5-2013

# PARALLEX FILE SYSTEM (PXFS): BRIDGING THE GAP BETWEEN EXASCALE PROCESSING CAPABILITIES AND I/O PERFORMANCE

Shane Snyder
*Clemson University*, sdsnyde@clemson.edu

# PARALLEX FILE SYSTEM (PXFS): BRIDGING THE GAP BETWEEN EXASCALE PROCESSING CAPABILITIES AND I/O PERFORMANCE

---

A Thesis
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Shane Snyder
May 2013

---

Accepted by:
Dr. Walter B. Ligon III, Committee Chair
Dr. Adam Hoover
Dr. Jim Martin

# Abstract

Due to processors reaching the maximum performance allowable by current technology, architectural trends for computer systems continue to increase the number of cores per processing chip to maximize system performance. Most estimates suggest massively parallel systems will be available within the decade, containing millions of cores and capable of exaFlops of performance. New models of execution are necessary to maximize processor utilization and minimize power costs for these exascale systems. ParalleX is one such execution model, which attempts to address inefficiencies of current execution models by exposing fine-grained parallelism, increasing system utilization using asynchronous workflow, and resolving resource contention through the use of adaptive and dynamic resource scheduling.

A particularly important aspect of these exascale execution models is the design of the I/O subsystem, which has seen limited performance increases compared to processor and network technologies. Parallel file systems have been designed to help alleviate the poor performance of storage technologies by distributing file data across multiple nodes of a parallel system to maximize the aggregate throughput attainable by file system clients. However, the design of parallel file systems needs to be modified to

explicitly address the inherent high-latency of remote file system operations without degrading file system performance and scalability.

We present modifications to OrangeFS, a high-performance, working model parallel file system geared towards the facilitation of research in the field of parallel I/O, to help address the inefficiencies of current file systems. We deem our resultant parallel file system implementation ParalleX File System (PXFS), as it attempts to support the features required by the I/O subsystem of the ParalleX execution model. Specifically, PXFS offers mechanisms for masking the latency of file system operations, defining meaningful computation to be overlapped with file system communication, and maintaining the high-performance and scalability exhibited by OrangeFS. Our results indicate PXFS successfully improves file system performance and supports the semantics of ParalleX with limited programmer intervention, potentially simplifying the design and increasing the performance of many ParalleX applications.

# Dedication

First, this work is dedicated to my family for their constant patience with me and their pushing me to succeed over the years. Also, this work is dedicated to all of my friends for their encouragement and the balance they have given me throughout my academic career.

# Acknowledgements

First and foremost I would like to thank my advisor, Dr. Walt Ligon for his consistent support and guidance in helping me complete this work. None of this would be possible without his assistance and wealth of knowledge in this field. Also, I would like to thank Dr. Adam Hoover and Dr. Jim Martin for taking the time to serve on my committee and providing valuable feedback regarding my research.

# Table of Contents

Table of Contents (Continued)

Page

# List of Tables

# List of Figures

List of Figures (Continued)

Figure Page

# Chapter 1

# Introduction

While the transistor density of integrated circuits continues to increase with Moore's Law, the clock speed of modern processors has peaked, due mostly to the energy efficiency of the underlying technology. Consequently, current processor design trends opt to include multiple compute elements on a single chip, rather than continue to push the performance of a single processor. For instance, according to the Top500 list, the four fastest computers in the world all have over 0.5M cores and offer nearly 20 petaFlops of performance [32]. According to most approximations, computer chips will offer hundreds or even thousands of compute elements per node by 2018, resulting in massively parallel exascale systems [2]. Of course, programmers will have to find a way to exploit this billion-way parallelism if they hope to achieve exaFlops of performance.

Unfortunately, current programming and execution models do not take advantage of the massive amount of parallelism available in systems with many cores. The cause of this lack of efficiency ranges from the lack of rich parallel programming constructs

available in most programming languages to the inability of current runtime systems to distribute work to avoid load imbalances. It is clear that for computer performance to continue to improve in these many-core environments, changes must be made to current execution models to allow compute resources to be leveraged efficiently and adaptively with limited burden on the programmer. Otherwise, the performance of systems with increasing numbers of compute elements will continue to scale poorly.

## 1.1 High-performance Computing

High-performance computing (HPC) is the application of supercomputers to solve large science and engineering problems, ranging from the simulation of complex physical systems to solving linear systems of equations. On standard computers, the computational requirements and large data sets inherent to these problems make it infeasible to obtain results in a reasonable amount of time or within a desired precision. Simulations are particularly useful applications, since some physical systems are difficult to observe experimentally or model analytically. For example, [11] present a biomolecular network simulator used to model complex biological processes, which are difficult to analyze due to their stochastic nature. Also, [27] claims their turbulence simulation engine executed on a 10,000 core supercomputer achieves similar results as an actual experiment conducted in a wind tunnel. This is convenient, as a scientist can obtain “real” results via simulation rather than going through the trouble of setting up a physical experiment.

Currently, most HPC systems are realized using *clusters*. A cluster is a set of interconnected nodes, in which each node consists of (often) the same commodity hardware components. Clusters have continued to gain popularity, due mostly to their ease of implementation and improved scalability over single node systems. Even the world's fastest supercomputers follow the architecture of computer clusters, although they often utilize specialized hardware and proprietary high-speed interconnection networks. As of this writing, typical hardware specifications for an HPC node are given as [9]:

- Processing elements – high-end multi-core processors (generally at least 8 cores) placed on multi-socket motherboards (2 sockets are typical). Some systems may also include GPUs or other acceleration hardware.

- Memory – tens of gigabytes of DDR3 DRAM.

- Interconnection network – a high-speed, dedicated interconnection network. Lower-end systems may use Ethernet, while higher-end systems may opt for InfiniBand or Myrinet technologies.

- Storage – a subset of nodes (*I/O nodes*) in the HPC system will serve data to the *compute nodes*, usually using a high-performance parallel file system (e.g. PVFS, Lustre, or GPFS) and the interconnection network.

As long as there have been HPC systems, there has been the need for software libraries and other development tools to allow programmers to obtain the highest performance from the underlying hardware. The Message Passing Interface (MPI)

Standard [19] is one such example, providing parallel programmers a standard API for passing messages between processes on a distributed-memory system, e.g. a cluster. Provided functionality includes both point-to-point communication between processes, collective communication between groups of processes, and synchronization primitives, such as global barriers. Another parallel programming tool available is OpenMP [20], which is used to parallelize code for shared-memory systems. Unlike MPI, OpenMP is a set of compiler directives used to extend the Fortran and C/C++ languages. In particular, OpenMP provides constructs for thread creation and work-sharing, which are most frequently used to parallelize large loops. In the case of clusters, it is not uncommon to use a hybrid programming model, in which MPI may be used for inter-node communication and OpenMP may be used for intra-node (inter-core) communication.

As the number of processing elements in HPC clusters continues to increase, it is necessary to reconsider the design of longstanding models of execution. Newly proposed models include innovative design principles for achieving higher utilization and lower power consumption than conventional models, making them appealing for more complex parallel systems. Since the datasets of many HPC applications are increasing rapidly, too, these execution models also require a high-performance parallel file system that supports highly concurrent access. Otherwise, data starvation may severely limit the performance of I/O intensive applications. This suggests that explicit cooperation between parallel execution models and parallel file systems is instrumental in achieving continued performance increases in HPC applications.

## 1.2 Parallel Execution Models

It would be extremely challenging to develop programs that execute correctly without a well-defined interface to the underlying hardware resources of the system. More specifically, the high-level software components of a system, such as the programming models, compilers, runtime environments, software libraries etc., must be aware of how to make efficient use of the system architecture. The execution model describes the interface between the software and hardware architecture of the system [14], as shown in Figure 1.1. Also, the execution model defines the governing principles of all computation, which include system semantics, referentiable structures, and policies of resource management [3].

For example, the execution model for the basic von Neumann architecture

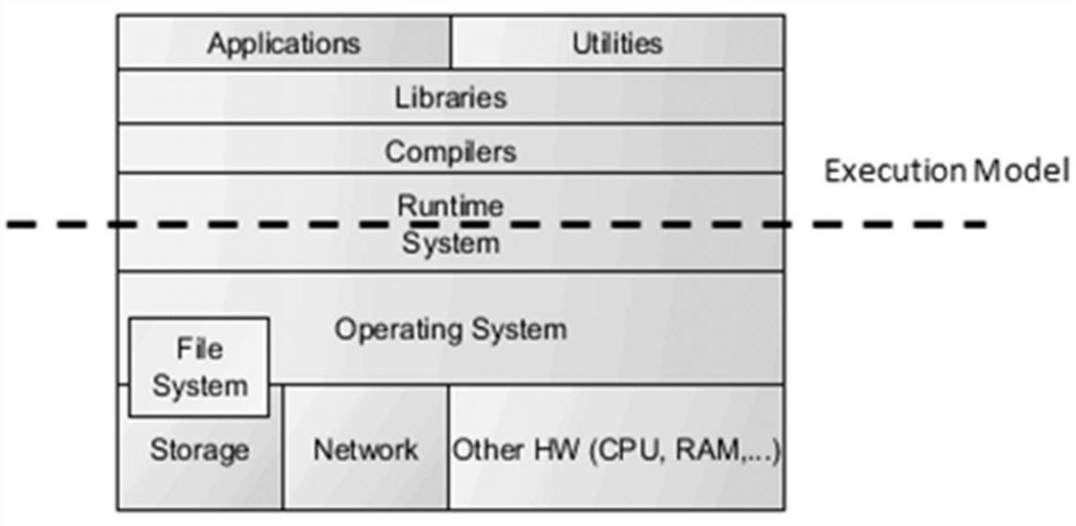


*Figure 1.1 – Basic execution model for an abstract computer system.*

involves the translation of application source code into a sequence of machine executable instructions via compilers or interpreters. Once the runtime system loads the executable program in memory, which includes instructions for managing the program stack and storing local variables, the processor can fetch and execute instructions sequentially. Concurrency may only be achieved by interleaving instructions from multiple tasks and through the use of instruction pipelines. Though the complexity of this model precludes it from use in highly concurrent, distributed systems, it has proven to be extremely robust and will remain relevant as long as von Neumann processors are the standard computing building blocks [13].

High-performance parallel systems obviously warrant a more sophisticated execution model, in which distributed concurrent processes may communicate and coordinate execution with each other. The most prevalent parallel execution model is that of communicating sequential processes (CSP), where a group of distributed processes communicate using a message passing library, such as MPI. A parallel application following this model creates a fixed number of processes at startup, where each distributed process has a globally unique name and a private address space. Generally, a single process per node is used, although multiple threads per process may be utilized to make more efficient use of multicore architectures. These processes operate on data contained locally in their own private address space and cooperate with other processes by sending and receiving messages. The sending and receiving of messages is often used as an implicit form of synchronization, but many CSP implementations include explicit synchronization mechanisms, like global barriers or mutexes.

With exascale HPC systems on the horizon, much research is needed in developing parallel execution models that more effectively abstract the massive pool of resources contained in each system. While the CSP execution model is well-understood and performs well on many classes of parallel applications, it offers limited functionality for efficiently representing the fine-grained parallelism and dynamic workloads typical of other classes of scaling-impaired applications [3]. Future execution models must provide more programmer and system support for the lightweight creation of tasks, synchronization between these tasks, and dynamic distribution and scheduling of these tasks across a system. This requires extensive co-design between all layers of the execution model: system architects will have to modify or redesign low-level system components of the execution model, such as operating systems, file systems, and network interfaces, while application programmers will have to rewrite or completely redesign portions of longstanding application codes. Clearly, the transition from a well-known, stable execution model to a revolutionary, untested one will be met with resistance, but a paradigm shift is necessary to meet the computing and power requirements of the exascale systems of the future [2].

## 1.3 Parallel File Systems

At the most elementary level, a file system is simply an abstraction of the storage hardware on a computer system. In other words, a file system provides a common interface to a range of storage devices (hard disk drives, SSDs, RAIDs, tape drives, etc.),

such that a user can create, update, and delete files without a detailed knowledge of the low-level mechanisms involved. It is important to note that, depending on the file system capabilities, files may be stored on either local or remote devices. Files include program executables, images, configuration files, and other data. On Unix systems, everything is stored as a file, meaning directories, sockets, devices, and links are all stored and accessed in the same manner as regular files, as far as the file system is concerned. File systems are tasked with not only storing data, but also metadata – that is, data about data. Metadata includes permissions, timestamps, physical layout of the file on storage devices, and other attributes associated with a particular file.

The advent of computer networks brought need for file systems that allowed sharing of files across networks, typically referred to as network file systems. One such file system is the Network File System (NFS) [28], which allows a single server to serve file data to multiple clients over a network. The benefits of NFS include: relaxed storage requirements on client nodes, ability to provide a consistent home directory for all clients network-wide, and the ability to share other storage devices, such as CDROM drives, over a network. However, a single file server causes a single point of failure in the system and is an obvious performance bottleneck for applications that are I/O intensive, as shown in Figure 1.2.

Since most HPC systems follow a distributed node architecture, a high-performance distributed file system is necessary for I/O intensive parallel applications. Parallel file systems were proposed to alleviate the performance issues inherent in single

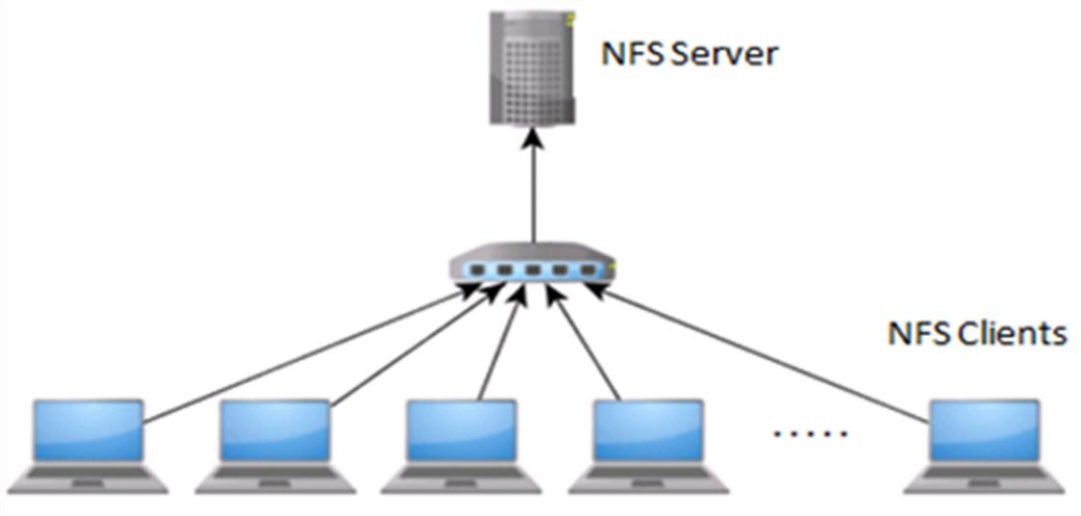


*Figure 1.2 – The single server architecture of NFS.*

server network file systems. Typically, this has been achieved by striping file data across multiple data servers in a manner similar to RAID 0, while maintaining a consistent network-wide namespace [7]. As shown in Figure 1.3, striping file data across multiple servers allow clients to utilize the aggregate network bandwidth of a set of file servers, rather than overloading a single server.

Of course, the file system must provide a standard interface to programmers to facilitate portable coding. Arguably, the most prevalent file I/O API is provided as a subset of the Portable Operating System Interface (POSIX) standard [23], which specifies a standard operating system interface and environment for providing software compatibility between conforming systems – Unix-like systems, historically. The POSIX file I/O standards include definitions for many basic file operations, such as read, write, open, close, and seek, among others. While these operations benefit from their simplicity and high portability, they lack support for the collective communication and

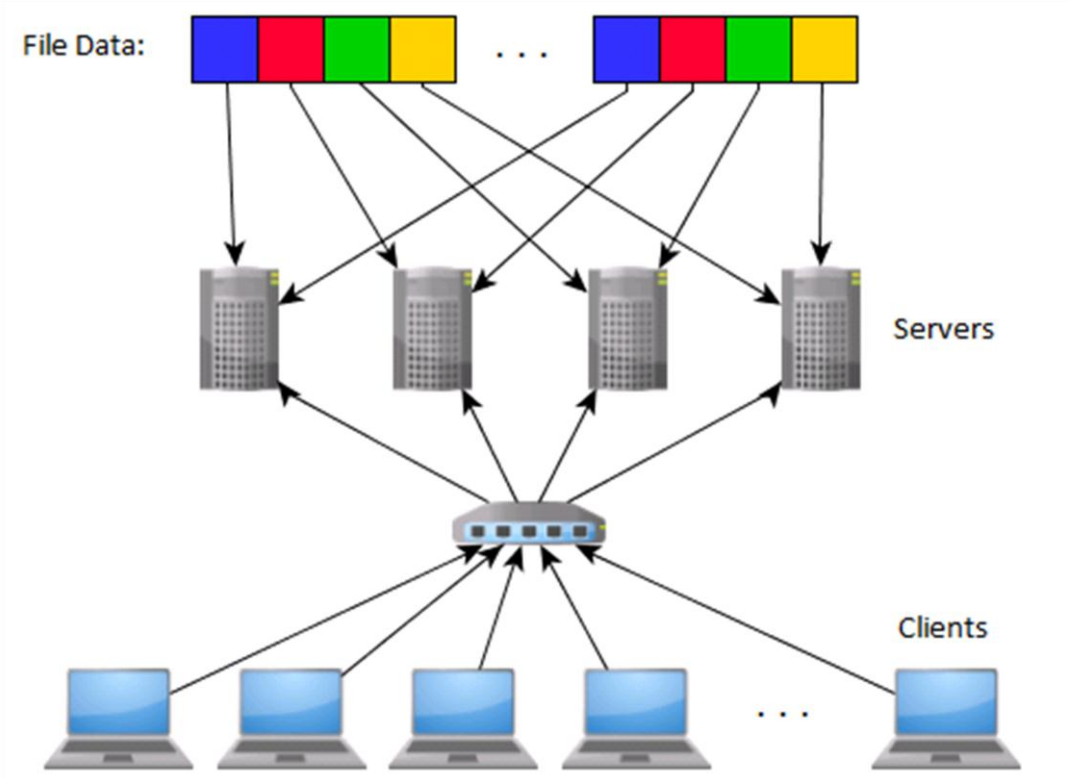


*Figure 1.3 – The multi-server architecture of parallel file systems allows for increased I/O performance.*

noncontiguous disk access typical of many parallel applications [30]. On the other hand, the MPI standard [19] includes definitions for collective I/O, noncontiguous accesses using derived data types, and other advanced I/O routines, which allow for more expressive parallel programming. File system developers may also decide to implement their own native APIs, which are generally optimized for high performance and allow for a range of specialized capabilities, including asynchronous operations and configurable data distribution parameters.

## 1.4 Goals

With massively parallel systems capable of exaFlops of performance expected within the decade, the time for researching, designing, and analyzing execution models to most efficiently utilize their resources is now. Current parallel execution models do not offer the fine-grained parallelism necessary for the high-performance of some classes of scaling-impaired applications. In exascale systems with orders of magnitude more processing elements, these execution inefficiencies will only worsen. A crucial design consideration for these exascale execution models is the I/O subsystem, which, historically, has seen limited performance improvements compared to processor and network technologies. Future parallel execution models must address the I/O bottleneck to prevent processors from wasting compute cycles waiting for the completion of high-latency I/O operations. Parallel file systems are an obvious choice for the basis of exascale I/O research, due to their high-performance and distributed architecture.

**We propose that the design characteristics of a high-performance parallel file system may be customized for integration with ParalleX, a new parallel execution model targeted for massively parallel HPC systems. Specifically, the semantics, capabilities, interfaces, and runtime environment of the file system may be modified for supporting the I/O requirements of the ParalleX execution model.** To demonstrate this supposition, we develop a proof-of-concept parallel file system dubbed PXFS (ParalleX File System) to integrate with the ParalleX model. In order to satisfy the ParalleX I/O model, PXFS should exhibit the following characteristics:

- High-performance, high-concurrency I/O to support exascale workloads
- Low overhead of I/O operations to maintain system scalability
- I/O operation semantics that accommodate the semantics of programming languages, the runtime system, and other higher level execution model components
- Modular architecture to facilitate incremental development and continued research in exascale I/O

## 1.5 Methodology

To assist in the development of our initial prototype of PXFS, we adhere to the following design methodology:

- *Select a suitable parallel file system to serve as a basis for our I/O model.* We choose OrangeFS for our basis, since it is a working model parallel file system and is highly modifiable. This allows us to redesign file system components, while, ideally, maintaining high-performance. We discuss OrangeFS in further detail in Section 2.2.
- *Select a suitable parallel execution model for our prototype.* As mentioned in the previous section, we choose ParalleX as our target parallel execution model. We discuss the design of ParalleX in detail in Section 2.1.

- *Characterize the I/O requirements of the ParalleX execution model.* We analyze the I/O model used by ParalleX to gain an understanding of what semantics a compatible I/O subsystem would provide.

- *Modify the architecture of OrangeFS to satisfy the I/O model of ParalleX.* Armed with a detailed specification of the I/O model of ParalleX, we make the necessary modifications to OrangeFS to satisfy these specifications, yielding our PXFS prototype.

- *Test the function and performance of our modified I/O subsystem.* We test the PXFS prototype in a production environment to ensure its functionality and high-performance.

- *Analyze the feasibility of PXFS.* To determine the feasibility of our modifications, we analyze the performance results obtained to evaluate if PXFS effectively satisfies the I/O requirements of the ParalleX model.

## 1.6 Thesis Organization

In Chapter 2 we present more detailed information concerning the design of the ParalleX execution model and OrangeFS file system. We also summarize related research in the area of exascale execution environments and file systems. In Chapter 3, we characterize the I/O requirements of the ParalleX execution model and outline modifications to be made to OrangeFS to satisfy these requirements. In Chapter 4, we

analyze the performance results from our testing to gauge the viability of our implementation. Finally, in Chapter 5, we offer conclusions we have drawn from this research project and, also, we propose future avenues of research regarding parallel file systems and exascale execution models.

# Chapter 2

# Background and Related Work

## 2.1 ParalleX Execution Model

ParalleX is a working hypothesis parallel execution model motivated by two primary goals in the field of HPC: the long-term objective of achieving exaFlop performance on the million core parallel systems expected within the next decade and the current objective of improving the performance of a diverse group of scaling-impaired parallel applications [29]. The predominant parallel execution model of present HPC systems, CSP (i.e., MPI), does not provide suitable mechanisms for representing fine-grained parallelism, intrinsically hiding latency, or balancing workloads across a system. These shortcomings suggest the necessity of a paradigm shift in the area of parallel execution, as the system semantics of current models offer little support for managing dynamic and irregular parallelism – a problem only intensified by the steady increase in the core counts of HPC systems. The designers of ParalleX present four critical bottlenecks affecting the efficiency of current parallel execution models [3]:

- Starvation due to an inability to utilize and manage application parallelism
- Latencies of accessing local and remote resources
- Overhead of managing parallel access to machine resources
- Waiting for contention resolution of multi-core chip I/O pins, memory banks, and network interfaces

A core design principle of the ParalleX model is to improve parallel performance by attempting to offset the effects of these inefficiencies. This goal transcends the design of any single component in the system, and, instead, encompasses considerable co-design between all layers of the execution model.

ParalleX aims to improve the parallel performance of future systems through the application of message-driven execution in the context of a global namespace using lightweight synchronization primitives [29]. The model is largely dependent on the use of active messages [34, 33] to implement the message-driven flow control, which allows for the overlapping of computation and communication. The global namespace framework facilitates the dynamic distribution of data and simplifies parallel programming, as data may migrate from node to node and may be referenced regardless of where it is physically stored. Lightweight synchronization mechanisms improve processor efficiency by avoiding the over-constraining nature of global synchronization. Other key design features of the model for improving parallel performance include the application of finer-

grained multithreading, inherent latency hiding of remote operations, and dynamic, adaptive resource management techniques for contention resolution.

To provide more insight into the ParalleX execution model, it is necessary to consider the function and cooperation of the fundamental components of the model [29], which include:

- *Active Global Address Space* (AGAS) - The AGAS module provides mechanisms for maintaining a system-wide global namespace, which decouples the access of a data object from its actual physical location.

- ParalleX *Processes* - A ParalleX process provides the full context of all computation contained in a particular parallel application, including threads, application data, methods, synchronization mechanisms, and child-processes. Processes may span multiple nodes and even share nodes, in contrast to contemporary processes which generally are statically mapped to a single processing core.

- *Threads* and Thread Management – ParalleX threads provide the smallest unit of computational work that may be represented and scheduled for execution on a single node. A thread manager is responsible for scheduling threads in the most efficient manner, given runtime system information.

- *Parcels* and Parcel Management - The ParalleX model achieves inter-node communication using parcels, active messages which typically encapsulate an

action to be performed, but may also reference user data. A parcel manager is used to demultiplex incoming parcels to different parts of the system.

- *Lightweight Control Objects* (LCOs) - LCOs represent a set of flow control mechanisms whose semantics allow for the event-driven instantiation of threads, dynamic work distribution, and the prevention of race conditions in parallel applications.

- *Percolation* – Percolation is a special technique for moving work to data to make efficient use of heterogeneous resources, like GPGUs, by hiding the latency of accessing the resource and reducing the overhead of using it.

As a first attempt for implementing the ParalleX execution model, the designers of ParalleX have developed High Performance ParalleX (HPX), a parallel runtime system meeting the specifications of the model. Although the ParalleX model suggests potential redesign of all system layers, including hardware architecture, an experimental runtime system like HPX can be used to validate the model and provide necessary feedback for guiding future parallel system design. The core design objective of the HPX runtime system is to provide an efficient, modular, and portable framework for the development and execution of ParalleX applications [29]. A modular diagram demonstrating the HPX runtime architecture is given in Figure 2.1. Currently, HPX does not support ParalleX processes or percolation, but implements all other ParalleX components, which are described in further detail below.

The AGAS component maintains a translation table that provides the global virtual address for all objects in the execution environment, both local and remote. This allows for a less restrictive programming model, as a programmer does not need to be aware of where an object is stored to reference it. The design of the AGAS component is largely based on previous research in Partitioned Global Address Space (PGAS) programming models, which provide a logical global address space composed of each contributing thread's local memory partition. PGAS models attempt to improve the productivity and performance of parallel programming by combining the convenience of shared memory programming with the performance control of message passing models [36]. The AGAS model improves upon previous PGAS implementations by allowing objects to migrate throughout the system without the added overhead of a virtual address translation. This seamless migration is crucial to supporting the dynamic load-balancing inherent to the ParalleX model. To assist in providing a global namespace abstraction, the ParalleX model introduces the notion of a *locality* to delineate the boundaries in a parallel system. A locality is defined as a contiguous physical domain, which guarantees atomic operations on local resources [3]. In a cluster environment, a locality is typically equated to a node, where intra-locality accesses require access to local memory and inter-locality accesses require access to the interconnection network.

The HPX thread manager is responsible for the runtime scheduling of a potentially large amount of independent HPX threads. To support dynamic and adaptive work-load balancing, the thread manager employs a work queue based scheduling discipline [29]. Also, the thread manager schedules threads in a cooperative manner to

limit the ill effects of context switching and cache thrashing. It is worth noting that HPX treats threads as first-class objects, so, they may be managed remotely or even migrated between localities. However, thread migration is typically avoided as it is more computationally efficient to send a parcel that spawns a thread remotely, rather than transmit the entire thread context across the network.

As mentioned previously, parcels are an extended form of active messages used for inter-locality communication. To perform an operation on a remote object, typically a HPX thread will send a parcel to that locality encapsulating the function to be executed and the corresponding parameters. When the remote locality receives the parcel, it decodes it and schedules the contained function for execution via the thread manager. This functionality allows the moving of work to data (instead of data to work), which is preferable in many scenarios since it takes advantage of data locality and allows overlapping of computation with communication.

Two of the more useful synchronization operations utilized by HPX are the *future* and *dataflow* LCOs, although conventional mechanisms like semaphores, mutexes, and conditions are also included. A future is used as a proxy for a result that has not yet been calculated [4]. A thread requesting the value of the future may suspend its execution until the value is available, allowing other threads to perform meaningful work while the remote operation completes. The dataflow LCO defines a set of pre-conditions that must be satisfied before a specific follow-on thread is instantiated [3]. This mechanism helps address the inefficiencies of typical global barriers by using a light-weight, event-driven

synchronization mechanism that allows each given precondition to be updated asynchronously.

The following discussion describes the flow of data and control throughout the HPX runtime environment (Figure 2.1). Incoming parcels are delivered over the interconnection network to the parcel port, which passes each parcel to a parcel handler to buffer. The action manager then fetches and decodes each parcel, scheduling the encapsulated thread for execution with the thread manager. The thread manager dynamically schedules a pool of threads, which operate on local and remote objects. The action manager queries the AGAS translation table on behalf of threads to determine if

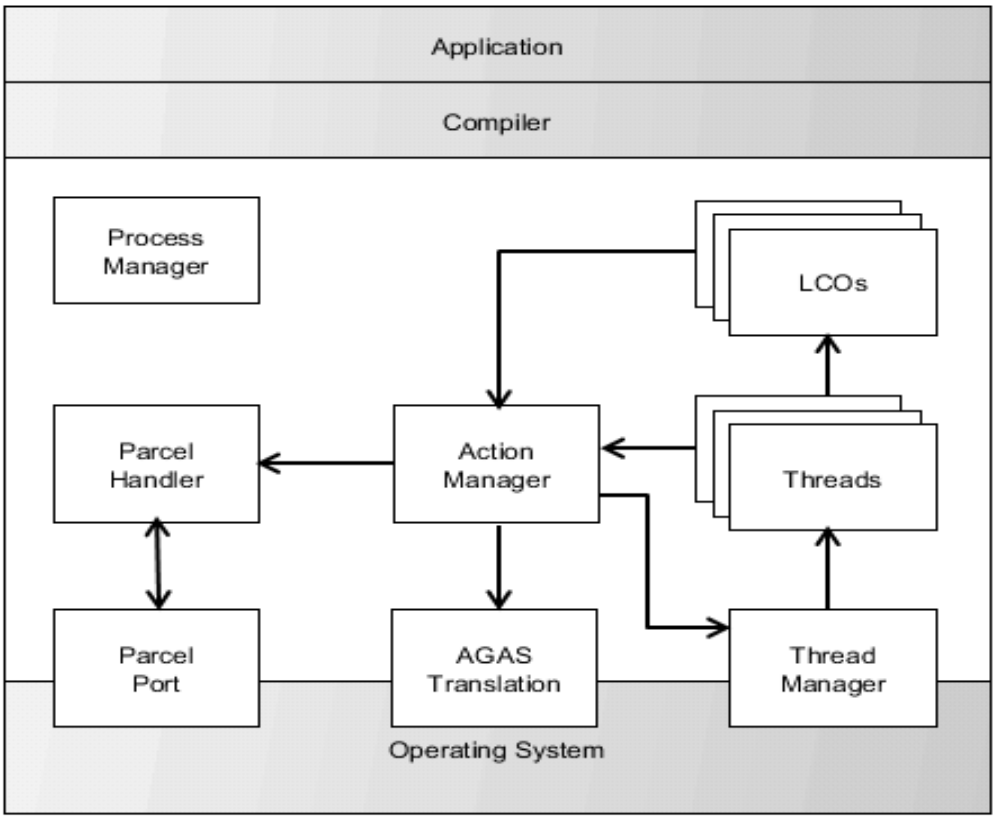


*Figure 2.1 - Modular diagram of the HPX runtime system.*

referenced objects are remote or local. If the object is local, the action manager simply creates a new thread, but if it is remote, the action manager encapsulates the necessary work and sends it to the remote object using the parcel port. LCOs (created by individual threads) notify the action manager when new or suspended threads may begin executing.

## 2.2 OrangeFS

OrangeFS is a recent branch of Parallel Virtual File System 2 (PVFS2), a high-performance, open-source parallel file system designed for use in HPC systems. Some of the more appealing features of PVFS2 include high parallel I/O performance, reliability, and hardware independence [24]. PVFS2 achieves high I/O throughput by dynamically distributing file data and metadata across a system. This strategy alleviates file system bottlenecks and improves system scalability. The reliability and performance of the file system also benefit from relaxed consistency semantics, which obviate the need for a file locking subsystem. PVFS2 maintains compatibility with a wide range of instruction-set architectures, storage systems, and network architectures, exhibiting a high level of hardware independence. Also, the modular software architecture of PVFS2 allows new hardware technologies to be integrated smoothly. PVFS2 offers an ideal platform for research in the area of parallel I/O, due mostly to the high-performance and modular nature of its architecture coupled with the fact the software is openly available for redesigning. OrangeFS offers many of the same features of PVFS2, but was branched specifically to provide additional capabilities that seek to improve the performance of a

wide range of parallel systems. This is in contrast to PVFS2, which focused design solely on improving the performance of specific parallel system architectures (typically, clusters or other large distributed systems) [22]. Among the new features provided by OrangeFS are improved scalability of metadata and directory operations, configurable redundancy and fail-over mechanisms, and secure access controls for protecting file system data.

OrangeFS utilizes an intelligent server architecture, in which server processes serve file data and metadata to client processes [21]. In a cluster environment, I/O nodes generally execute the server processes (due to their specialized storage hardware), while the compute nodes execute the client applications. The file system may be configured to use any number of servers, with each serving file data and/or metadata. Tunable data distribution parameters allow users to increase file system performance by distributing data in a manner that complements the file access patterns of a particular program. By default, file data is striped evenly across a set of servers to take advantage of the aggregate sum of network bandwidth, however, numerous data distribution configurations are supported. Similarly, metadata may be distributed among servers at a per directory granularity to avoid overloading individual servers [8]. Also, the file system utilizes server-to-server collective communication to improve the performance and scalability of many metadata operations [22].

The software architecture of OrangeFS clients and servers is given below in Figure 2.2. Typically, a client will make calls into a high-level I/O library (e.g., an MPI-IO implementation) to utilize the file system. These libraries must leverage the *system*

*interface*, the lowest level API available to programmers for accessing the file system. Essentially, the system interface provides programmers with the necessary building blocks for developing both general and domain-specific I/O libraries. The system interface implements file system operations using *state machines*, a software module that allows multi-step file operations to be represented using explicit state machines. Specifically, state machines define the function to be executed for each state (step) and the order in which these functions are executed. State transitions typically occur after completion of a low-level I/O operation, where the operation's error code is used to determine which state to transition to next [8]. State machines are particularly valuable to a parallel file system, as they allow independent I/O operations to be serviced in a concurrent and asynchronous manner. The server-side libraries also utilize state machines to manage the execution of concurrent file system operations. The *job interface's* primary purpose is to bind together the high-level file system libraries described above with the

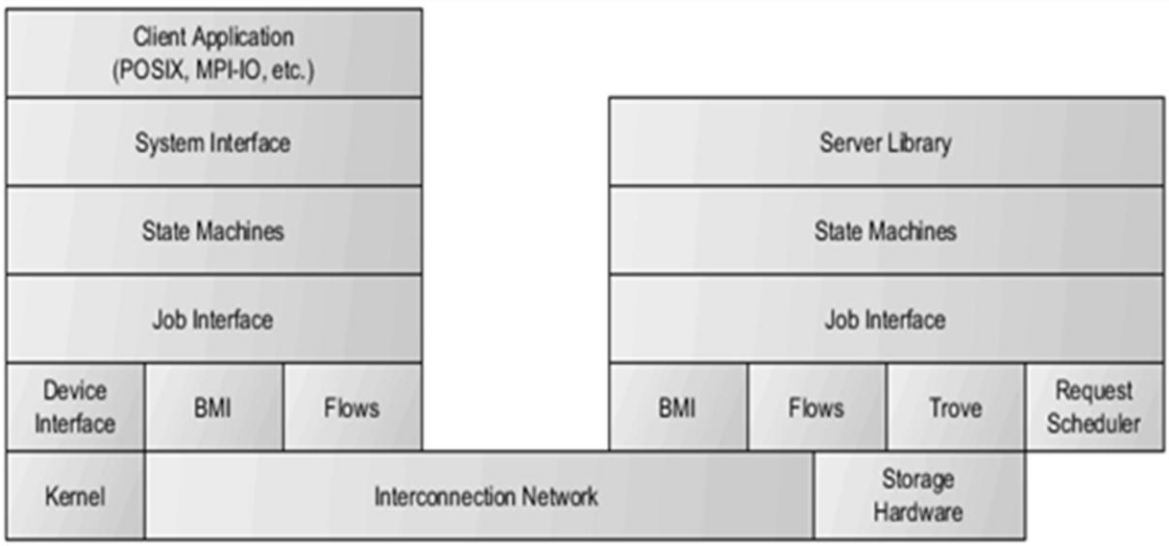


*Figure 2.2 - Software stacks for OrangeFS clients and servers.*

low-level hardware components of the file system architecture [22]. It provides a single interface for the posting and completion checking of storage and network operations.

The lowest level of the OrangeFS software stack is composed of hardware abstractions for the storage and network subsystems. Both the client and the server utilize the Buffered Message Interface (*BMI*), which provides consistent access to a variety of network architectures, including TCP/IP and InfiniBand. The design of BMI simplifies the integration of new network architectures, due to its layered interface model – a high-level API is provided to BMI users while implementation-specific API's are maintained in separate software modules [8]. The OrangeFS server leverages the *Trove* storage abstraction to provide bytestream and key/value access to available storage resources. Trove uses regular files to store bytestreams, while a database implementation is used to store key/value pairs (metadata). The *flows* subsystem is used to coordinate the network and disk transfers initiated by BMI and Trove. Particularly, it handles the buffering, scheduling, and datatype processing necessary for bulk data transfers [8]. The server process contains an additional software component for managing the consistency between concurrent file system requests. The *request scheduler* analyzes incoming requests and orders them in a manner that protects the consistency of the file system. Also, the client software stack includes the optional *kernel module*, which allows an OrangeFS file system to be mounted and used like a standard Linux file system.

## 2.3 Related Work

Many languages have been proposed to satisfy the PGAS execution model, with UPC [5] and Titanium [31] representing two of the more well-known attempts. UPC (Unified Parallel C) is an explicit parallel extension to ISO C, which coordinates independent threads in a SPMD fashion. Titanium offers similar features as UPC, except that it is a dialect of the Java language targeted for application in large HPC systems like clusters. Both UPC and Titanium support the PGAS global memory model which allows programmers to control the processor affinity of shared data, which increases potential performance of parallel applications. Also, both languages utilize GASNet [6], a low-level, language-independent networking layer that provides a portable interface for high-performance one-sided communication. The GASNet core API is largely based on the use of active messages, while an extended API offers more expressive operations like collective communication and remote memory access.

APGAS (Asynchronous Partitioned Global Address Space) languages improve upon standard PGAS languages by allowing the asynchronous creation of both local and remote tasks. One such APGAS language is X10 [26], an object-oriented, high-performance parallel programming language (Java-based) targeted for application on non-uniform memory access (NUMA) compute clusters. X10 distributes asynchronous computation *activities* among a set of *places*, abstractions that encapsulate local data and computation. Chapel [12] is another global address space language that offers many high-level operations for expressing data parallelism, task parallelism, and nested parallelism.

[37] present a fine-grained parallel execution model based on compiling parallel applications into small code snippets called codelets. High-performance parallel languages like X10 and Chapel can be implemented on top of this runtime model, as long as a compiler exists that can translate the respective parallel applications to a set of independent codelets.

There has been extensive research into adding parallelism constructs to C++, one of the most popular programming languages for application developers. Cilk++ [17] is a proposed parallel programming environment (incorporating a compiler, runtime system, and race-detection toolkit) that extends C++ with operations that allow for thread spawning, automatic loop parallelization, and local barrier synchronization. The Cilk++ runtime environment guarantees to effectively load-balance parallel computation, due in large part to the use of a work-stealing scheduler. Charm++ [16] is an object-oriented parallel programming language based on C++, which utilizes message-driven computation on parallel processes called *chares*. An important design feature of Charm++ is the extension of traditional object-oriented principles like inheritance and dynamic binding to concurrent objects. Intel presents a C++ library for supporting scalable parallel programming called Threading Building Blocks (TBB) [25]. This library allows for convenient parallel programming through the specification of concurrent tasks that are dynamically scheduled by the runtime environment, freeing the programmer from the burden of creating, synchronizing, and destroying actual threads.

Many parallel file systems have been proposed improve the performance, portability, and consistency of concurrent I/O operations. Lustre [18] is a POSIX-compliant file system for use in large-scale clusters. Lustre is renowned for its ability to support a large number of client processes while maintaining high I/O throughput. IBM's General Parallel File System (GPFS) [15] provides another high-performance parallel file system which offers high availability through dynamic file system management and data replication.

Since the I/O subsystem has historically been the largest hindrance to the high performance of parallel applications, much research is necessary in improving the performance and scalability of I/O architectures for exascale systems. [10] propose a new I/O architecture that dynamically coordinates I/O accesses according to program access patterns, network topology, network condition, and physical data distribution on storage devices to reduce contention and preserve data locality. [35] present numerous tools for utilizing latent I/O asynchrony in HPC applications. Latent I/O asynchrony is essentially the decoupling of ancillary I/O operations from the core computation of a parallel application. [1] propose a new I/O paradigm, referred to as I/O forwarding, in which compute nodes ship I/O calls to dedicated I/O nodes. The dedicated I/O nodes then perform requests on behalf of the compute nodes, optimizing I/O performance by aggregating and caching file system operations.

# Chapter 3

# Design and Methodology

In this chapter, we first propose the desired features for an I/O subsystem fitting the ParalleX execution model. We then outline modifications to be made to the OrangeFS file system to accommodate these features, yielding our PXFS prototype.

## 3.1 ParalleX I/O Characteristics

Before developing our PXFS prototype, we first characterize the I/O requirements of the ParalleX model. Above all, the ParalleX I/O subsystem must support the overall goals of the execution model, which include the use of asynchronous communication to mask high-latency operations. While there are numerous parallel I/O libraries available with support for asynchronous operations, none directly support the advanced semantics required by the ParalleX model. Essentially, a ParalleX thread attempting an I/O operation will make a call to the I/O subsystem to submit the asynchronous request to the

underlying file system. It is desirable that this asynchronous I/O operation incur as little overhead as possible, to limit the blocking time of calling thread. This allows the calling thread to continue performing meaningful computation while waiting for the I/O operation to complete. The I/O subsystem should also utilize *continuations* to notify of the completion of an operation, where a continuation is a specifier for some follow-on action to be taken. This continuation can be used to notify the calling thread of the completion of the operation, submit another I/O operation, instantiate a new thread of execution, or perform any other defined functionality.

Obviously, the I/O subsystem must scale with the increasing number of concurrent I/O operations that are probable in exascale HPC applications. Otherwise, any potential I/O performance gains will be offset by the high contention for accessing the I/O subsystem. Also, the I/O subsystem should include a complete set of high-level file operations, not just basic file reads and write. ParalleX applications require the ability to perform a range of file operations, such as open, close, seek, make directory, create file, etc., and expect that all operations follow the same asynchronous model discussed above. These I/O operations should all be contained within a well-defined, consistent user interface, which provides the necessary ParalleX I/O semantics with limited programmer intervention.

So, we conclude that any I/O subsystem attempting to satisfy the I/O requirements of the ParalleX model should provide the following features:

- Asynchronous I/O operations

- Low overhead for submitting operations
- Continuation support for notifying of I/O completions
- Support for high-performance, high-concurrency I/O
- Breadth of high-level I/O operations
- Well-defined user interface providing the ParalleX I/O model semantics

## 3.2 PXFS Design

As stated previously, we will implement our PXFS prototype by modifying the OrangeFS file system source code to accommodate the changes required by the ParalleX model. To support the asynchronous submission of high-level file system operations, modifications must be made to the OrangeFS client-side software stack (Figure 2.2). The server-side libraries remain unchanged, because client requests are handled the same way by the server, regardless of whether they are synchronous or asynchronous. The modifications to the client software stack are detailed in the following sections, starting with the highest level (the PXFS I/O library) and moving to the lowest level (client-side state machines).

### 3.2.1 PXFS I/O Library

The PXFS I/O library is the high-level interface exposed to ParalleX components or applications needing to perform I/O operations. Therefore, this library must explicitly

support the semantics of the ParalleX I/O model, but must do so by leveraging functionality contained within the OrangeFS file system. Before focusing on the actual implementation of the library, we first decide on the set of I/O operations we wish to support. Leaning on convention, we initially seek to implement many of the file and I/O operations included in the POSIX specification. POSIX was chosen as a reference because most of its I/O operations and datatypes are well-understood by programmers and straight-forward to implement – we are not interested in designing a POSIX-conformant I/O library. Basically, we borrow the naming conventions and functional parameters of POSIX operations as a basis for our PXFS I/O library. Since the PXFS I/O functions are asynchronous and do not return the desired value immediately, output values are passed as pointers to the functions and are updated after the completion of the I/O operation. A complete list of operations currently supported by PXFS is given in Table 3.1.

From our characterization of the ParalleX I/O model, we already know that a ParalleX thread calling into the PXFS I/O library expects to submit an asynchronous operation that blocks for as short as possible, requiring that the I/O library incur minimal overhead. To support this requirement, we design the PXFS I/O library to perform only the necessary computation before submitting an operation to the asynchronous I/O (AIO) subsystem. This necessary computation includes checking and modifying functional parameters, specifying the asynchronous operation to be performed, and submitting the operation to the AIO subsystem. To specify an asynchronous operation, a data structure called an *asynchronous control block* is allocated and populated with the necessary

| PXFS I/O Operation | Purpose |
|---|---|
| pxfs_open(64) | Open a file. |
| pxfs_creat(64) | Create a new file. |
| pxfs_unlink | Remove a file. |
| pxfs_rename | Rename a file. |
| pxfs_read, pxfs_pread(64), pxfs_readv | Variations of file read. |
| pxfs_write, pxfs_pwrite(64), pxfs_writev | Variations of file write. |
| pxfs_lseek(64) | Repositions a specific file pointer. |
| pxfs_ftruncate(64), pxfs_truncate(64) | Truncate a file to a desired length. |
| pxfs_close | Close a file. |
| pxfs_fstat(64), pxfs_stat(64), pxfs_lstat(64) | Retrieve a file's status. |
| pxfs_fchown | Change a file's user and group ownership. |
| pxfs_fchmod | Change a file's permissions. |
| pxfs_mkdir | Make a new directory. |
| pxfs_rmdir | Remove a directory. |
| pxfs_symlink | Create a symbolic link to a file. |

*Table 3.1 - I/O operations currently implemented by PXFS*. Note that 64-bit functions allow files over 2 GB to be manipulated by the file system.

parameters. These parameters include original function arguments, as well as internal file system data structures necessary to servicing the operation. The asynchronous control block is all that is required by the AIO subsystem to service a particular I/O request. Further details explaining the design and implementation of the AIO subsystem are provided in Section 3.2.2.

Since the ParalleX I/O model also requires support for continuations, the PXFS I/O library should include the necessary definitions for specifying continuations for completed I/O operations. Accordingly, we first define a continuation specifier, which simply represents a continuation as two separate pointers: a pointer to a function to be executed upon the completion of a particular I/O operation and a pointer to a block of

user data to be passed as input to this function. A valid continuation function must also accept a resultant I/O error code as input, which states whether the I/O operation succeeded, and if not, the specific error (as defined by the POSIX error code definitions). Basically, a continuation specifier defines an arbitrary function that operates on a specific block of data supplied by the programmer, depending on the error code of the I/O operation. With a suitable definition for a continuation specifier, we update all operations in the PXFS I/O library to accept this specifier as a functional parameter, along with the operation's respective POSIX arguments. This specifier is subsequently stored in the asynchronous control block to be leveraged by the AIO subsystem upon completion of the I/O operation. To demonstrate the general structure of the PXFS I/O library functions, we provide an example function call in Figure 3.1.

A data flow diagram illustrating the asynchronous nature of the PXFS architecture is given in Figure 3.2. This diagram provides insight into how ParalleX threads submit I/O operations and respond to their completion. As stated earlier, a ParalleX thread blocks for a short time to submit an I/O operation then resumes normal execution after it

```
int pxfs_write(int fd, const void *buf, size_t count,
               ssize_t *bcnt, pxfs_cb cb, void *cdat);
```

*Figure 3.1 – Prototype for the PXFS write operation.* The first three arguments correspond directly to the POSIX write definition: *fd* represents the associated file descriptor, *buf* points to the buffer of data to be written, and *count* specifies the amount of bytes to be written from the buffer. The next argument, *bcnt*, points to the location where the output value (number of bytes written) should be written. The last two arguments correspond to the continuation specifier: *cb* stores the associated callback function for the continuation and *cdat* stores the pointer to the user-supplied data block. The function returns 0 on a successful asynchronous submission to PXFS and returns -1 if an error occurred before submission.

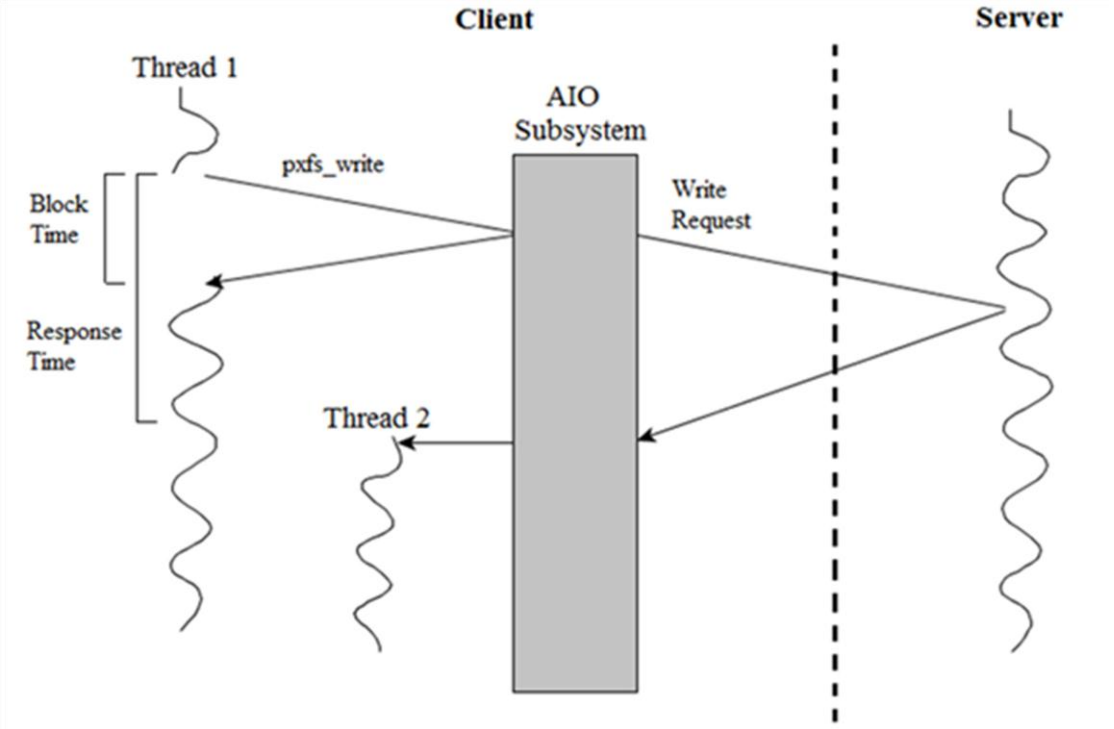


*Figure 3.2 – Data flow diagram illustrating an example I/O operation in the PXFS I/O library.* Thread 1 submits a PXFS write operation with a continuation which spawns Thread 2.

has been submitted, allowing for autonomous overlapping of computation with file system communication. After a thread has submitted an I/O operation, the PXFS AIO subsystem manages all client-side computation and communication necessary to service the operation, clearly alleviating the programmer from the burden of repeatedly checking for its completion. When the operation has finally completed, the AIO subsystem utilizes the provided continuation specifier to notify the ParalleX application. In the case of Figure 3.2, the continuation spawns a new thread of execution, but could be easily modified to notify the original thread or perform any other functionality.

### 3.2.2 AIO Common Interface

The main purpose of the AIO Common interface is to offer a consistent interface to high-level I/O libraries, such as the PXFS I/O library, wishing to utilize the asynchronous file system operations supported by PXFS. The benefit of layering libraries in this manner is that it directly supports our design goal of providing modular software architecture. Changes or optimizations may be made to the AIO common interface without requiring any modifications to be made to high-level libraries. Also, unnecessary complexity is removed from high-level I/O libraries – these libraries may utilize functionality contained in the AIO common interface which abstracts away the low-level details of performing an asynchronous I/O operation. This modularity not only increases programmer productivity, but also reduces the likelihood of developing buggy I/O libraries.

To simplify the design of the AIO common interface, we expose a single function (*aiocommon_submit_op*) for submitting asynchronous operations to PXFS. This function simply queues the referenced control block for service by the file system then returns control back to the caller, which minimizes the blocking time for user threads. However, managing this queue in a highly concurrent environment represents a potential bottleneck, as many threads contend for queue access simultaneously. The submit function accepts an asynchronous control block as its only parameter, so, clearly, the asynchronous control block must contain all information required by the file system to service the operation and notify the caller of its completion. Table 3.2 provides an

| Field | Data Type | Purpose |
|---|---|---|
| op_id | PVFS_sys_op_id | Unique identifier used to track the progress of a particular file system operation. |
| op_code | PVFS_aio_op_code | Code indicating the particular file system operation represented by the control block (open, read, write, etc.). |
| error_code | PVFS_error | Code indicating the current status of the asynchronous operation (i.e., in progress, finished, or in error). |
| link | qlist_head | Linked list node used to store the control block in a waiting or running queue. |
| call_back_fn | void (*)(void **c_dat*, int *status*) | Function pointer which points to a user-supplied callback function to be executed upon the completion of an operation. *c_dat* points to a block of data given by the user and *status* provides the final error code of the operation. |
| call_back_dat | void * | Void pointer used to store an arbitrary block of user data that will be passed to the given callback function. |
| u | union | Union used to store the operation-specific parameters and internal data structures necessary to service an asynchronous file system operation. |

*Table 3.2 – Definition of the PXFS asynchronous control block.*

overview of each of the fields that compose a PXFS asynchronous control block. It is important to note that the file system relies solely on the use of callback functions to notify a user of the completion of an I/O operation, requiring user applications to supply a function meeting the specifications given in the asynchronous control block. While this could be considered burdensome to library developers, it also allows for a wide range of

high-level functionality by providing mechanisms for defining some arbitrary function to be performed upon the completion of an operation.

A major design consideration of the AIO common interface is the architecture of its runtime system, which services asynchronous file system operations and notifies application threads when these operations complete. An overview of our proposed AIO common runtime system is given in Figure 3.3. As illustrated, the runtime architecture utilizes three primary components: a *waiting queue*, a *running queue*, and a *progress*

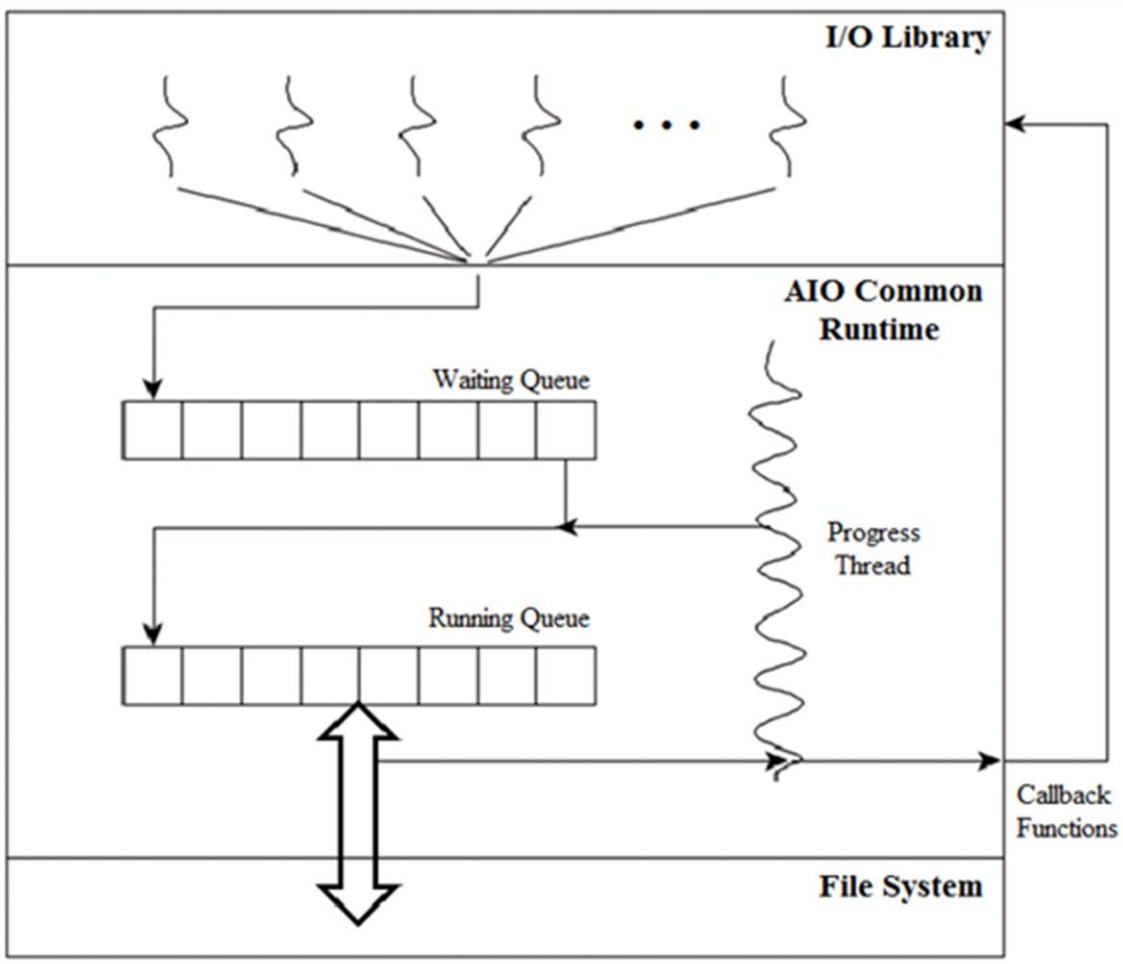


*Figure 3.3 – Architecture of the AIO common runtime system.*

*thread*. Asynchronous control blocks submitted by a high-level I/O library are buffered at the tail of the waiting queue, where they wait to be serviced by the file system. The AIO common progress thread then moves the control blocks from the head of the waiting queue to the running queue, where they are finally submitted to the file system. Note that the progress thread and a potentially large number of application threads may access the waiting queue simultaneously, so a mutex lock is required to protect the integrity of the queue. On top of submitting asynchronous operations to the file system, the progress thread is also tasked with forcing progress on any currently running operations, removing completed operations from the running queue, and notifying applications of operation completions using supplied callback functions.

To provide a more complete understanding of the functionality necessary to servicing asynchronous file operations, a flow diagram of the AIO common progress thread is given in Figure 3.4. The progress thread first moves any waiting operations to the running queue, assuming it has not reached capacity. It is necessary to impose a limit on the number of running operations, because submitting many concurrent operations can negatively affect the performance of the file system. The waiting queue may grow arbitrarily long, depending on the rate of incoming requests and the service time of the file system. If there are no operations waiting or running, the progress thread suspends execution – a user thread that submits an asynchronous I/O operation and finds the progress thread idle signals the progress thread to wake up and service its request. If operations are running, progress is forced on each eligible operation (i.e., file operations with work available) using a test function. The test function returns any operations which

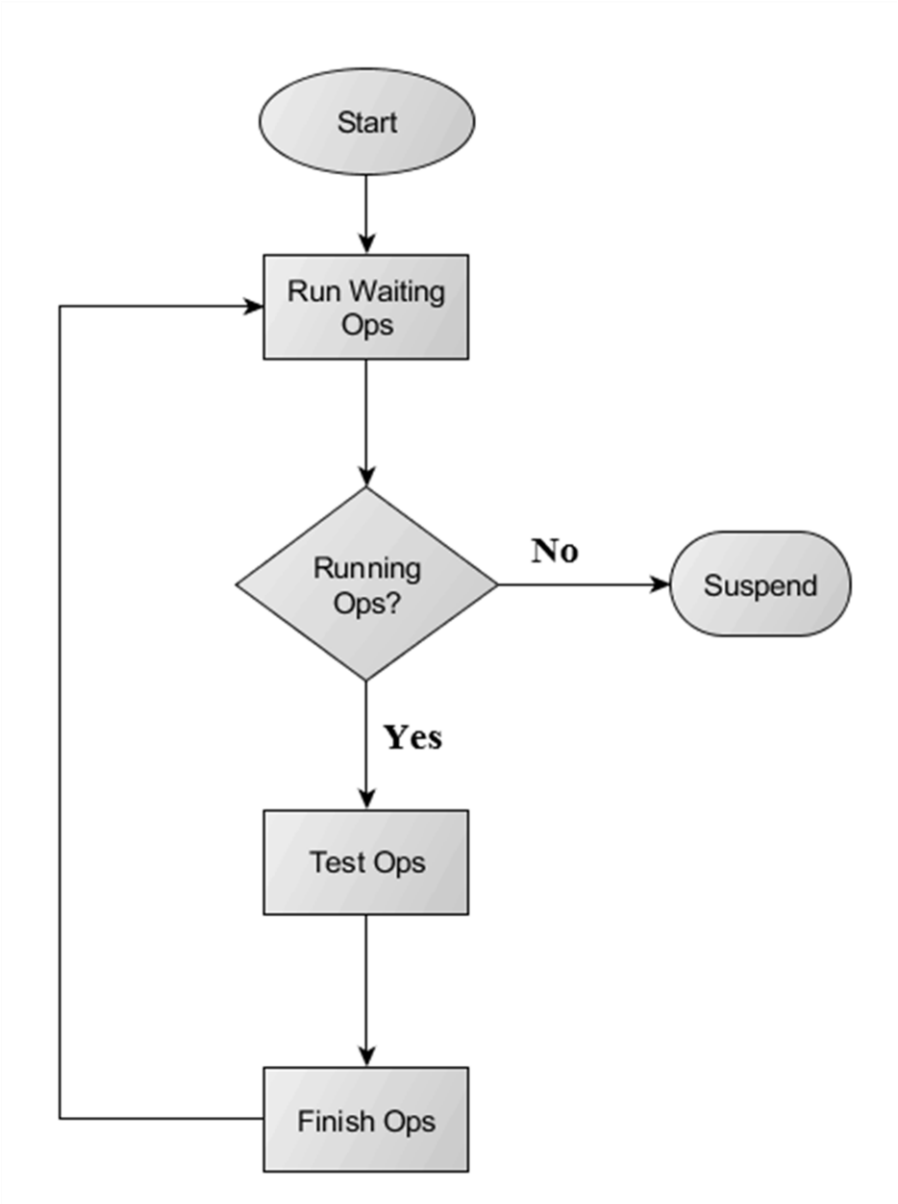


*Figure 3.4 – Flow diagram for the AIO common progress thread.*

executed to completion, whether they be successful or in error. These completed operations are then removed from the running queue and their respective callback functions are executed to notify the caller of their completion. This entire process is repeated until there are no requests left to service. To aid in our implementation of this

algorithm, we rely on the POSIX Pthreads API, which provides mechanisms for creating threads and synchronizing their execution.

For each supported asynchronous operation, the AIO common interface defines functionality for submitting the request to the file system (*initializers*) and returning the file system response (*finalizers*). Initializers perform any necessary transformation on functional parameters contained in an asynchronous operation's control block then initiate an operation-specific state machine to service the request. When the operation completes, the finalizer copies the file system response into the appropriate output fields, as specified by the operation. For example, the finalizer for the asynchronous open operation would copy the file descriptor returned by the file system to a user-supplied variable so that it may be accessed by the original application.

### 3.2.3 State Machines

OrangeFS utilizes a finite-state machine implementation to represent and service low-level, multi-step I/O operations. Each state represents a specific function to be executed, which typically includes the posting of a remote file system operation. The nesting of state machines is also allowed, so a single state may represent an entire state machine to be executed. This is convenient as it simplifies state machine complexity and facilitates code reuse. The return value of a particular state function or nested state machine is used to determine which state to transition to next, a process that continues until the file system operation succeeds or fails. Essentially, state machines offer a way

to define the structure and manage the flow of multi-step file system operations, which may require multiple nested operations, such as a file lookups, file creates, etc.

Conventionally, high-level OrangeFS file system operations have been defined as a sequence of calls to the system interface, as shown in Figure 3.5. The system interface, the lowest level interface available for accessing OrangeFS files, offers a comprehensive set of low-level file system operations in synchronous and asynchronous form – the synchronous code simply calls the asynchronous code and waits for its completion before

```
Operation  orangefs_rename(old_path, new_path) :

    old_dir, old_name = split_path(old_path);
    new_dir, new_name = split_path(new_path);

    old_ref = orangefs_sys_lookup(old_dir);
    if (error) return error;

    new_ref = orangefs_sys_lookup(new_dir);
    if (error) return error;

    orangefs_sys_rename(old_name, old_ref, new_name, new_ref);
    if (error) return error;
    else return success;

End
```

*Figure 3.5 – Pseudocode for the file rename operation in OrangeFS.* This high-level operation accepts two file paths as input (old_path and new_path) and renames the first path to the second path. After breaking these paths into respective directory and file names, the function looks up the references for both directories using the system interface lookup call. After obtaining the parent directory references, the system interface rename call may be used to submit the rename request to OrangeFS. It can be seen that the OrangeFS file rename operation requires 3 separate system interface calls, each of which requires an operation-specific state machine.

returning. Each asynchronous operation concludes by starting a state machine for servicing the given file system request. After a state machine has been started, it must be repeatedly polled by a test function to check for its completion. If the operation has not been completed, the test function will force progress on that operation and any other operation with an incomplete state machine. This demonstrates the concurrent processing abilities of state machines, as the execution of states from independent machines may be interleaved with each other. Once a state machine has completed execution, the data from the server response is returned to the user through the system interface.

So, while OrangeFS supports the asynchronous submission of low-level file system operations via state machines, there is no explicit support for asynchronous high-level operations, like file open, close, seek, etc. This is mostly because these high-level operations involve a possibly complex sequence of state machines, making it extremely cumbersome for library developers to manage the flow between the operations in a manner that appears completely asynchronous to the user. Obviously, new low-level mechanisms are required for representing and executing asynchronous file system operations without adding unnecessary complexity to high-level I/O libraries or requiring an inordinate amount of programmer intervention. To support this functionality, we must provide a meaningful and efficient way of managing the necessary state and dynamic flow of high-level I/O operations.

Therefore, we propose the addition of new PXFS state machines to represent these high-level file operations, allowing each state to specify some nested state machine

or necessary function for providing the desired result. Essentially, we extend the state machine abstraction to define high-level file operations, which generally require the execution of a sequence of nested state machines and state functions. An example state machine implementing the rename operation defined in Figure 3.5 is presented below in Figure 3.6. The benefits of using state machines to implement high-level file system operations are two-fold: the potentially complex flow of these operations is easily specified using finite-state machines and the use of state machines allows an arbitrary

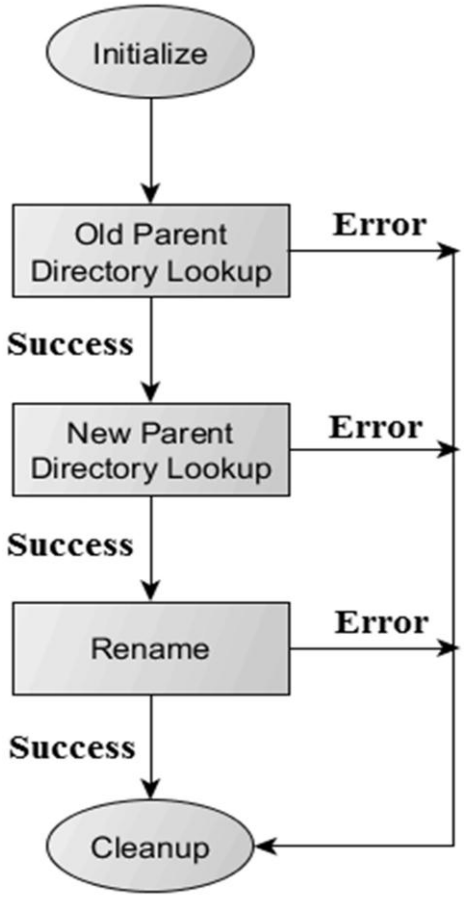


*Figure 3.6 – Example PXFS state machine for the file rename operation*. Note that an operation's state trajectory depends on the error code of each nested operation.

function to be serviced asynchronously by the file system. This allows higher level libraries, such as the AIO common interface, to define asynchronous file operations in terms of a single state machine, which is much more manageable than an elaborate sequence of state machines.

## 3.3 Summary

Initially, we analyzed the ParalleX execution model in an attempt to define a suitable I/O subsystem to be used with model. We concluded that an I/O subsystem meeting the semantics of the ParalleX execution model must support the following features: asynchronous I/O operations, low operation submission overhead, continuation support, high operation concurrency, a wide-range of I/O operations, and a well-defined interface for utilizing the library.

We then proposed modifications and additions to the OrangeFS file system to satisfy these requirements, resulting in our first PXFS prototype. Our implementation involves the design of the high-level PXFS I/O library, the AIO common interface, and new client-side state machines. The PXFS I/O library defines a wide-range of I/O operations supporting the semantics of the ParalleX I/O model. The AIO common interface provides functionality for servicing asynchronous operations using a runtime system that manages these requests and forwards their response back to users. New state machines were developed to allow high-level I/O operations to be represented and

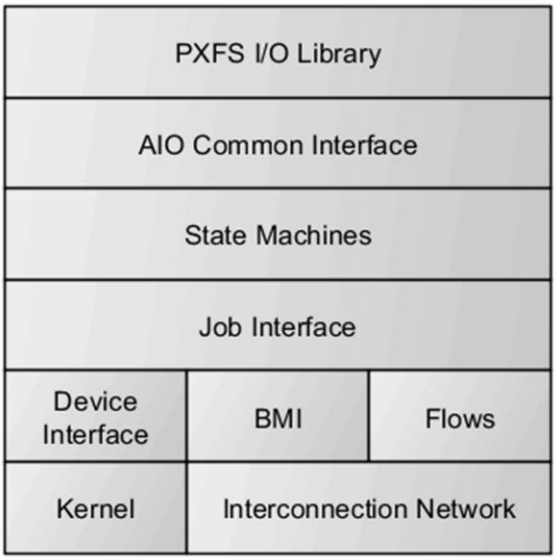


*Figure 3.7 – Modified client-side stack for PXFS clients.*

managed using a single asynchronous state machine. The resultant client-side software stack is shown in Figure 3.7.

It is important to note that this is only our first prototype of the PXFS file system, and, thus, is not fully compatible with the ParalleX execution model. Eventually, we hope to fully integrate PXFS with the execution model by modifying PXFS to utilize ParalleX parcels and threads directly. This reduces potential I/O overhead and allows the ParalleX thread manager to explicitly manage the execution of all threads, including the PXFS progress thread. However, our initial implementation will allow us to analyze the long-term feasibility of integrating our current I/O model with the ParalleX execution model.

# Chapter 4

# Results

In this chapter we outline experiments to benchmark the performance of PXFS with the original OrangeFS file system, as well as, verify the feasibility of PXFS as a suitable I/O subsystem for the ParalleX execution model.

## 4.1 Read/Write Throughput Results

To compare the potential performance of PXFS with the original OrangeFS file system, we first outline experimental application test cases for submitting file reads and writes to the underlying file system. This allows us to analyze the performance of the file systems from the perspective of numerous application I/O patterns. These test cases are outlined below:

- Case 1: A number of threads simultaneously read and write a set of files. Each thread waits for the completion of its previous I/O operation before submitting a new one, and each operation references a random block in the file.
- Case 2: A single thread submits a list of sequential I/O operations to either read or write a file one block at a time. It is unnecessary to wait for the

completion of prior operations to submit new ones, as there is no file or buffer overlap between operations.

- Case 3: A single thread repeatedly submits sequential I/O operations then simulates computation after each operation has been submitted. Each I/O operation references equal size blocks and the length of the computation time is fixed. Similarly to case 1, the PXFS application must wait for the completion of the current I/O operation (and computation) before submitting a new operation.

Before running any experiments, it is important to note the hardware configuration details of our testing environment. For simplicity, we use a single file system client and a single file server for our testing environment, both containing identical hardware, detailed in Table 4.1. Both the client and server nodes run the CentOS 5.6 Linux distribution.

### 4.1.1 Random Read/Write (Case 1)

For test case 1, a simple OrangeFS application was developed that spawned a specified number of threads, each of which performs a sequence of I/O operations on a set of files. Each I/O operation is completely random – it is equally likely that the

| CPU | Memory | Storage | Network Interconnect |
|---|---|---|---|
| Intel Xeon 3040 (dual-core 1.86 GHz) | 2 GB DDR2 RAM | Seagate 500 GB HDD | Gigabit Ethernet |

*Table 4.1 – Client and server hardware specifications for PXFS testing.*

operation is a read or write and the referenced block of data (512 KB) is randomly selected, as well. When a thread finishes its sequence of I/O operations, it terminates. Once all threads complete, the program calculates each thread's read and write throughput and then terminates. A similar PXFS application was developed, however changes were made to account for the asynchronous nature of its interface. Specifically, a thread submits an asynchronous I/O operation specifying a continuation for recursively submitting more I/O operations. The main thread suspends and awaits the completion of this chain of asynchronous operations. This functionality was required to enforce the requirement that I/O operations may not be submitted before previous operations have completed.

The average read and write throughput results for case 1 using PXFS and OrangeFS are given in Figure 4.1 and Figure 4.2, respectively. It is clear from these graphs that the potential read and write throughput for PXFS slightly exceeds that of the original OrangeFS file system, regardless of the number of application threads used. This was not anticipated, as the OrangeFS I/O library is much more established, in the sense that it has been around for a long time and has been optimized for high-performance. Most likely, PXFS is able to attain higher throughput performance due to the lightweight and simplistic nature of the AIO common interface, which simply submits I/O operations to the underlying file system and awaits their completion – all optimizations are concealed from the interface and handled internally by the file system (the state machines and job interface, specifically). These results prove that PXFS is capable of providing high-performance, high-concurrency I/O operations for this test case. However, the slight

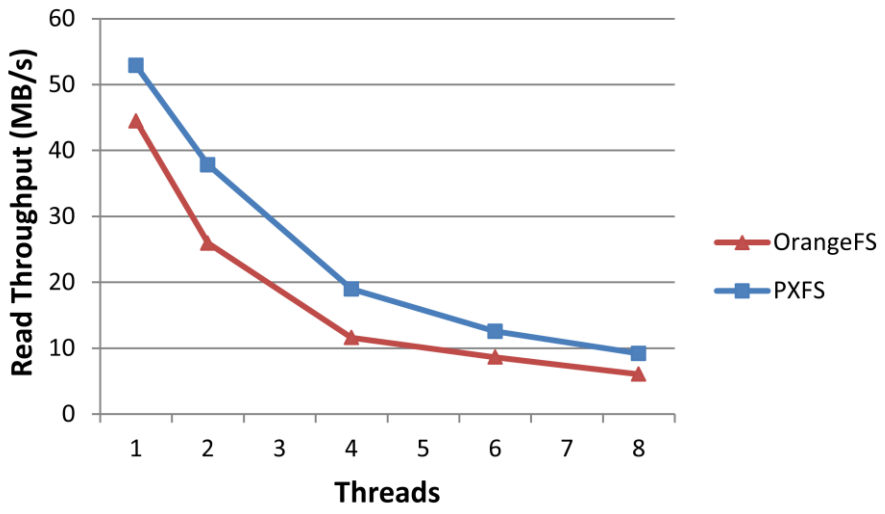


*Figure 4.1 – Average read throughput for case 1 using OrangeFS and PXFS.*

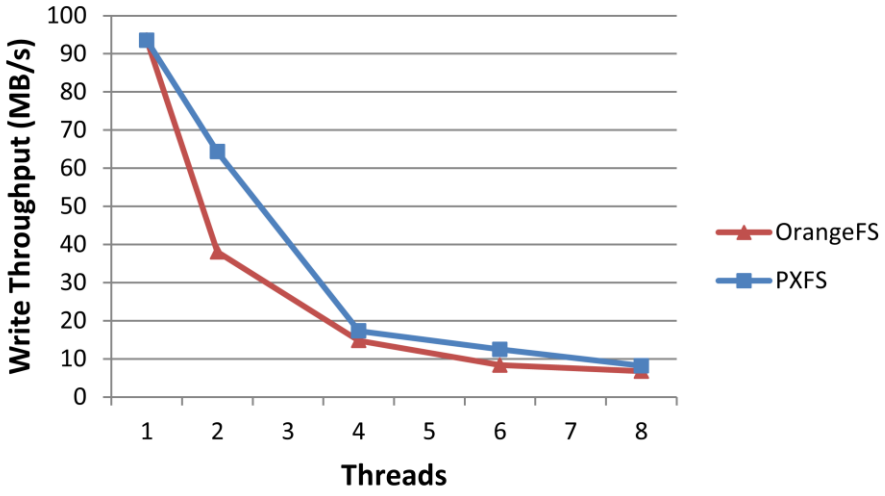


*Figure 4.2 – Average write throughput for case 1 using OrangeFS and PXFS.*

throughput increase possible with PXFS does not justify rewriting all applications using the PXFS I/O library, as applications that perform asynchronous I/O are inherently more complex than their synchronous counterparts, since they must specify callback functions and often require synchronization mechanisms to be notified of I/O completions.

### 4.1.2 Sequential Read/Write (Case 2)

In test case 2, OrangeFS and PXFS applications were developed in which a single thread reads or writes a 1 GB file one block at a time. The time required to finish this operation was calculated and used to determine which file system was able to obtain the highest aggregate throughput. The OrangeFS application is very similar to the one developed for case 1, except the sequence of reads or writes is sequential, not random. Clearly, since the OrangeFS I/O library utilizes synchronous file operations, the application thread must wait for the completion of previous I/O operations before submitting additional ones, just as in case 1. However, since the PXFS I/O library is asynchronous and no overlap exists between the sequence of I/O operations, the PXFS application is able to submit all I/O operations at once. After all asynchronous I/O operations have been submitted, the PXFS application thread suspends execution until a specified continuation function notifies it that all operations have completed.

Figures 4.3 and 4.4 illustrate the aggregate read and write throughput results achieved by both file systems using varying block sizes. It is clear from these graphs that the performance of PXFS easily exceeds that of OrangeFS for reads and writes across all block sizes. This makes sense, because the PXFS application is able to submit all I/O operations at once, while the OrangeFS application must submit one operation at a time. Essentially, the PXFS AIO subsystem may run as many simultaneous operations as possible and queues the remaining operations. As operations complete, the AIO subsystem runs new operations, achieving maximum utilization of the underlying file system. Conversely, the OrangeFS application may only submit one I/O operation at a time and cannot submit further I/O operations until the current one has been returned to the application thread. This does not allow for high file system utilization, because each I/O operation must traverse the file system stack before a new operation may be submitted, due to the synchronous nature of the library. The only way to achieve higher

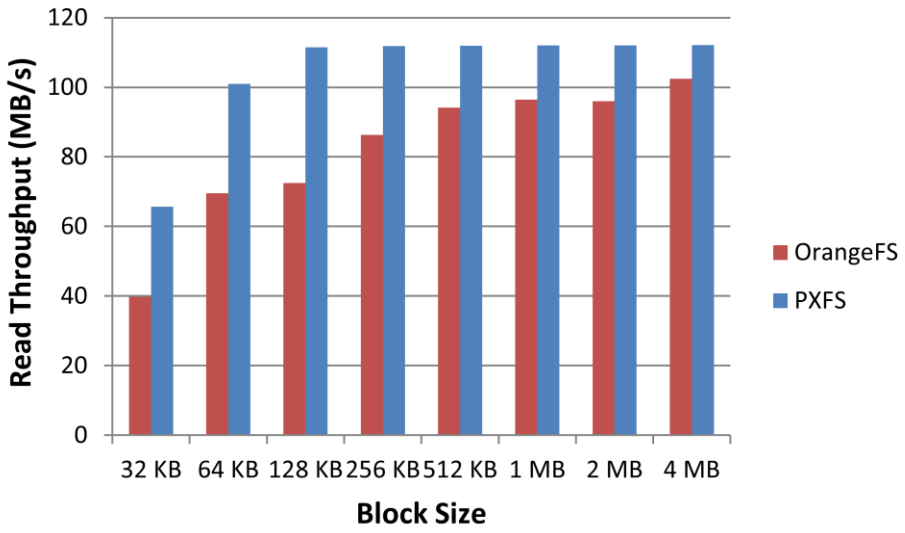


*Figure 4.3 – Aggregate read throughput results for case 2.*

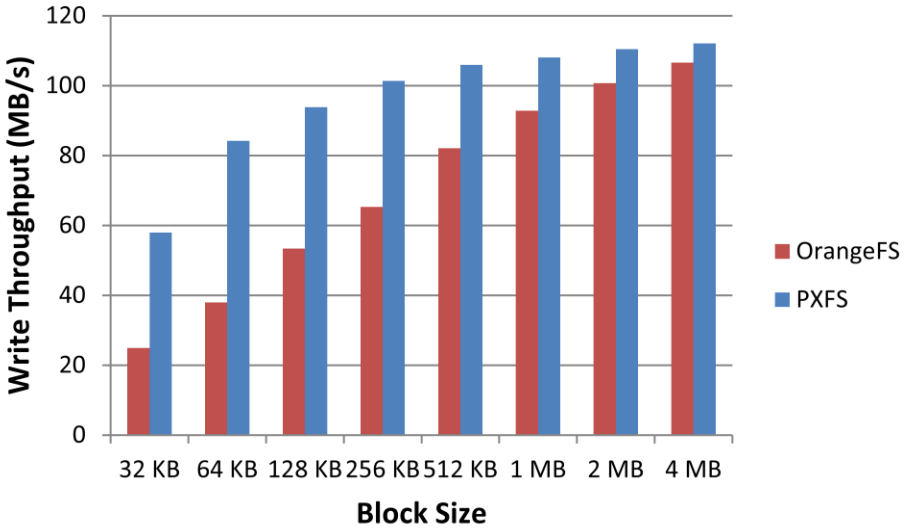


*Figure 4.4 – Aggregate write throughput results for case 2.*

performance with OrangeFS is to use multiple application threads or modify the file system to define an operation for submitting a list of I/O operations at once.

### 4.1.3 Sequential Read/Write with Computation Overlap (Case 3)

The goal of test case 3 was to analyze the ability of each file system at overlapping I/O with computation, an important requirement for satisfying the ParalleX I/O model. The ability to overlap I/O with computation is important as it allows applications to continue performing meaningful computation while waiting for potentially high latency I/O operations to complete, with negligible effects on the I/O throughput. Obviously, multiple processing cores are required to overlap computation with I/O – one or more cores to perform computation and at least one core dedicated to performing I/O. For test case 3, the same OrangeFS application from case 2 was modified

to perform a constant amount of computation after the submission of each I/O operation. Again, we simply calculate the aggregate throughput for reading or writing a 1 GB file, except we include simulated computation. Similarly, a PXFS application was developed which reads or writes a 1 GB file by sequentially submitting asynchronous operations referencing each block, simulating computation, and waiting for completion of the submitted operation before continuing. Unlike case 2, the PXFS application may not submit a list of operations but must submit operations one at a time. Note that the processor affinities for the PXFS application thread and the AIO progress thread were set to different cores to obtain the most accurate results.

Case 3 results for OrangeFS reads and writes are given in Figures 4.5 and 4.6. Also, Table 4.2 gives the percentage decrease in I/O throughput between an OrangeFS application with no computation overlap (like case 1 or case 2) and an application which performs computation after I/O completions. As expected, OrangeFS aggregate I/O throughput clearly suffers when overlapping computation with I/O. This can be attributed to the synchronous nature of the OrangeFS I/O library. Specifically, OrangeFS application threads are blocked while waiting for the completion of an I/O operation, and thus, cannot perform any other computation before the operation completes. Computation may only be overlapped with I/O by spawning additional threads, which is burdensome to programmers and leads to complex applications.

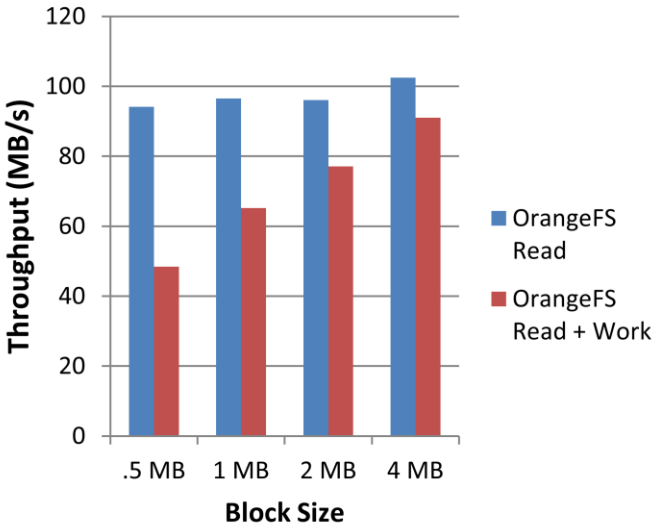


*Figure 4.5 – OrangeFS aggregate read throughput results for case 3.*

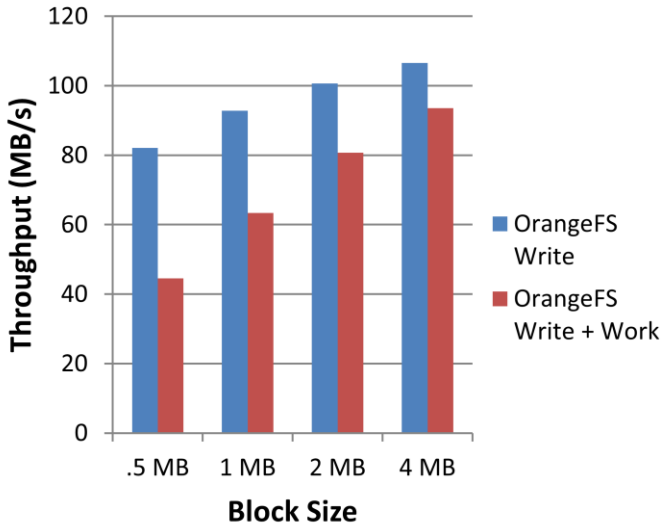


*Figure 4.6 – OrangeFS aggregate write throughput results for case 3.*

| Block Size (MB) | Read Throughput (MB/s) | Read + Work Throughput (MB/s) | Percentage Decrease | | Write Throughput (MB/s) | Write + Work Throughput (MB/s) | Percentage Decrease |
|---|---|---|---|---|---|---|---|
| .5 | 94.17 | 48.41 | 48.6 % | | 86.06 | 44.51 | 48.3 % |
| 1 | 96.50 | 65.17 | 32.5 % | | 92.80 | 63.34 | 31.7 % |
| 2 | 96.02 | 77.13 | 19.7 % | | 100.67 | 80.74 | 19.8 % |
| 4 | 102.43 | 91.05 | 11.1 % | | 106.60 | 93.58 | 12.2 % |

*Table 4.2 – OrangeFS aggregate throughput percentage decrease results for case 3.*

The case 3 results for a PXFS application are given below in Figures 4.7 and 4.8, and percentage decrease values like that of Table 4.2 are given in Table 4.3. It is obvious from these results that PXFS is much more suitable for writing applications which seek to overlap meaningful computation with I/O operations. The read and write aggregate throughput graphs clearly show that a PXFS application is able to obtain nearly the same performance regardless of whether computation is overlapped with I/O operations. This confirms the PXFS I/O library effectively offloads I/O operations from application

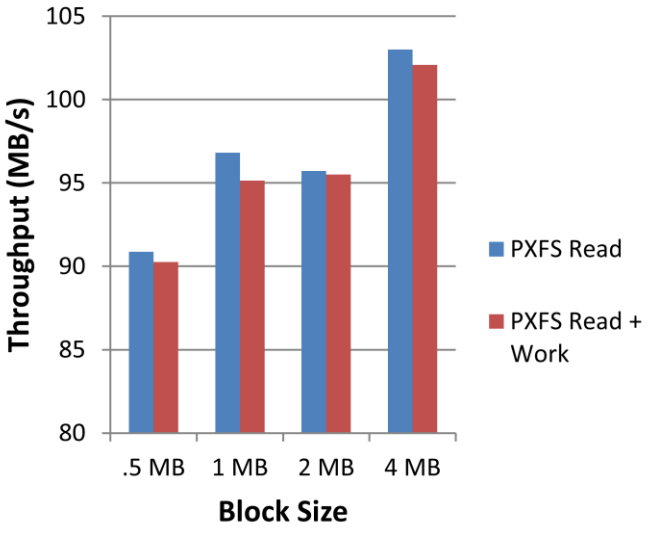


*Figure 4.7 – PXFS aggregate read throughput results for case 3.*

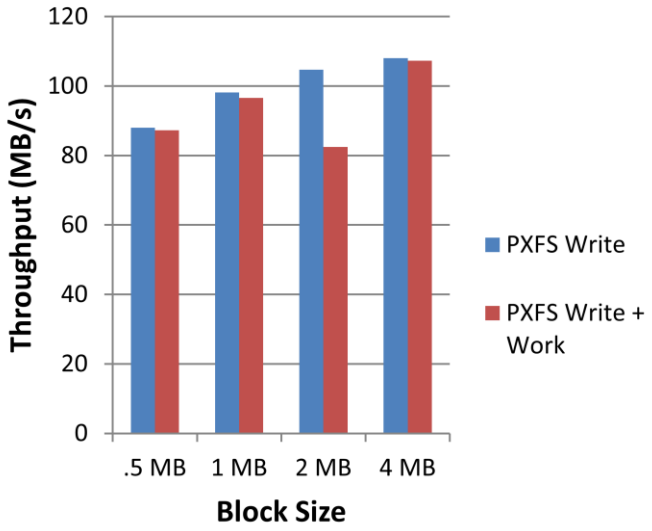


*Figure 4.8 – PXFS aggregate write throughput results for case 3.*

threads and efficiently notifies these applications of the completion of these operations. This is extremely beneficial as PXFS applications can achieve maximum utilization of the file system without sacrificing the utilization of application threads.

To further illustrate the benefits of using PXFS for overlapping computation and I/O, Figure 4.9 shows the effect of computation time on the apparent I/O time as seen by an application thread in OrangeFS and PXFS. With OrangeFS, the apparent length of an

| Block Size (MB) | Read Throughput (MB/s) | Read + Work Throughput (MB/s) | Percentage Decrease | | Write Throughput (MB/s) | Write + Work Throughput (MB/s) | Percentage Decrease |
|---|---|---|---|---|---|---|---|
| .5 | 90.87 | 90.25 | .7% | | 88.03 | 87.28 | .9 % |
| 1 | 96.81 | 95.15 | 1.7 % | | 98.13 | 96.56 | 1.6 % |
| 2 | 95.72 | 95.51 | .2% | | 104.70 | 82.44 | 21.2 % |
| 4 | 103.00 | 102.09 | .9 % | | 108.07 | 107.34 | .7 % |

*Table 4.3 – PXFS aggregate throughput percentage decrease results for case 3.*

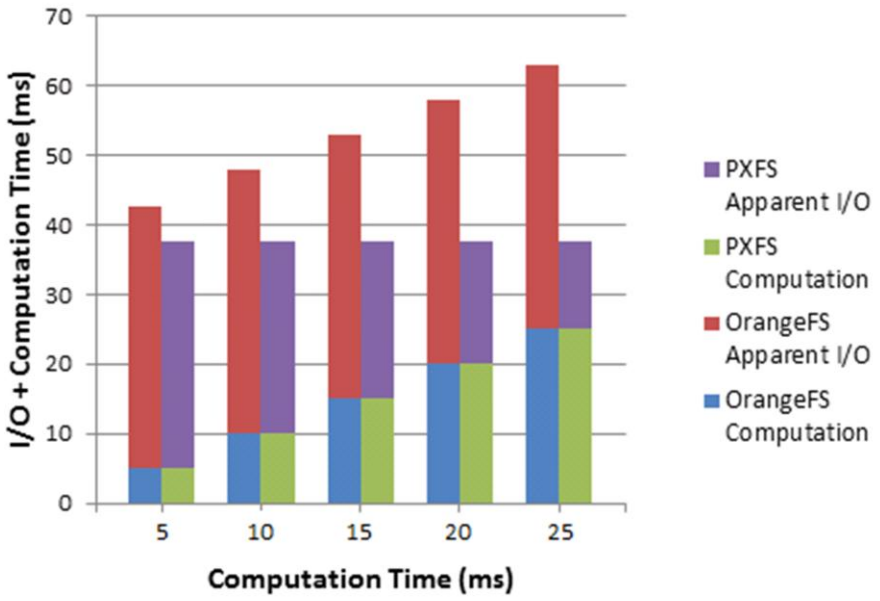


*Figure 4.9 – Effect of computation time on apparent I/O time in PXFS and OrangeFS.*

I/O operation remains constant independent of the computation time, as expected. However, with PXFS, as the overlapping computation time of an application increases, the apparent length of the I/O operation decreases. If the computation length is long enough, the apparent length of the I/O operation may actually be completely hidden from the application. Obviously, the I/O operation requires roughly the same amount of time using PXFS as OrangeFS, but the PXFS AIO subsystem effectively masks this latency by offloading the operation to a thread dedicated to performing I/O (the progress thread).

## 4.2 Metadata Results

While the results of file reads and writes seem to support the feasibility of the PXFS file system, it is also important to consider the potential performance of metadata

operations. Many parallel applications not only read and write large amounts of data, but also create, update, and remove potentially large numbers of files and directories. Thus, poor metadata performance can drastically reduce the overall performance of many parallel applications. OrangeFS and PXFS applications were developed which perform a sequence of metadata operations to determine the comparative performance of each file system. The metadata operations tested include: file create, file open, file close, file truncate, file stat, file remove, directory create, and directory remove.

The average time to complete each metadata operation for both file systems is given below in Table 4.4, as well as the percentage decrease in operation time from the PXFS library to the OrangeFS library. These results indicate minimal performance differences between the two file systems, with OrangeFS offering better performance for half of the operations (file create, file open, file close, and file stat) and PXFS offering better performance for the other half (file truncate, file remove, directory create, and directory remove). OrangeFS offers particularly high percentage increases for file opens and file stats, but there is only a negligible 70 microsecond and 180 microsecond difference for these operations, respectively, compared to PXFS. We conjecture that OrangeFS likely offers higher metadata performance for metadata operations whose minimal computational complexity suffer from the overhead of the high-level state machines used in PXFS.

| Metadata Operation | OrangeFS Operation Time (ms) | PXFS Operation Time (ms) | Percentage Decrease |
|---|---|---|---|
| File Create | 3.75 | 3.76 | -0.27 |
| File Open | 0.21 | 0.28 | -33.33 |
| File Close | 6.59 | 6.82 | -3.49 |
| File Truncate | 2.59 | 2.58 | 0.39 |
| File Stat | 0.93 | 1.11 | -19.35 |
| File Remove | 3.26 | 3.22 | 1.23 |
| Directory Create | 3.39 | 3.19 | 5.9 |
| Directory Remove | 3.32 | 2.93 | 11.75 |

*Table 4.4 – OrangeFS and PXFS file metadata performance comparison.*

## 4.3 File Operation Blocking Time Results

Another important requirement of the ParalleX I/O model is the minimization of the blocking time necessary for submitting file operations to the underlying file system. Lower blocking times mean higher computational efficiency for applications, as application threads spend less time waiting for the file system to relinquish control of processing resources. Ideally, the blocking time required for an asynchronous file system operation should remain constant independent of the type of request (I/O or metadata), the size of an I/O request, and the number of simultaneous threads accessing the file system. To determine the blocking characteristics of PXFS we simply measured the average blocking time for a range of operations and compared them to the average blocking times achieved using OrangeFS.

Figure 4.10 shows the comparative blocking times of file read operations using PXFS and OrangeFS. Similarly, Figure 4.11 shows the comparative blocking times of file writes using both file systems. These graphs clearly illustrate the disparity of the blocking times between each file system, with OrangeFS blocking times growing proportional to the block size of the I/O operation and PXFS blocking times remaining constant independent of the referenced block size. This was expected, as OrangeFS blocking times are directly related to the time required to manage the transfer of file data to or from the file server. On the other hand, PXFS blocking times depend solely on the time required to copy functional parameters, obtain a mutex lock, and add the file operation to the AIO subsystem waiting queue. The obtained results indicate the average blocking time for a PXFS file operation is consistently less than a microsecond, orders of magnitude less than that of an identical OrangeFS file operation. Also, since the critical section of submitting a PXFS operation only involves adding the operation to the tail of the waiting queue, the

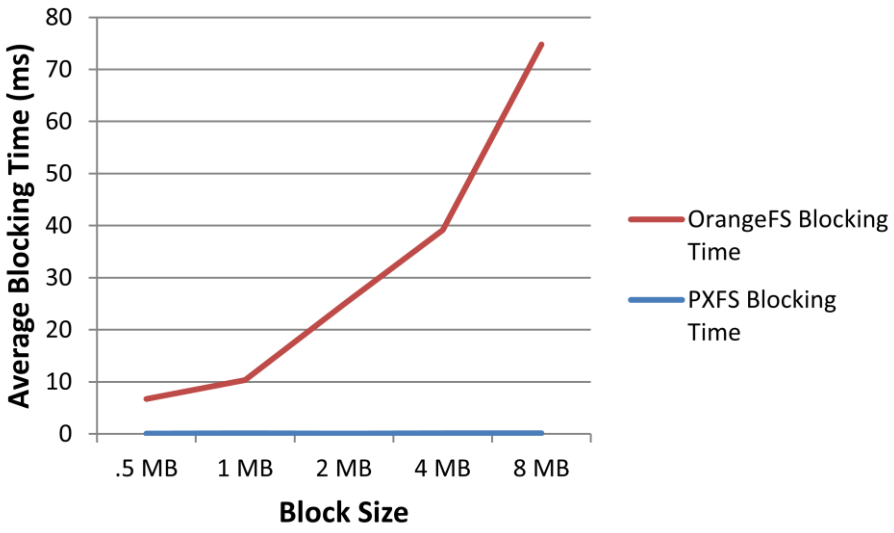


*Figure 4.10 – OrangeFS and PXFS blocking times for file reads.*

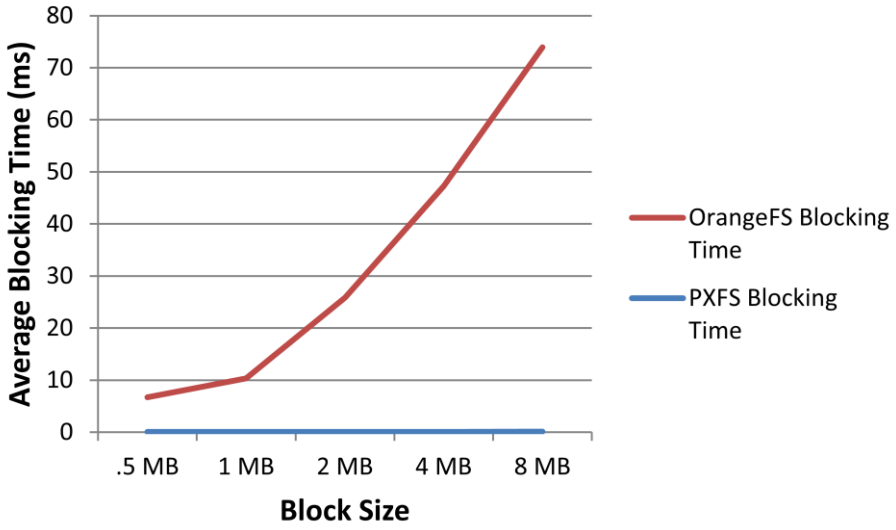


*Figure 4.11 – OrangeFS and PXFS blocking times for file writes.*

blocking time exhibits little variance, regardless of the number of application threads accessing the file system. Thus, it can be seen that PXFS I/O library definitively satisfies the blocking requirements of a ParalleX application, allowing application threads to achieve maximum processor utilization.

## 4.4 Effect of PXFS Continuation Complexity

Before presenting any conclusions, it is important to consider the unobvious effect high complexity continuation functions have on the overall performance of the file system. Since the AIO subsystem is currently designed to execute given continuations using the progress thread, the performance of the file system will suffer if these continuations consume too many compute cycles. To confirm this, a series of tests were run in which I/O operations specified continuation functions that required increasing

amounts of execution time. These results were then analyzed to determine the extent of file system performance degradation caused by complex continuation functions.

The resulting performance of the PXFS file system when subjected to varying amounts of continuation complexity is given below in Figure 4.12. As projected, file system performance degrades proportional to the amount of computation performed in specified continuation functions. This suggests programmers should explicitly design PXFS applications to utilize continuations with minimal computational requirements. However, this requirement can severely limit the expressability of PXFS applications and place unnecessary burden on application developers. The most sensible way to avoid this problem is to implement a thread manager for controlling the execution of user continuations, such that continuations of arbitrary complexity may be specified with limited impact on file system performance. Since the ParalleX execution model defines such a thread manager, this requirement can be easily satisfied when PXFS is fully

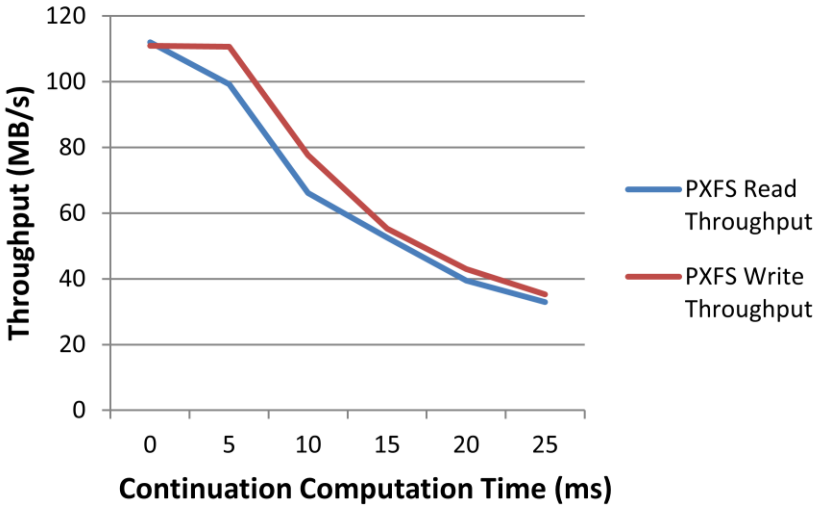


*Figure 4.12 – PXFS throughput versus continuation computational complexity.*

integrated with ParalleX. As mentioned earlier, this is our first PXFS prototype and is intended to simply determine the feasability of our file system design – further performance enhancements may be realized (such as the utilization of ParalleX threads and thread management) in future PXFS implementations.

# Chapter 5

# Conclusion

We have identified the necessary characteristics for an I/O subsystem satisfying the I/O model required by ParalleX, a proposed exascale parallel execution model. These characteristics were integrated into the OrangeFS file system, resulting in our first implementation of the ParalleX File System (PXFS). Performance results obtained using PXFS confirmed our initial hypothesis that an existing parallel file system may be easily modified to meet the semantics of future exascale execution models, which attempt to achieve exaFlops of performance in part by addressing the inherent inefficiencies of I/O subsystems by explicitly masking the high latency of remote file system operations.

Specifically, we have shown that our initial PXFS prototype offers increased file read and write performance relative to OrangeFS, an already well-established high performance file system. Also, PXFS provides comparable performance to OrangeFS for metadata operations, an important requirement for applications which create or modify a large number of files or directories simultaneously. These results suggest that PXFS is a suitable I/O subsystem for exascale workloads, which demand high-concurrency, high-performance management of file data and metadata.

More importantly, our results prove that PXFS directly supports the advanced semantics required by exascale applications, and in particular ParalleX applications. Specifically, PXFS offers a wide range of asynchronous file operations within a well-defined user interface. These asynchronous operations allow computation to be easily and efficiently overlapped with necessary file system communication with limited programmer intervention. The utilization of user-specified continuations allows for the execution of arbitrary functions immediately upon the completion of file system operations, leading to much more expressive parallel applications. Also, the low overhead incurred by the PXFS I/O library limits blocking time as seen by ParalleX threads, allowing for higher utilization of processing resources.

## 5.1 Future Work

While the results obtained from our first implementation of PXFS support the feasibility of our design, there is still room for further research and improvements. As mentioned previously, PXFS performance could benefit from a tighter integration with the ParalleX execution model. In particular, PXFS may be redesigned to explicitly utilize ParalleX threads and parcels. The ParalleX thread manager may allow application threads and the PXFS progress thread to be scheduled in a more efficient and cooperative manner. A more intelligent thread manager could help alleviate the performance issues that arise from continuations with high computational complexity, as discussed in Section 4.4. Also, modifying PXFS to directly utilize ParalleX parcels could lead to further performance improvements, as they may be used by the file servers to deliver file data

directly to ParalleX threads or to seamlessly migrate stored data between servers. Ultimately, our goal is to offer the same execution model semantics for objects stored on disk as in-memory objects, effectively unifying the namespace of local memory and remote storage devices in exascale parallel systems.

# Bibliography


[1] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, P. Sadayappan, "Scalable I/O Forwarding Framework for High-Performance Computing Systems," in *Cluster '09*, New Orleans, LA, 2009, pp. 1-10.

[2] S. Amarasinghe et al., "Exascale Programming Challenges," in *Report 2011 Workshop Exascale Programming Challenges*, Marina del Ray, CA, 2011, pp. 10-17.

[3] M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "An Application Driven Analysis of the ParalleX Execution Model," Louisiana State University, Baton Rouge, LA, Technical report, Sept. 2011.

[4] H.G. Baker Jr. and C. Hewitt, "The Incremental Garbage Collection of Processes," in *Proc. 1977 Symp. Artificial Intelligence and Programming Languages*, Rochester, NY, 1977, pp. 55-59.

[5] Berkeley Unified Parallel C (UPC) Project [Online]. (Jan. 2013). Available: http://upc.lbl.gov.

[6] D. Bonachea and J. Jeong, "GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages," *CS258 Parallel Computer Architecture Project*, 2002.

[7] P.H. Carns, W.B. Ligon III, R.B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *Proc. Extreme Linux Track: 4th Annu. Linux Showcase and Conf.*, Atlanta, GA, 2000, pp. 317-327.

[8] P.H. Carns, "Achieving Scalability in Parallel File Systems," Ph.D. dissertation, ECE Dept., Clemson Univ., Clemson, SC, 2005.

[9] D. Chavarría-Miranda, Z. Huang, and Y. Chen, "High-Performance Computing (HPC): Application & Use in the Power Grid," in *Power and Energy Society General Meeting*, Richmond, WA, 2012, pp. 1-7.

[10] Y. Chen, "Towards Scalable I/O Architecture for Exascale Systems," in *Proc. 2011 ACM Int. Workshop on Many Task Computing on Grids and Supercomputers*, Seattle, WA, 2011, pp. 43-48.

[11] Y. Chushak, B. Foy, and J. Frazier, "Biomolecular Network Simulator: Software for Stochastic Simulation of Cellular Biological Processes," in *Proc. High Performance Computing Symp.*, Norfolk, VA, 2007, pp. 345-349.

[12] Cray Inc., Seattle, WA. Chapel Specification 0.92 [Online], 2012. Available: http://chapel.cray.com/spec/spec-0.92.pdf.

[13] I. Foster, "Parallel Computers and Computation," in *Designing and Building Parallel Programs*, Version 1.3, ch. 1, sec. 3 [Online]. Available: http://www.mcs.anl.gov/~itf/dbpp/text/book.html.

[14] G.R. Gao. (2012). Introduction of Parallel Program Execution and Architecture Models [Online]. Available: http://www.capsl.udel.edu/courses/eleg652/2012/slides/02_pxm.pdf.

[15] General Parallel File System (GPFS) Wiki [Online], 2013, Available: http://www.ibm.com/developerworks/connect/GPFS.

[16] L.V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based On C++," in *Proc. 8th Annu. Conf. on Object-oriented Programming Systems, Languages, and Applications*, Washington, D.C., 1993, pp. 91-108.

[17] C.E. Leiserson, "The Cilk++ Concurrency Platform," in *Proc. 46th Annual Design Automation Conf.*, San Francisco, CA, 2009, pp. 522-527.

[18] Lustre File System Operations Manual [Online], Version 2.0, 2011. Available: http://wiki.lustre.org/manual/LustreManual20_HTML/index.html.

[19] MPI: A Message-Passing Interface Standard, Version 2.2, 2009.

[20] OpenMP Application Programming Interface, Version 3.0, 2008.

[21] OrangeFS Developer's Guide [Online]. (Dec. 2012). Available: http://www.orangefs.org/documentation/releases/current/doc/pvfs2-guide/pvfs2-guide.php.

[22] Orange File System (OrangeFS) Project [Online]. (Dec. 2012). Available: http://www.orangefs.org.

[23] POSIX.1-2008, Standard for Information Technology – Portable Operating System Interface (POSIX), IEEE Standard 1003.1, 2008.

[24] Parallel Virtual File System (PVFS) Project [Online]. (Dec. 2012). Available: http://www.pvfs.org.

[25] J. Reinders, "Why Threading Building Blocks?" in *Intel Threading Building Blocks*, 1st edition, O'Reilly Media, 2007.

[26] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. The X10 language specification. Technical report, IBM T.J. Watson Research Center, 2008.

[27] P. Schlatter, J. Malm, G. Brethouwer, A.V. Johansson, and D.S. Henningson, "Large-scale Simulations of Turbulence: HPC and Numerical Experiments," in *7th IEEE Conf. e-Science*, Stockholm, Sweden, 2011, pp. 319-224.

[28] Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*. IETF RFC 1094, Mar. 1989.

[29] A. Tabbal, M. Anderson, M. Brodowicz, H. Kaiser, and T. Sterling, "Preliminary Design Examination of the ParalleX System from a Software and Hardware Perspective," *SIGMETRICS Performance Evaluation Review*, 38:4, Mar. 2011.

[30] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *Proc. 6th Workshop I/O in Parallel and Distributed Systems*, Atlanta, GA, 1999, pp. 23-32.

[31] Titanium Project [Online]. (Jan. 2013). Available: http://titanium.cs.berkeley.edu/.

[32] Top500.org (2012, Jun.). Top 500 Supercomputing Sites [Online]. Available: http://www.top500.org/lists/2012/11.

[33] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proc. 19th Int. Symp. on Computer Architecture*, Gold Coast, Australia, 1992, pp. 256-267.

[34] D.W. Wall, "Messages as Active Agents," in *ACM Symp. Principles of Programming Languages (POPL)*, Albuquerque, NM, 1982, pp. 256-267.

[35] P. Widener, M. Payne, P. Bridges, M. Wolf, H. Abbasi, S. McManus, K. Schwan, "Exploiting Latent I/O Asynchrony in Petascale Science Applications," in *Proc. 2009 Int. Conf. on Parallel Processing Workshops*, Vienna, Austria, 2009, pp. 105-112.

[36] K. Yelick et al., “Productivity and Performance Using Partitioned Global Address Space Languages,” in *PASCO ’07: Proc. 2007 Int. Workshop on Parallel Symbolic Computation*, London, Ontario, Canada, 2007, pp. 24-32.

[37] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G.R. Gao, “Position Paper: Using a “Codelet” Program Execution Model for Exascale Machines,” in *EXADAPT '11: Proc. 1st Int. Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, San Jose, CA, 2011, pp. 64-69.