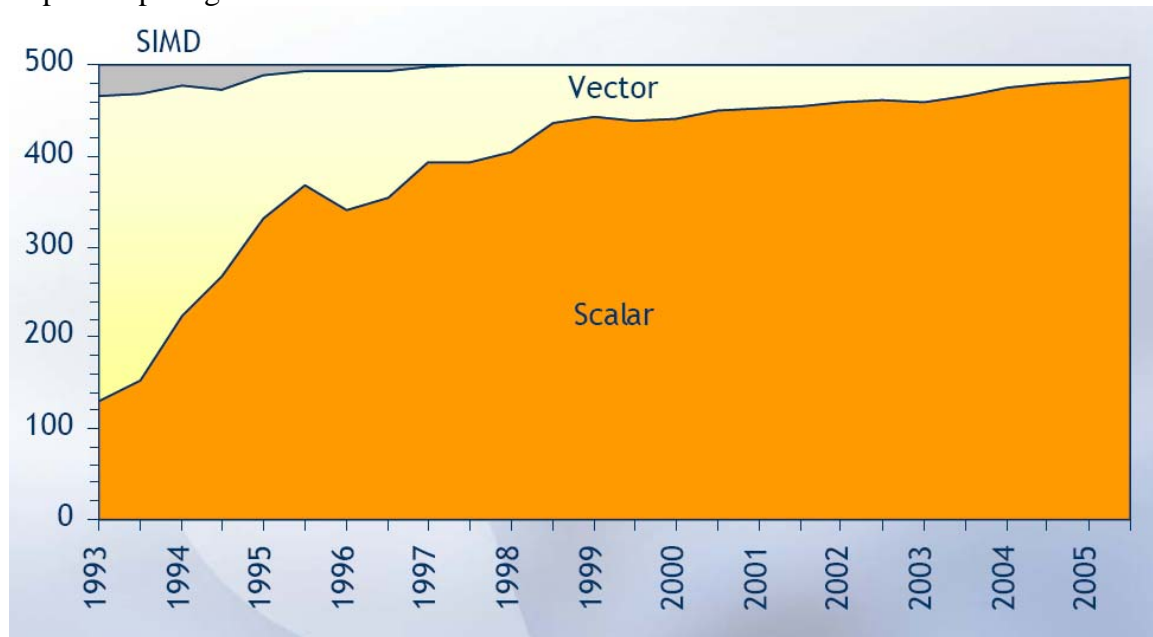
## Introduction

One of the fastest growing markets today is that of cluster computing. IDC has had to revise its growth predictions up in both 2003 and 2004. Cluster computing presents unique I/O and file system issues. Cluster computing is one example of distributed memory computing. Distributed memory computing is becoming very common in many disciplines and in many industry verticals including data mining, engineering, scientific, and entertainment computing. Distributed memory computing is actually an old concept for the high end computing, or supercomputing, industry. This industry has used distributed memory computing for at least two decades at all scales including small clusters of a few desktop class computers to massively parallel computers with tens of thousands of processors, each with their own memory space. The demands of the rapidly growing industry using distributed memory architectures have produced globally accessible parallel file systems, file systems that allow for the many related distributed memory process to have global access to the file space and even concurrently to a single file. Additionally this growing industry has begun to have an increasing effect on file system standards and to that end, the IETF working group on Network File Systems (NFS) has recently incorporated into its agenda support for highly concurrent distributed memory access to global parallel file systems and parallel NFS server resources. Called the Parallel NFS (pNFS) extensions ([www.ietf.org/internet-drafts/draft-ietf-nfsv4-pnfs-00.txt](http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-pnfs-00.txt)), these protocol extensions allow for direct, parallel access to a variety of storage servers. Distributed memory applications in this growing market segment typically accomplish I/O using the POSIX standard API calls or leveraging support libraries that do. The POSIX I/O API was not designed for this distributed memory application paradigm. Unfortunately, POSIX mandates a "stream of bytes" view of the file address space and access to globally coherent values for file metadata. The distributed nature of the machine makes the overhead involved in maintaining such views of files and metadata untenable for the high performance application for which these machines were designed. Further, POSIX forces repetition of certain operations on all processes, such as name resolution, when one process might instead perform these operations on behalf of all processes in the distributed application. These disparities between the facilities and I/O model of POSIX and the needs of distributed memory applications encourage the development of extensions for POSIX I/O in this environment.

## Best Practices for I/O in High End and Distributed Memory Computing Applications

The computing world has clearly turned to a distributed model of computation in order to solve larger and harder problems. The cluster or multi-program, parallel (MPP) machine architecture supporting this model of computation range from the easily assembled mini-cluster in the basement to the world's most powerful machines.

More, this architecture has become dominant for high end computing as can be seen by the following graphic, extracted from a [Highlights](#) talk given by Horst Simon at Supercomputing '05.



Such machines require special programming. Typically, an application that wishes to harness the power of such a machine is aware that it's many parts are distributed among multiple computational units, or nodes, and that data values must be explicitly moved between the piece-parts. Naturally, a cooperative programming model has typically been used. An application, at start, will partition the problem into discrete units. Each unit is farmed off to a computational unit. Those computational units proceed to work independently on their part of the problem. At the end, the solutions for each of the parts is assembled and integrated.

At many times during the above described process the application will write intermediate results, in order to restart after failure, or for other reasons. It will also write it's final result in the end. Often, especially with coupled problems, the final result data is the same format as the intermediate result.

Applications may use one, or more, or a hybrid of three methods to accomplish this I/O:

- A single file per process/node; Each node or process will create a unique file to hold the result of the local computation thus far. This is a problem in that a restart, after failure or interruption, must use the same number of nodes. It is also

problematic with respect to the final results as many files must be integrated or mined by special tools and post-processors.

- A smaller number of files than processors or nodes used in the computation; The application will do some integration work at each checkpoint, and for the final result, in order to allow restart on a different number of nodes or processors. While this handles the restart problem nicely, it still, frequently, requires special post-processing and tools.
- A single file; The application integrates all of its piece-wise information into a single file for restart purposes and to deliver a nice package for the result.

Finally, like many other applications, these high end applications desire portability. New, more powerful, machines are introduced every year. Users move, and they want to take their applications with them to new facilities and machines. This is enabled, then, by writing portable programs adhering to common standards, typically POSIX.

## The Standard POSIX I/O Model and Implementation

The standard POSIX I/O model derives directly from that found in the Unix operating system. The inventors of UNIX chose to simplify the number of supported access methods for file I/O to only one: A stream of bytes. In this model the view includes a flat, linear, address space for the file and a pointer giving the current location in that address space. All I/O operations begin at the place indicated by the file pointer and continue for a specified length. There is no concept of a record or any unit other than the byte. To facilitate random access to files a repositioning, or `seek`, operation is supplied.

The model incorporates simple, consistent, and powerful semantics with respect to the file address space. For instance, the case where two or more processes attempting to write at the same location in the file address space is resolved by allowing the last writer's data to be the content. As well, altered or new content is immediately visible to all processes immediately after a write operation.

POSIX also mandates that information about a file is always up to date. Each read or write operation alters a time stamp, with single second resolution. Similarly, each write operation adjusts counters reflecting the size of the file in bytes and how much file system space is allocated to the file.

Implementation of these semantics has been relatively easy. For instance, the resolution of two or more processes writing to the same file at the same location is trivially solved by using barriers, or mutually exclusive regions, allowing only one of many process threads to pass. In this way, potentially unordered access becomes ordered. It is the last process thread to pass through the barrier that becomes the one to leave its data in the file at the conflicting region.

Taken together, this model and its semantics have provided a usable application programmer's interface for accomplishing file I/O in operating systems for the last 40 years. Implementations have remained relatively simple and, so, easy to construct. At the heart of the matter, the one thing that has enabled such powerful, easy, and efficient implementations of this model and its semantics is the availability of fast primitives for synchronization. Many machines provide a test-and-set, test-and-increment, or other barrier instruction. Those that did not have access to such instructions

rely on tricks such as a non-interruptible routine in the executive to provide ordering. In all cases, though, these synchronization primitives are very fast; Often only on the order of only a few hundreds of processor cycles.

## **Why POSIX I/O is a Poor Fit in High End and Distributed Memory Computing**

The fundamental, implicit, assumption in the POSIX I/O programming model is that an implementation enjoys fast synchronization primitives in order to resolve competing access to shared data. As mentioned previously, this is typically accomplished by leveraging special memory-barrier operations, such as test-and-set, or leveraging mutually-exclusive code regions enabled by the operating system.

In the MPP-only, cluster, or distributed memory machine, though, no such analogous function is at all efficient on a machine-wide basis. The machine is composed of many discrete nodes, each possessing it's own clock, memory sub-system, network interface(s), etc. Any such barrier operation may only be accomplished by mutual agreement or by implicitly ordering requests through a monitor mechanism.

Any form of global synchronization in the MPP then, inevitably, leads to the use of the machine network. The relative levels of performance difference between memory and a single network transaction is huge. In memory, test and set of a single value can be accomplished in only a few tens of nanoseconds. In the MPP, a similar operation must use the network and, even in the fastest networks available now, takes on the order of tens of microseconds. This is a three orders of magnitude difference! That considers only one node. Additional overhead is imposed if many nodes simultaneously wish access as the network is forced to deal with congestion. Thus, while implementations may, and do, exist to provide synchronization to globally shared data they are glacially slow when compared to the speed of the processor(s) in more classical non-distributed memory architectures.

All this is not to say that high performance I/O is impossible with POSIX in the MPP architecture. It is highly restricted, though. Many applications accomplish high performance IO by using the single file per process or node solution, discussed previously. While this does yield good performance it leads to the already discussed problems; namely, they must restart on a similar number of processors and there is an intense post-processing task employing special tools for that application.

In short, applications leverage the distributed nature of the file systems in these high end machines. What they really desire, though, is to use a parallel approach without artificial constraint. They would often like to deposit all of their data into only one file or, perhaps, a small set of files.

Such applications by nature and design are already intensely cooperative. They are perfectly capable of arranging their file access so as to avoid conflicts in the file address space. They do not often require access to globally maintained time stamps. Nor do they often need to know the size of the file while it is being generated.

Unfortunately, programmers have no way to inform the file system, using POSIX, that they are cooperating, or that they won't need the highly volatile timestamps and file size. They are, simply, stuck using an API and semantics designed for a machine that enjoys uniform access to all memory on the machine if they wish to write portable applications.

These methods of access they use, serial per processor or parallel for all, do provide a usable interface; however, deposition of their data in a reasonable time frame is virtually impossible.

## **Proposed work and schedule**

There is a group of people representing high end computing users, universities, and manufacturers of high end computing solutions that are already working on a proposed set of extensions to the POSIX I/O API to address the needs in this growing sector. The people in this group are all involved in high end computing I/O and file systems research and development. We intend to harness the momentum of the High End Computing Revitalization Task Force (HECRTF) as a mechanism to enable the work required to flesh out and prototype the proposed extension set. The HECRTF is a government interagency working group (HECIWG) which promotes high end computing needs and coordinates government funding of research and development activities in the high end computing area. The HECIWG has a committee that manages I/O and file systems activities. The HECIWG has all the contacts within the high end computing community to very effectively draw in the required resources, and even fund work, if necessary, to assist with this POSIX I/O API extensions effort. This group will facilitate the process of bringing together the users, university researchers, and the high end computing solution providers to specify, document, and prototype the proposed POSIX I/O API extensions. Once these specifications and background documents have been drafted, the output will be posted in the appropriate forum within the Open Group to be vetted in the normal Open Group methodology. We expect to be able to present a draft of the proposed API extensions in summer of 2006, with prototype extensions and results of testing prototype extensions by late fall or early winter of 2006.

## **Conclusion**

The POSIX I/O model and its semantics are simple yet powerful. However, they were designed with the assumption that the underlying hardware provided an implementation with very fast synchronization primitives. The multi-program parallel architecture now dominant in high-end computing invalidates that basic assumption. This leads to applications in the distributed memory environment either restricting their use of the file system to a very specific, not always natural case, using non-standard extensions, or relying on implementation details that are not necessarily implementable in existing and contemplated file system solutions. So, it is not that POSIX I/O calls are incapable of providing access to a high performance file system solution. Instead, the mandated constraints and games the programmer must resort to in order to get around these constraints are onerous and lead to solutions that make what was originally attractive now moot. The POSIX API and semantics very effectively deny the ability for processes in a distributed memory machine to efficiently share a single file's address space.

We propose a set of work to specify, document, prototype, and test a set of POSIX I/O API extensions to address the needs of the growing high end computing sector.