2014

A Persistent Storage Model for Extreme Computing

Shuangyang Yang
Louisiana State University and Agricultural and Mechanical College

A PERSISTENT STORAGE MODEL FOR EXTREME COMPUTING

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Shuangyang Yang
B.S., Zhejiang University, 2006
M.S., University of Dayton, 2008
December 2014

Dedicated to my wife Miao Yu and our daughter Emily.

Acknowledgments

This dissertation would not be possible without several contributions. It is a great pleasure to thank Dr. Hartmut Kaiser @ Louisiana State University, Dr. Walter B. Ligon III @ Clemson University and Dr. Maciej Brodowicz @ Indiana University for their ongoing help and support.

It is a pleasure also to thank Dr. Bijaya B. Karki, Dr. Konstantin Busch, Dr. Supratik Mukhopadhyay at Louisiana State University for being my committee members and Dr. Kenneth Fasching-Varner at Louisiana State University for being my dean's representative.

A special thanks to Dr. Qiwen Zhan @ University of Dayton and Dr. Peter E. Powers @ University of Dayton (1965-2014) for their kindly support in both research and life all these years.

A warm thanks to the STE||AR group @ Louisiana State University and the Orange file system group @ Clemson University and Omnibond LLC for all the help and great working environment.

This work was supported by NSF, Clemson University, the Center for Computation & Technology at Louisiana State University, and Center for Research in Extreme Scale Technologies at Indiana University. I would like to thank them for their continued efforts to fund projects that can bring the next generation research in high performance computing.

Final thanks goes to my parents, my parents-in-law, my wife and our daughter Emily, for their great support throughout the journey.

Table of Contents

Acknowledgments iv

List of Tables vii

List of Figures viii

Abstract xiii

Chapter

1 Introduction 1
- 1.1 Research Objective 2
  - 1.1.1 Goal 2
  - 1.1.2 Hypothesis 3
  - 1.1.3 Objective 3
- 1.2 Technical Strategy 4
- 1.3 Main Contributions 5
- 1.4 Dissertation Outline 7

2 Background 8
- 2.1 State of Parallel File System 8
  - 2.1.1 Overview 8
  - 2.1.2 PVFS and OrangeFS 9
- 2.2 State of High Performance Computing System 13
  - 2.2.1 MPI - Message Passing Interface 13
  - 2.2.2 ParalleX - an Execution Model for Extreme Scale 15
  - 2.2.3 HPX - An Implementation of ParalleX 19
- 2.3 Linda: Coordination and Communication Model 23

3 PXFS: A Persistent Storage Model for Extreme Scale 25
- 3.1 PXFS Design 25
- 3.2 Asynchronous I/O Operations 26
  - 3.2.1 OrangeFS Asynchronous I/O Interface 26
  - 3.2.2 HPX Asynchronous I/O Class APIs 30
- 3.3 Active TupleSpace Model 35
  - 3.3.1 Introduction 35
  - 3.3.2 HPX TupleSpace 35
  - 3.3.3 The Implementation 37
  - 3.3.4 Use Case: Micro Check-pointing 39
- 3.4 Performance Evaluation 42
  - 3.4.1 Method 42
  - 3.4.2 Synchronous I/O v.s. Asynchronous I/O 43
  - 3.4.3 Asynchronous I/O Throughputs of HPX File Classes 54

4 Distributed Directory Service in Orange File System 91
4.1 Background 92
4.2 Design and Implementation 93
4.2.1 OrangeFS Structure 93
4.2.2 Scalable Distributed Directory Design 96
4.3 Performance Evaluation 100
4.3.1 Experiment Setup 100
4.3.2 Results and Analysis 100
4.4 Chapter Conclusions 101

5 Conclusions and Future Work 105

References 107

Vita 113

List of Tables

2.1 Comparison of the message passing computing system and virtual shared memory computing system. 13

List of Figures

2.1 OrangeFS System Architecture Diagram. 12

2.2 Architecture of the HPX runtime system. HPX implements the supporting functionality for most of the elements needed for the ParalleX model: Parcels (parcel-port and parcel-handlers), HPX-threads (thread-manager), LCOs, AGAS, HPX-processes, performance counters and a means of integrating application specific components. 20

3.1 Design of PXFS storage component as a part of HPX runtime. 25

3.2 Prototype for the OrangeFS AIO write operation. 26

3.3 Data flow diagram illustrating an example I/O operation in the AIO library. Thread 1 submits a `pxfs_write` operation with a continuation which spawns Thread 2. 27

3.4 An example code to show the integration of HPX and OrangeFS AIO interface to make an asynchronous `write` call in HPX applications. 29

3.5 This code example shows the asynchronous file oriented API exposed by the implementation of the **local_file** class presented persistent storage model. The **local_file** class is responsible to operate on local file systems using the HPX asynchronous interfaces and C++ standard file I/O libraries. 31

3.6 This code example shows the asynchronous file oriented API exposed by the implementation of the **orangefs_file** class presented persistent storage model. The **orangefs_file** class is responsible to operate on OranegeFS using the HPX asynchronous interface and OrangeFS synchronous user interfaces. 32

3.7 This code example shows the asynchronous file oriented API exposed by the implementation of the **pxfs_file** class presented persistent storage model. The **pxfs_file** class is responsible to operate on OranegeFS using the OrangeFS AIO asynchronous user interfaces. 33

3.8 This code example demonstrates an exemplar use of the file oriented **pxfs_file** API to asynchronously create a new file and to write some chunks of data to it. The I/O operation is performed fully overlapping any other work which needs to be performed before synchronizing with the result of the I/O. 34

3.9 Diagram of HPX Tuple structure. HPX tuple is composed of an ordered list of any object, which can hold any type of object. 37

3.10 Diagram of HPX TupleSpace component. 38

3.11 This code example demonstrates an exemplar use of the TupleSpace and the three operations. There are synchronous and asynchronous tuplespace operations. 39

3.12 This code example demonstrates an exemplar use of the TupleSpace serialization interfaces. 40

3.13 Use case of check-pointing of HPX application using the active TupleSpace. . . . 41

3.14 I/O performance of file test case `w-5f-100x64KB` between PXFS and direct interface on HPX and OrangeFS. 45

3.15 I/O performance of file test case `r-5f-100x64KB` between PXFS and direct interface on HPX and OrangeFS. 46

3.16 I/O performance of file test case `w-5f-100x1MB` between PXFS and direct interface on HPX and OrangeFS. 47

3.17 I/O performance of file test case `r-5f-100x1MB` between PXFS and direct interface on HPX and OrangeFS. 48

3.18 I/O performance of file test case `w-5f-1x100MB` between PXFS and direct interface on HPX and OrangeFS. 49

3.19 I/O performance of file test case `r-5f-1x100MB` between PXFS and direct interface on HPX and OrangeFS. 50

3.20 I/O performance of file test case `w-100f-1x64KB` between PXFS and direct interface on HPX and OrangeFS. 51

3.21 I/O performance of file test case `r-100f-1x64KB` between PXFS and direct interface on HPX and OrangeFS. 52

3.22 I/O performance of all four file test cases between HPX+PXFS module on OrangeFS and HPX on NFS. 53

3.23 I/O performance of file test case `w-50f-100x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 56

3.24 I/O performance of file test case `r-50f-100x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 57

3.25 I/O performance of file test case `w-50f-10x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 58

3.26 I/O performance of file test case `r-50f-10x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 59

3.27 I/O performance of file test case `w-5f-1x100MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 60

3.28 I/O performance of file test case `r-5f-1x100MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 61

3.29 I/O performance of file test case `w-5f-100x1MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 62

3.30 I/O performance of file test case `r-5f-100x1MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces. 63

3.31 HPX local-file asynchronous I/O performance of test case `w-50f-100x64KB` with different number of threads on client nodes. 65

3.32 HPX local-file asynchronous I/O performance of test case `r-50f-100x64KB` with different number of threads on client nodes. 66

3.33 HPX local-file asynchronous I/O performance of test case `w-50f-10x64KB` with different number of threads on client nodes. 67

3.34 HPX local-file asynchronous I/O performance of test case `r-50f-10x64KB` with different number of threads on client nodes. 68

3.35 HPX local-file asynchronous I/O performance of test case `w-5f-1x100MB` with different number of threads on client nodes. 69

3.36 HPX local-file asynchronous I/O performance of test case `r-5f-1x100MB` with different number of threads on client nodes. 70

3.37 HPX local-file asynchronous I/O performance of test case `w-5f-100x1MB` with different number of threads on client nodes. 71

3.38 HPX local-file asynchronous I/O performance of test case `r-5f-100x1MB` with different number of threads on client nodes. 72

3.39 HPX orangefs-file asynchronous I/O performance of test case `w-50f-100x64KB` with different number of threads on client nodes. 74

3.40 HPX orangefs-file asynchronous I/O performance of test case `r-50f-100x64KB` with different number of threads on client nodes. 75

3.41 HPX orangefs-file asynchronous I/O performance of test case `w-50f-10x64KB` with different number of threads on client nodes. 76

3.42 HPX orangefs-file asynchronous I/O performance of test case `r-50f-10x64KB` with different number of threads on client nodes. 77

3.43 HPX orangefs-file asynchronous I/O performance of test case `w-5f-1x100MB` with different number of threads on client nodes. 78

3.44 HPX orangefs-file asynchronous I/O performance of test case `r-5f-1x100MB` with different number of threads on client nodes. 79

3.45 HPX orangefs-file asynchronous I/O performance of test case `w-5f-100x1MB` with different number of threads on client nodes. 80

3.46 HPX orangefs-file asynchronous I/O performance of test case `r-5f-100x1MB` with different number of threads on client nodes. 81

3.47 HPX pxfs-file asynchronous I/O performance of test case `w-50f-100x64KB` with different number of threads on client nodes. 82

3.48 HPX pxfs-file asynchronous I/O performance of test case `r-50f-100x64KB` with different number of threads on client nodes. 83

3.49 HPX pxfs-file asynchronous I/O performance of test case `w-50f-10x64KB` with different number of threads on client nodes. 84

3.50 HPX pxfs-file asynchronous I/O performance of test case `r-50f-10x64KB` with different number of threads on client nodes. 85

3.51 HPX pxfs-file asynchronous I/O performance of test case `w-5f-1x100MB` with different number of threads on client nodes. 86

3.52 HPX pxfs-file asynchronous I/O performance of test case `r-5f-1x100MB` with different number of threads on client nodes. 87

3.53 HPX pxfs-file asynchronous I/O performance of test case `w-5f-100x1MB` with different number of threads on client nodes. 88

3.54 HPX pxfs-file asynchronous I/O performance of test case `r-5f-100x1MB` with different number of threads on client nodes. 89

4.1 Illustration of a three-level splitting process in GIGA+. The hash space is divided evenly for each split. 93

4.2 Diagram of structures of OrangeFS objects. A directory metadata object only has one dirdata object, which resides on the same metadata server. Usual file attributes fields are omitted in this diagram. 95

4.3 Diagram of object structures in scalable distributed directory OrangeFS design. A directory metadata object holds a number of dirdata objects across all metadata servers. Dirdata handles array and dirdata bitmap are added on the directory metadata object and the dirdata object. 96

4.4 Illustration of the lookup process in scalable distributed directory on OrangeFS. *N* denotes the total number of dirdata objects. In this example, N=8 and the key starts at 7. 99

4.5 Scale and performance of scalable distributed directory implementation on OrangeFS with different number of OrangeFS servers. The UCAR metarates benchmark is used and the number of clients are selected to be 2x the number of servers. 103

4.6 Scale and performance of 64 servers scalable distributed directory implementation on OrangeFS with different number of clients. 104

Abstract

The continuing technological progress resulted in a dramatic growth in aggregate computational performance of the largest supercomputing systems. Unfortunately, these advances did not translate to the required extent into accompanying I/O systems and little more in terms of architecture or effective access latency. New classes of algorithms developed for massively parallel applications, that gracefully handle the challenges of asynchrony, heavily multi-threaded distributed codes, and message-driven computation, must be matched by similar advances in I/O methods and algorithms to produce a well performing and balanced supercomputing system. This dissertation proposes PXFS, a storage model for persistent objects inspired by the ParalleX model of execution that addresses many of these challenges. The PXFS model is designed to be asynchronous in nature to comply with ParalleX model and proposes an active TupleSpace concept to hold all kinds of metadata/meta-object for either storage objects or runtime objects. The new active TupleSpace can also register ParalleX actions to be triggered under certain tuple operations. An first implementation of PXFS utilizing a well-known Orange parallel file system as its back-end via asynchronous I/O layer and the implementation of TupleSpace component in HPX, the implementation of ParalleX. These details are also described along with the preliminary performance data. A house-made micro benchmark is developed to measure the disk I/O throughput of the PXFS asynchronous interface. The results show perfect scalability and 3x to 20x times speedup of I/O throughput performance comparing to OrangeFS synchronous user interface. Use cases of TupleSpace components are discussed for real-world applications including micro check-pointing. By utilizing TupleSpace in HPX applications for I/O, global barrier can be replaced with fine-grained parallelism to overlap more computation with communication and greatly boost the performance and efficiency. Also the dissertation showcases the distributed directory service in Orange file system which process directory entries in parallel and effectively improves the directory metada operations.

Chapter 1
Introduction

Over the coming decade the performance of the largest computing systems is expected to shift from Terascale and Petascale to Exascale. According to the TOP500 supercomputer list [1] of June 2014, the performance of the top 37 systems breaks the PetaFLOPs barrier. These systems are composed of tens of thousands of cores and nodes running in parallel. For example, the top supercomputer, Tianhe-2, has 16,000 nodes equipped with 3,120,000 computing cores and achieves the performance of 33.86 petaFLOP/s on the Linpack benchmark [1]. The rapid growth in computing cores and high demand for parallelism imposes significant challenges for the parallel runtime and storage system.

Applications employing conventional parallel runtime systems, such as the Communicating Sequential Processes (CSP) [2] execution model as reflected by the Message Passing Interface (MPI) [3] programming model, are getting more and more difficult to make effective use of the ever increasing number of processors to achieve desired scalability and performance. The main limiting factors are:

Starvation : insufficient concurrent work to maintain high utilization of resources,

Latencies : delay of remote resource access and services,

Overheads : work for management of parallel actions and resources on critical path which is not necessary in a sequential variant,

Waiting for contention resolution : delays due to lack of availability of oversubscribed shared resources.

All of these factors (**SLOW**) are difficult to avoid using today's prevalent programming models, and a new computational strategy is required to achieve dramatic increases in performance. The ParalleX execution model [4, 5, 6] is offered as a means of addressing these critical computational requirements. It is striving to expose myriad forms of parallelism, hide system wide latencies, decouple hardware execution resources from executing software tasks to prevent the blocking of processor cores, and to enable runtime dynamic adaptive scheduling to employ real-time system state to resource management decisions.

In the meantime, parallel storage system is expected to handle the input/output (I/O) requests from parallel applications with good performance and great scalability. Many parallel file systems are developed to answer that challenge [7, 8, 9, 10]. Orange File System (OrangeFS) [8, 11, 12] is a production-quality parallel file system designed for use on high end computing (HEC) systems that provides very high performance access to disk storage for parallel applications. However, I/O for Exascale high-end computing (HEC) systems is still hampered by several unfortunate issues, including orders of magnitude slower speed and response time, storage distribution problem, fast growing volume of application data and complex data structures.

The challenges of Exascale computing suggest that not a simple extension of our current model of computation, but rather a new model of computation is needed upon which a new framework for mass storage may be built. It is both prudent and essential to consider a corresponding model of I/O along with the design of next generation computation ecosystem.

1.1 Research Objective

1.1.1 Goal

This dissertation presents PXFS (ParalleX File System), a new persistent storage model aiming for Exascale computing. It will be taking advantage of ParalleX execution model and OrangeFS parallel storage research to extract the maximum of parallelism and performance out of the storage resources. Based on some preliminary work [13, 14], this dissertation, for the first time, presents a novel design of PXFS model and a complete implementation using HPX and OrangeFS interfaces.

To be specific, the goals of PXFS research lie below

Goal 1 : Establish a persistent storage model that utilizes asynchronous interface for I/O operations to hide the latency of I/O operations. And make it seamlessly incorporate into ParalleX model.

Goal 2 : Unify the storage objects and runtime objects under the same namespace. With PXFS, the objects in memory and in storage are treated in the same scope with different corresponding implementations. And all objects can be moved to and restored from storage and memory.

Goal 3 : Better metadata management. Extend the concept of metadata of storage file to runtime objects and hold all information of different forms in one place for any entity to use, either program or user.

1.1.2 Hypothesis

The hypothesis of this dissertation are listed below.

Hypothesis 1 : I/O operations can be made asynchronous to hide I/O latencies with communication and computation.

Hypothesis 2 : Storage items can be viewed as objects and managed under the ParalleX namespace.

Hypothesis 3 : Metadata of storage objects and runtime objects can be merged and processed at the same scope and can help improve the performance and increase parallelism

1.1.3 Objective

The proposed research designs a new persistent storage model which takes advantage of develop experiences of parallel file system and parallel runtime system to explore the maximum of parallelism and performance in Exascale computing era. This model will utilize the asynchronous nature of ParalleX execution model and distributed file system design of OrangeFS to develop an innovative model to enhance parallel I/O performances.

The research provides a new and efficient method to target the upcoming storage challenges in the next decade.

The detailed objectives are list below.

1. Design asynchronous I/O interface with local file system, network file system, parallel file system using HPX asynchronous APIs.

2. Design PXFS asynchronous I/O interface which employs the Orange file system asynchronous I/O APIs and provide HPX interface.

3. Evaluate and compare performances of synchronous/asynchronous network file API, synchronous/asynchronous Orange file API and asynchronous PXFS API.

4. Design and implement HPX active TupleSpace to actively hold and keep all meta information of storage objects and runtime objects.

5. Migrate HPX objects to and from storage system by utilizing the meta information in TupleSpace.

1.2 Technical Strategy

The technical strategy that used to test the hypothesis is composed of the following steps.

Step 1 : Develop the asynchronous API in Orange file system which can register a callback function to be executed automatically when an I/O operation is completed.

Step 2 : Write HPX file classes using HPX asynchronous interface and synchronous file operations of local file system, network file system and Orange file system.

Step 3 : Write PXFS file class using HPX and asynchronous Orange file system APIs.

Step 4 : Design the active ParalleX TupleSpace and implement in HPX.

Step 5 : Enable locality/application movement between memory and storage which utilizes TupleSpace and design other scenarios for TupleSpace.

1.3 Main Contributions

The goal of this research is to create a persistent storage framework for ParalleX execution model with a parallel file system OrangeFS. As such it promises to have high impact on a broad spectrum of applications depending on high performance parallel I/O.

The main contributions of dissertation can be summarized as below:

- Pioneering attempt to address the challenges of parallel runtime and storage system of the next Exascale computing decade.

- Contributed new understanding, concepts, and methods for realizing advanced persistent storage management. It merges two innovative system classes: an advanced parallel file system, OrangeFS, and a unique execution model, ParalleX. The result will be an exploratory vehicle for establishing a new paradigm for mass storage at extreme scale. It will provide an event driven dynamic adaptive computation environment in response to the uncertainty of data access times, related asynchrony and imposed overheads through the embedding of futures-based synchronization.

- Proposed new syntax and semantics for asynchronous I/O in object-oriented parallel runtime system. Developed asynchronous AIO interface in Orange file system which is doing asynchrony on the file bytes level, and designed the asynchronous I/O operation semantics in ParalleX model for object level asynchrony. The work also provide basis to created storage classes for different file systems including network file system, local file system and parallel file systems, which enables asynchronous semantics on different layer in an abstract form.

- Unified the namespace of memory objects and storage objects. Translate traditional file-based storage system into object-based storage entities and include them into the same namespace scope with runtime objects. It will eliminate the division of programming imposed by conventional file systems through the unification of name spaces and their management.

- Introduced the active TupleSpace model for better metadata management. Metadata is becoming more and more important in parallel computing and the TupleSpace idea can provide an active and flexible way to manage metadata from all kinds of objects and entities. It can help the user and program to make smart decisions to improve performance and parallelism.

- Designed and implemented distributed directory service in Orange file system, which distributed directory entries among multiple data servers. This move will help parallelize the directory operations and help improve metadata performances.

- Provided insights and directions on how to target the challenges of efficient I/O in the context of future Exascale systems. The conclusions will help scientists efficiently process an unprecedented amount of data in a science computation and shine the light on computer scientists of a new way to design the next generation storage system for massively parallel runtime systems.

- The research creates outputs that can lead to a thriving continuous work for both parallel file system and parallel runtime systems. The work follows can help shape the next generation computation system in the next decade and can improve productivity in both scientific and industrial fields

- This work of the dissertation has resulted in two full-length published papers: [12] and [15], one tech talk at [13] and one poster at [16].

1.4 Dissertation Outline

The remaining of the dissertation is organized as follows.

Chapter 2 introduces the related work of the parallel file system, the high performance computing system and the Linda coordination model. It provides the background of these research areas and provision the importance of this dissertation.

Chapter 3 details PXFS, the innovative persistent storage model established for extreme computing. It is designed as a HPX component and provides an asynchronous storage interface to the underlying storage system, including OrangeFS file system and network file system. At the meantime, an active TupleSpace is proposed to store the metadata of all metadata of all object entities in one place including the storage objects and runtime objects. It can kill the gap between storage objects and memory objects. A micro benchmark is used to evaluate the performance of PXFS and the result shows much better I/O throughput.

Chapter 4 outlines the design and implementation of distributed directory service in Orange file system. By distributing the directory entries among storage servers, a much higher metadata throughput of directory creation and deletion is achieved.

Chapter 5 draws the conclusion and derives the key contribution of this dissertation. This chapter also points out the research directions that is beyond the scope of the dissertation. It also highlights the specifics on future work to be carried out beyond this research.

Chapter 2
Background

2.1 State of Parallel File System

2.1.1 Overview

The high-performance computing (HPC) community is walking into an new era where top clusters and systems are able to achieve a performance of petaFLOPs (10^{15}) scale easily. The rapid growth in computing power and high demand for parallelism imposes significant challenges for the storage system, which is expected to handle the input/output (I/O) requests from parallel applications with good performance and great scalability.

Many parallel and distributed file systems are developed to answer that challenge[17, 7, 8, 9, 10]. File data are distributed across multiple processing nodes in parallel file systems, each with its own storage resources. In that case, concurrent I/O requests can be spread across several servers to process rather than focusing I/O on a single server [18].

These parallel and distributed file systems can be roughly categorized into four types: commercial parallel file systems, distributed file systems, high performance general-purpose parallel file system and research parallel file systems.

The first type comprises commercial parallel file systems such as PFS for the Intel Paragon [19] , PIOFS and GPFS for the IBM SP [20], HFS for the HP Exemplar [21], and XFS for the SGI Origin2000 [22]. These file systems provide high performance and functionality desired for I/O-intensive applications but are available only on the specific platforms on which the vendor has implemented them.

Examples of distributed file systems are NFS [23], AFS/Coda [24], xFS [25], and GFS [26]. These file systems are designed to provide distributed access to files from multiple client machines, and their consistency semantics and caching behavior are designed accordingly for such access. The types of workloads resulting from large parallel scientific applications usually do not mesh well with file systems designed for distributed access; particularly, distributed file systems are not designed for high-bandwidth concurrent writes that parallel applications typically require.

Several parallel file systems are general-purposed and actively maintained/developed by companies, community and research organizations, including Lustre [7, 27] and Parallel Virtual File System (PVFS) [10] / Orange File System (OrangeFS) [8]. They are designed for high performance computing and can be installed on computing clusters built on commodity hardwares. These file system are highly tunable and can provide reasonable amount of support. Lustre and OrangeFS are deployed on a lot of data centers at national labs and research institutions. Lustre runs in kernel mode and supports high availability. PVFS/OrangeFS runs most of the server/client in the user mode and are easier to install and configure. It also supports high availability and can support a lot more features detailed below.

A number of research projects exist in the areas of parallel I/O and parallel file systems, such as PIOUS [28], Ceph [17] and Galley [29]. PIOUS focuses on viewing I/O from the viewpoint of transactions, Ceph is designed to be both self-healing and self-managing and strives to reduce both administrator and budget overhead, and Galley looks at disk-access optimization and alternative file organizations. These file systems may be freely available but are mostly research prototypes, not intended for everyday use by others.

2.1.2 PVFS and OrangeFS

PVFS is a parallel file system that supports both distributing file data and metadata among multiple servers and coordinated access to file data by multiple tasks of a parallel program. PVFS has an object based architecture. The software consists of two major components: a storage server that runs as a daemon on an IO node and a client library that provides an interface for user programs running on a compute node. The storage server stores all data in objects known as dataspaces and all IO operations are preformed relative to one or more of these objects. A data space consists of two storage areas: a bytestream that stores arbitrary binary data as a sequence of bytes, and a collection of key/value pairs that allow structured data to be stored and quickly retrieved when needed.

Trove is a subsystem of PVFS that implements dataspaces with a non-blocking post and poll interface. The current implementation of Trove uses the host's file system to implement bytestreams and a set of Berkeley DB databases to manage key/value pairs and dataspaces themselves. PVFS interacts over the network through an interface known as BMI. BMI provides a non-blocking post and poll interface for sending messages, receiving messages and checking to see if outstanding posts have completed. A special class of messages is unexpected messages those for which the server cannot predict their arrival. These are mostly requests from clients. Unexpected messages are limited in size but are useful for most client requests. Larger messages are managed using Flows. Flows combine network messages over BMI either with IO through Trove or directly to memory. Flows manage multiple buffers and complex data routing algorithms to transfer data associated with file IO from disk, across the network, to the client's memory. BMI has implementations for TCP/IP, GM, MX, IB, and other networking fabrics. Multiple networks can be used at the same time, though there are performance issues in doing this and most installations do no. The PVFS server manages requests to Trove, BMI, and Flows through a Job layer. The Job interface is also a non-blocking post and poll design and provides a common interface for having many outstanding operations in flight on the server at one time. Jobs are issued by the server Request processor which is built using a custom state-machine language SM. SM allows programs to define fundamental steps in the processing of each request each ending in posting a job. Once a job is posted the state-machine is suspending until completion at which point it automatically resumes. SM allows return codes to drive the processing to different states. SM is designed to make the coding of server requests simpler by abstracting away many details including interacting with BMI and Trove, encoding and decoding messages, and asynchronous issues. The PVFS client code is built from many of the same components as the server including BMI, Flows, Jobs, and state-machines. The primary difference is that the state-machines are written to implement each function in the PVFS System Interface (sysint). The sysint is designed based on operations typically found in a modern OS virtual file system. Thus, for example, there isn't an open call, but rather a lookup that takes a path name and returns a reference to the file's metadata. Metadata is read or written with getattr and setattr respectively. User programs are expected to use PVFS via one of several user level interfaces. These include a VFS kernel module for Linux, a FUSE interface, support via ROMIO (MPI-IO) and a User Interface (usrint) provided with PVFS.

Orange File System is a branch of the Parallel Virtual File System. Like PVFS, OrangeFS is a parallel file system designed for use on high end computing (HEC) systems that provides very high performance access to disk storage for parallel applications. OrangeFS is different from PVFS in that we have developed features for OrangeFS that are not presently available in the PVFS main distribution. While PVFS development tends to focus on specific very large systems, Orange considers a number of areas that have not been well supported by PVFS in the past. OrangeFS development tends to be driven by input from users with specific needs. To become one of these users open a dialog with the OrangeFS development team. PVFS design characteristics that are common with OrangeFS include:

- Performance
- Reliability
- Optimized MPI-IO support
- Hardware Independence
- Painless Deployment
- Research Platform

The system architecture is illustrated in Figure 2.1. There are a number of specific areas where Orange development has been focused over the last couple of years including:

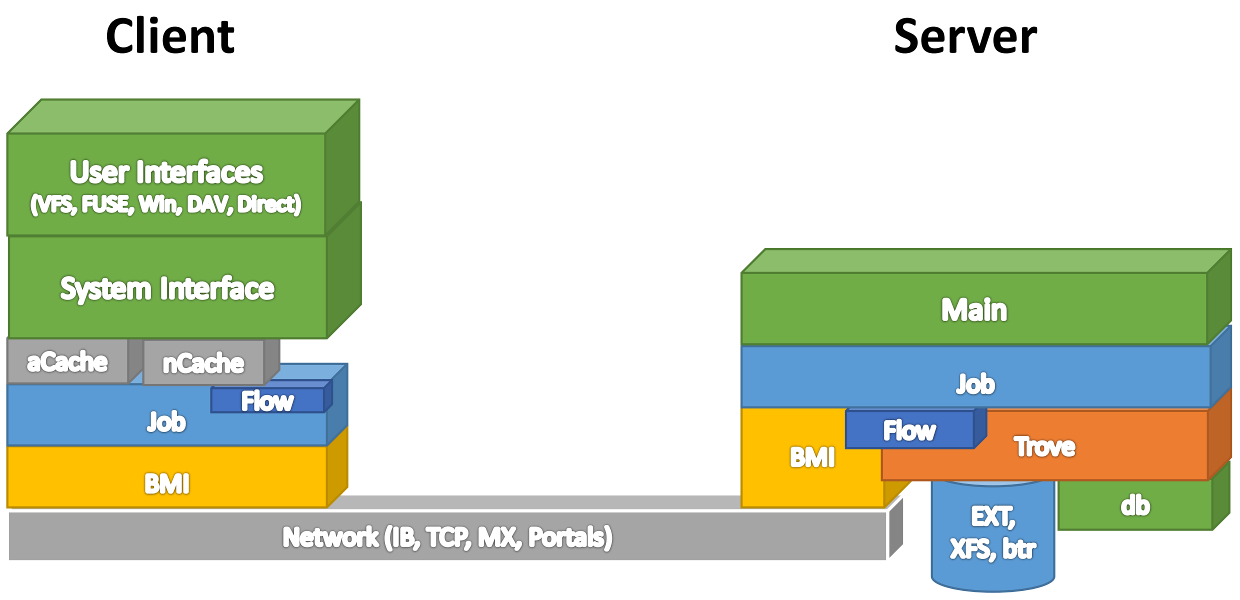


Figure 2.1: OrangeFS System Architecture Diagram.

Metadata operations OrangeFS uses server-to-server collective communication to improve the scalability of metadata operations. OrangsFS is implementing distributed directories to make directory operations more scalable. Another project is developing the ability to search metadata as an alternative to traditional path lookup.

Small, unaligned accesses OrangeFS is developing middleware driven caching on the client side including configurable semantics that provide a trade-off between performance and consistency management.

Cross-server redundancy Hardware failover provides a invaluable services for large storage systems. Some installations want additional redundancy, and smaller installations want redundancy without the high cost of hardware solutions. OrangeFS is developing configurable redundancy and fail-over mechanisms into the file system. These allow different files to have difference levels of redundancy as required by the application and also allow FS redundancy to the turned off for maximum performance.

Secure access control OrangeFS has developed an access control method based on signed credentials and capabilities that works with federated authentication mechanisms and yet maintains performance characteristics. Current research is working on flexible access control schemas beyond simple userid based permissions.

Improved documentation The OrangeFS project is working to be production ready including improving the documentation base for all of PVFS.

2.2 State of High Performance Computing System

There are two popular parallel computing systems in the high performance computing field when running applications on multiple machines: message passing system and virtual shared memory system. The message passing systems treat each computing node as an independent system and use message communication to coordinate and synchronize work. The best known protocol is Message Passing Interface (MPI). On the other end, the virtual shared memory model for parallel computing includes the computing resources on all computing nodes under the same namespace and depends on fine-grained synchronization among computing tasks as a coordination method. ParalleX is such a new execution model and HPX is one implementation. Table 2.1 shows the pros, cons and models of these two computing systems.

Table 2.1: Comparison of the message passing computing system and virtual shared memory computing system.

	message-passing	**virtual shared-memory**
Pros	scalable; workers are independent; data protection	scalable; easy to manage resources; fine-grained parallelism
Cons	global barrier to synchronize; need to manage resources explicitly	setup overhead; concurrency and synchronization problems
Models	MPI, PVM, Titanium	ParalleX, UPC

2.2.1 MPI - Message Passing Interface

MPI (Message Passing Interface) [3] is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation. It was defined by the MPI Forum [30], a broadly based group of parallel computer vendors, library writers, and applications specialists.

MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes. Furthermore, MPI provides abstractions for processes at two levels.

- First, processes are named according to the rank of the group in which the communication is being performed.
- Second, virtual topologies allow for graph or Cartesian naming of processes that help relate the application semantics to the message passing semantics in a convenient, efficient way. Communicators, which house groups and communication context (scoping) information, provide an important measure of safety that is necessary and useful for building up library-oriented parallel code.

There are several MPI implementations [31, 32]. Most MPI implementations consist of a specific set of routines (i.e., an API) directly callable from C, C++, Fortran and any language able to interface with such libraries, including C#, Java or Python. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs).

- **MPI-IO - Parallel I/O for MPI**

MPI-IO [33, 34] interface is designed as a widely used standard for describing parallel I/O operations within an MPI message-passing application. The interface establishes a flexible, portable, and efficient standard for describing independent and collective file I/O operations by processes in a parallel application. In a nutshell, MPI-IO is based on the idea that I/O can be modeled as message passing: writing to a file is like sending a message, and reading from a file is like receiving a message. MPI-IO intends to leverage the relatively wide acceptance of the MPI interface in order to create a similar I/O interface. The MPI-IO interface is intended to be submitted as a proposal for an extension of the MPI standard in support of parallel file I/O. The need for such an extension arises from three main reasons.

First, the MPI standard does not cover file I/O. Second, not all parallel machines support the same parallel or concurrent file system interface. Finally, the traditional UNIX file system interface is ill-suited to parallel computing.

Currently there are several implementation of MPI-IO library such as ROMIO [35]. In this paper we are trying to build a persistent storage model for the shared-memory model of distributed and parallel computing model ParalleX. The relation is comparable to MPI vs MPI-IO.

2.2.2 ParalleX - an Execution Model for Extreme Scale

ParalleX is an experimental execution model devised to support co-design and interoperability of the component layers of extreme scale (strong scale and Exascale) numeric and symbolic computing systems. It is a synthesis of a number of key concepts from prior art extending back multiple decades (in some cases) combined with original contributions that together comprise a new paradigm (in detail) for governing the structure and operation of future generation of efficient, scalable, and programmable high performance computing. The development and employment of a new execution model is motivated by underlying changes in the trends of enabling technologies and the need for responding to them in order to continue the exponential growth in delivered sustained performance for numeric applications in science and technology as well as new application domains in knowledge management and data understanding. ParalleX is devised principally to expose and exploit new forms of parallelism to extend scalability by the essential orders of magnitude required for extreme scale while hiding intrinsic latencies for necessary efficiencies in time and energy required for practical computer systems at the end of this decade including but not limited to Exascale processing capability.

ParalleX provides a conceptual framework for permitting dynamic adaptive runtime methods to achieve yet further improvements in efficiency by exploiting runtime information (not known at compile time) for adaptive load balancing and the circumvention of hotspots due to contention for physical (e.g., memory banks, network channels) and logical (e.g., semaphores) objects. In support of latency mitigation and adaptive contention avoidance ParalleX greatly expands the control space/state of a parallel execution by adding to conventional program counter and global barrier based flow control the use of a broad array of distributed control objects (LCO) based on dataflow semantics and the futures construct (among others). This second global layer of control allows the control state to migrate (literally move) across the computation and its physical computing medium. This global distributed structure of synchronization control state allows the management of asynchrony and at the same time reduces much of the data and service access latency.

A brief summary of the ParalleX model provides a foundation for the innovation anticipated and the research proposed in the domain of persistent storage model. There are seven primary constructs that comprise the ParalleX model, although substantial additional detail in their form and use is beyond the scope of this synopsis.

ParalleX complexes Like many emerging models of computation, ParalleX includes a set of lightweight task objects that are often referred to as user runtime threads or equivalent terminology but here are identified as computational abstract complexes of just complexes. The change in naming is to reflect that such executables in ParalleX incorporate internal fine-grain static dataflow control semantics where intermediate private variables are concerned (suggestive of single-assignment semantics) to preclude anti-dependencies where possible and further that they are first class objects capable of being named and manipulated in the same name space as any program variable. Unlike some PGAS languages such as UPC, ParalleX complexes are ephemeral in that they can be created and terminated at any time as well as being suspended and transformed in to a synchronization object (LCO) with the same name.

ParalleX process Such complexes exist in the context of a ParalleX process. A process defines a logically localized name space in which is maintained and managed multiple first class objects besides the complexes such as other processes, data, code, and mapping of logical process resources to system physical resources (e.g., nodes). This last suggests that unlike almost all other models, ParalleX permits logical processes to span multiple physical nodes. Multiple processes may also share the same node. A process is instantiated with operand values or references. Internal state may also be accessed by a limited set of methods by other processes. The hierarchical name space of a program comprises the tree of parent-child processes.

Local Control Objects LCO serve to provide the global layer of control state and are lightweight synchronization constructs that may be linked together in dynamic graphs for access and manipulation. While there are many forms including conventional semaphores and mutexes, the most important among these are the dataflow and futures constructs that manage asynchrony, establish constraint-based flow control, support anonymous producer-consumer computation, can exploit graph meta-data parallelism, balance eager versus lazy evaluation based on resource availability, and many other important functionality.

Parcels Parcels are a form of active message that facilitates message-driven computation to move the work to the data rather than always moving the data to the work. Parcels support split-phase transactions across nodes to reduce latency effects and improve both time and energy efficiencies. Parcels provide the means of asynchronous operation across the global distributed system within the name space of the process hierarchy.

Percolation Percolation is a variant of parcels that supports heterogeneous computation by providing the means to move both work and the data upon which it is to be performed to an ancillary subsystem like a new system node to add resources or a GPU accelerator to optimize specific idioms of execution. Percolation hides the latency for these costly resources and offloads their overhead to achieve their highest possible utilization.

Active Global Address Space AGAS is an advanced version of virtual address space that extends PGAS beyond conventional practices. Its critical distinction is that the virtual address of a first class object in AGAS is not localized to a particular pre-designated system node as in PGAS. Instead, AGAS permits the migration of first class objects across physical resources without having to change their virtual address. This is essential for load balancing and for the management of persistent storage data objects.

Synchronous Domains They are physical resources that serve as the nodes of conventional systems. They establish the boundaries within which key operational properties can be guaranteed and between which asynchronous operation via Parcels is assumed. Such attributed include bounded operational time properties amenable to compile time optimization and atomic compound sequences of memory operations. It is also assumed that all threads within a Synchronous Domain sees the same address space which may include cache coherence.

ParalleX, is solidly rooted in the following governing principles:

- The utilization of an Active Global Address Space (AGAS), without the assumption of cache coherence. This implies the preference of using adaptive locality management over purely static data placement strategies.
- The exposure of new forms of program parallelism, including fine-grained parallelism, a fundamental paradigm shift from CSP and MPI as today's prevalent programming model.
- The preference for mechanisms which allow hiding latencies over methods for latency avoidance.
- The preference for moving work to data over moving data to work.
- The elimination of (implicit and explicit) global barriers, replacing them with constraint-based synchronization built on Local Control Objects (LCOs) to enable the efficient

use of fine-grain parallelism.

- The facilitation of dynamic and heuristic resource management and task-queue based scheduling, to utilize information provided by runtime introspection and to allow active, dynamic, resource management, and locality control for applications.

2.2.3 HPX - An Implementation of ParalleX

High Performance ParalleX (HPX [36, 5, 37, 38]) is the first open-source implementation of the ParalleX execution model. HPX is a state-of-the-art runtime system developed for conventional architectures and, currently, Linux-based systems, such as large Non Uniform Memory Access (NUMA) machines and clusters. Strict adherence to Standard C++11 [39] and the utilization of the Boost C++ Libraries [40] makes HPX both portable and highly optimized. This modular framework facilitates simple compile- or runtime-configuration and minimizes the runtime footprint. HPX has been carefully designed as an alternative to mainstream parallel frameworks such as MPI. We have focused on overcoming conventional limitations such as global barriers, poor latency hiding, and lack of support for fine-grained parallelism. The current implementation of HPX (see Figure 2.2) supports most of the key ParalleX elements: Parcels, PX-threads, Local Control Objects (LCOs) and the Active Global Address Space (AGAS) .

Parcel Transport Layer: HPX parcels are a form of active messages used for communication between localities [41]. In HPX, parcels encapsulate remote method calls. A parcel contains the global name of an object to act on, a reference to one of the object's methods and the arguments to call the method with. Parcels are used to either migrate work to data by invoking a method on a remote entity, or to bring pieces of data back to the calling locality. Each locality has a parcel port which reacts to inbound messages and asynchronously transmits outbound messages. After a parcel port receives and de-serializes a message, it passes the parcel to a parcel handler. The parcel handler is responsible for deciding how to process the parcel. The primary parcel handler currently used by HPX

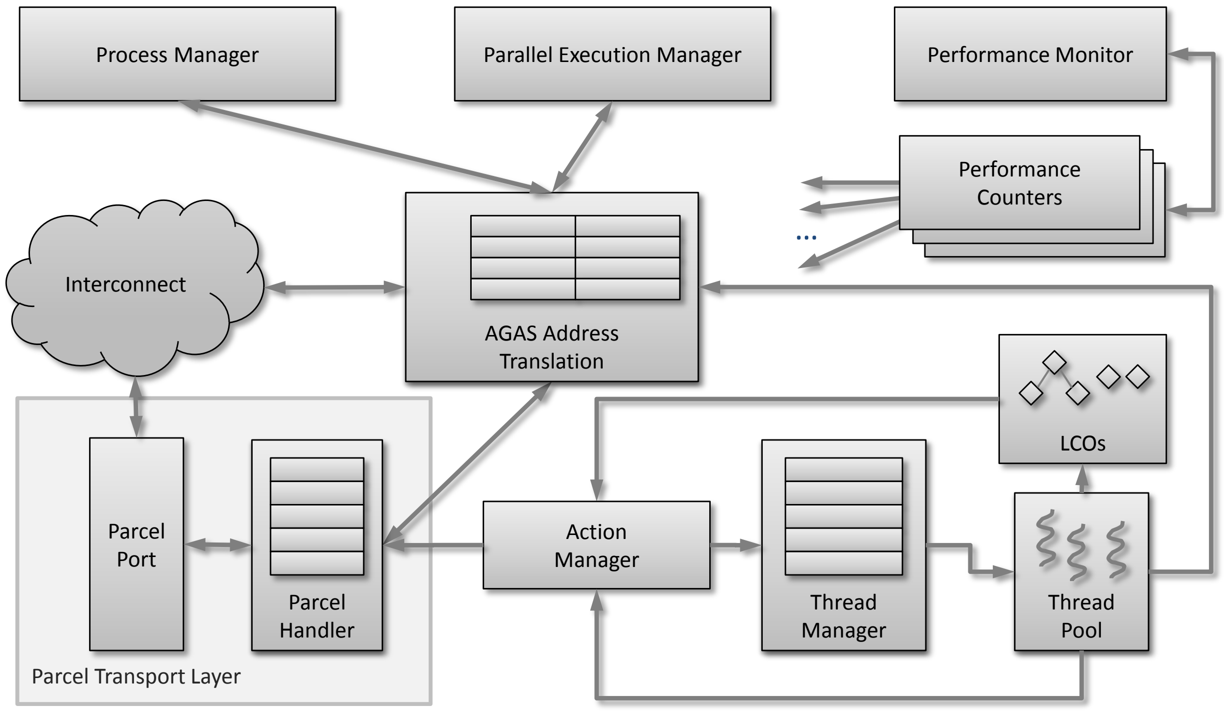


Figure 2.2: Architecture of the HPX runtime system. HPX implements the supporting functionality for most of the elements needed for the ParalleX model: Parcels (parcel-port and parcel-handlers), HPX-threads (thread-manager), LCOs, AGAS, HPX-processes, performance counters and a means of integrating application specific components.

is the action manager. If the target object of a parcel is local, then the action manager converts the parcel into a HPX-thread, which is scheduled by the HPX thread-manager.

HPX-threads and their management: The HPX thread-manager is responsible for the creation, scheduling, execution and destruction of HPX-threads. In HPX, threading uses an $M : N$ or hybrid threading model. In this model, N HPX-threads are mapped onto M kernel threads (OS-threads), usually one OS-thread per core. This threading model was chosen to enable fine-grained parallelization and low overhead context switches; HPX-threads can be scheduled without a kernel call, reducing the overhead of their execution and suspension. The thread-manager uses a work-queue based execution strategy with work stealing similar to systems such as Cilk++ [42], Intel Threading Building Blocks (TBB [43]) and the Microsoft Parallel Patterns Library (PPL [44]). HPX-threads are scheduled cooperatively, that is, they are not preempted by the thread-manager. HPX-threads voluntarily suspend themselves when they must wait for data that they require to continue execution, I/O operations, or synchronization.

Local Control Objects (LCOs): LCOs provide a means of controlling parallelization and synchronization in HPX. Any object that may create a new HPX-thread or reactivate a suspended HPX-thread exposes the required functionality of an LCO. Support for event-driven HPX-thread creation, protection of shared data structures, and organization of flow control are provided by LCOs. They are designed to allow for HPX-threads to proceed in their execution as far as possible, without waiting for a particular blocking operation, such as a data dependency or I/O, to finish. Some of the more prominent LCOs provided by HPX are:

- **Futures** [45, 46, 47] are proxies for results that are not yet known, possibly because they have not yet been computed. A future synchronizes access to the result value associated with it by suspending HPX-threads requesting the value if the value is not available at the time of the request. When the result becomes available, the future reactivates all suspended HPX-threads waiting for the value. These semantics allow execution to proceed unblocked until the actual value is required for computation.

- **Dataflow objects** [48, 49, 50] provide a powerful mechanism for managing data dependencies without the use of global barriers. A dataflow LCO waits for a set of values to become available and triggers a predefined function passing along all input data.
- **Traditional concurrency control mechanisms** such as various types of mutexes [51], counting semaphores, spinlocks, condition variables and barriers are also exposed as LCOs in HPX. These constructs can be used to cooperatively block an HPX-thread while informing the HPX thread-manager that other HPX-threads can be scheduled on the underlying OS-thread.

LCOs are first class objects in HPX, they enable intrinsic overlapping of computation and communication. This not only hides latencies, but also allows many phases of a computation to overlap, exposing greater application parallelism. They can be used to control parallelism across multiple localities. The mechanisms for naming and referencing first class objects such as LCOs are provided by AGAS.

The Active Global Address Space (AGAS): In HPX, AGAS currently is a set of services that implement a 128-bit global address space that spans all localities. AGAS provides two naming layers in HPX. The primary naming service maps 128-bit unique, global identifiers (GIDs) to a tuple of meta-data that can be used to locate an object on a particular locality. The higher-level layer maps hierarchical symbolic names to GIDs. Unlike systems such as X10 [52], Chapel [53], or UPC [54], which are based on PGAS [55], AGAS exposes a dynamic, adaptive address space which evolves over the lifetime of an HPX application. When a globally named object is migrated, the AGAS mapping is updated, however, its GID remains the same. This decouples references to those objects from the locality that they are located on.

Performance Counter Framework: HPX performance counters provide an intrusive method of instrumenting the environment in which an HPX application is running, exposing metrics from hardware, the OS, HPX runtime services, and applications. The data provided by performance counters facilitate the development of heuristic algorithms that use introspection to make smarter runtime decisions. A performance counter is a first class object associated with a symbolic name which exposes a uniform interface for collecting arbitrary performance data on demand. Instrumentation utilities can connect to a running HPX application through the parcel transport layer, query performance counters, and then disconnect.

2.3 Linda: Coordination and Communication Model

Programming a distributed or parallel system can be seen as the combination of two distinct activities: the actual computing part comprising a number of processes involved in manipulating data and a coordination part responsible for the communication and cooperation between the processes. Thus, coordination can be used to distinguish the computational concerns of some distributed or parallel application from the communication ones.

In a lot of computing systems, computation and coordination does not have clear distinction. For example, the coordination in MPI is achieved by synchronizations like global barriers. And for the virtual shared memory model, LCOs can be used for coordination. However, the computation problem in the next decade can have larger coordination problems on thousands and thousands of synchronizing problems which is very hard or inefficient by only using the current small-scale synchronization methods. Separating the coordination model from the computation model is one way to target the problem from a brand new perspective.

Linda [56, 57] is historically the first genuine member of the family of coordination languages. It provides a simple and elegant way of separating computation from communication concerns. Linda is based on the so-called generative communication paradigm: if two processes wish to exchange some data, then the sender generates a new data object (referred to as a tuple) and places it in some shared dataspace (known as a tuple space) from which the receiver can retrieve it. This paradigm decouples processes in both space and time: no process need to know the identity of other processes, nor is it required of all processes involved in some computation to be alive at the same time. In addition to passive tuples containing data, the tuple space can also contain active tuples representing processes which after the completion of their execution, turn into ordinary passive tuples.

Linda is in fact not a fully fledged coordination language but a set of some simple coordination primitives. In particular, `out(t)` is used to put a passive tuple `t` in the tuple space, `in(t)` retrieves a passive tuple t from the tuple space, `rd(t)` retrieves a copy of `t` from the tuple space (i.e., `t` is still there) and `eval(p)` puts an active tuple `p` (i.e., a process) in the tuple space. The primitives `rd` and `in` are blocking primitives and will suspend execution until the desired tuple has been found. The primitives `out` and `eval` are nonblocking primitives. A process that executes `eval(p)` will carry on executing in parallel with `p`, which will turn into a passive tuple when it completes execution. Over the years, a number of additional primitives were introduced into the basic model; for instance `rdp(t)` and `inp(t)` are non-blocking variants of `rd(t)` and `in(t)`, respectively, which when the desired tuple is not found in the tuple space will return FALSE. Tuples are actually sequences of typed fields. They are retrieved from the tuple space by means of associative pattern matching. More to the point, the parameter t of the primitives `in`, `inp`, `rd` and `rdp` is actually a tuple schemata containing formal parameters; pattern matching of `t` with an actual tuple `ta` in the tuple space will succeed provided that the number, position, and types of `t`s fields match those of `ta`.

The Linda primitives are indeed completely independent of the host language; thus, it is possible to derive natural Linda variants of almost any programming language or paradigm (imperative, logic, functional, object-oriented, etc.).

There are several implementations of Linda TupleSpace on various platforms, such as JavaSpaces [58], TSpaces [59] and GigaSpaces [60], etc.

Chapter 3
PXFS: A Persistent Storage Model for Extreme Scale

3.1 PXFS Design

The PXFS model is designed as a layer on top of storage media and file systems, and as a part of the ParalleX runtime system. The system diagram is depicted in Fig. 3.1. It is implemented as a component of HPX and manages I/O operations and object meta properties through different levels.

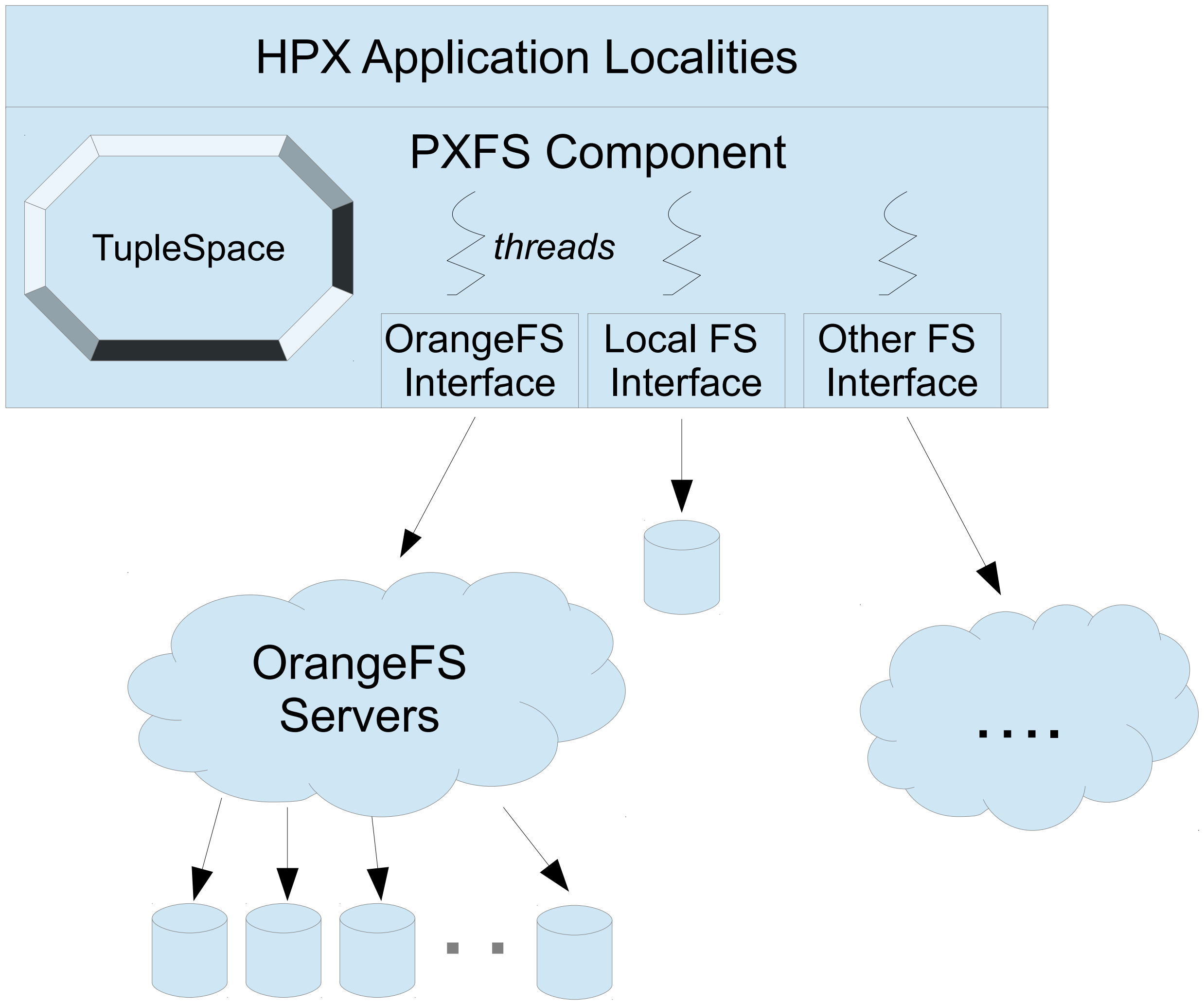


Figure 3.1: Design of PXFS storage component as a part of HPX runtime.

The interfaces of different file systems can be expanded to more general storage media like tape or network. HPX threads are essential to guarantee the asynchronous properties of pxfs calls. The TupleSpace entities will be detailed below to active manage meta properties of both runtime objects and storage objects.

3.2 Asynchronous I/O Operations

3.2.1 OrangeFS Asynchronous I/O Interface

As a coordinated work, an OrangeFS asynchronous interface is designed and a prototype has been implemented [14]. It is part of the PXFS model to interact with parallel file system in an asynchronous fashion. The asynchronous interface will use the OrangeFS system calls to schedule I/O operations and execute a callback function when the task is finished. PXFS will create HPX futures for I/O operations and set the future value in the callback function to integrate the parallel file system into the HPX runtime system seamlessly.

The HPX asynchronous interface will use the OrangeFS asynchronous I/O (AIO) subsystem which add a thread managing layer on top of OrangeFS system interface. The AIO subsystem provides a set of asynchronous I/O functions to perform basic I/O operations. It will store the result in a pointer and enable a callback function.

Examplar OrangeFS AIO Write Call

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

Figure 3.2: Prototype for the OrangeFS AIO write operation.

As an example, the asynchronous `write` function call is listed in Fig. 3.2. 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.

The AIO library contains most of the I/O operations provided in OrangeFS user interface, including **read/pread**, **write/pwrite**, **lseek** and directory operations.

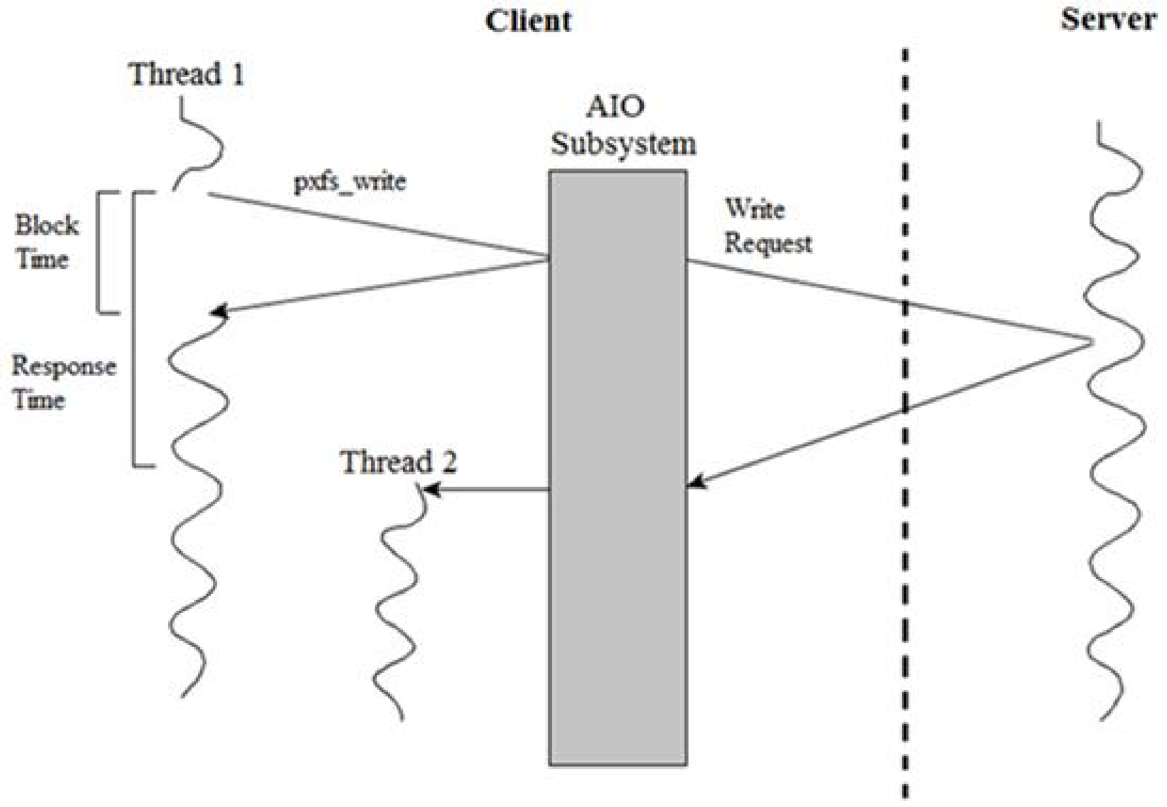


Figure 3.3: Data flow diagram illustrating an example I/O operation in the AIO library. Thread 1 submits a `pxfs_write` operation with a continuation which spawns Thread 2.

A data flow diagram illustrating the asynchronous nature of the AIO subsystem is given in Fig. 3.3. This diagram provides an insight into how external threads submit I/O operations and respond to their completion. 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 initiating application. In the case of Fig. 3.3, the continuation spawns a new thread of execution, but could be easily modified to notify the original thread or perform any other functionality.

To integrate the OrangeFS AIO interface with HPX, an HPX application can pass a future as the argument of the callback function and set the value of the future in the callback function. A simple code example is illustrated in Fig. 3.4. The callback function `set_promise_cb` will set the value of the future passed through the argument to notify the external HPX application when the I/O operations is finished. There are some register and unregister functions in the callback function to deal with the different thread mechanisms between HPX threads and posix threads.

Moreover, the AIO interface is highly configurable and the number of internal threads and the size of thread waiting queues can be tuned and changed according to the actual environment to fit the requirements.

Examplar HPX AIO Call

```
typedef hpx::lcos::local::promise<int> int_promise_type;
typedef hpx::runtime rt_type;

// structure to pass HPX runtime information
struct promise_rt_ptr_type
{
    std::string thread_name_;
    int_promise_type* p_p_;
    rt_type* rt_p_;
}

// this function will be executed by an HPX thread
void set_value(
    hpx::lcos::local::promise<int> *p,
    int result)
{
    // notify the waiting HPX thread and return a value
    p->set_value(result);
}

// the callback function to set the future value
int set_promise_cb(void *cdat, int status)
{
    promise_rt_ptr_type *pr_p = (promise_rt_ptr_type *) cdat;

    pr_p->rt_p_->register_thread(pr_p->thread_name_.c_str());

    // Create an HPX thread to guarantee that the promise::set_value
    // function can be invoked safely.
    hpx::threads::register_thread(hpx::util::bind(&set_value, pr_p->p_p_, status));

    pr_p->rt_p_->unregister_thread();

    return status;
}

// init promise_rt_ptr
promise_rt_ptr_type promise_rt_ptr;
// make the AIO call
pxfs_write(fd, buf, bufsiz, &num_written, set_promise_cb, &promise_rt_ptr);
```

Figure 3.4: An example code to show the integration of HPX and OrangeFS AIO interface to make an asynchronous `write` call in HPX applications.

3.2.2 HPX Asynchronous I/O Class APIs

While designing the API for the implementation of the presented persistent storage model special attention was directed towards a natural integration with the existing highly asynchronous programming model exposed by HPX. By design, all functionality in HPX which can potentially take longer than 100 microseconds to execute is exposed through asynchronous functions. An asynchronous function in HPX is a function which is returning a future representing the result of that function. The caller can continue executing immediately without being suspended or without having to wait for the function result. Three C++ classes, **local_file**, **orangefs_file** and **pxfs_file** are written to incorporate the asynchronous nature into I/O calls. The code example in Fig. 3.5, Fig. 3.6 and Fig. 3.7 show part of the developed file oriented APIs, respectively.

Note that the HPX future type exposes an interface consistent with the C++11 Standard [39] with extensions as proposed to the C++ standardization process [61, 62].

The **file_class** and **orangefs_class** interfaces are using the client/server architecture in HPX which creates a component on a designated server side and on current locality calls the server actions through a thin client interface to deal with the asynchronism mechanism. The client side interface is an asynchronous wrapper which schedules the asynchronous task and manages the results processed at the server side which is utilizing the synchronous I/O functions provided by the C++ fstream library and the OrangeFS synchronous user interfaces.

On the other hand, the **pxfs_file** is directly employing the OrangeFS AIO interface which is already asynchronous by itself. Thus the **pxfs_file** is merely an HPX client class which setup the interface and thread then handle the function to the OrangeFS AIO interface in OrangeFS client and then communicate with the remote OrangeFS servers.

HPX local_file Asynchronous class API

```
namespace hpx { namespace io {
class local_file {
  ~local_file();          // closes file synchronously

  // asynchronous function by default
  lcos::future<void> open(std::string const& name, std::string const& mode);
  void open_sync(std::string const& name, std::string const& mode)

  lcos::future<void> close();
  void close_sync();

  lcos::future<int> remove_file(std::string const& file_name);
  int remove_file_sync(std::string const& file_name);

  // read/write operations
  lcos::future<std::vector<char> > read(size_t const& count)
  std::vector<char> read_sync(size_t const count)
  lcos::future<std::vector<char> > pread(ssize_t const count, off_t const offset);
  std::vector<char> pread_sync(size_t const count, off_t const offset)
  lcos::future<ssize_t> write(std::vector<char> const& buf);
  ssize_t write_sync(std::vector<char> const& buf);
  lcos::future<ssize_t> pwrite(std::vector<char> const& buf, off_t const offset);
  ssize_t pwrite_sync(std::vector<char> const& buf, off_t const offset);
  lcos::future<int> lseek(off_t const offset, int const whence);
  int lseek_sync(off_t const offset, int const whence);
};
}}
```

Figure 3.5: This code example shows the asynchronous file oriented API exposed by the implementation of the **local_file** class presented persistent storage model. The **local_file** class is responsible to operate on local file systems using the HPX asynchronous interfaces and C++ standard file I/O libraries.

HPX orangefs_file Asynchronous class API

```
namespace hpx { namespace io {
class orangefs_file {
  ~orangefs_file();           // closes file synchronously

  // asynchronous function by default
  lcos::future<void> open(std::string const& name, int const& flag);
  void open_sync(std::string const& name, std::string const& mode)

  lcos::future<void> close();
  void close_sync();

  lcos::future<int> remove_file(std::string const& file_name);
  int remove_file_sync(std::string const& file_name);

  // read/write operations
  lcos::future<std::vector<char> > read(size_t const& count)
  std::vector<char> read_sync(size_t const count)
  lcos::future<std::vector<char> > pread(ssize_t const count, off_t const offset);
  std::vector<char> pread_sync(size_t const count, off_t const offset)
  lcos::future<ssize_t> write(std::vector<char> const& buf);
  ssize_t write_sync(std::vector<char> const& buf);
  lcos::future<ssize_t> pwrite(std::vector<char> const& buf, off_t const offset);
  ssize_t pwrite_sync(std::vector<char> const& buf, off_t const offset);
  lcos::future<int> lseek(off_t const offset, int const whence);
  int lseek_sync(off_t const offset, int const whence);
};
}}
```

Figure 3.6: This code example shows the asynchronous file oriented API exposed by the implementation of the **orangefs_file** class presented persistent storage model. The **orangefs_file** class is responsible to operate on OranegeFS using the HPX asynchronous interface and OrangeFS synchronous user interfaces.

HPX pxfs_file Asynchronous class API

```
namespace hpx { namespace io {
class pxfs_file {
  ~pxfs_file();           // closes file synchronously

  // asynchronous function by default
  lcos::future<void> open(std::string const& name, int const& flag);
  void open_sync(std::string const& name, std::string const& mode)

  lcos::future<void> close();
  void close_sync();

  lcos::future<int> remove_file(std::string const& file_name);
  int remove_file_sync(std::string const& file_name);

  // read/write operations
  lcos::future<std::vector<char> > read(size_t const& count)
  std::vector<char> read_sync(size_t const count)
  lcos::future<std::vector<char> > pread(ssize_t const count, off_t const offset);
  std::vector<char> pread_sync(size_t const count, off_t const offset)
  lcos::future<ssize_t> write(std::vector<char> const& buf);
  ssize_t write_sync(std::vector<char> const& buf);
  lcos::future<ssize_t> pwrite(std::vector<char> const& buf, off_t const offset);
  ssize_t pwrite_sync(std::vector<char> const& buf, off_t const offset);
  lcos::future<int> lseek(off_t const offset, int const whence);
  int lseek_sync(off_t const offset, int const whence);
};
}}
```

Figure 3.7: This code example shows the asynchronous file oriented API exposed by the implementation of the **pxfs_file** class presented persistent storage model. The **pxfs_file** class is responsible to operate on OranegeFS using the OrangeFS AIO asynchronous user interfaces.

The advantage of the chosen asynchronous programming model is demonstrated in the code example shown in Fig.3.8, which demonstrates how easy it is to fully overlap the asynchronous IO operations (create a file followed by writing several chunks of data to it) with other useful work.

Examplar Usage of Asynchronous pxfs_file API

```
using namespace hpx;
using namespace hpx::io::pxfs_file;

std::vector<char> data1 = {...};
std::vector<char> data2 = {...};
std::vector<char> data3 = {...};

pxfs_file pf;

pf.open_sync(file_name, O_WRONLY|O_CREAT);

std::vector<hpx::lcos::future<ssize_t> > futures;

// spawn asynchronous file write operations
futures.push_back(pf.pwrite(data1, 0));
futures.push_back(pf.pwrite(data2, data1.size()));
futures.push_back(pf.pwrite(data3, data1.size() + data2.size()));

// do other operations here, concurrently to I/O

hpx::lcos::wait_each(futures); // synchronize with whole I/O operations

// do other I/O operations to the file

pf.close();
```

Figure 3.8: This code example demonstrates an exemplar use of the file oriented **pxfs_file** API to asynchronously create a new file and to write some chunks of data to it. The I/O operation is performed fully overlapping any other work which needs to be performed before synchronizing with the result of the I/O.

Unlike the traditional I/O operations which takes a local byte buffer as the input and fill in the contents, the asynchronous I/O operations are all about objects. The data that used to read from or write to files are encapsulated inside a **vector¡char¿** object and delivered to the data source. This is also conform to the idea of ”send work to data” which can efficiently reduce the overhead in communication and send less amount of data between client and server.

3.3 Active TupleSpace Model

3.3.1 Introduction

Linda[56, 57] is the first programming language of a coordination model for distributed applications. It is built around TupleSpace, a virtual shared memory holding a collection of ordered tuples. A tuple is an ordered list of typed fields which can contain executable code or passive data values. Linda supports a small, yet elegant, set of operations on TupleSpace.

- **out**: insert a tuple into the TupleSpace.
- **in**: retrieve a tuple from the TupleSpace and remove it from the space.
- **rd**: retrieve a tuple from the TupleSpace and preserve it in the space.

To find a tuple in the TupleSpace, a tuple template which contains a subset of exact matching or wildcards matching of tuple fields. Both **in** and **rd** operations are blocking, which combines synchronization and communication in an extremely simple model with a high level of abstraction.

3.3.2 HPX TupleSpace

HPX implements the Active Global Address Space (AGAS) which exposes a dynamic, adaptive address space which evolves over the lifetime of an HPX application. The shared memory model of HPX is a perfect hot bed to implement TupleSpace within.

The HPX TupleSpace will implement and extend Linda model under HPX framework. TupleSpace can be located in one locality or distributed over multiple localities. A tuple can contain active HPX actions/continuations or passive data values of any type.

Besides the three basic operations (**out**, **in** and **rd**), an asynchronous notification interface is added which can trigger an HPX action when a certain operation on a matched tuple is carried out. This capability will fit into the “move work to data” principle of HPX nicely.

Also the TupleSpace will support serialize and de-serialize itself to move to and from the data. The serialization and de-serialization can be stored as a closure to be triggered in the TupleSpace under certain circumstances. Some use cases will be illustrated in the next few chapters. It can be used as a snapshot holder to store temporary states and resume.

The role of HPX TupleSpace can not only be served as a coordinating language, but also as a meta-space for objects. The TupleSpace can hold all attributes and properties of objects, localities, files, processes, applications, and etc. These information can be called meta entities because they are holding attributes and properties of an entity, just as metadata vs data in file system fields. A meta entity lives along with the corresponding entity instance. It has the same life time as the base entity. It can be stored at the same or different places with the entity instance. It can contain all aspects of information about the entity instance. The TupleSpace can also be a knowledge base to connect with other libraries/applications, including OrangeFS.

Some examples of meta entity can be illustrated below.

- The status(suspend/running), gid, properties(some features enabled or not, etc), attributes(life time, user-defined etc), performance counter, etc
- All namespace information, relation to other objects and the knowledge of where some certain object is located(partial AGAS space upon interaction with AGAS or other objects). For example, process hierarchy tree, data dependence graph.
- security properties(permission for owner,group,others), protection framework.
- metadata of OrangeFS file objects, such as data server address and data distribution topology.

Since all meta entities are kept in TupleSpace, it is really easy to conduct meta space search. Search criteria can be composed as a tuple template and the result can be retrieved as another tuple. It is also ideal for debugging at the runtime.

3.3.3 The Implementation

The HPX TupleSpace is implemented as a HPX component. The tuple structure is described in Fig. 3.9. It utilizes the `hpx::util::any` type which can hold any type of objects.

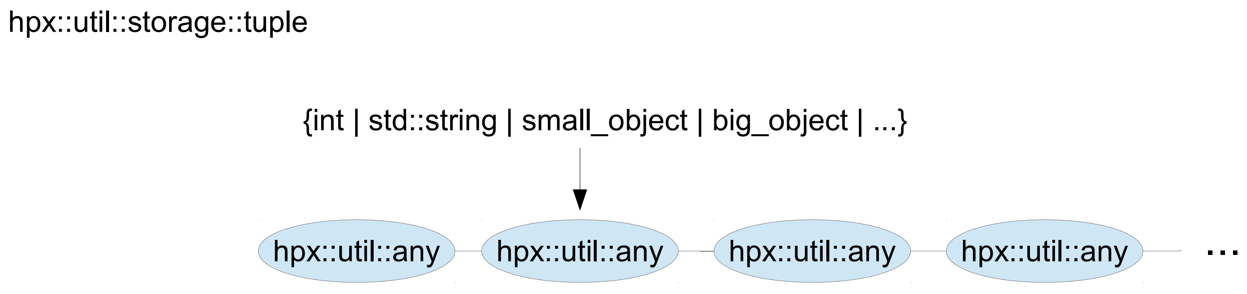

hpx::util::storage::tuple
{int | std::string | small_object | big_object | ...}
hpx::util::any
hpx::util::any
hpx::util::any
hpx::util::any
...

Figure 3.9: Diagram of HPX Tuple structure. HPX tuple is composed of an ordered list of any object, which can hold any type of object.

The diagram of the HPX TupleSpace is shown in Fig. 3.10. All tuples are kept in tuple_warehouse and the asynchronous interface is also included. From outside, HPX threads can perform `read`, `take` and `insert` operations and third-party libraries can also interact with TupleSpace. If all HPX threads store and update their status and all other attributes in the TupleSpace, then it can be treated as a huge meta space based on which better decisions about any aspect can be made. It also support serialize and de-serialize to and from disk files, which can store a snapshot of current application status so that in the future it can load all the information back and resume running.

A central TupleSpace has been implemented in HPX. Tuples are indexed in `tuple_warehouse` and three operations, `insert`,`read` and `take` are provided. TupleSpace can examine each element of an incoming template tuple to find the best-matched tuple. HPX threads are able to find the TupleSpace via AGAS and communicate with it using HPX actions. TupleSpace is able to serialize and de-serialize itself to and from storage media, which provides the capability of store and load a snapshot of itself in disk.

A code example of using the HPX TupleSpace is listed in Fig. 3.11. A TupleSpace can be created with a symbol name and other threads can connect to the TupleSpace

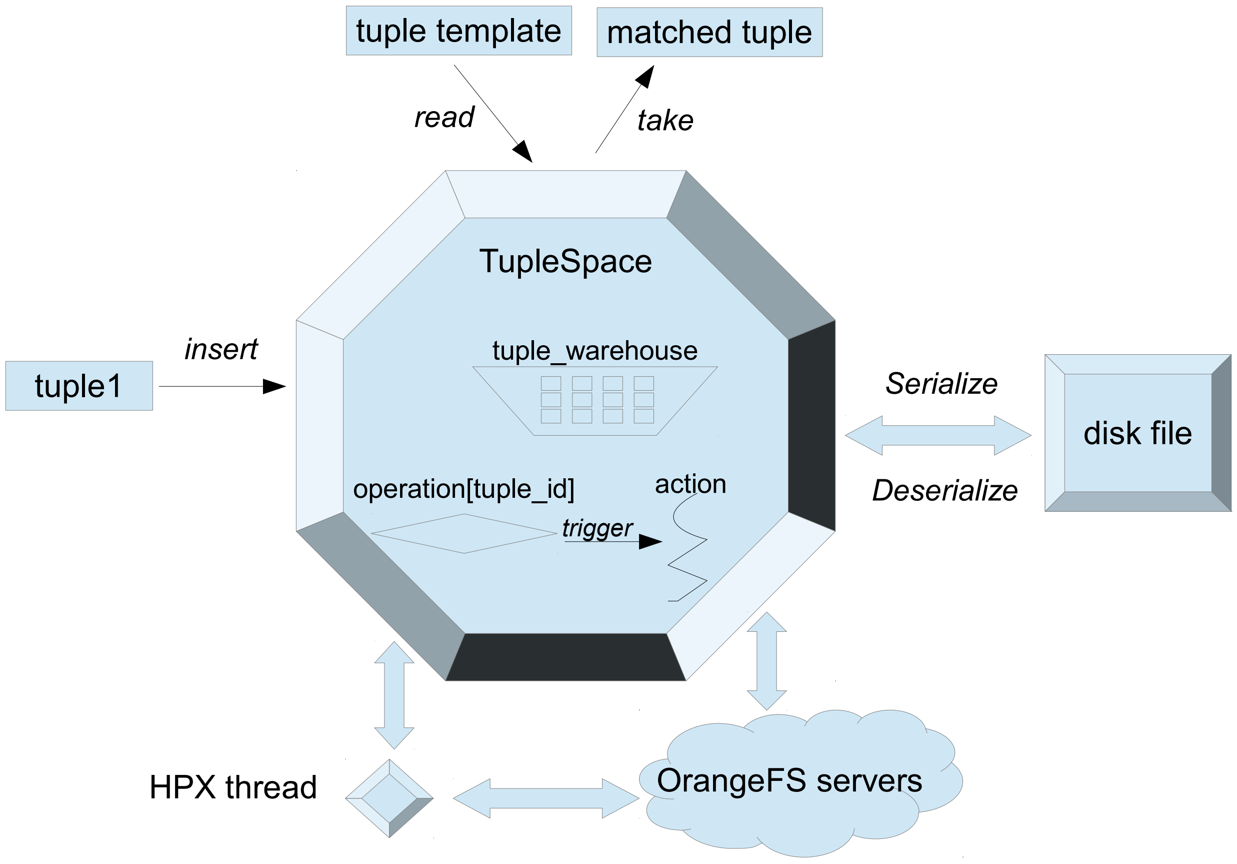


Figure 3.10: Diagram of HPX TupleSpace component.

through the symbol name. Three operations, `write`, `read` and `take` are supported in both synchronous and asynchronous fashion. The tuple retrieval functions, `read` and `take` will take a template tuple and return the best match tuple.

Examplar HPX TupleSpace Usage

```
// Find the localities to hold the tuplespace.
std::vector<hpx::id_type> locality;

const std::string tuplespace_symbol_name = "/tuplespace"; // tuplespace name
examples::simple_central_tuplespace central_tuplespace;

// create tuplespace
central_tuplespace.create(tuplespace_symbol_name, locality);

// create tuple
tuple_type tuple1;
tuple1.push_back(std::string("first"))
.push_back(10) // first elem: int
.push_back(small_object(20)) // second elem: small_object
.push_back(big_object(30, 40)); // third elem: big_object

// write to tuplespace asynchronously
future<int> write_op = central_tuplespace.write(tuple1);

// tuple template
tuple_type partial_tuple = <...>;
// read operation, 2nd argument means infinity waiting time, blocking
future<tuple_type> read_op = central_tuplespace.read(partial_tuple, 0);
// take operation, 2nd argument means infinity waiting time, blocking
future<tuple_type> take_op = central_tuplespace.take(partial_tuple, 0);

// doing other computation and communications

// get the results when needed
int ret = write_op.get();
tuple_type read_result = read_op.get();
tuple_type take_result = take_op.get();
```

Figure 3.11: This code example demonstrates an exemplar use of the TupleSpace and the three operations. There are synchronous and asynchronous tuplespace operations.

Another code example of using the HPX TupleSpace serialization interfaces is shown in Fig. 3.12. A TupleSpace client can connect to a server-side tuplespace identified by a symbol name and issue a serialization command to stroe the whole contents into storage file objects. Also a TupleSpace can load new contents from a serialized disk file and start from there.

3.3.4 Use Case: Micro Check-pointing

This section describes a check-pointing use case of HPX applications by utilizing the active TupleSpace. The basic scenario is that a large scientific application will have a large

```
Examplar TupleSpace Serialization Usage

examples::simple_central_tuplespace tuplespace;

// connect to existing tuplespace
tuplespace.connect(tuplespace_symbol_name);

// serialize the whole tuplespace into a file
future<int> store_op = tuplespace.store(file_name);

// doing other computation and communications

int store_ret = store_op.get();

// deserialize the tuplespace from a file
future<int> load_op = tuplespace.load(another_file_name);

// doing other computation and communications

int load_ret = load_op.get();
```

Figure 3.12: This code example demonstrates an exemplar use of the TupleSpace serialization interfaces.

amount of data and needs a long time to run, even days or weeks. Nowadays when running application on large computing clusters which has a lot of computing nodes, the probability of nodes failure will increase dramatically. It could result in a halt of the computation in the middle, which could waste a lot of time and resources.

Check-pointing is one way to solve this problem by storing the current data and status into a snapshot file. The if the computation is interrupted later and restarted, the application can load the snapshot file and resume computation from a middle point instead of start over. However, current implementation of the check-pointing step will bring the whole application to a stop and then store every necessary parts to storage files, whose process might cost a huge time and all computer resources are wasted because of the blocking nature of the process. When using MPI to implement check-pointing, a global barrier is essential to bring every process to a synchronized stage and then start the possible lengthy check-pointing process.

Here within the ParalleX model with the help of the active TupleSpace model, it is possible to overcome these disadvantages by using a micro-check-pointing mechanism. The basic diagram is illustrated in Figure 3.13.

The HPX application is composed of the AGAS component and other components

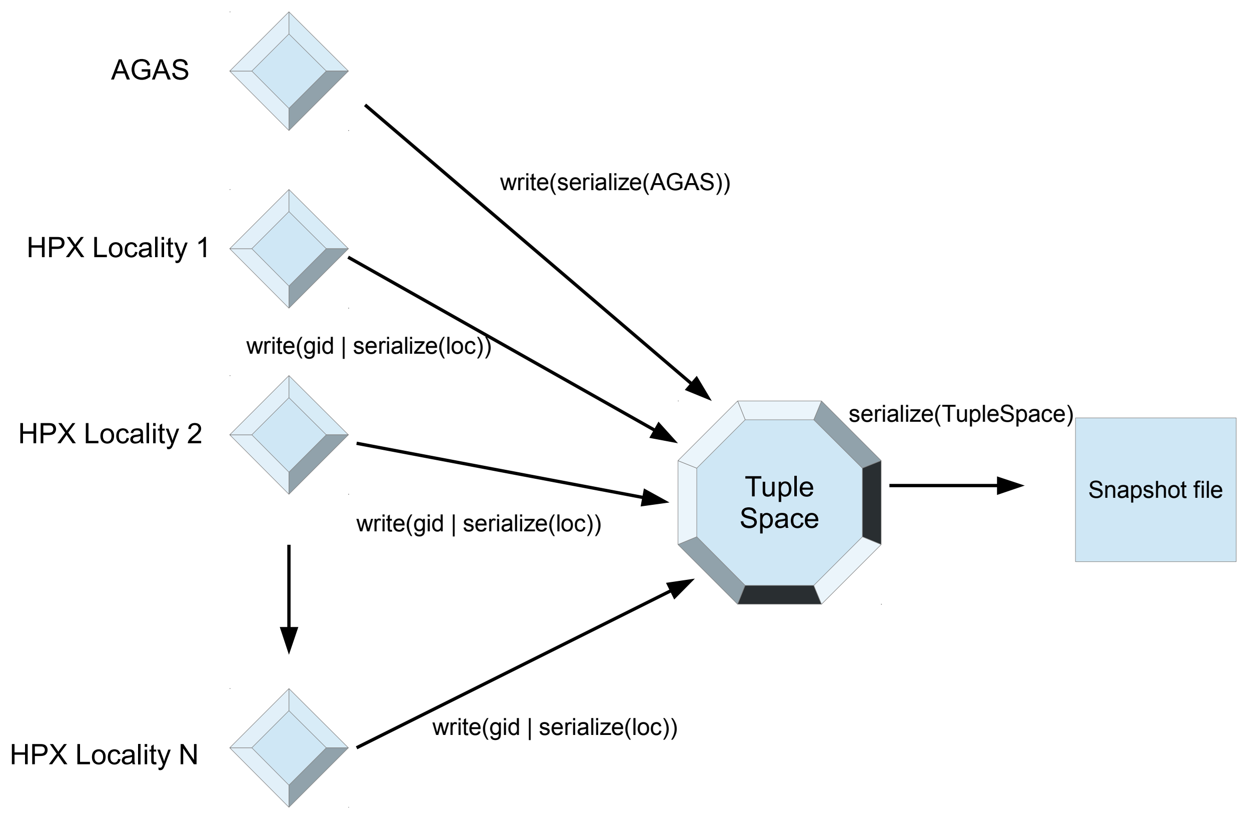


Figure 3.13: Use case of check-pointing of HPX application using the active TupleSpace.

residing in different HPX localities. In the middle of the computation, when the application reaches one point to store its whole data and status, each locality or component can decide on their own when to start the check-pointing process. And the meaning of micro check-pointing is that it does not need a global barrier to force all parts to run the check-pointing at the same time, rather it can only depend on a small set of localities/objects to start the check-pointing on its own schedule.

The process of check-pointing is that every component stores itself to the TupleSpace and the AGAS could register an action to serialize the TupleSpace when all objects/localities have submitted their data and status. The use of asynchronism ensures that the check-pointing is done automatically and will not interfere with the computation flow, which will improves the performance.

One big problem remains to solved is to determine the global consistent state of all the object in one snapshot file. This topic remains out of the scope of this dissertation.

3.4 Performance Evaluation

3.4.1 Method

A disk performance micro benchmark is developed in HPX to measure the total throughput of PXFS module on I/O reading and writing performance. The benchmark can adjust the following parameters:

- **nc**: number of client nodes
- **nt**: number of threads on each client node, a thread is carrying out the actual I/O operations.
- **r/w**: select between read and write tests.
- **nf**: number of test files.
- **nb**: number of total blocks composed in one file. A thread is operating on one block when issuing one I/O commands.

- **sb**: number of bytes in one block.

These test files are read or written in a sequential fashion at the current time. Then the read/write throughput can be calculated as

$$Throughput = \frac{nc * nt * nf * nb * sb}{Elapsed\ Time}$$

It is obvious that larger throughput means a better performance.

3.4.2 Synchronous I/O v.s. Asynchronous I/O

This experiment will compare the performances of synchronous I/O and asynchronous I/O on local file system, network file system and orange file system. It will use the benchmark mentioned in Section 3.4.1 to measure the I/O throughput of synchronous and asynchronous I/O operations on these file systems.

- **Experiment Setup**

All the tests are executed on the Hermione cluster at STE||AR [37] group in Center for Computation and Technology (CCT)[63] at Louisiana State University (LSU) [64]. At the time when the experiments are conducted, the cluster is a heterogeneous system consisting of 39 computing nodes connected with Gigabit Ethernet. The nodes are running Linux and using SLURM as the scheduler.

Four OrangeFS systems are started with 2,4,8,16 server nodes. The client nodes are distributed in the cluster and not using the server nodes. The number of client nodes is selected to be comparable to the number of server nodes. HPX runtime system is responsible for managing threads on the client nodes. Large number of blocks and files are picked to keep the system under its full capacity.

Several sets of test cases is performed in this experiment on HPX with and without PXFS module systems on top of OrangeFS file system and Network File System (NFS) [23].

- `w/r-5f-100x64KB`: writing or reading 5 files with 100 blocks and block size 64KB;
- `w/r-5f-100x1MB`: writing or reading 5 files with 100 blocks and block size 1MB;

- `w/r-100f-1x64KB`: writing or reading 100 files with 1 block and block size 64KB;
- `w/r-5f-1x100MB`: writing or reading 5 files with 1 block and block size 100MB;

These four file test cases can represent reading and writing small, medium and large block size cases in large scale scientific applications which read and create sets of data frequently.

As an alternative use case, the OrangeFS user interface is called directly from the benchmark. The user interface can only support 1 thread per client node and deploys a synchronous API set. In the PXFS module, multiple threads can be enabled and multi-core architecture can be utilized.

- **Results and Discussions**

The I/O throughput results of four test cases are presented in Fig. 3.14, Fig. 3.15, Fig. 3.16, Fig. 3.17, Fig. 3.18, Fig. 3.19, Fig. 3.20 and Fig. 3.21 respectively. The number of clients are chosen to be the same as the number of OrangeFS servers and 1.5x times number of OrangeFS servers.

It is crystal clear that the PXFS module has increased the throughput dramatically than the direct synchronous interface. For one thread, 24 clients and 16 OrangeFS servers, the PXFS has 10x speedup for `w-5f-100x64KB`, 20x speedup for `r-5f-100x64KB`, 4x speedup for `w-5f-100x1MB`, 3x speedup for `r-5f-100x1MB`, 1.5x speedup for `w-5f-1x100MB`, 2x speedup for `r-5f-1x100MB`, 2x speedup for `w-100f-1x64KB`,2x speedup for `r-100f-1x64KB`. When using multiple threads, `w-5f-100x64KB` nearly doubles the throughput, `w/r-100f-1x64KB` has better performances, `r-5f-100x64KB` and `w-5f-1x100MB` sees some downgrades, and `w/r-5f-100x1MB` and `r-5f-1x100MB` have little variations. It is indicated that the thread management can influence the throughput and `w-5f-100x64KB` case still has potentials for higher performance, while `r-5f-100x64KB` might have some thread contention and needs some fine tune.

In all of the four cases, PXFS has shown perfect scalability when the number of OrangeFS servers are changing from 2 through 16. The throughput increases at the same ratio as the number of OrangeFS servers. The great scalability is a strong argument for

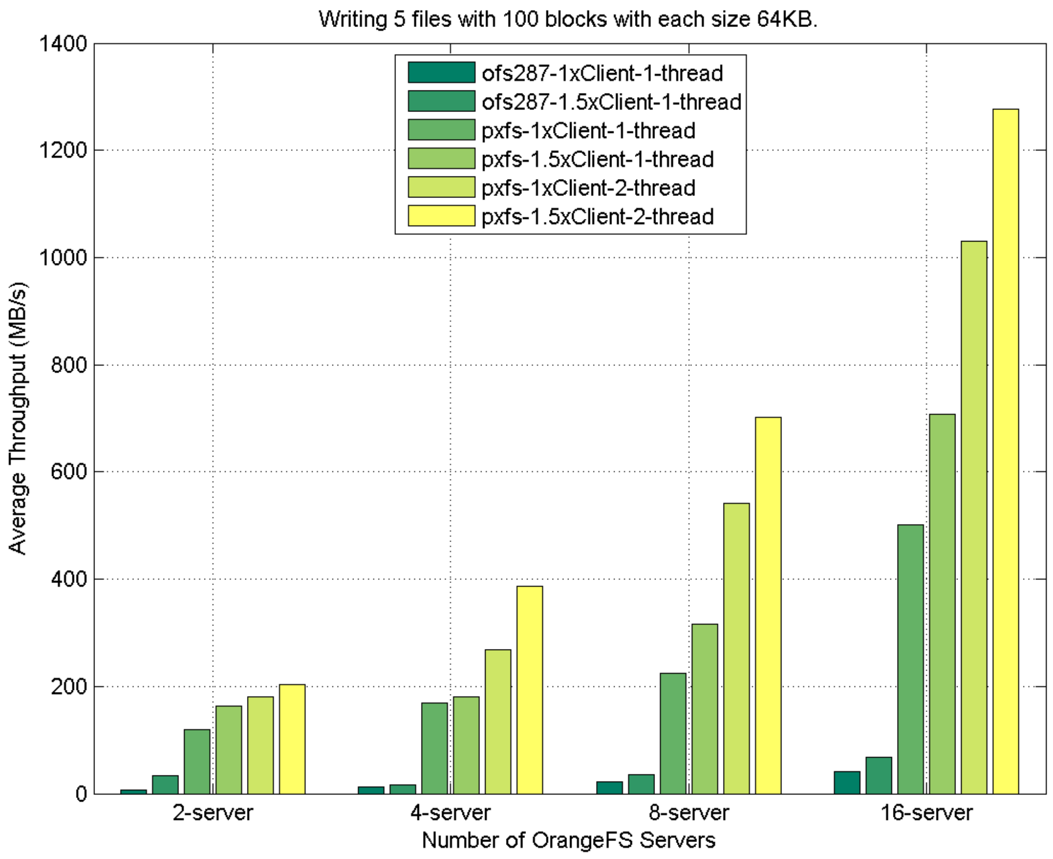


Figure 3.14: I/O performance of file test case `w-5f-100x64KB` between PXFS and direct interface on HPX and OrangeFS.

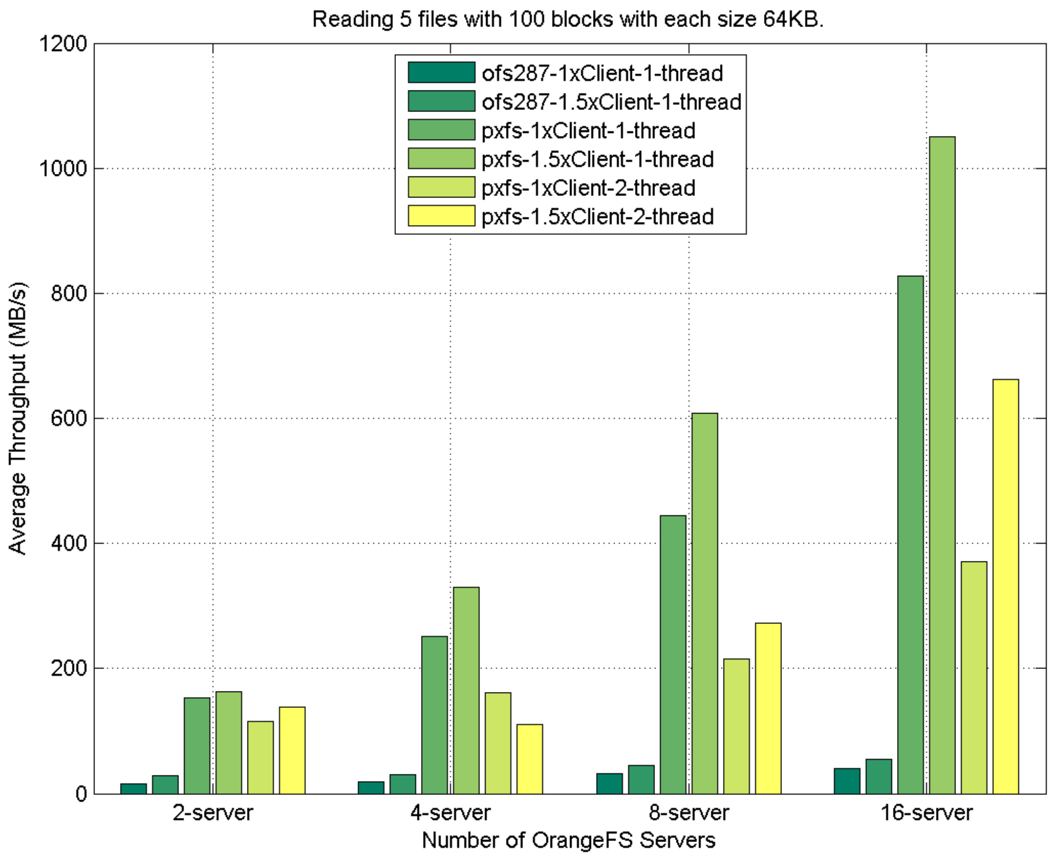


Figure 3.15: I/O performance of file test case `r-5f-100x64KB` between PXFS and direct interface on HPX and OrangeFS.

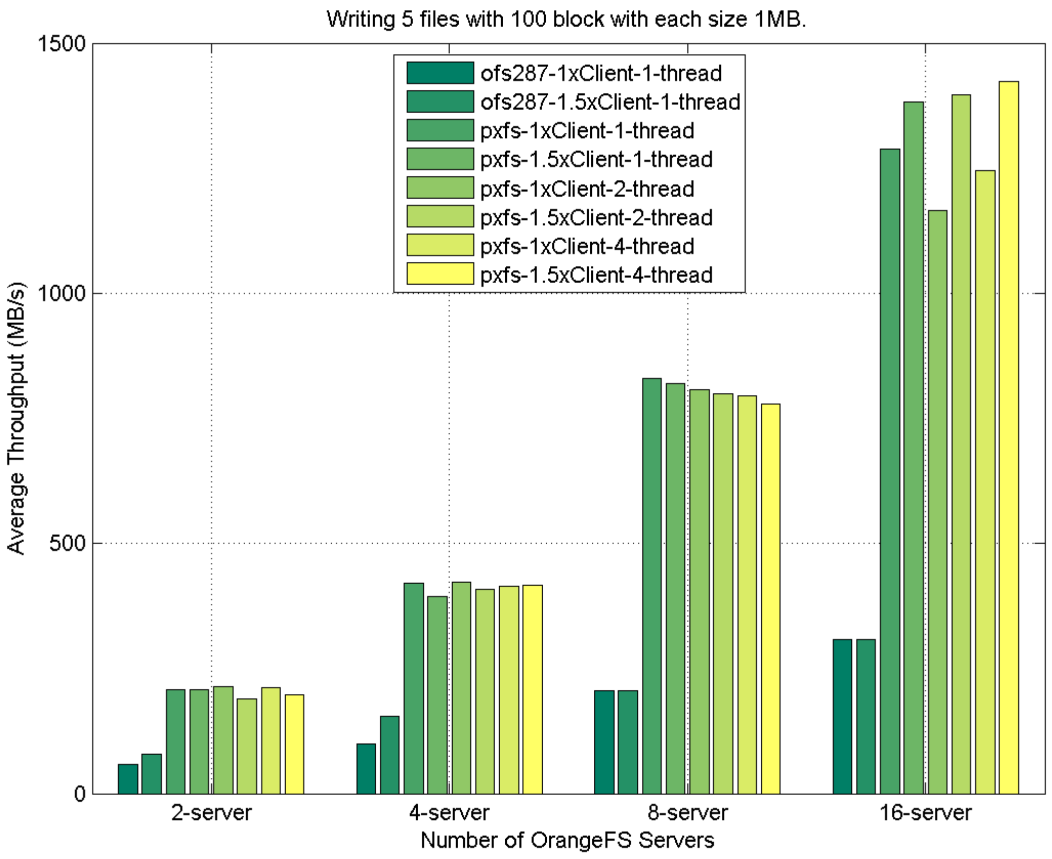


Figure 3.16: I/O performance of file test case `w-5f-100x1MB` between PXFS and direct interface on HPX and OrangeFS.

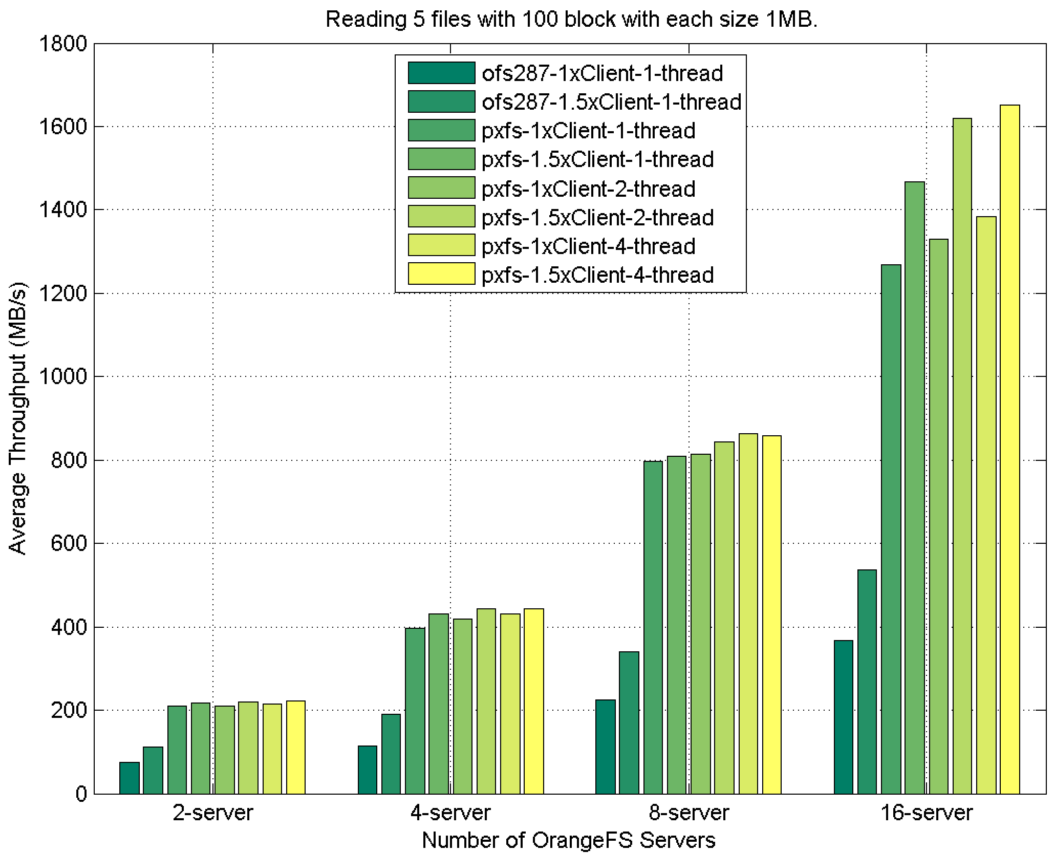


Figure 3.17: I/O performance of file test case `r-5f-100x1MB` between PXFS and direct interface on HPX and OrangeFS.

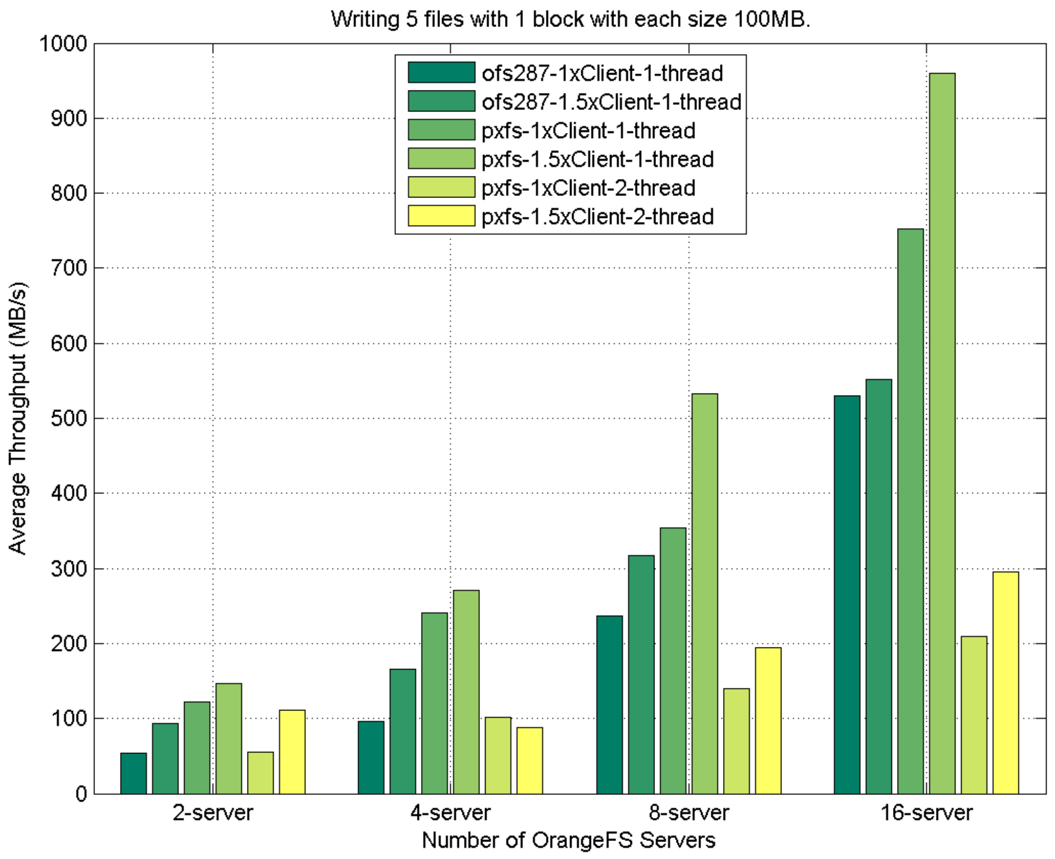


Figure 3.18: I/O performance of file test case `w-5f-1x100MB` between PXFS and direct interface on HPX and OrangeFS.

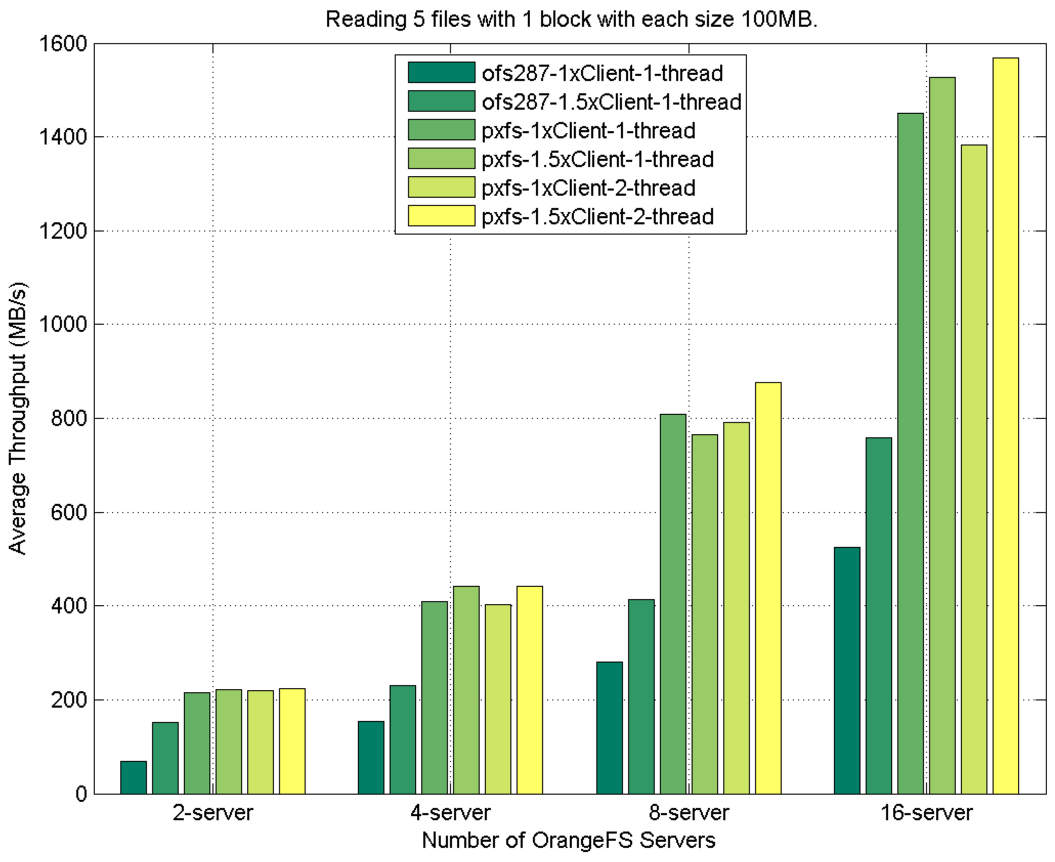


Figure 3.19: I/O performance of file test case `r-5f-1x100MB` between PXFS and direct interface on HPX and OrangeFS.

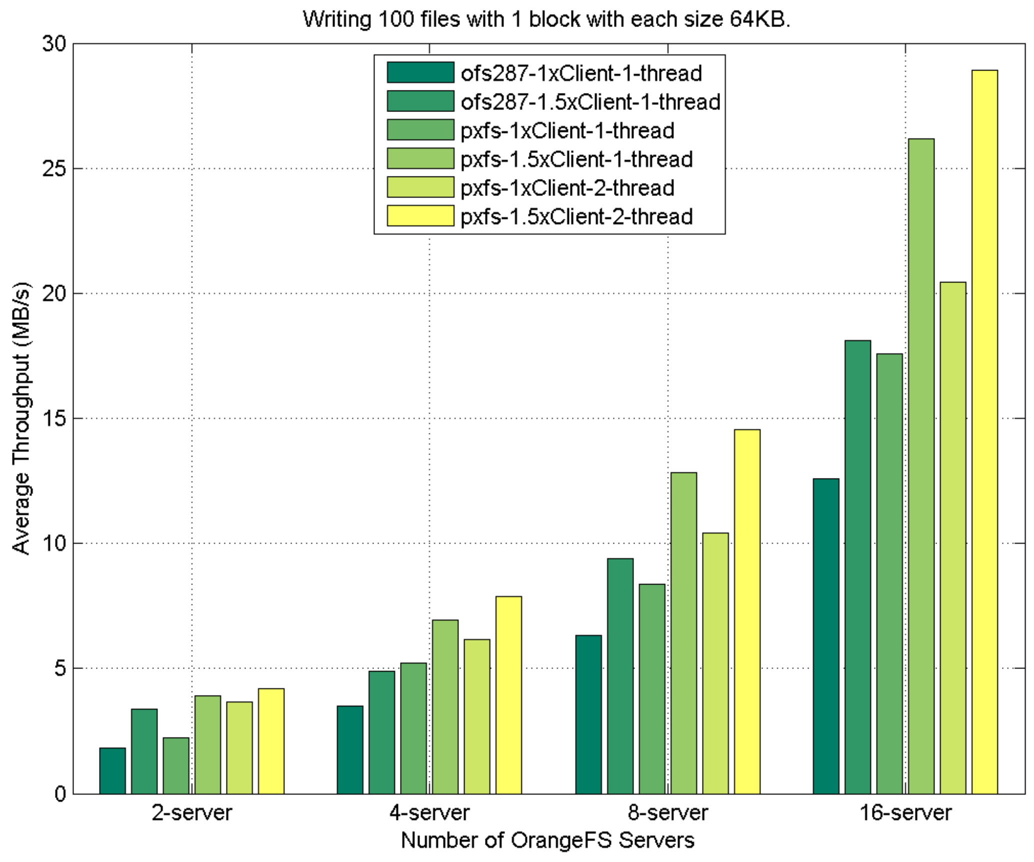


Figure 3.20: I/O performance of file test case `w-100f-1x64KB` between PXFS and direct interface on HPX and OrangeFS.

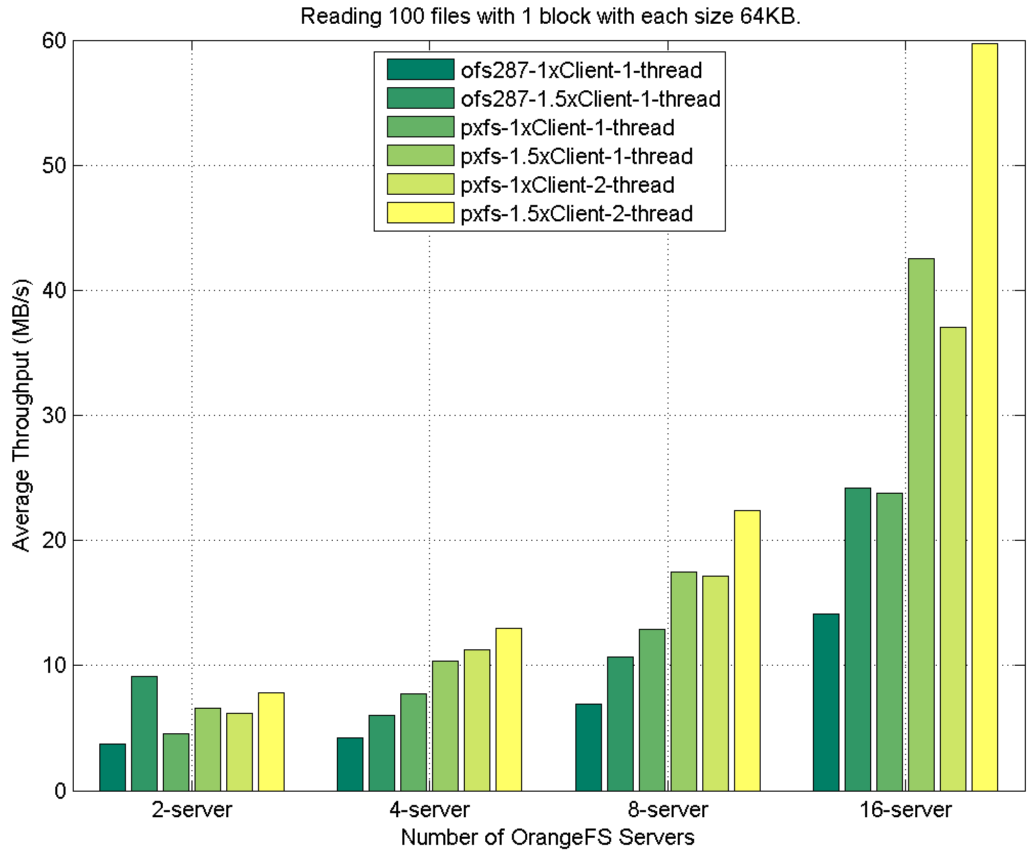


Figure 3.21: I/O performance of file test case `r-100f-1x64KB` between PXFS and direct interface on HPX and OrangeFS.

running large scale application on thousands of nodes now and in the future.

One more thing to mention is that sometimes the writing throughput is better than the reading throughput for the same file block size and test cases. One possible reason is that when doing write, it might write to cache first and then write to disk afterwards, while for reading they directly go to disk, which will cause congestion in the I/O system.

To compare the performance of a parallel file system and a centralized file system, the tests are also run on NFS with 24 HPX clients and the results are displayed in Fig. 3.22.

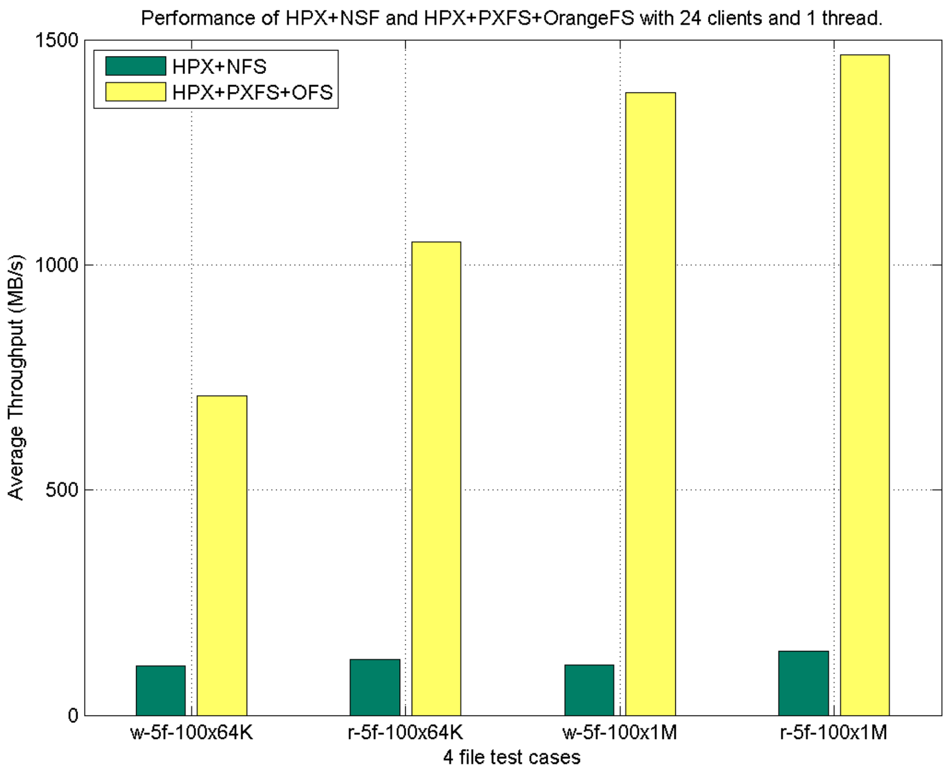


Figure 3.22: I/O performance of all four file test cases between HPX+PXFS module on OrangeFS and HPX on NFS.

As seen from the results, the performance with the PXFS model and a parallel system underneath greatly surpasses the NFS system with a 5 to 10 times improvement in I/O throughput with 24 HPX clients. It is shown that in Exascale era, the new storage model should take into account the experience of parallel file system development.

3.4.3 Asynchronous I/O Throughputs of HPX File Classes

As seen in Section 3.4.2, different number of threads is having a substantial influence on the I/O throughput performance. In this section, performances of the HPX asynchronous file classes (**local_file**, **orangefs_file** and **pxfs_file**) are measured using the method mentioned in Section 3.4.1. The purpose is to compare the throughputs of different asynchronous file classes and their performance under different number of threads.

- **Experiment Setup**

All the tests are executed on the Palmetto Cluster housed by Clemson Computing and Information Technology [65]. At the time when the experiment was conducted, the status of Palmetto cluster is listed as below:

- Both multicore large-shared-memory systems and standard multicore distributed-memory configurations available
- Jobs allocated on a per-core basis (i.e., small jobs may share nodes with other jobs)
- Benchmarked operating at a sustained 551 TFlops using 276 nodes
- Ranked #4 among academic research clusters on June 2014 Top 500 list
- Currently comprises 1,978 compute nodes (20,728 cores) + 598 NVIDIA Tesla GPU accelerators
- InfiniBand and Myrinet network interconnects
- Operating system on all nodes: Scientific Linux 6 (based on RedHat Enterprise Linux)
- Job queuing system: PBS Professional 12.0
- Connected to Internet2's 100 GbE (gigabit Ethernet) Advanced Layer 2 Service

Three OrangeFS systems are started with 4,8,16 server nodes and the number of client nodes are set to twice the number of servers, meaning 8, 16 and 32 client nodes respectively.

The client nodes are distributed in the cluster and not using the server nodes. HPX runtime system is responsible for managing threads on the client nodes. Different test cases of files which has different number and size of file blocks are picked to evaluate the system from different prospectives.

Several sets of test cases is performed in this experiment on HPX with **local_file**, **orangefs_file** and **pxfs_file** asynchronous interfaces on top of OrangeFS file system and Network File System (NFS) [23].

- `w/r-50f-100x64KB-Xp`: X threads on each client nodes, writing or reading 50 files with 100 blocks and block size 64KB;
- `w/r-50f-10x64KB-Xp`: X threads on each client nodes, writing or reading 50 files with 100 blocks and block size 64KB;
- `w/r-5f-1x100MB-Xp`: X threads on each client nodes, writing or reading 5 files with 1 block and block size 100MB;
- `w/r-5f-100x1MB-Xp`: X threads on each client nodes, writing or reading 5 files with 100 blocks and block size 1MB;

These four file test cases can represent reading and writing small, medium and large block size cases in large scale scientific applications which read and create sets of data frequently.

All test cases are run with varying number of threads ranging from 1 to 4 to test the throughput under multiple threads in detail.

First of all, the horizontal comparison of all three file classes are conducted to measure the scalability and absolute throughput of these asynchronous interfaces.

Secondly, another bunch of experiments are carried out to study the influences of number of I/O threads in each file class and test cases. This is to show if multiple threads per node will affect the I/O throughput or not.

The results are shown and analysed in the next few sections.

- **Horizontal Comparison: All Asynchronous File Classes with 1 Thread**

The throughput results of the asynchronous file classes with 1 thread and all the test cases are illustrated in Figure 3.23, Figure 3.24, Figure 3.25, Figure 3.26, Figure 3.27, Figure 3.28, Figure 3.29 and Figure 3.30.

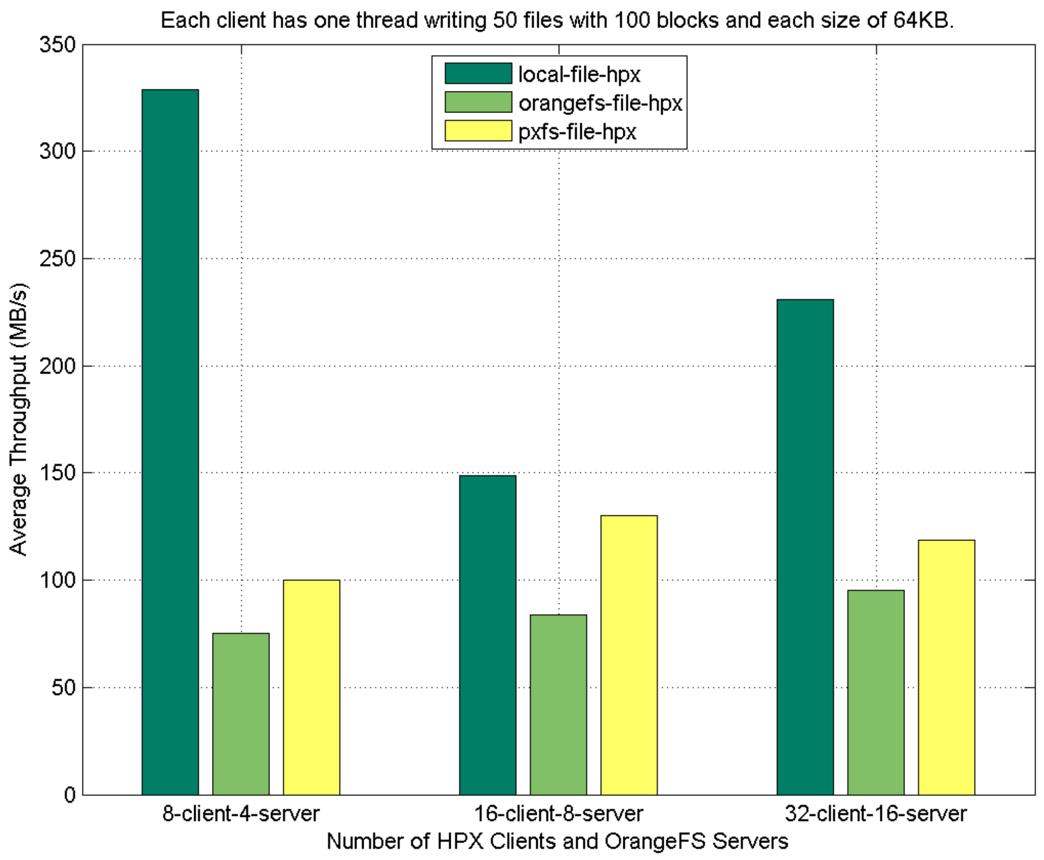


Figure 3.23: I/O performance of file test case `w-50f-100x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

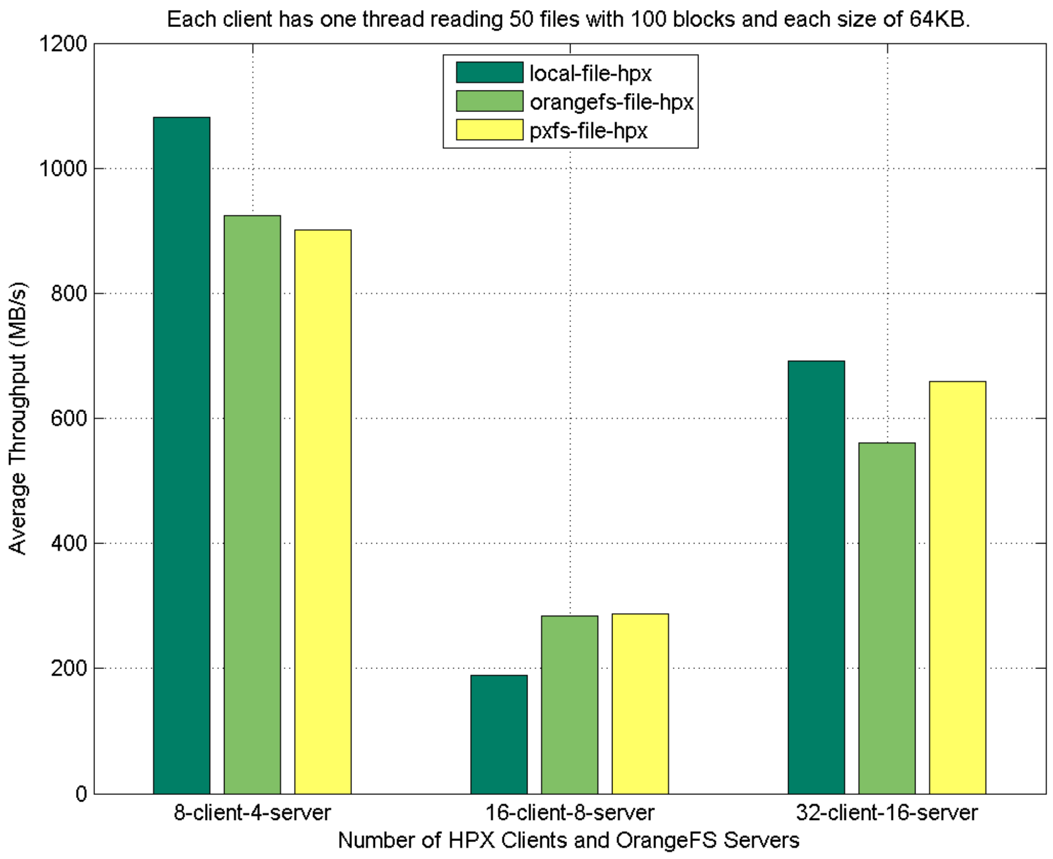


Figure 3.24: I/O performance of file test case `r-50f-100x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

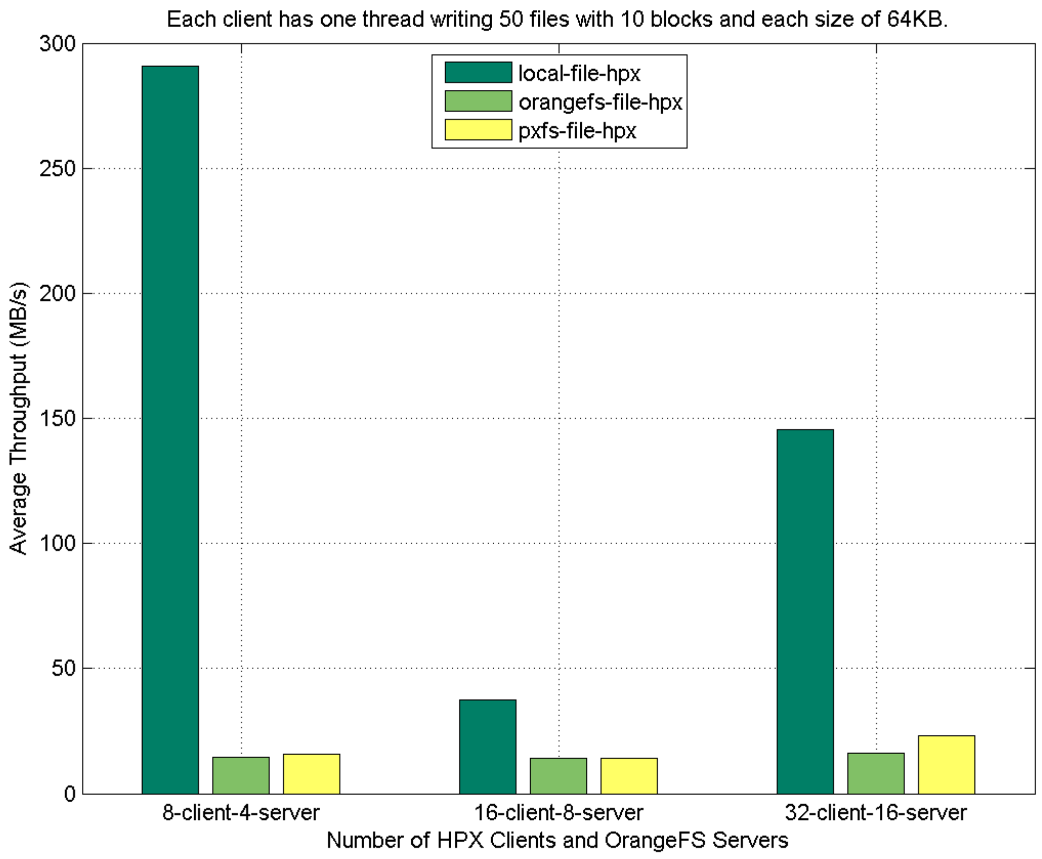


Figure 3.25: I/O performance of file test case `w-50f-10x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

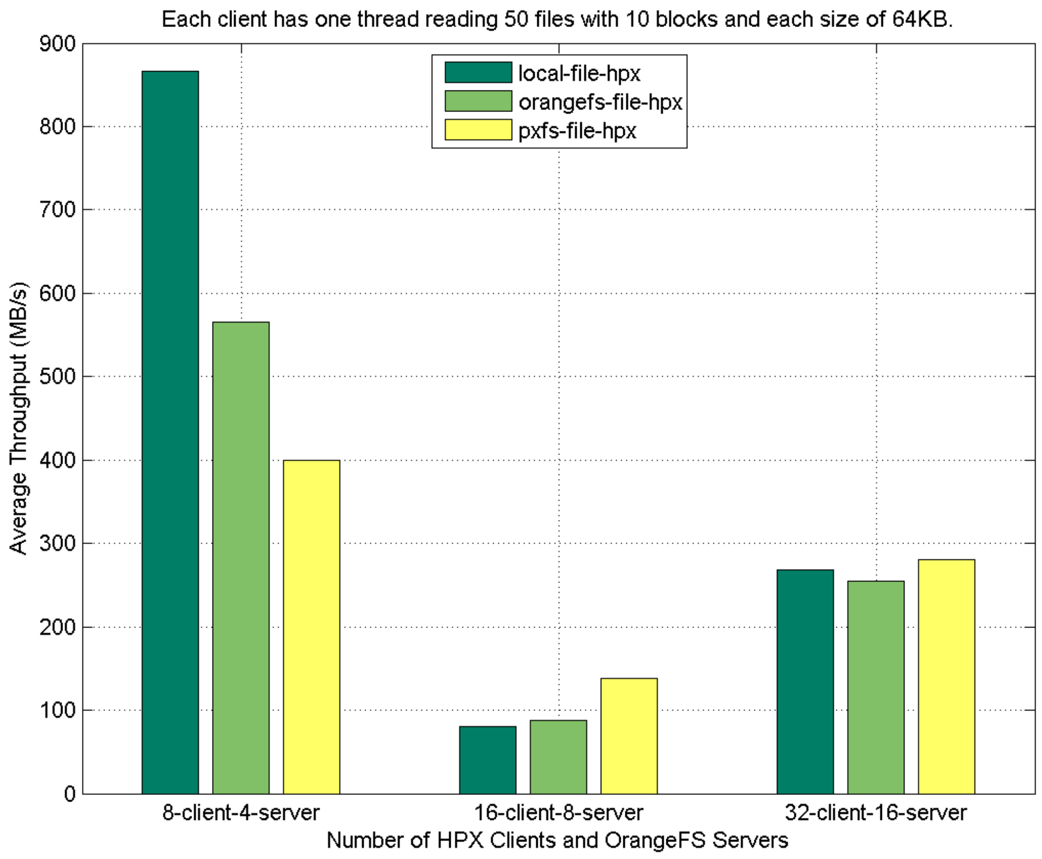


Figure 3.26: I/O performance of file test case `r-50f-10x64KB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

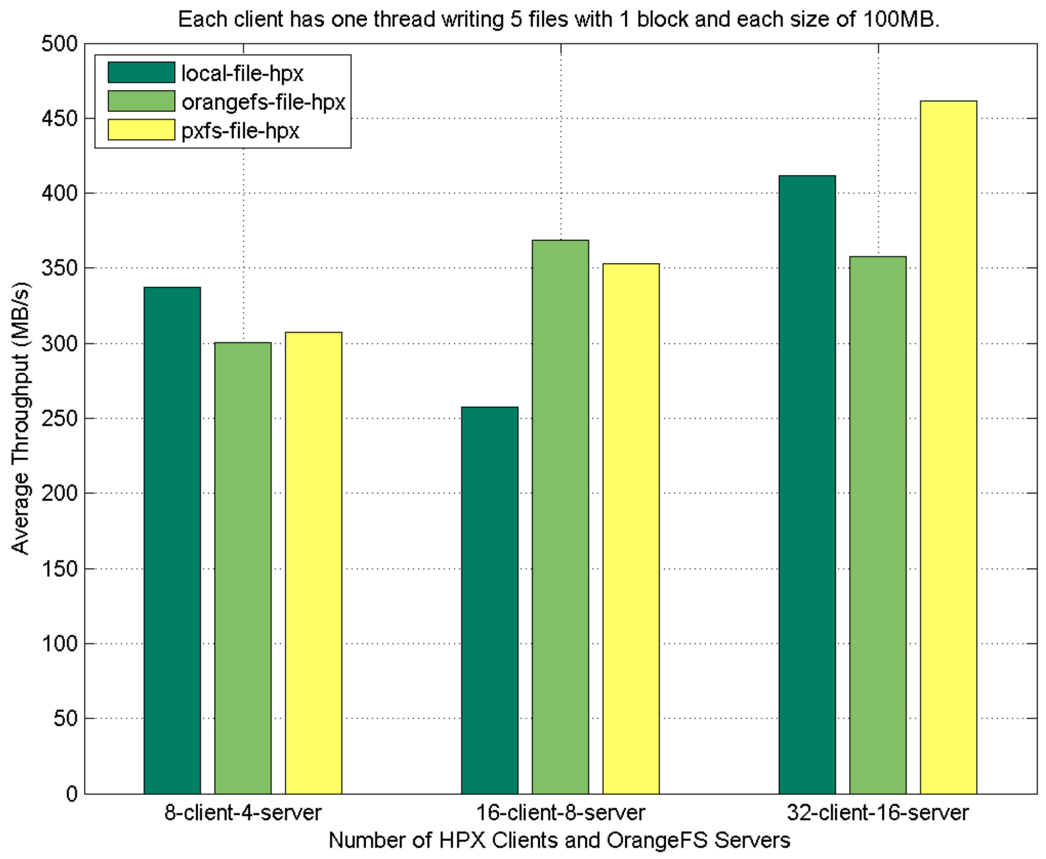


Figure 3.27: I/O performance of file test case `w-5f-1x100MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

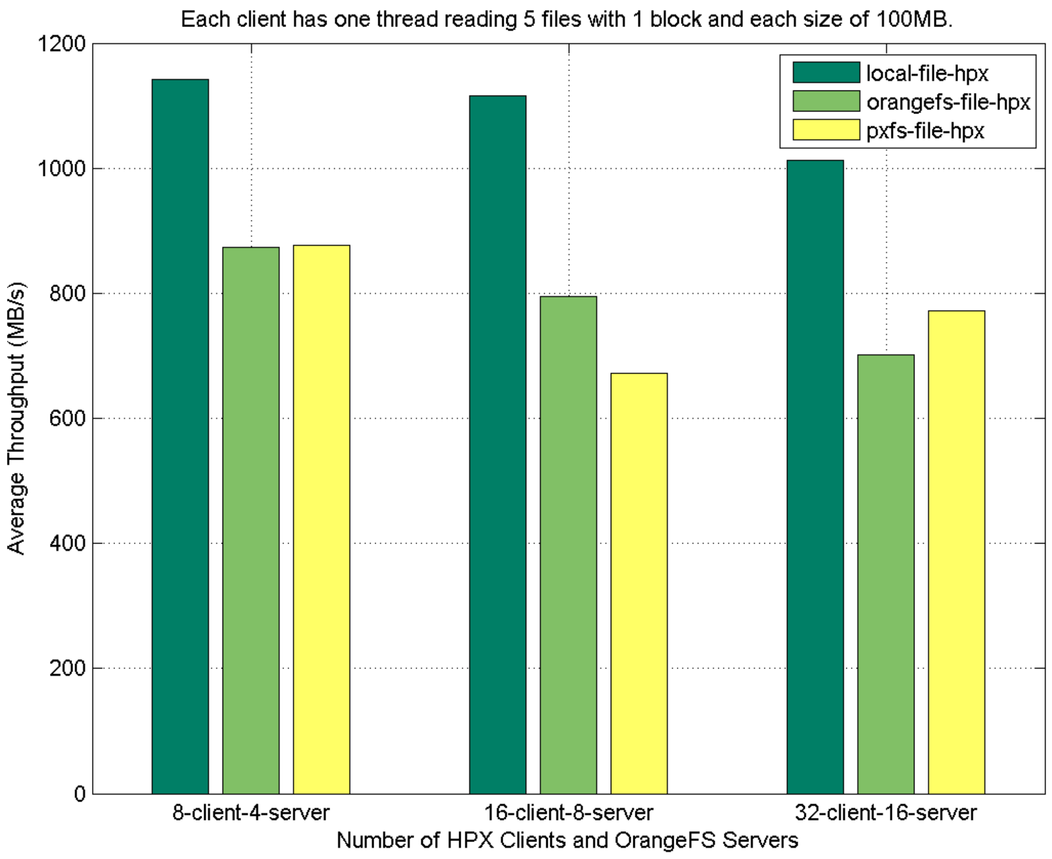


Figure 3.28: I/O performance of file test case `r-5f-1x100MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

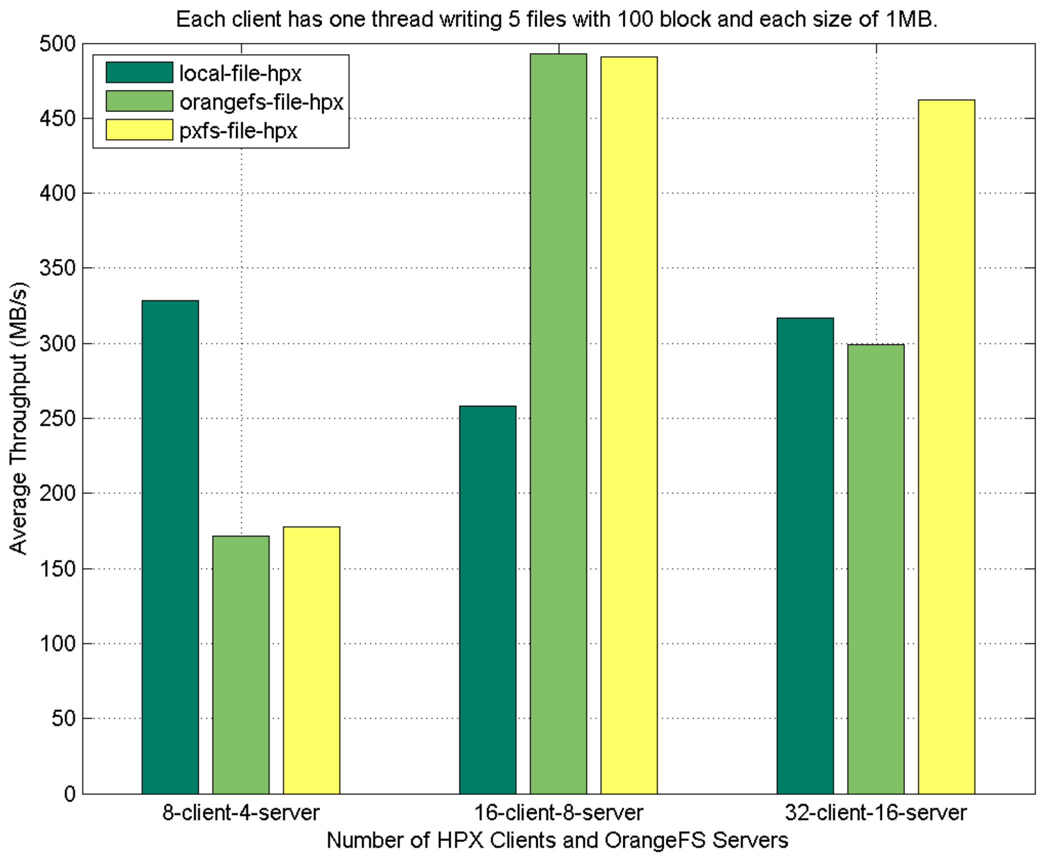


Figure 3.29: I/O performance of file test case `w-5f-100x1MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

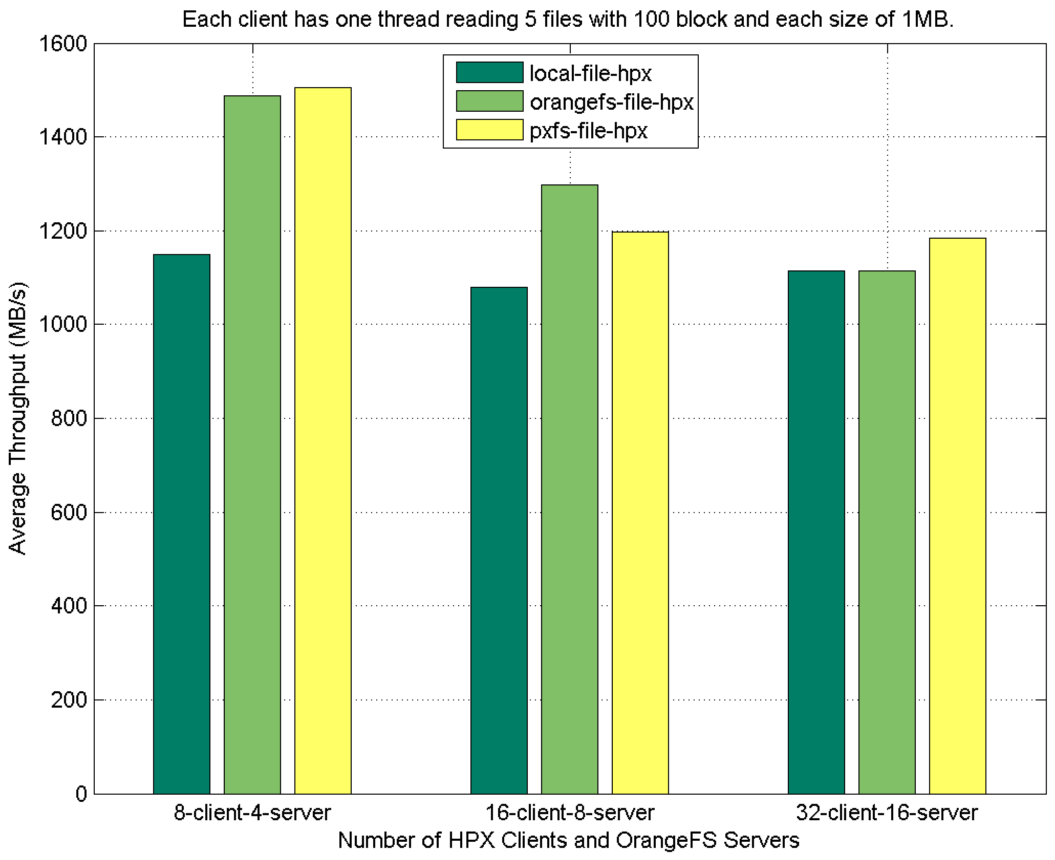


Figure 3.30: I/O performance of file test case `r-5f-100x1MB-1p` of HPX local-file, orangefs-file and pxfs-file asynchronous interfaces.

The following observations about these results can be made without much effort.

- Writing throughput is much less than reading in all cases because writing data to disk cost more time.

- For small file block sizes like $64KB$ in the first 2 test cases, **local_file** performs much better because it does not have a lot of overhead for small files like in parallel file systems. However, **orangefs_file** and **pxfs_file** are improving with more client nodes and data servers, while **local_file** has less throughput with more client nodes and data servers. This clearly shows a lack of scalability for **local_file** class.

- For large file block sizes ($100MB$), **orangefs_file** and **pxfs_file** are doing much better than small files and can be comparable or better than the **local_file** class. This is due to the parallel nature of OrangeFS to distribute large files on multiple data servers.

- For medium file block sizes ($1MB$), **orangefs_file** and **pxfs_file** are doing great on writing comparing to **local_file**, and slightly better on reading medium file blocks. This might due to the reason that OrangeFS has a better balance of communication overhead and actual I/O operations for medium file blocks.

- These files shows small scalability for **orangefs_file** and **pxfs_file** for reading and writing in 4,8,16 data servers. The reason might be the small number of data servers and clients in these test cases. More servers causes instability of the benchmark and did not return reasonable results.

- In all cases, **pxfs_file** is showing better performance of **orangefs_file** in I/O operations. The difference between them is that **orangefs_file** handles asynchronism in HPX runtime, while **pxfs_file** deals with asynchronism on OrangeFS side. The better performance of **pxfs_file** indicates that moving the asynchronism closer to the actual data will improve the I/O throughput.

- **Vertical Comparison: local_file class with Different Number of Threads**

The throughput results of the asynchronous **local_file** class with varying number of threads on all the test cases are displayed in Figure 3.31, Figure 3.32, Figure 3.33, Figure 3.34, Figure 3.35, Figure 3.36, Figure 3.37 and Figure 3.38.

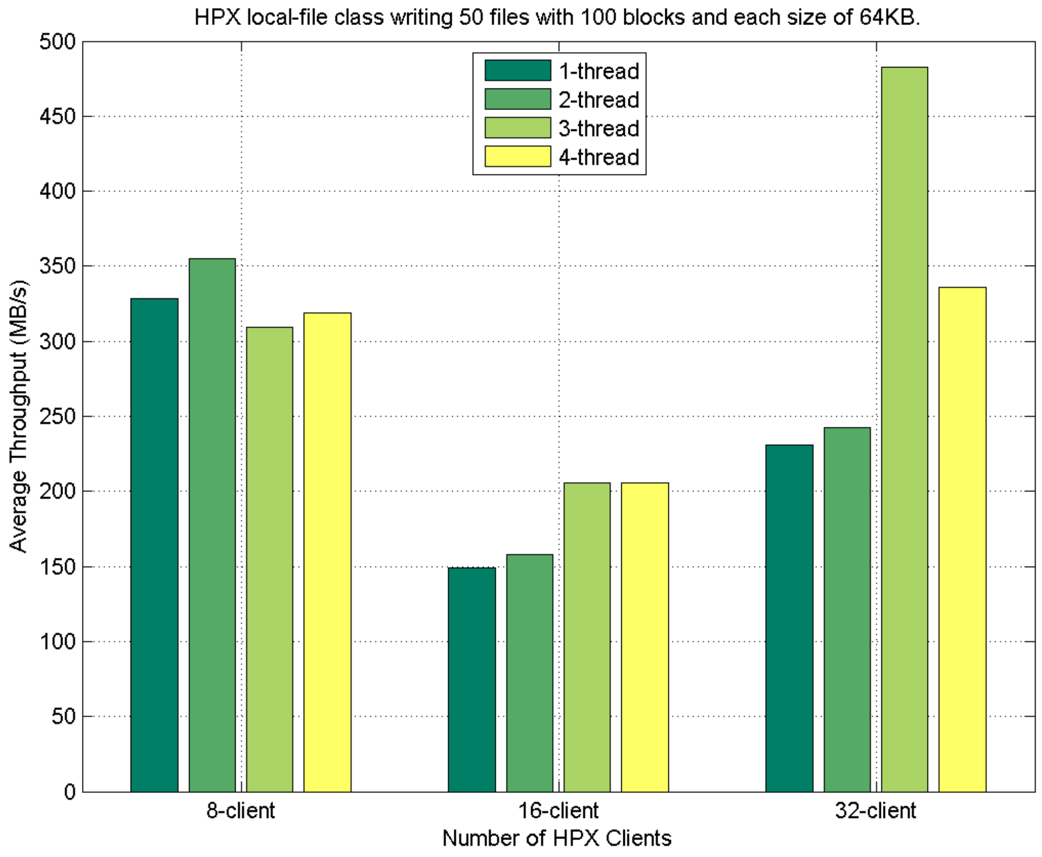


Figure 3.31: HPX local-file asynchronous I/O performance of test case `w-50f-100x64KB` with different number of threads on client nodes.

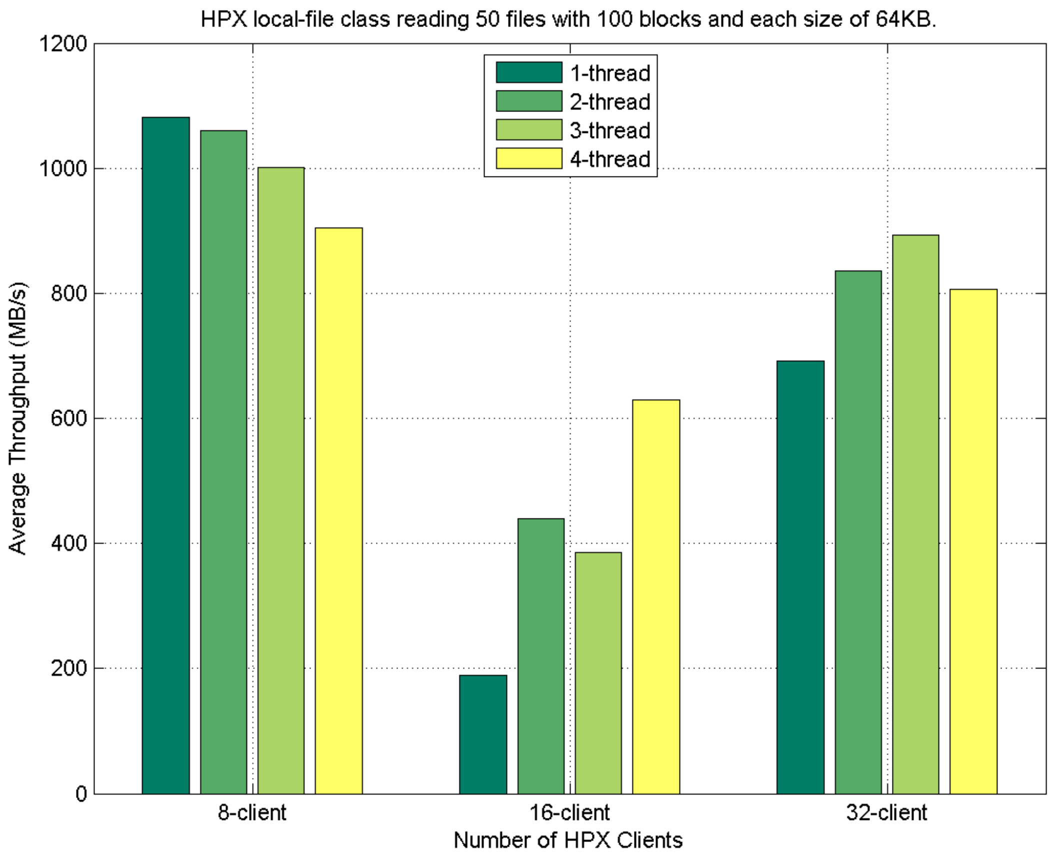


Figure 3.32: HPX local-file asynchronous I/O performance of test case `r-50f-100x64KB` with different number of threads on client nodes.

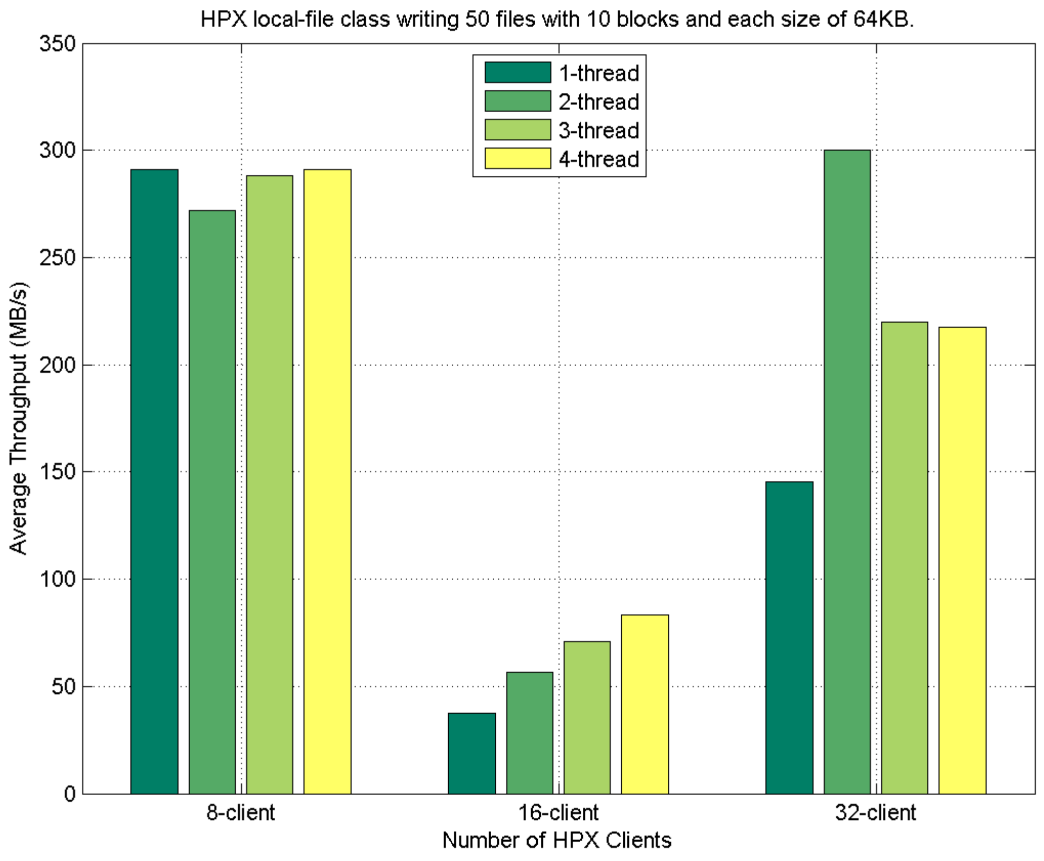


Figure 3.33: HPX local-file asynchronous I/O performance of test case `w-50f-10x64KB` with different number of threads on client nodes.

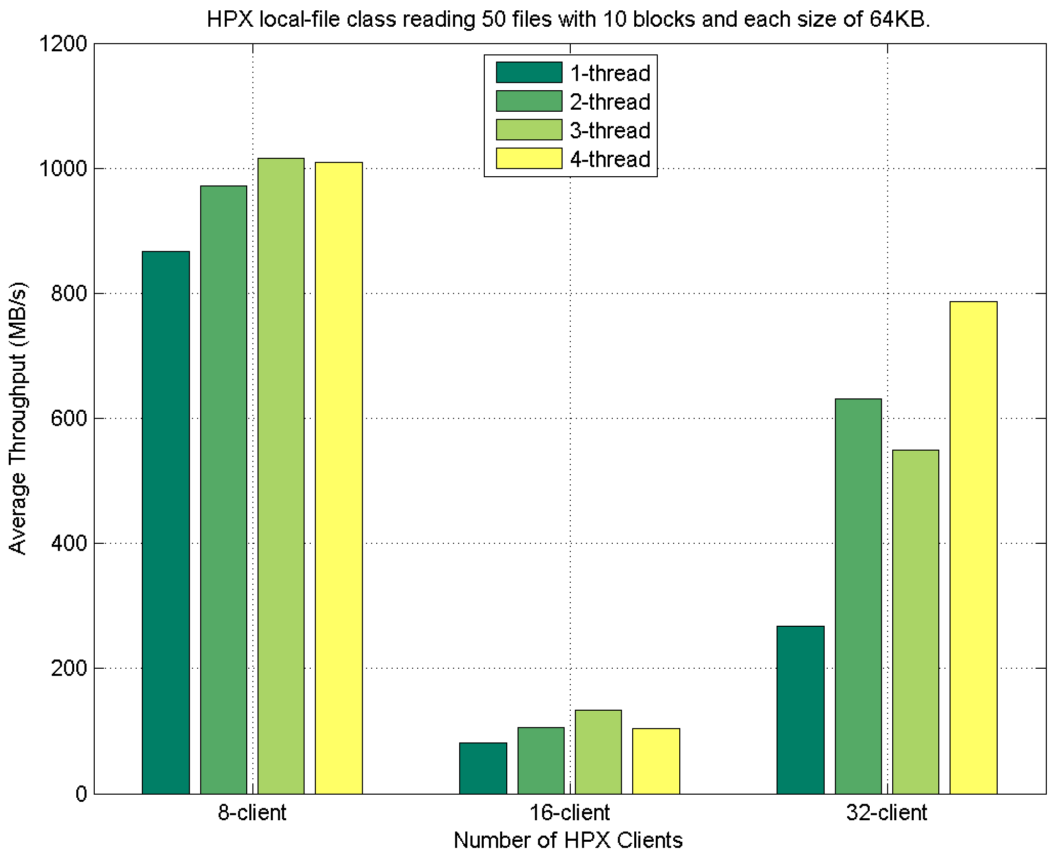


Figure 3.34: HPX local-file asynchronous I/O performance of test case `r-50f-10x64KB` with different number of threads on client nodes.

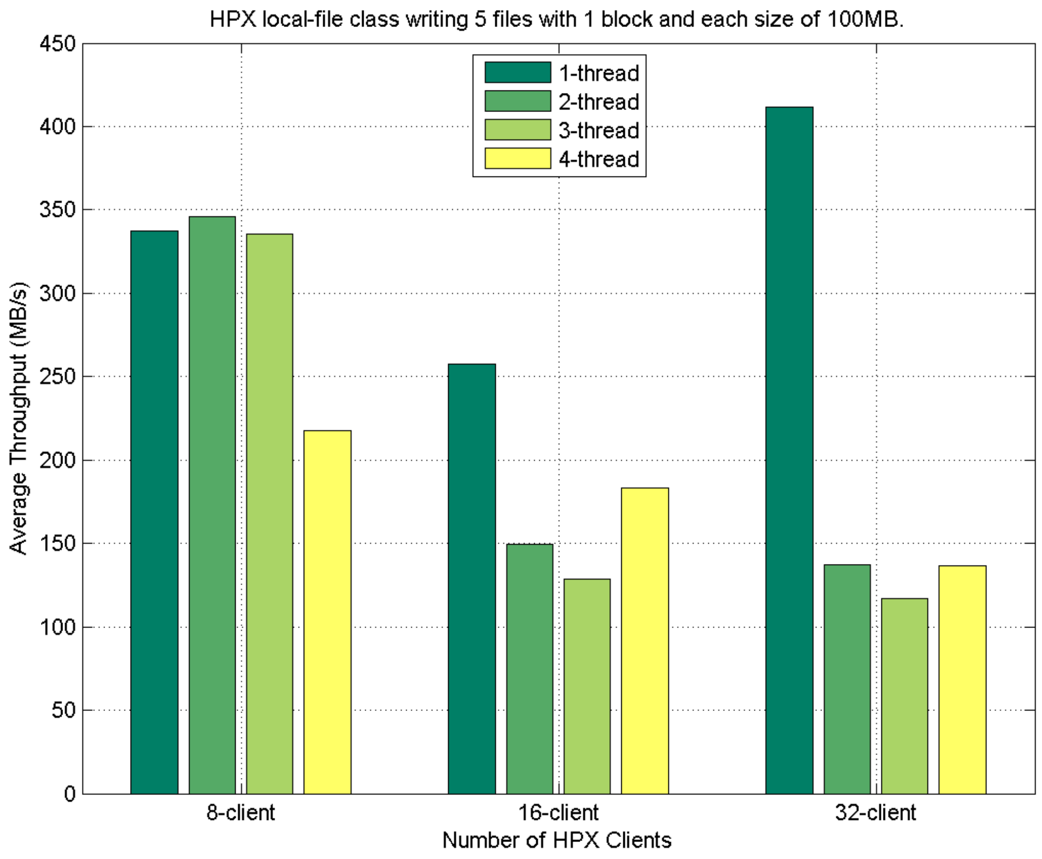


Figure 3.35: HPX local-file asynchronous I/O performance of test case `w-5f-1x100MB` with different number of threads on client nodes.

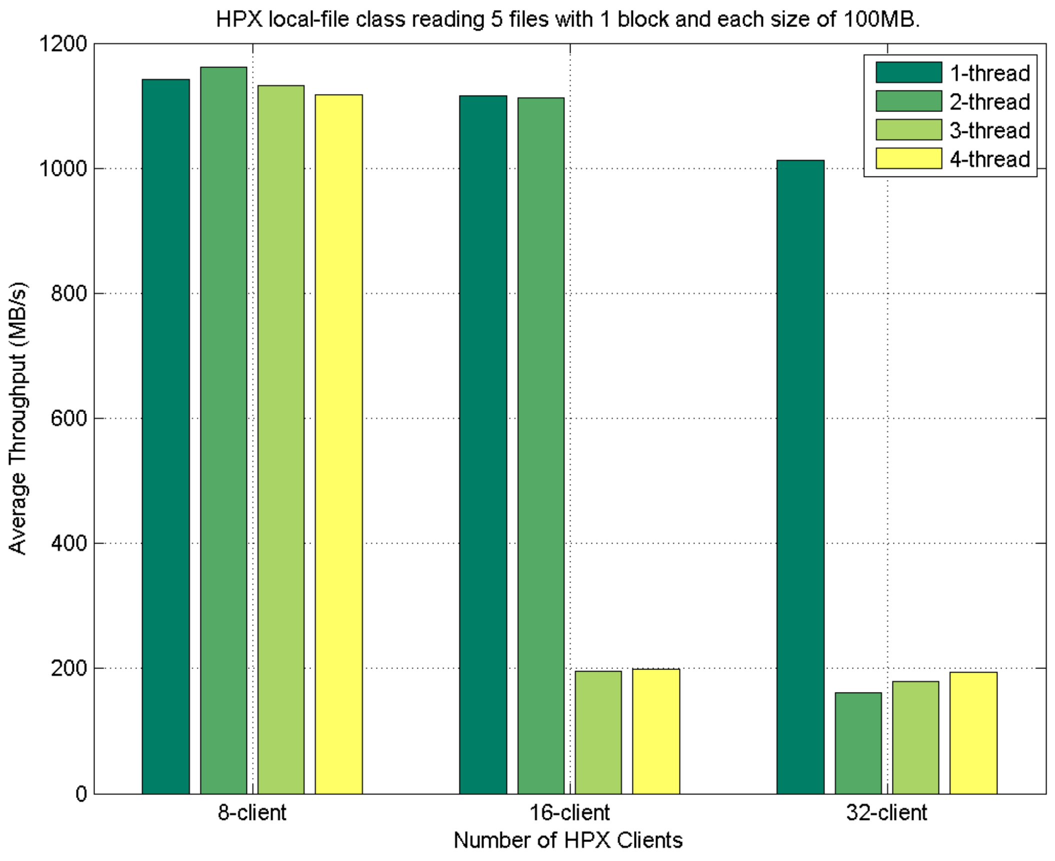


Figure 3.36: HPX local-file asynchronous I/O performance of test case `r-5f-1x100MB` with different number of threads on client nodes.

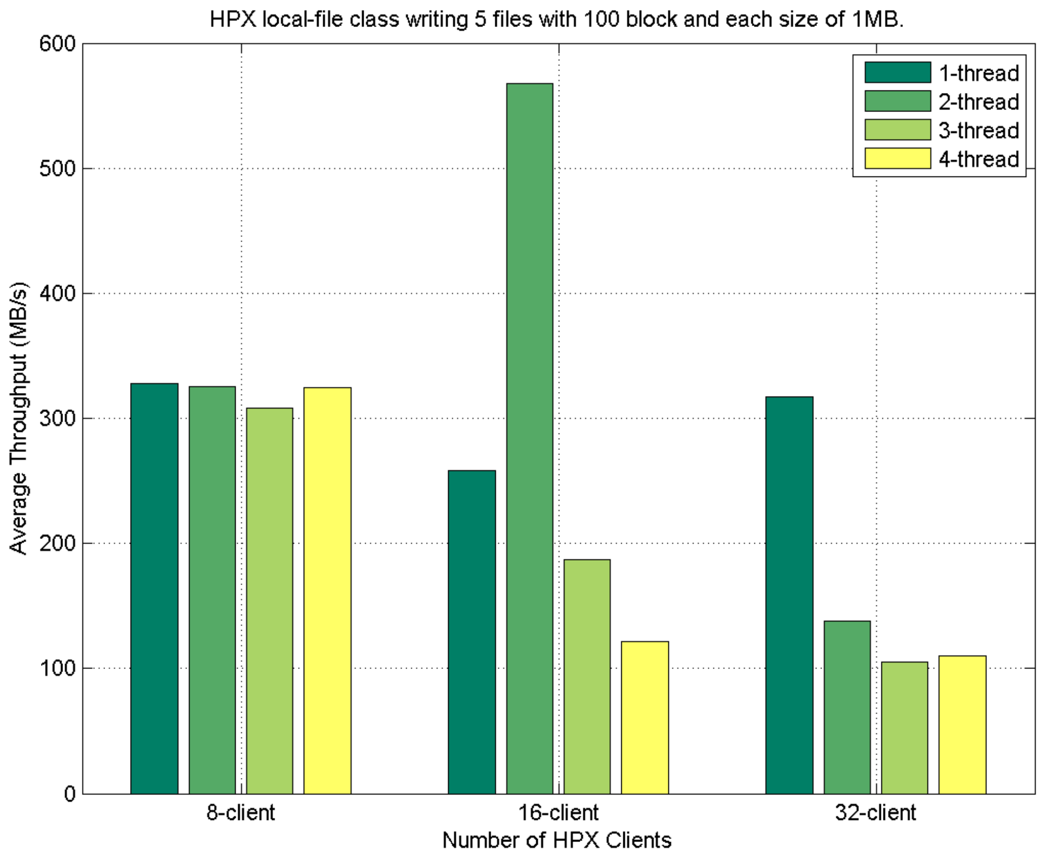


Figure 3.37: HPX local-file asynchronous I/O performance of test case `w-5f-100x1MB` with different number of threads on client nodes.

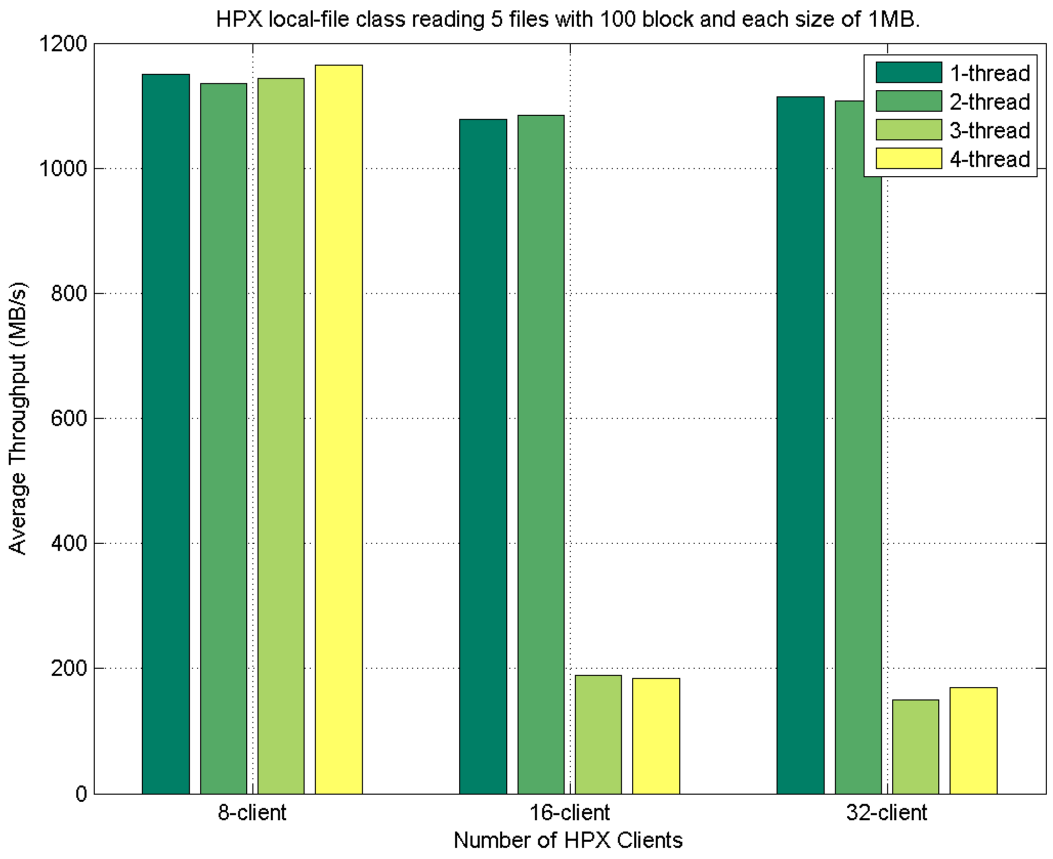


Figure 3.38: HPX local-file asynchronous I/O performance of test case `r-5f-100x1MB` with different number of threads on client nodes.

The following observations of these diagrams can be made without much efforts.

- For small file block size ($64KB$), **local_file** does not see a lot of variation of the throughputs when increasing the number of threads for different number of clients. In reading, increasing the number of threads will improve the throughput a little bit.

- For large file block size ($100MB$) and medium file block size ($1MB$), increasing the number of threads will largely downgrade the reading and writing throughputs. And with more clients, smaller number of threads are necessary to maintain a good throughput.

Thus in conclusion, the number of threads will largely affect the reading and writing throughput of **local_file** asynchronous classes, and with more clients less number of threads are allowed to prevent poor performances.

- **Vertical Comparison: orangefs_file class with Different Number of Threads**

The throughput results of the asynchronous **orangefs_file** class with varying number of threads on all the test cases are displayed in Figure 3.39, Figure 3.40, Figure 3.41, Figure 3.42, Figure 3.43, Figure 3.44, Figure 3.45 and Figure 3.46.

The following observations of these diagrams can be made without much efforts.

- For all cases of file block size ($64KB, 1MB, 100MB$), **orangefs_file** does not see a lot of variation of the throughputs when increasing the number of threads for different number of clients. In reading, increasing the number of threads will improve the throughput.

- More data servers shows near-scalable performance for writing and comparable performance for reading. It might be caused by the small number of OrangeFS data servers and small number of client nodes. These test cases did not reach the point where scalability can be shown.

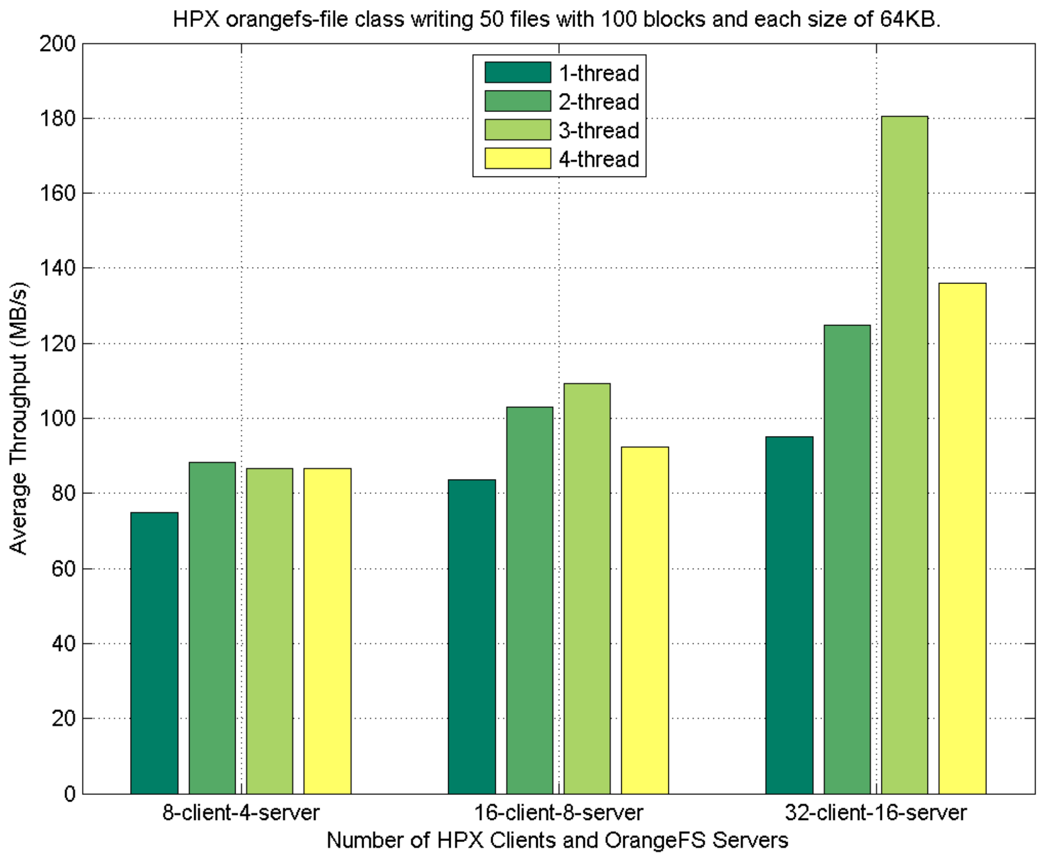


Figure 3.39: HPX orangefs-file asynchronous I/O performance of test case `w-50f-100x64KB` with different number of threads on client nodes.

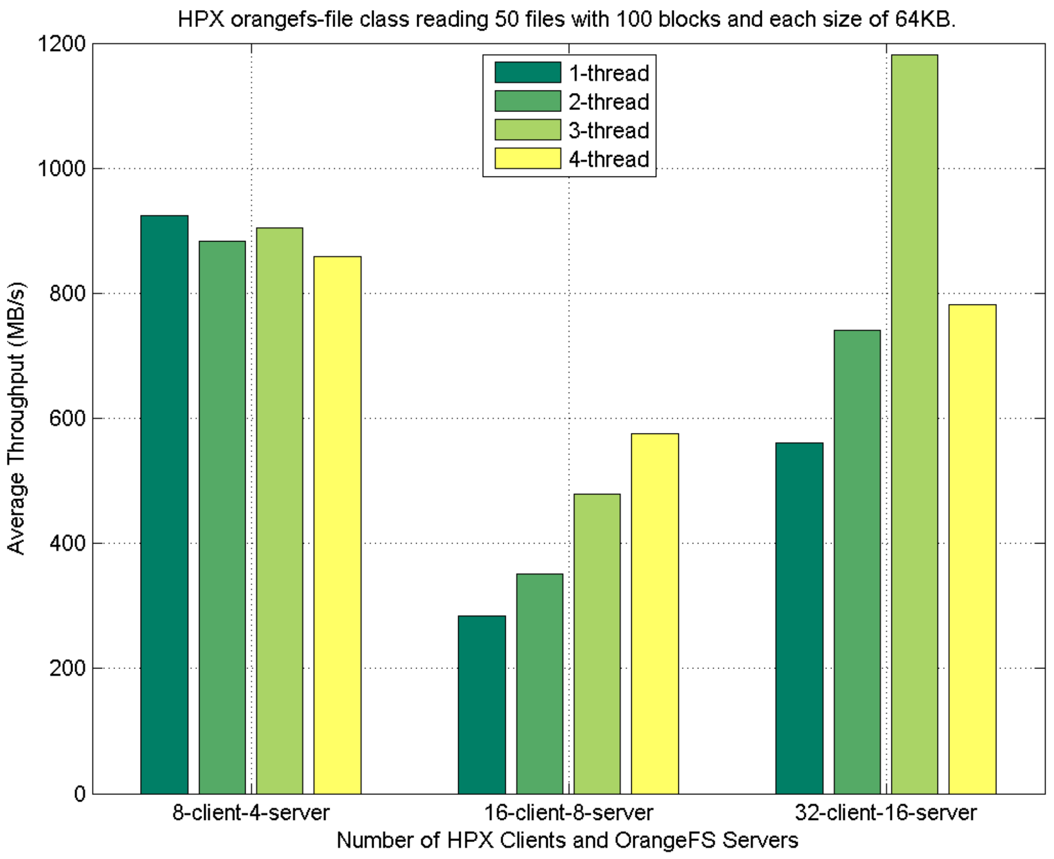


Figure 3.40: HPX orangefs-file asynchronous I/O performance of test case `r-50f-100x64KB` with different number of threads on client nodes.

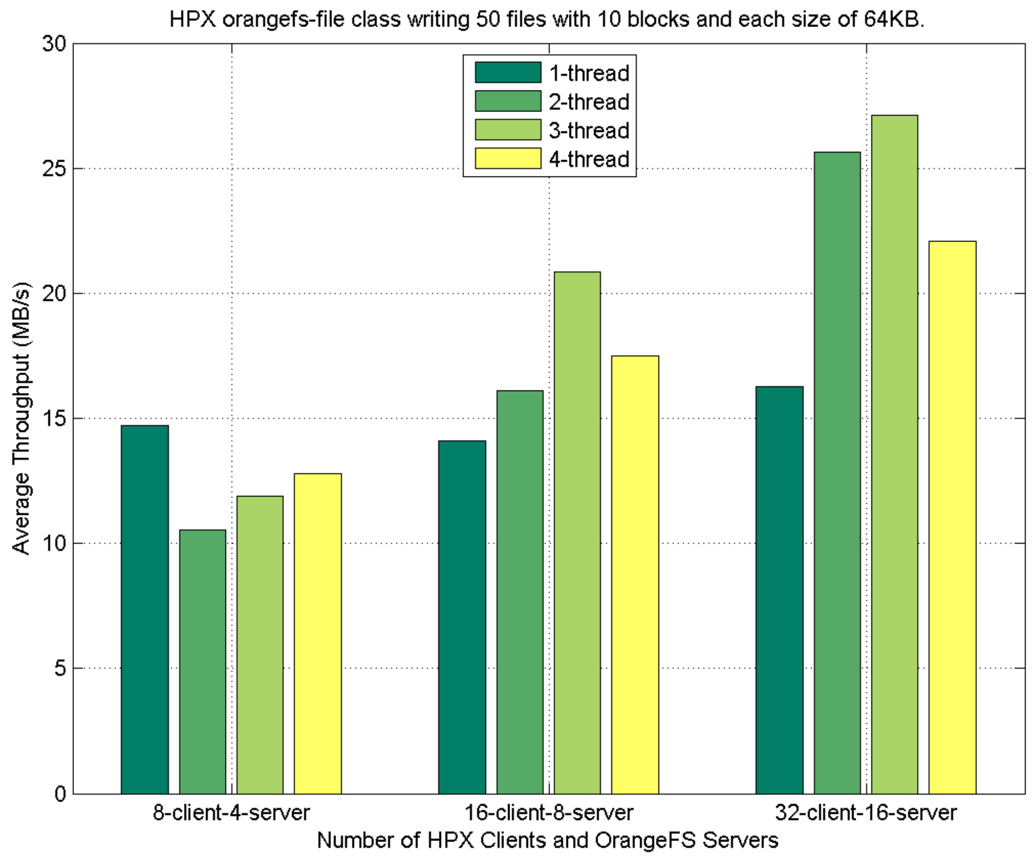


Figure 3.41: HPX orangefs-file asynchronous I/O performance of test case `w-50f-10x64KB` with different number of threads on client nodes.

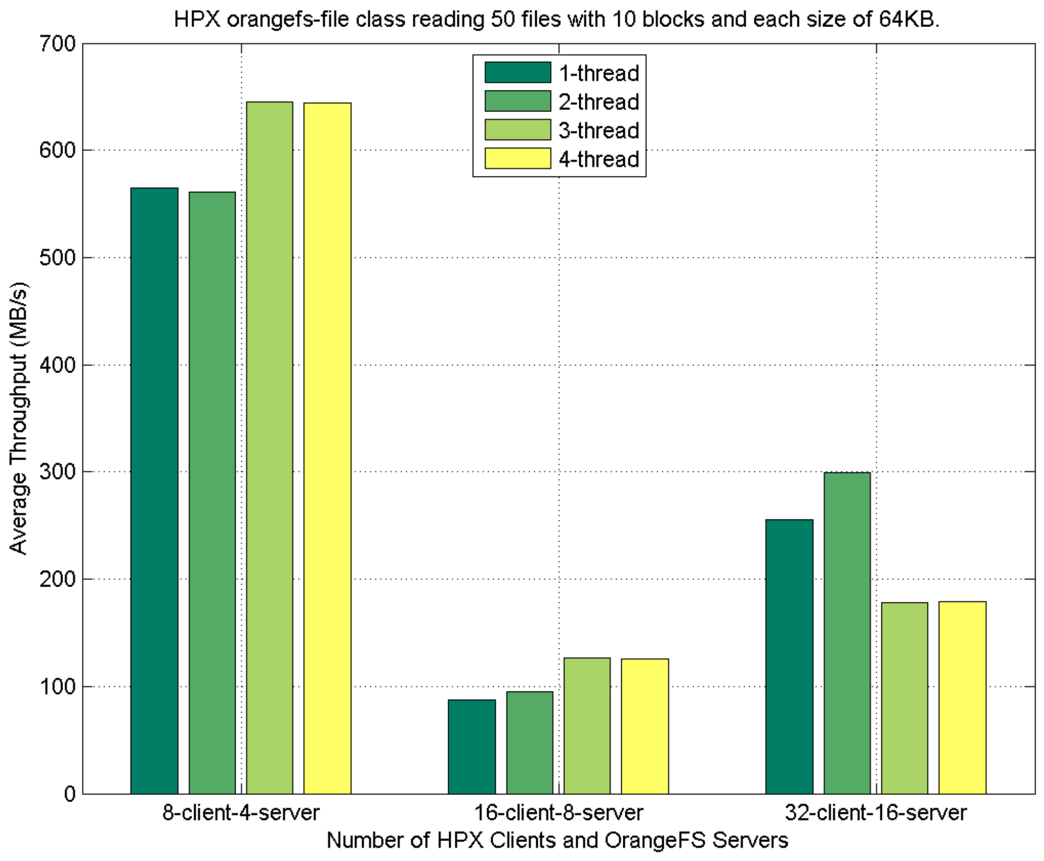


Figure 3.42: HPX orangefs-file asynchronous I/O performance of test case `r-50f-10x64KB` with different number of threads on client nodes.

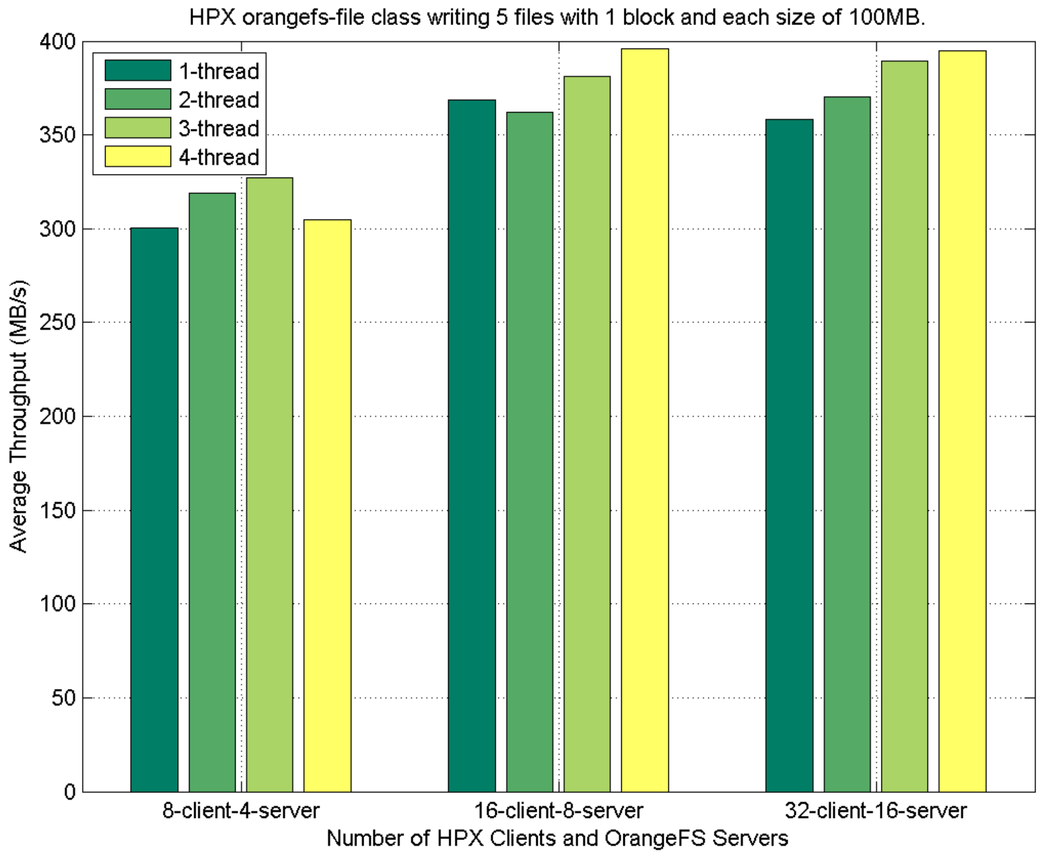


Figure 3.43: HPX orangefs-file asynchronous I/O performance of test case `w-5f-1x100MB` with different number of threads on client nodes.

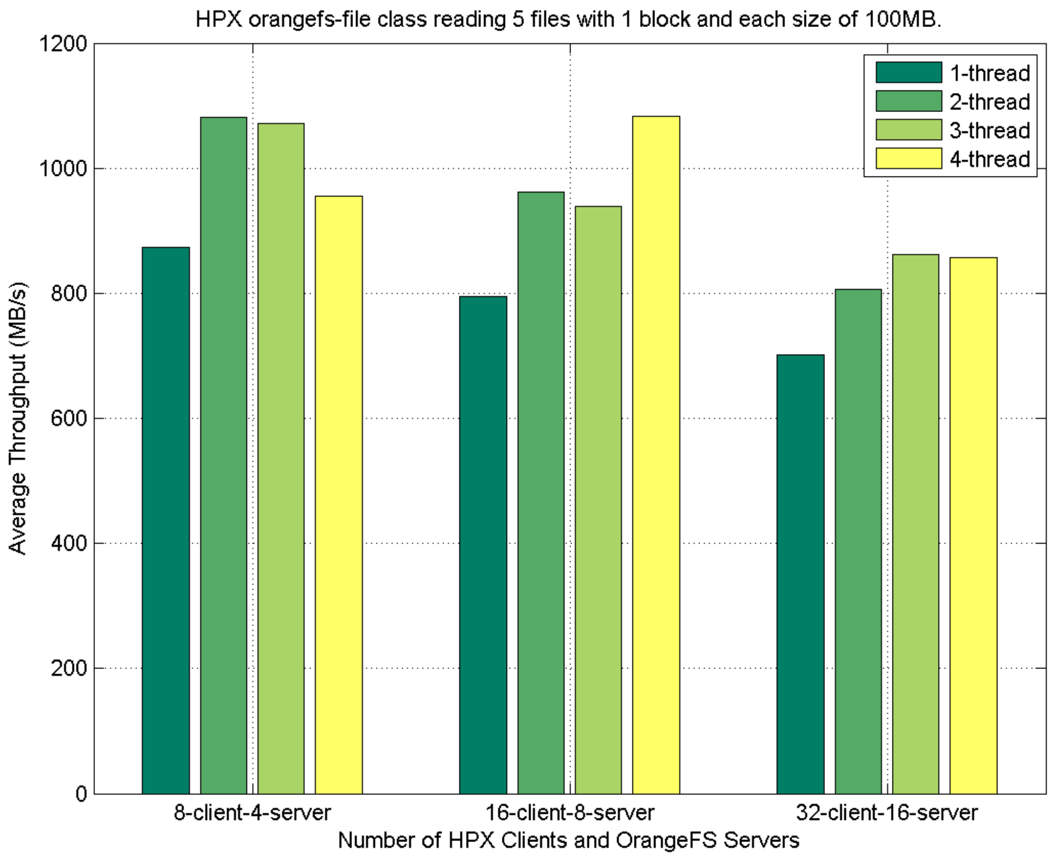


Figure 3.44: HPX orangefs-file asynchronous I/O performance of test case `r-5f-1x100MB` with different number of threads on client nodes.

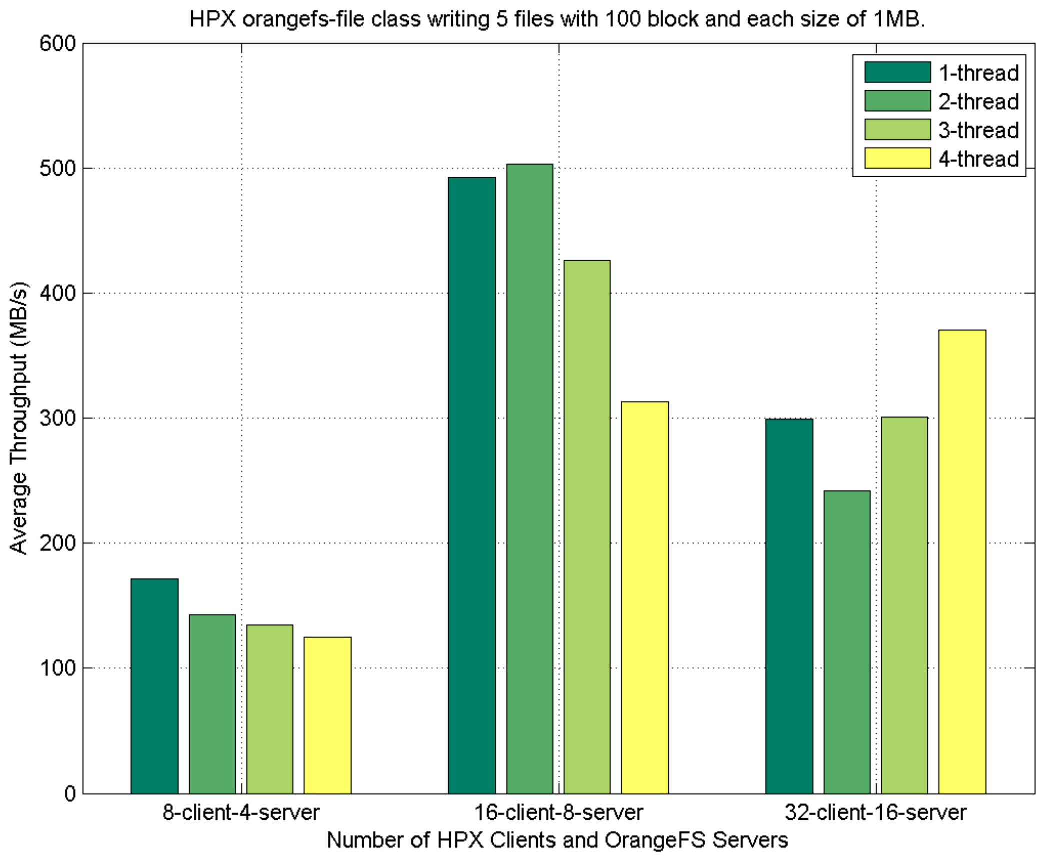


Figure 3.45: HPX orangefs-file asynchronous I/O performance of test case `w-5f-100x1MB` with different number of threads on client nodes.

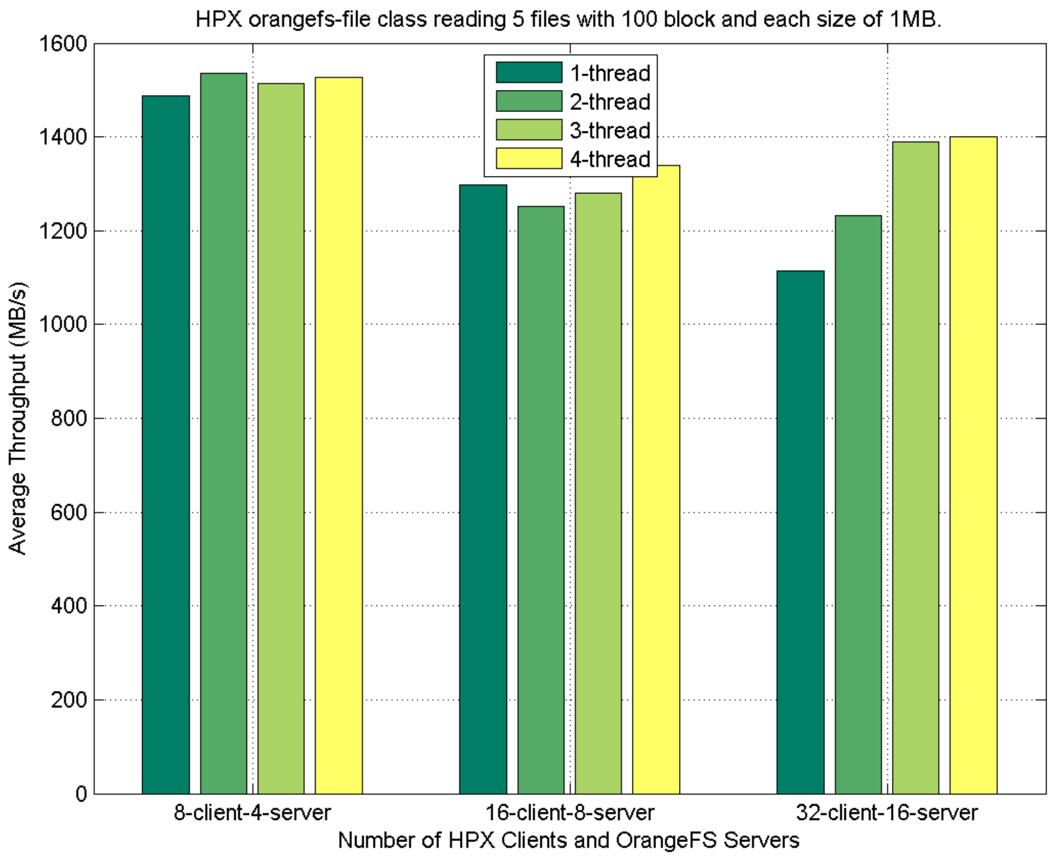


Figure 3.46: HPX orangefs-file asynchronous I/O performance of test case `r-5f-100x1MB` with different number of threads on client nodes.

Thus in conclusion, the number of threads does not affect the reading and writing throughput of **orangefs_file** asynchronous classes, and with more clients and more servers, better performances can be expected which indicates a scalable system.

- **Vertical Comparison: pxfs_file class with Different Number of Threads**

The throughput results of the asynchronous **pxfs_file** class with varying number of threads on all the test cases are displayed in Figure 3.47, Figure 3.48, Figure 3.49, Figure 3.50, Figure 3.51, Figure 3.52, Figure 3.53 and Figure 3.54.

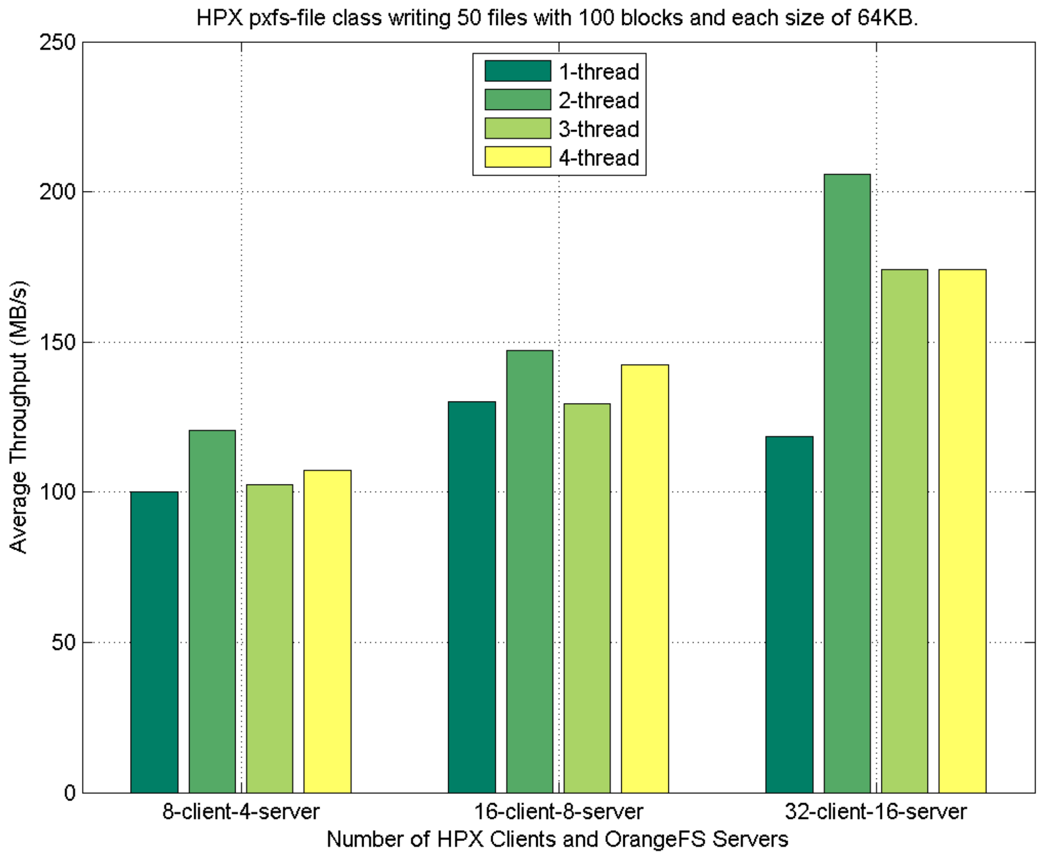


Figure 3.47: HPX pxfs-file asynchronous I/O performance of test case `w-50f-100x64KB` with different number of threads on client nodes.

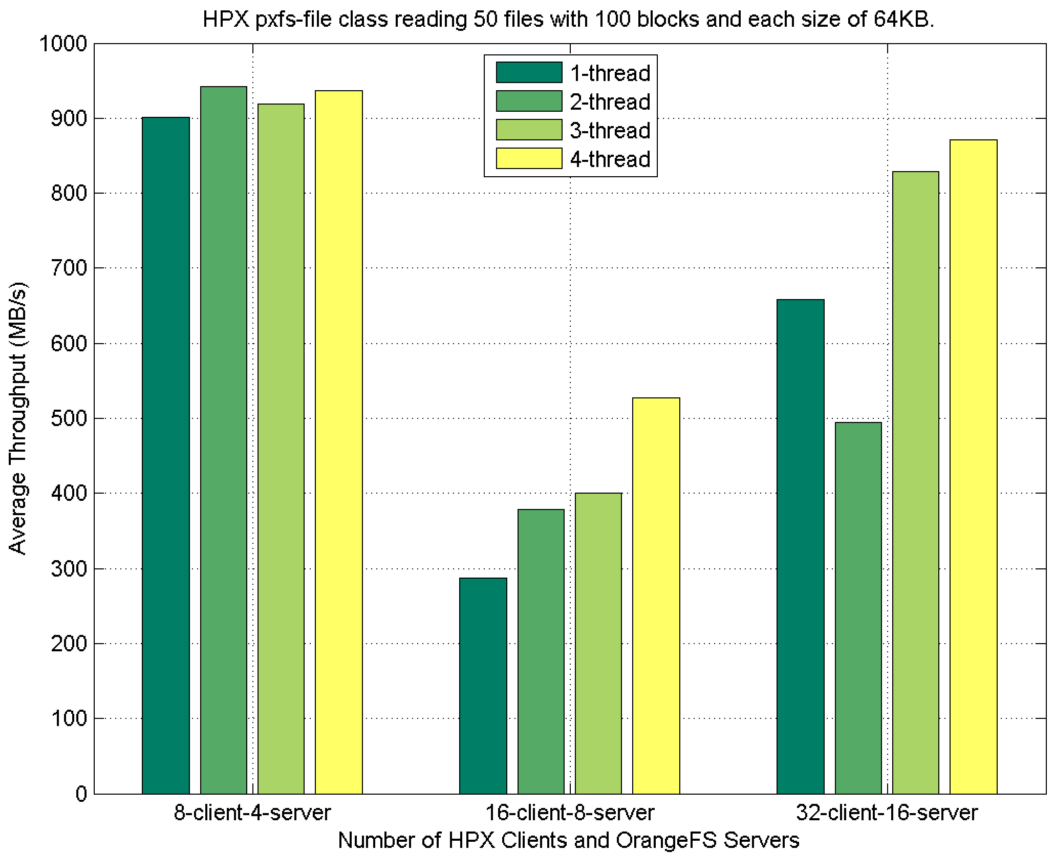


Figure 3.48: HPX pxfs-file asynchronous I/O performance of test case `r-50f-100x64KB` with different number of threads on client nodes.

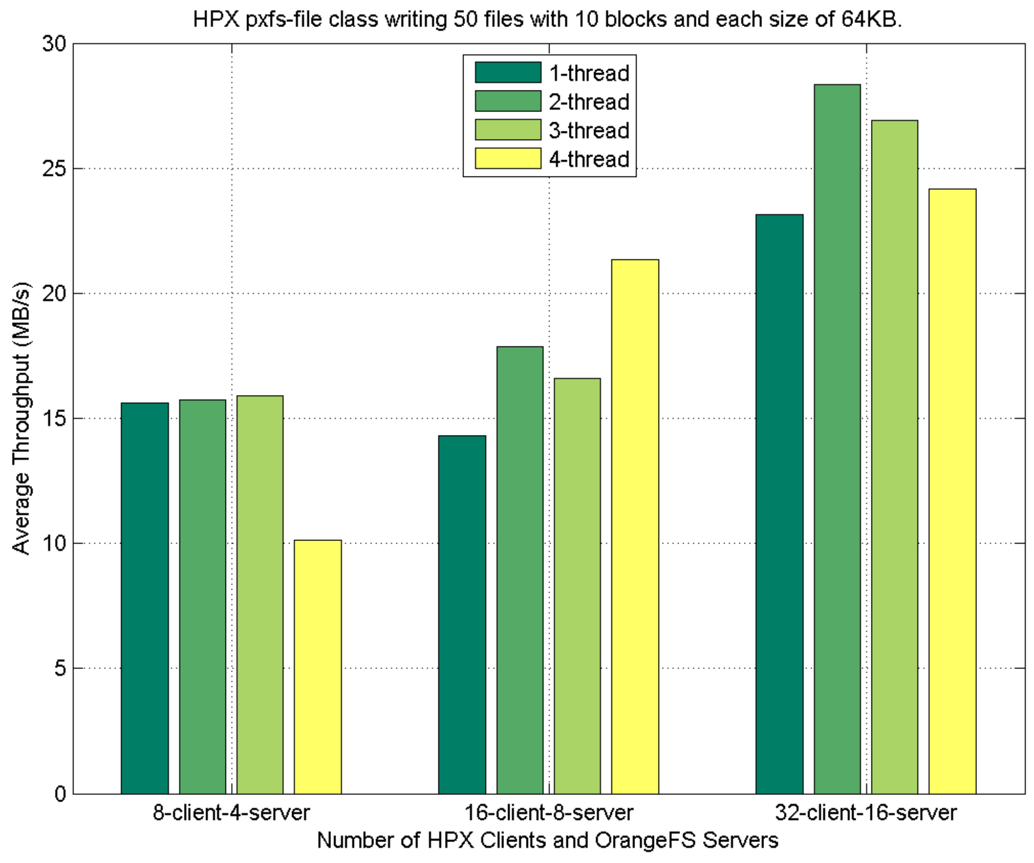


Figure 3.49: HPX pxfs-file asynchronous I/O performance of test case `w-50f-10x64KB` with different number of threads on client nodes.

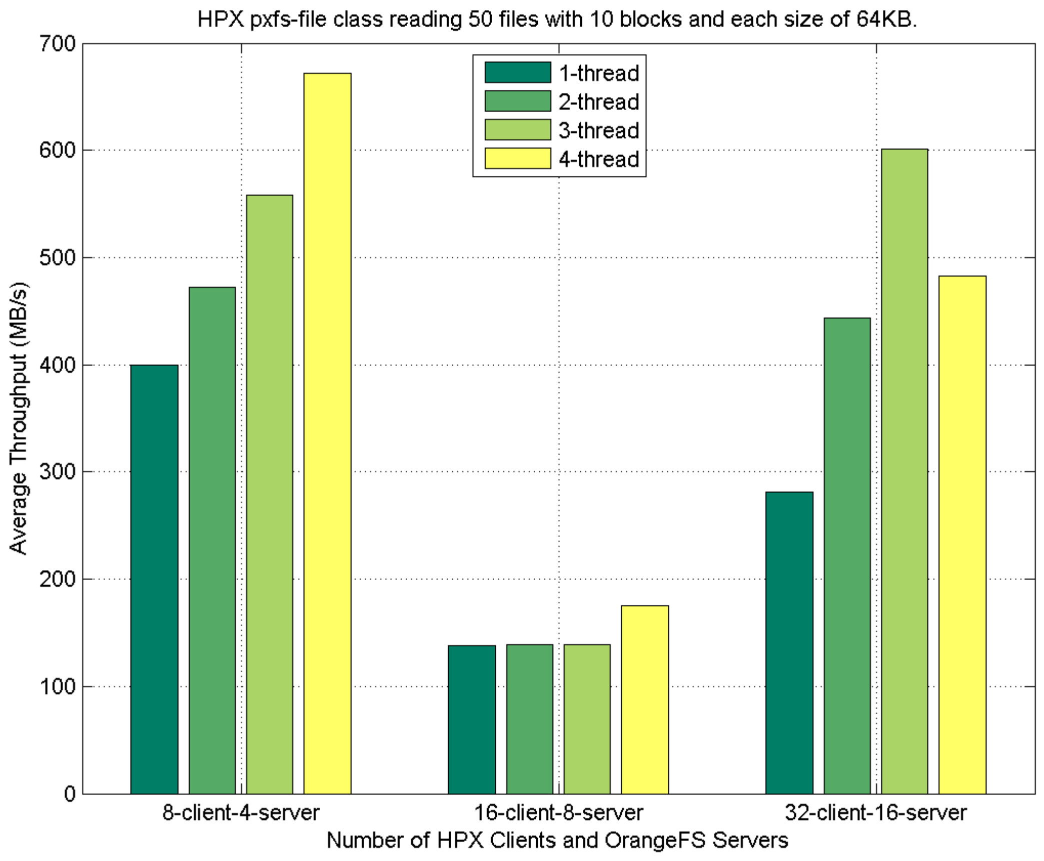


Figure 3.50: HPX pxfs-file asynchronous I/O performance of test case `r-50f-10x64KB` with different number of threads on client nodes.

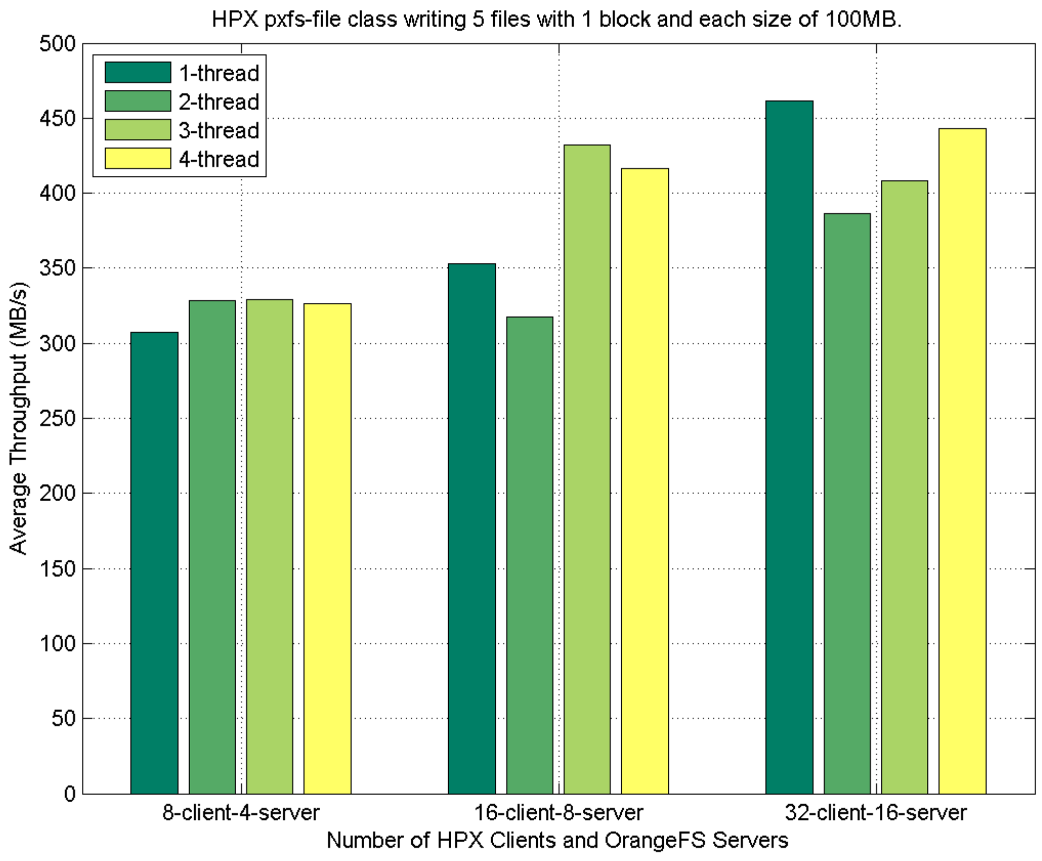


Figure 3.51: HPX pxfs-file asynchronous I/O performance of test case `w-5f-1x100MB` with different number of threads on client nodes.

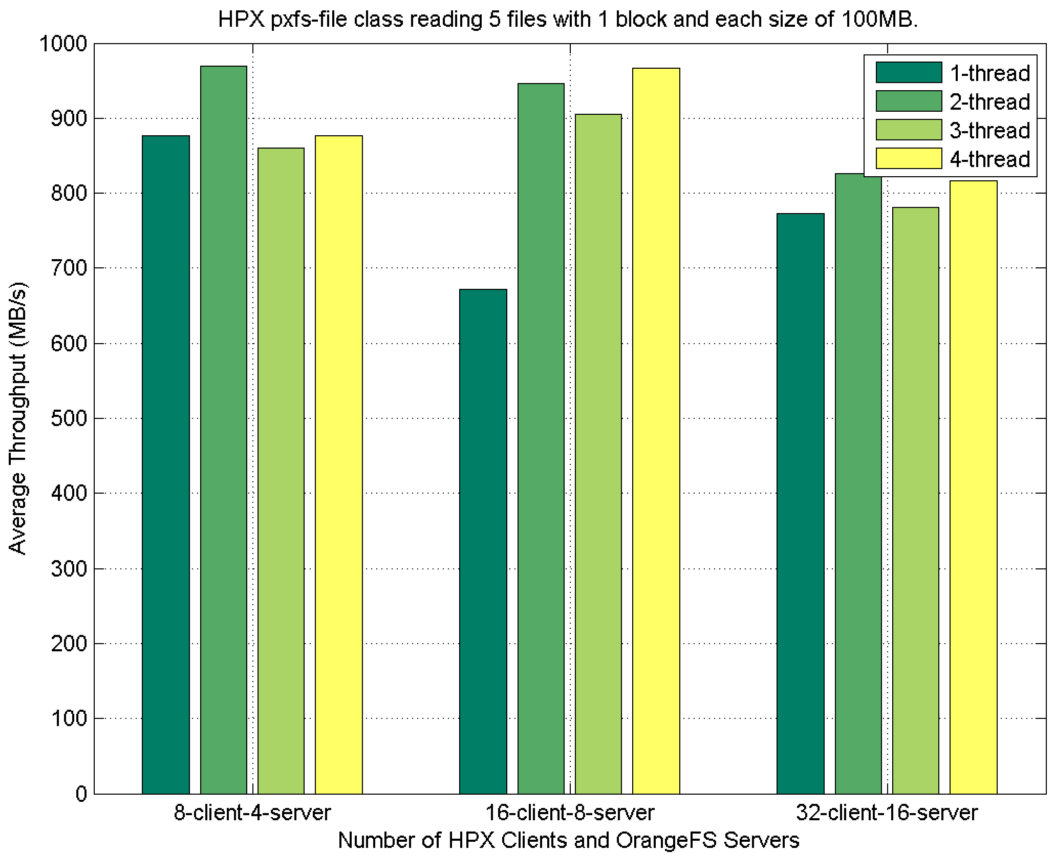


Figure 3.52: HPX pxfs-file asynchronous I/O performance of test case `r-5f-1x100MB` with different number of threads on client nodes.

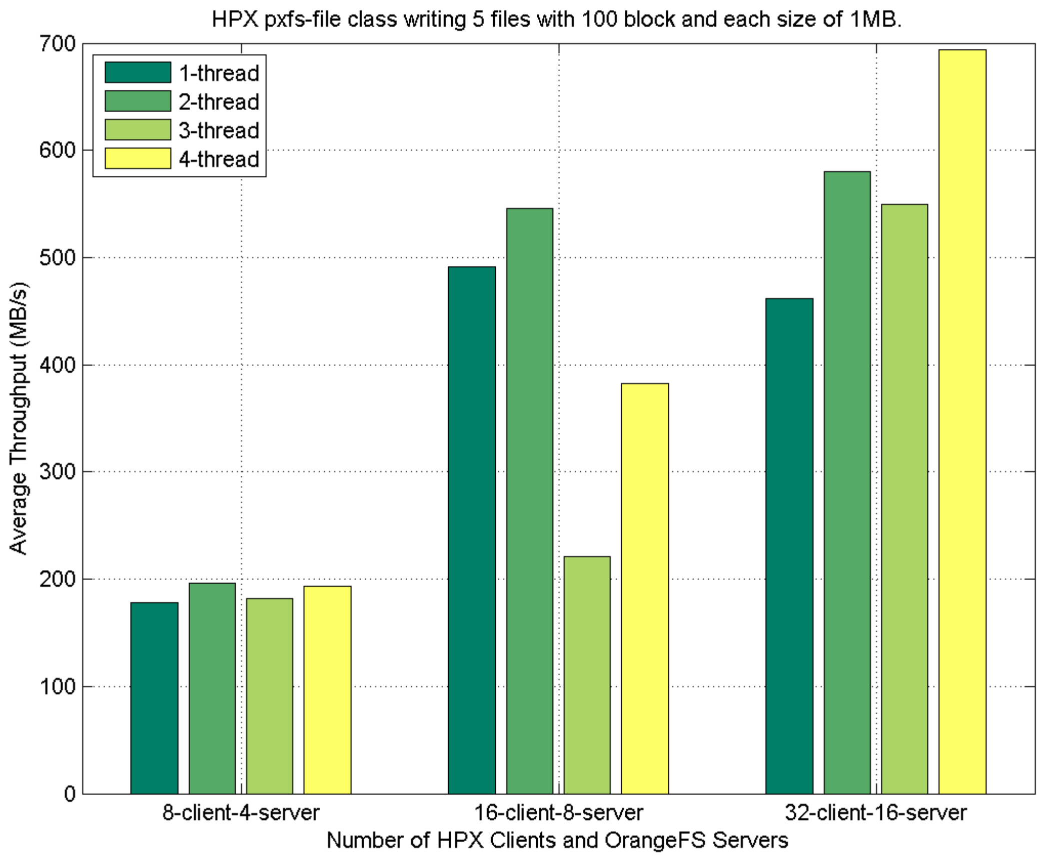


Figure 3.53: HPX pxfs-file asynchronous I/O performance of test case `w-5f-100x1MB` with different number of threads on client nodes.

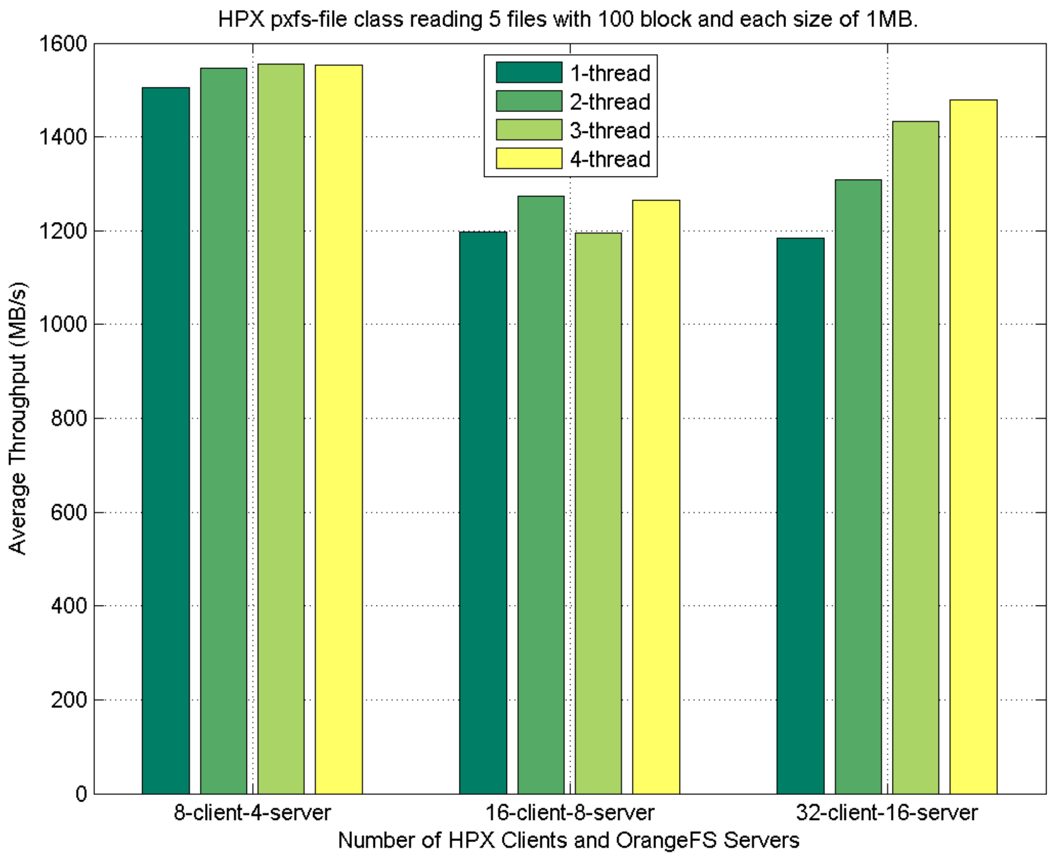


Figure 3.54: HPX pxfs-file asynchronous I/O performance of test case `r-5f-100x1MB` with different number of threads on client nodes.

The following observations of these diagrams can be made without much efforts.

- For all cases of file block size $(64KB, 1MB, 100MB)$, **pxfs_file** does not see a lot of variation of the throughputs when increasing the number of threads for different number of clients. In reading, increasing the number of threads will improve the throughput.

- More data servers shows near-scalable performance for writing and comparable performance for reading. It might be caused by the small number of OrangeFS data servers and small number of client nodes. These test cases did not reach the point where scalability can be shown.

Thus in conclusion, the number of threads does not affect the reading and writing throughput of **pxfs_file** asynchronous classes, and with more clients and more servers, better performances can be expected which indicates a scalable system.

Chapter 4
Distributed Directory Service in Orange File System

The high-performance computing (HPC) community is walking into an new era where top clusters and systems are able to achieve a performance of petaFLOPs (10^{15}) scale easily. The rapid growth in computing power and high demand for parallelism imposes significant challenges for the storage system, which is expected to handle the input/output (I/O) requests from parallel applications with good performance and great scalability.

Many parallel file systems are developed to answer that challenge[17, 7, 8, 9, 10]. File data are distributed across multiple processing nodes in parallel file systems, each with its own storage resources. In that case, concurrent I/O requests can be spread across several servers to process rather than focusing I/O on a single server [18]. Recently there's growing concern with making directories scalable on parallel file systems. It is a typical scenario for applications which do data mining and real-time application monitoring to create numerous small files under the same directory every second. As a result, it is important to build scalable directory services for parallel file systems to support efficient concurrent access to even larger directories in the future.

OrangeFS distributes large files across multiple servers, however, in prior versions a directory object and its data object are still stored on a single server. This design of a directory is inadequate to scale large directories when multiple clients are accessing the same directory concurrently.

A scalable directory service [66] designed for PVFS was demonstrated based on GIGA+ [67]. A prototype was built on a parallel file system simulator, where it achieved high throughput and scalability while minimizing bottlenecks and synchronization overheads. It is necessary to incorporate the scalable directory service into a production-quality high-performance parallel file system to get benefits. In this chapter, a scalable distributed directory is designed based on previous work and implemented into OrangeFS seamlessly.

The main contributions of this chapter include:

- Implemented a complete scalable distributed directory service which distributes directory entries across multiple servers, and incorporated the service into Orange File System seamlessly. The number of directory entry partitions is configurable during directory creation and can be dynamically incremented with usage.

- Evaluated throughput performance in large scale under a production-level environment and compared the outcomes with that of vanilla OrangeFS. The result shows great scalability when the number of servers and clients are varied.

In the rest of the chapter, Section 4.1 lists some related works by other researchers. Section 4.2 describes the design and implementation of the scalable distributed directory. Section 4.3 gives results of performance evaluation and Section 4.4 draws the conclusion.

4.1 Background

GPFS [9] supports efficient file name lookup in large directories through *extensible hashing* [68]. Directory entries of a large directory are stored in multiple disk blocks. To map an entry to a disk block, a hash function is applied to the entry name and the n low-order bits of the hash value is used as the block number, where n depends on the size of the directory. As a directory grows, a disk block can be split in two. The logical block number of the new directory block is derived from the old block number by adding a '1' in the $n + 1^{st}$ bit position, and directory entries with a '1' in the $n + 1^{st}$ bit of their hash value are moved to the new block.

Lustre File System [7] is a massively parallel distributed file system for cluster computing. Lustre is working on distributed metadata service which seems to support one level of splitting, and some results of a pre release [27] shows over 10,000 files creates per second.

Ceph [17] is a research distributed file system which offers dynamic distributed metadata management based on current access patterns. Ceph writes a directory's content to the object storage devices using the same striping and distribution strategy as file data.

Each metadata server keeps a record of the popularity of metadata within the directory and adaptively distributes metadata hierarchically with dynamic subtree partitioning strategy [69].

GIGA+ [67] is a scalable directory design for shared file systems which divides each directory into a scalable number of fixed-size partitions that are distributed across multiple servers in the cluster. A *bitmap* is used to represent a tree of partitions of the corresponding servers where '1' indicates presence and '0' for absence. At first, only the zero-th bit position in the bitmap is set to '1'. As a directory is filled, an overflowing partition with index i and depth r will move half of its hash space to a partition $i + 2^r$, and both partitions will be at depth $r + 1$. Each server manages its partitions independently. An illustration of the splitting process in GIGA+ is shown in Figure 4.1.

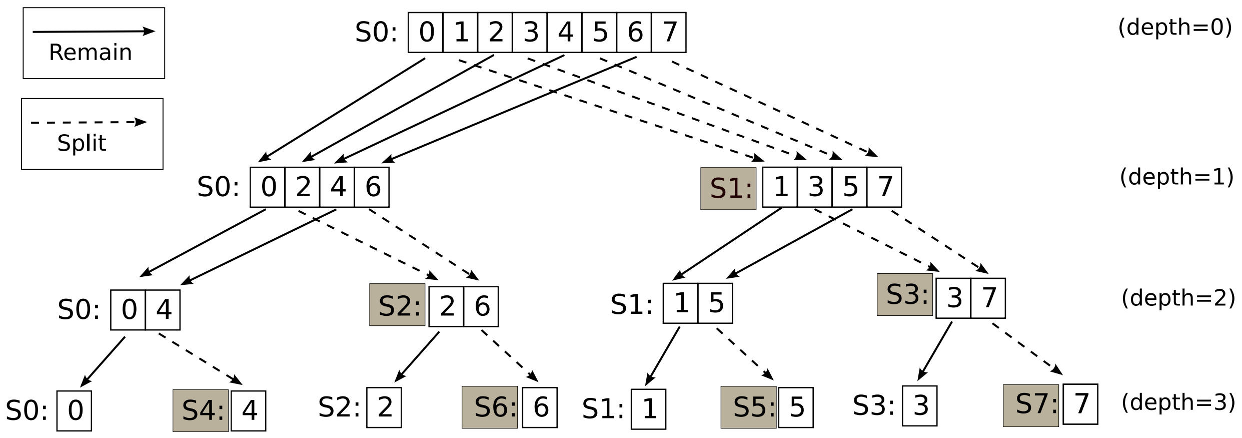


Figure 4.1: Illustration of a three-level splitting process in GIGA+. The hash space is divided evenly for each split.

The design of scalable distributed directory on OrangeFS employs the extensible hashing technique and the splitting strategy of GIGA+, however, it differs in other aspects in order to be incorporated into OrangeFS seamlessly.

4.2 Design and Implementation

4.2.1 OrangeFS Structure

OrangeFS organizes files and directories in the form of several storage objects. Some important objects are

- *Metadata* objects store data about files or directories. Besides usual file attributes like owner, group, permissions, etc. OrangeFS metadata object keeps file distribution information of the actual data. Metadata objects do not contain contents of files or directories.

- *Datafile* objects are blocks of actual file contents. OrangeFS stripes a file across multiple datafiles on multiple servers to facilitate parallel access.
- *Dirdata* objects contain contents of directories, or directory entries. Each directory entry is a pair of entry name and the identifier to its metadata object, either a file metadata object or a directory metadata object.

OrangeFS objects are uniquely identified by *handles*, which are unique, opaque, integer-like identifiers. This provides a concise, non path dependent mechanism for specifying what object to operate on when clients and servers communicate.

The metadata object of a file keeps a list of datafile handles identifying the actual data blocks, while the metadata object of a directory keeps only one dirdata handle representing the dirdata object at this time. The structures of these objects are visualized in Figure 4.2.

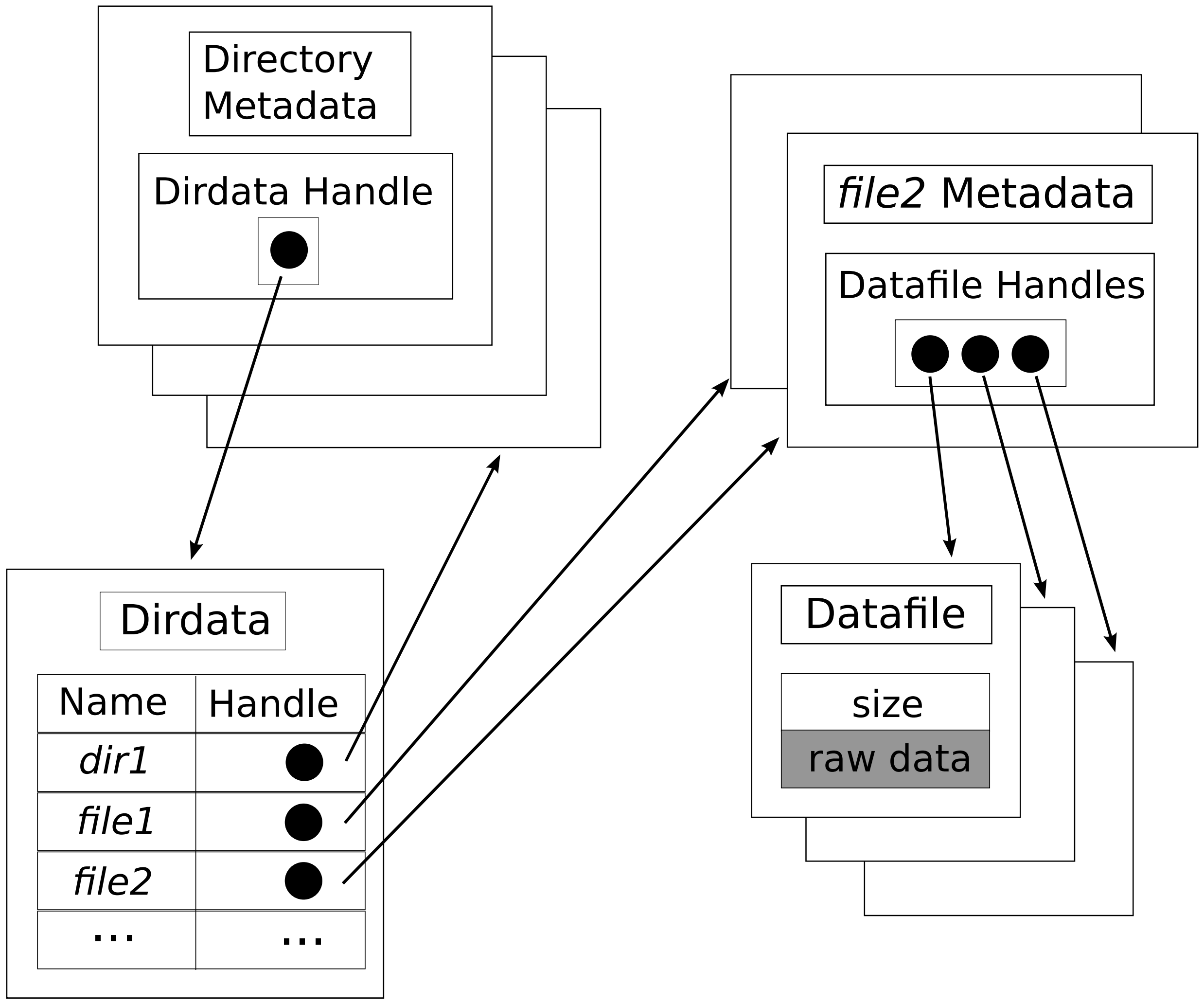


Figure 4.2: Diagram of structures of OrangeFS objects. A directory metadata object only has one dirdata object, which resides on the same metadata server. Usual file attributes fields are omitted in this diagram.

4.2.2 Scalable Distributed Directory Design

- **Data Structure**

An array of *dirdata handles* and a *dirdata bitmap* are added to a directory metadata object and dirdata objects to enable distribution of directory entries across multiple servers. The structures are shown in Figure 4.3.

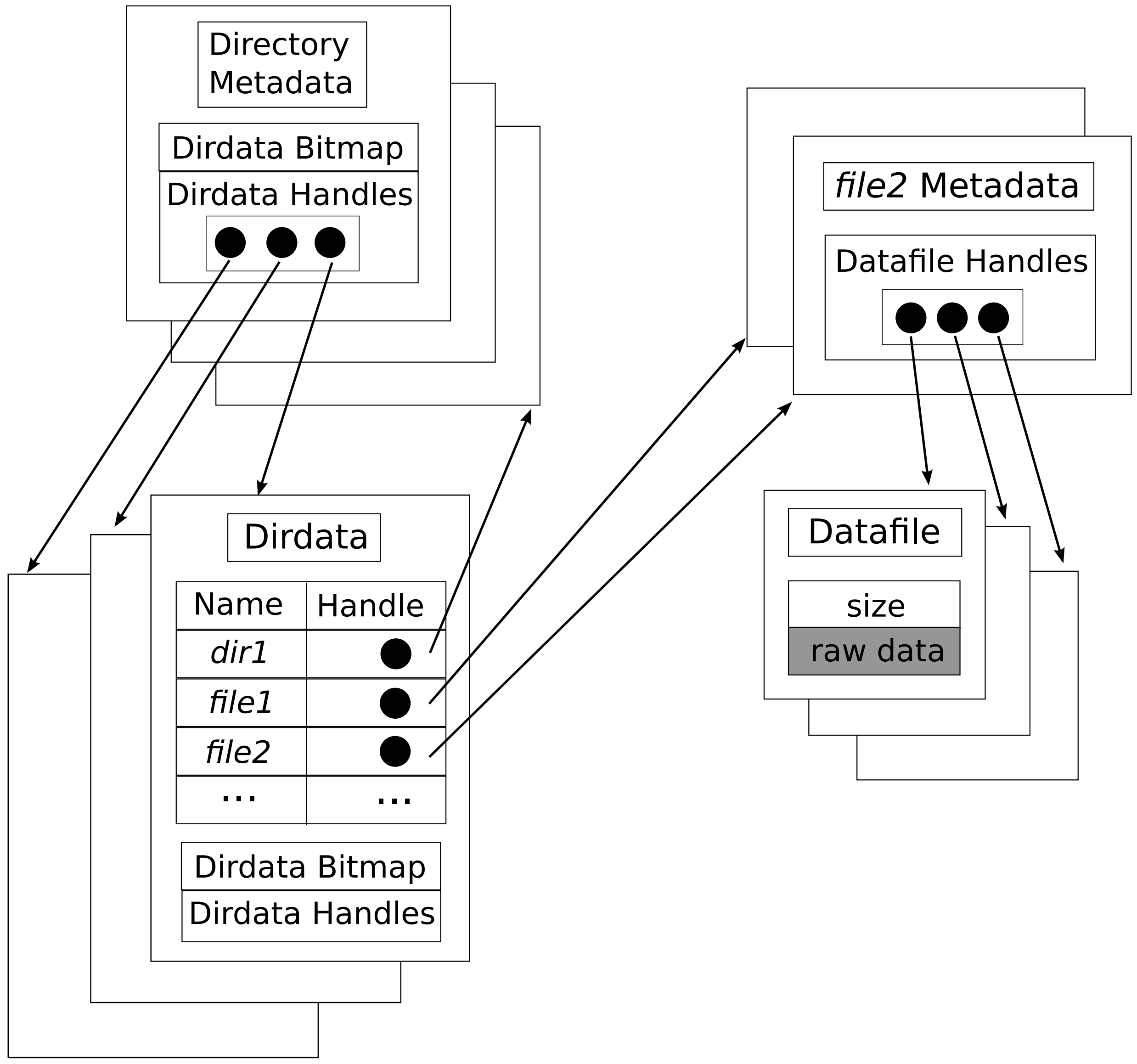


Figure 4.3: Diagram of object structures in scalable distributed directory OrangeFS design. A directory metadata object holds a number of dirdata objects across all metadata servers. Dirdata handles array and dirdata bitmap are added on the directory metadata object and the dirdata object.

When a directory is created, an array of dirdata objects is allocated with one dirdata object on each metadata server. Within each dirdata object, directory entries are indexed by Berkeley DB [70], which uses B+ tree to provide low-cost lookup, insert and delete operations plus efficient sequential access. Consequently there is no need to keep multiple dirdata objects of the same directory on one metadata server.

The dirdata bitmap keeps a one-to-one mapping between the bit position and the index of the dirdata handle array. A bit value of '1' indicates an *active* dirdata object while a bit value of '0' indicates an *inactive* dirdata object. The bitmap is initialized during directory creation to set the initial number of active dirdata objects to use. Unlike GIGA+ which always starts from one partition and increases the number of partitions gradually, the initial number of active dirdata objects is configurable. The splitting process is found to be expensive [66] and for a directory which is expected to be large, it is better to utilize all the dirdata objects to enjoy better scalability from the start.

The metada object of a directory keeps a most up-to-date copy of dirdata bitmap, which is done by accepting bitmap updates from dirdata objects. Each dirdata object also keeps a copy of dirdata bitmap and array of dirdata handles. One purpose is to verify the incoming directory entry request belongs to its hash space and another is to find the new dirdata object when issuing a split operation. Thus the dirdata bitmap on a dirdata object only needs to be locally updated and accurately reflects its own status.

- **Directory Entry Lookup**

When initiating a directory request, the OrangeFS client library is responsible for fetching the scalable distributed directory attributes, including the dirdata array and dirdata bitmap, and deciding which dirdata object to contact to complete directory operations. We made this decision to simplify the design on the server side and reduce metadata server waiting time for each client request, since metadata and dirdata are probably not sitting on the same server.

Another reason is that OrangeFS uses client-side caching of object metadata to optimize the performance of file system access patterns [71]. There is an *acache* module, which stands for attribute cache, in OrangeFS system. It is used to cache metadata of storage objects including ownership, timestamp and distributed file information. By keeping a copy of distributed directory attributes on the client side, we can take advantage of the *acache* module.

The extensible hashing technique mentioned in Section 4.1 is used to map a directory entry to a dirdata object using the algorithm as detailed in Figure 4.4. A directory entry is assigned to an active dirdata handle based on its name. To achieve a random distribution, the directory entry name is encoded by a strong hash algorithm (*MD5* in our case) first. The hashed value then serves as the key to dirdata object selection. The lower R bits ($R = \lceil log_2(N) \rceil$, where N is the size of dirdata array) are taken as an initial matching index. If the dirdata bitmaps shows an inactive dirdata handle at that index, the highest bit is taken off and the dirdata bitmap is checked again with the new matching index. In the end, an active dirdata handle is picked and the corresponding operations can be processed. The dirdata object at index 0 is always active which guarantees a dirdata object is selected under any circumstances.

- **Dirdata Splitting**

Dirdata splitting is supported when the number of directory entries in one dirdata object exceeds a threshold value, which is a configurable parameter with a default value. Because the maximum number of dirdata objects of a directory is fixed at the number of metadata servers, splitting is not possible, or necessary, when all of the dirdata objects are in use. In that case, the number of directory entries stored in the dirdata object is allowed to exceed the threshold. If new servers are added to the system, overloaded dirdata objects can resume splitting to dirdata objects on new servers.

The dirdata splitting process follows the splitting method of GIGA+ mentioned in Section 4.1. If the initial number of active dirdata objects is n, then during directory creation, the first n bits of dirdata bitmaps will be set to ‘1’ and their depths will be calculated. Upon splitting from a dirdata object with an index of i and depth of r, the

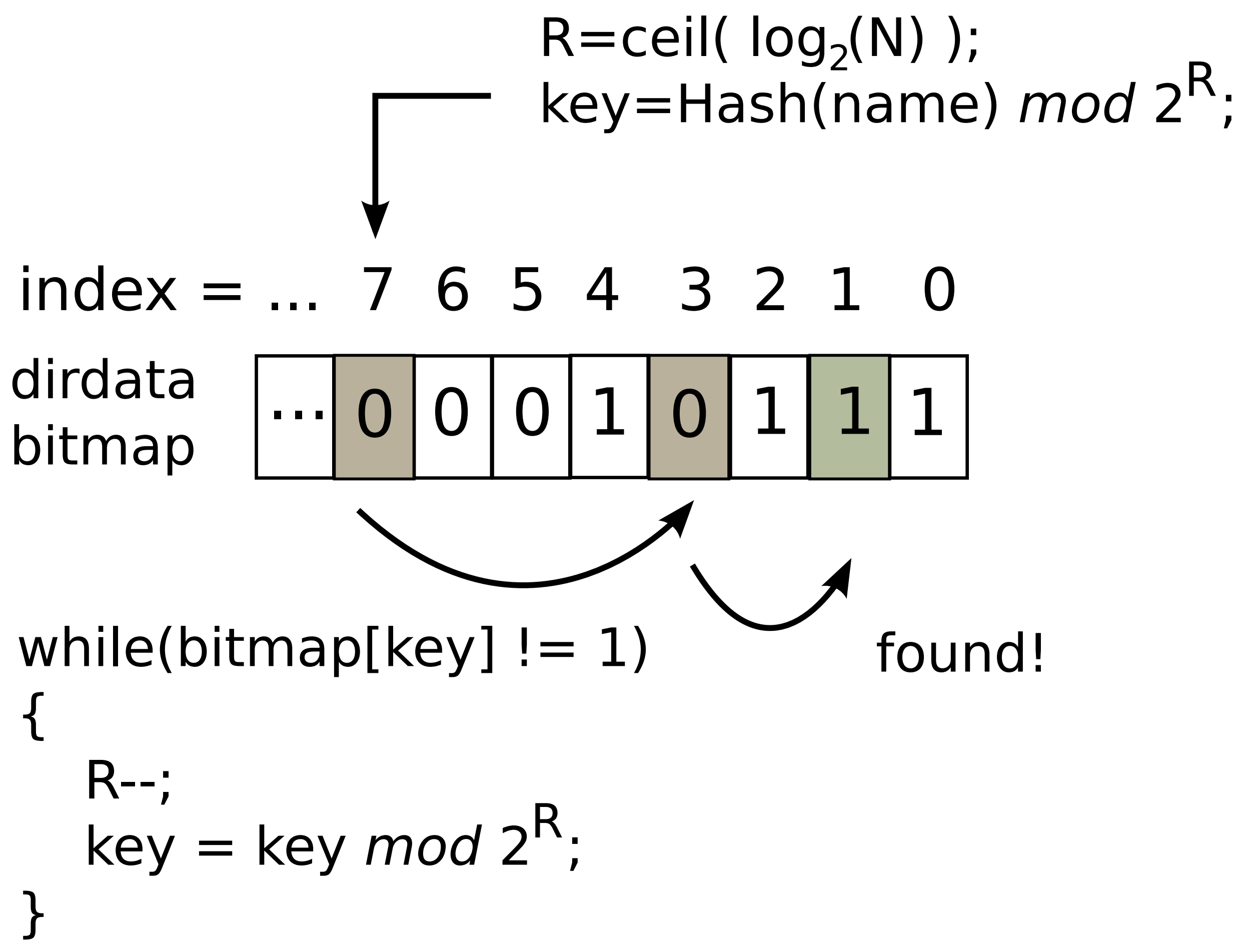


Figure 4.4: Illustration of the lookup process in scalable distributed directory on OrangeFS. N denotes the total number of dirdata objects. In this example, N=8 and the key starts at 7.

index of the new dirdata object is $i + 2^r$ if the index is still within range. Directory entries with a '1' on the $r + 1^{st}$ bit position of their hash values will be moved to the new dirdata object. Both dirdata objects will bump to a depth of $r + 1$ afterwards.

After splitting is completed between two dirdata objects, an updated bitmap is sent to the server holding the metadata object of the directory to update the bitmap there, basically setting the bit at the new dirdata position as '1'. With this mechanism, the metadata object will have the most up-to-date information of the directory at all times. If the client uses an out-of-date bitmap in its cache and finds that the directory entry is no longer on the indicated dirdata server because of splitting, it can always turn to the metadata object for the newest copy of the bitmap and start again. This simplifies the synchronization process and a client only needs a maximum of two probes to locate the correct dirdata object, comparing to possible multiple probes in GIGA+ because of the lazy client update mechanism they used [67].

- **Maintenance and Removal**

  Some attributes, such as the timestamp information and the number of directory entries, will be distributed with the dirdata and collected by the client if needed. Only active dirdata objects are contacted for these distributed attributes and directory entries. Other attributes like permissions are kept in the metadata object, similar to regular files.

  The removal of a scalable distributed directory is conducted mainly by the server which holds the metadata object to simplify the error handling procedure. It will make sure all the dirdata objects are deleted before the removal of the metadata object of the directory. If anything goes wrong when removing dirdata objects, the dirdata array can be rebuilt to restore the functionality of the directory.

4.3 Performance Evaluation

4.3.1 Experiment Setup

The performance of the scalable distributed directory was evaluated on the Palmetto Cluster housed by Clemson Computing and Information Technology [65]. The cluster was configured as follows at the time of our experiments. There were 1,541 nodes in total with 8 cores per node. The nodes were running Linux 2.6.18 and equipped with Intel Xeon E5345/E5410/L5420 at 2.33GHz/2.5GHz or AMD Opteron 2356 at 2.3GHz and memory sizes of 12GB/16GB. The interconnect network utilizes Myrinet 10G and high throughput storage is attached to all nodes.

4.3.2 Results and Analysis

The UCAR metarates benchmark [72] is an MPI application that measures throughput of file creation rates in one directory with multiple clients concurrently. We modified the benchmark program to use OrangeFS native APIs to communicate with OrangeFS servers and add a function to measure the file removal rates under a directory. The throughput results from creating and removing empty files under one directory with different numbers of OrangeFS servers is illustrated in Figure 4.5. The number of clients is twice the number of OrangeFS servers. The throughput of scalable distributed directory shows great scalability as the number of servers increase and can reach more than 8,000 file creations and over 11,000 file removals on average in one second with 64 servers.

Next, the number of clients is varied when running the benchmark on a 64 server setup. The results are illustrated in Figure 4.6. The results show that OrangeFS with scalable distributed directories can scale with the number of clients up to twice the number servers, while the vanilla OrangeFS can not scale up with the number of clients. Figure 4.6 also shows the results with 192 clients and 256 clients, where the average file creations per second maintains over 8,000 and the average file removals per second drops a little from 11,000 but still lingers above 8,000. This means the servers are overloaded with client requests and are reaching their full capacities. The faster drops in file removal may be due to the OrangeFS file removal implementation because the vanilla OrangeFS shows similar trend as servers are saturated.

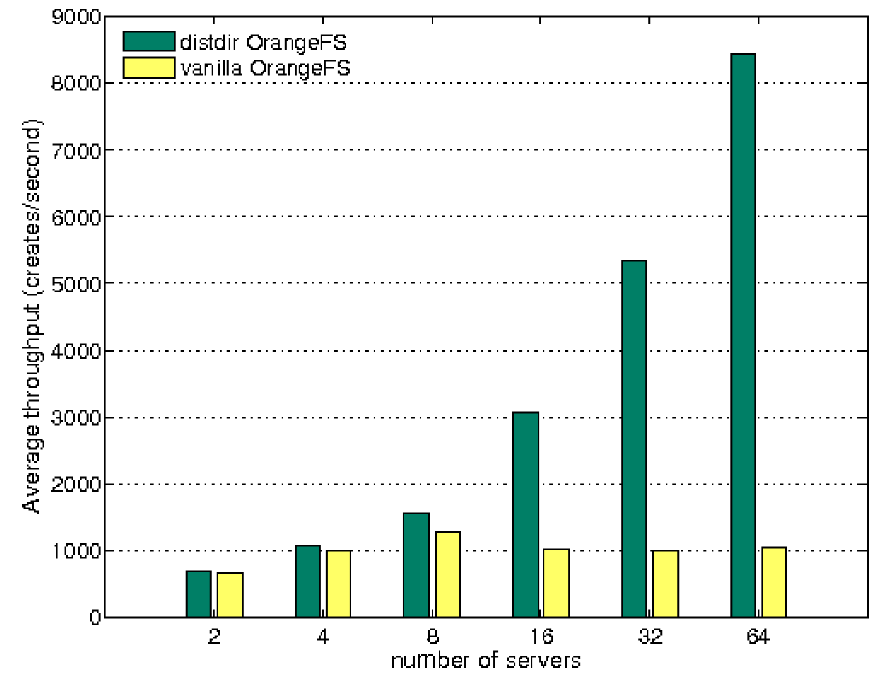
4.4 Chapter Conclusions

This chapter describes a scalable distributed directory design and implementation on Orange File System. Similar to data files, directory entries are distributed across multiple servers. The number of partitions can be specified during directory creation and dynamically incremented when filling the directory. Files are assigned to a specific partition by hashing their names to achieve load balancing among partitions. Clients are responsible for contacting the correct server when initiating a request.

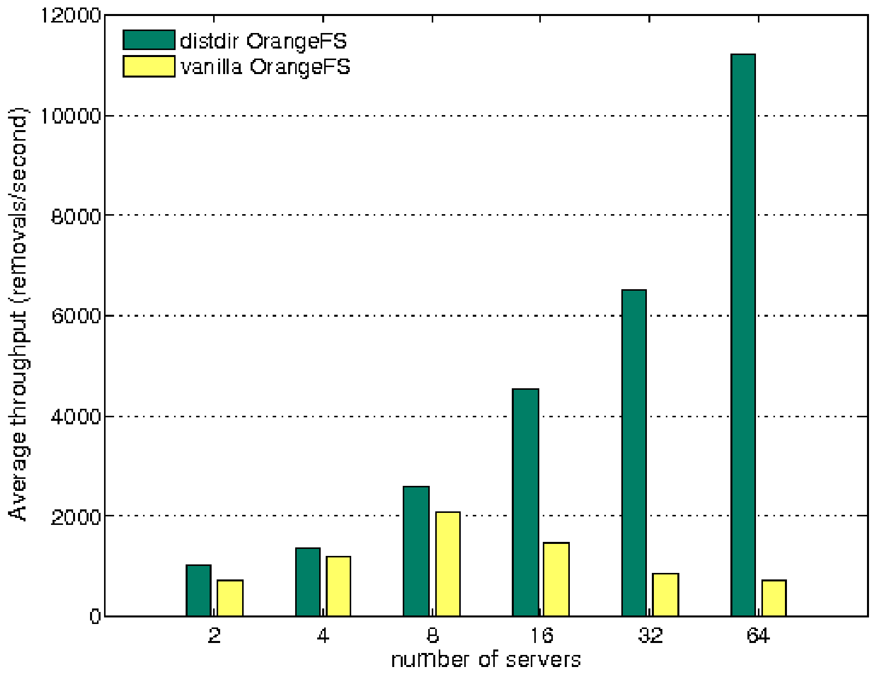
A complete scalable distributed directory implementation on OrangeFS has been carried out except the dynamic splitting function when the paper was prepared. A modified version of UCAR metarates benchmark is used to evaluate the performance of the scalable distributed directory. Comparing with vanilla OrangeFS, the scalable distributed directory shows great scalability in creating and removing large number of files by multiple clients concurrently. On a 64-servers setup and 128 clients accessing the same directory concurrently, the scalable distributed directory can achieve more than 8,000 file creations per second and over 11,000 file removals per second on average.

The scalable distributed directory feature is available in current 2.8 release of OrangeFS at

`http://orangefs.org/download/`

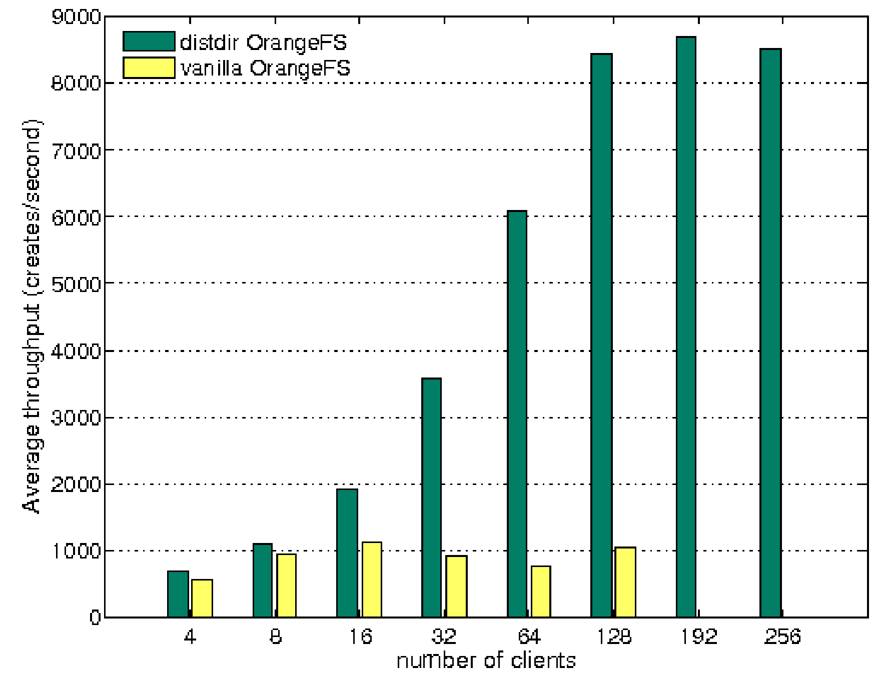


(a) average file creation throughput with different numbers of servers

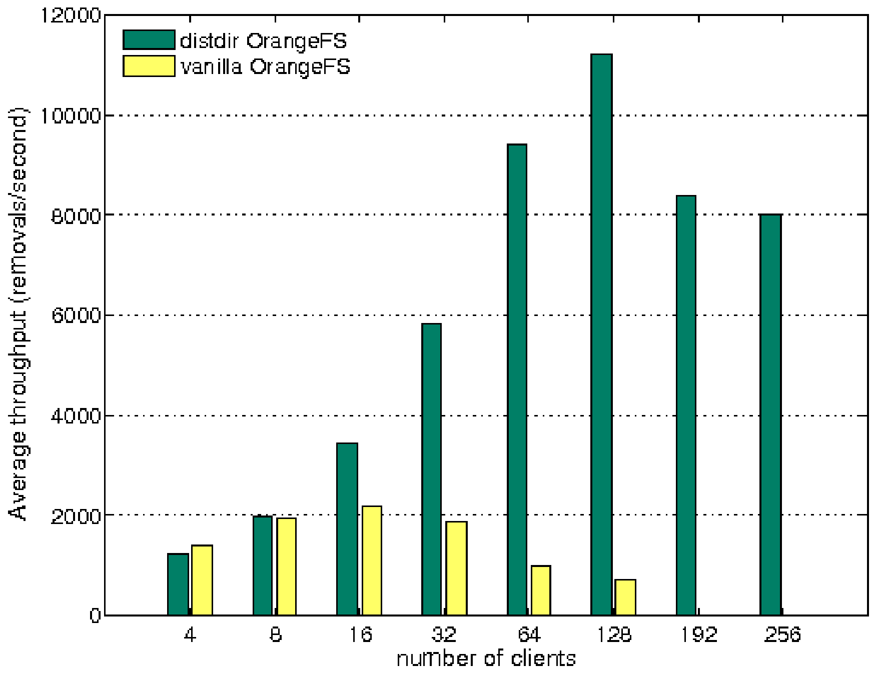


(b) average file removal throughput with different numbers of servers

Figure 4.5: Scale and performance of scalable distributed directory implementation on OrangeFS with different number of OrangeFS servers. The UCAR metarates benchmark is used and the number of clients are selected to be 2x the number of servers.



(a) average file creation throughput with different numbers of clients on 64-server system



(b) average file removal throughput with different numbers of clients on 64-server system

Figure 4.6: Scale and performance of 64 servers scalable distributed directory implementation on OrangeFS with different number of clients.

Chapter 5
Conclusions and Future Work

This dissertation presents PXFS, a novel persistent storage model for extreme scale. It aims to explore the maximum of parallelism and performance out of the storage layer in the Exascale computing era, which is expected to arrive in the next decade. The early implementation of PXFS is composed of two parts, asynchronous I/O operations and an active TupleSpace model.

The asynchronous I/O interface utilizes Orange parallel file system as the back-end and incorporate an asynchronous I/O interface into HPX, an implementation of ParalleX execution model. The I/O performance is evaluated with a homemade benchmark and the I/O throughput shows a perfect scalability, along with a 3x to 20x times speedup against the OrangeFS user interface and a 5x to 10x times higher throughput than NFS with 24 clients. Another benchmark is conducted to study the performance of different asynchronous HPX file classes and the effects of number of client threads. The result shows that **local_file** class is not scalable and its performance is largely affected by the number of client threads, while **orangefs_file** and **pxfs_file** are not affected by the number of client threads under all cases, which shows a sign of scalability. And by comparing the result of **orangefs_file** and **pxfs_file** class, another conclusion can be made that move the asynchronism closer to the data object will improve the I/O operations.

The other part of PXFS is the innovative design of active TupleSpace model which can be served as a coordinating places for concurrent operations and a meta-space for all objects, including storage objects and runtime objects. The meta-properties of objects can be used to aid application and programmers to make smarter decisions based on all information. A micro check-pointing use case is illustrated to showcase the power of the active HPX TupleSpace.

Another part of the dissertation is the design of a scalable distributed directory service and its implementation on Orange file system. It enables the scalability of directory operations and the result shows perfect scalability and excellent metadata operation throughputs.

In the future, more layers of Orange parallel file system and ParalleX model will be analyzed to develop a more sophisticated storage model. For the benchmark, the effect of threads and various architectures on I/O performance will be tested thoroughly. More real world applications will be used to evaluate the performance of PXFS.

Another part of work in the future could be detailed research in metadata management in object level runtime system. The TupleSpace could be utilized to include all metadata from either storage objects and runtime objects. More use cases can be found and applications be implemented to benefit from the unified metadata management.

References

[1] TOP500 Supercomputing Sites. `http://www.top500.org`, June 2014.

[2] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2.* High Performance Computing Center Stuttgart (HLRS), Stuttgart, Germany, September 2009.

[4] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6, 2007.

[5] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Parallel Processing Workshops*, pages 394–401, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[6] Alexandre Tabbal, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, and Thomas Sterling. Preliminary design examination of the ParalleX system from a software and hardware perspective. *SIGMETRICS Performance Evaluation Review*, 38:4, Mar 2011.

[7] Lustre a Network Clustering FS. `http://www.lustre.org`.

[8] Orange File System. `http://orangefs.org`.

[9] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the Conference on File and Storage Technology (FAST'02)*, Berkeley, CA, USA, January 2002.

[10] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th annual Linux Showcase & Conference*, Berkeley, CA, USA, 2000. USENIX Association.

[11] Scott Roberts. Next generation storage for the hltcoe. 2013.

[12] Shuangyang Yang, Walter B Ligon, and Elaine C Quarles. Scalable distributed directory implementation on orange file system. *Proc. IEEE Intl. Wrkshp. Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011.

[13] Shuangyang Yang, Wlater B. Ligon III, Maciej Brodowicz, and Hartmut Kaiser. PXFS: A Persistent Storage Model for Extreme Scale. In *Scientific Computing Around Louisiana*, February 2013.

[14] Shane Snyder. ParalleX file system (PXFS): Bridging the Gap Between Exascale Processing Capabilities and I/O Performance. Master's thesis, Clemson University, Clemson, SC, USA, May 2013.

[15] Shuangyang Yang, Wlater B. Ligon III, Maciej Brodowicz, and Hartmut Kaiser. Pxfs: A persistent storage model for extreme scale. In *Computing, Networking and Communications (ICNC), 2014 International Conference on*, pages 900–906, Feb 2014.

[16] Michael Moore, Bonnie David, Becky Ligon, Mike Marshall, Walt Ligon, Nicholas Mills, Elaine Quarles, Sam Sampson, Shuangyang Yang, and Boyd Wilson. OrangeFS: Advancing PVFS. *FAST '11: 9th USENIX Conference on File and Storage Technologies*, 2011.

[17] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrel D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on Operation systems design and implementation*, Berkeley, CA, USA, 2006.

[18] Philip H. Carns. *Achieving Scalability in Parallel File Systems.* PhD thesis, Clemson University, Clemson, SC, USA., 2005.

[19] Intel Scalable Systems Division. Paragon system user's guide. May 1995.

[20] P.F. Corbett, D.G. Feitelson, J.-P. Prost, G.S. Almasi, S.J. Baylor, A.S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B.D. Herr, J. Kavaky, T.R. Morgan, and A. Zlotek. Parallel file systems for the ibm sp computers. *IBM Systems Journal*, 34(2):222–248, 1995.

[21] Rajesh Bordawekar, Steven Landherr, Don Capps, and Mark Davis. Experimental evaluation of the hewlett-parkard exemplar file system. *SIGMETRICS Perform. Eval. Rev.*, 25(3):21–28, December 1997.

[22] XFS: A next generation journalled 64-bit filesystem with guaranteed rate I/O. `http://www.sgi.com/Technology/xfs-whitepaper.html`.

[23] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Network file system (nfs) version 4 protocol, 2003.

[24] Peter J. Braam. The coda distributed file system. *Linux J.*, 1998(50es), June 1998.

[25] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *ACM TRANSACTIONS ON COMPUTER SYSTEMS*, pages 109–126, 1995.

[26] K.W. Preslan, A.P. Barry, J.E. Brassow, G.M. Erickson, E. Nygaard, C.J. Sabol, S.R. Soltis, D.C. Teigland, and M.T. O'Keefe. A 64-bit, shared disk file system for linux. In *Mass Storage Systems, 1999. 16th IEEE Symposium on*, pages 22–41, 1999.

[27] R. Scott Studham and Rajagopal Subramaniyan. Lustre: A Future Standard for Parallel File Systems? Invited presentation at International Supercomputer Conference, June 2005.

[28] Steven Moyer and V. S. Sunderam. Pious: A scalable parallel i/o system for distributed computing environments. In *in Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[29] Nils Nieuwejaar and David Kotz. The galley parallel file system. In *Parallel Computing*, pages 374–381. ACM Press, 1996.

[30] Message Passing Interface Forum. `http://www.mpi-forum.org`.

[31] Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[32] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[33] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. Overview of the mpi-io parallel i/o interface. In *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146. Springer, 1996.

[34] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.

[35] ROMIO: A High-Performance, Portable MPI-IO Implementation. `http://press3.mcs.anl.gov/romio/`.

[36] Hartmut Kaiser, Bryce Adelstein-Lelbach, et al. HPX source code repository, 2007-2013. Available under the Boost Software License (a BSD-style open source license).

[37] The STE——AR Group. Systems Technologies, Emerging Parallelism, and Algorithms Reseach, 2011-2013. http://stellar.cct.lsu.edu.

[38] Chirag Dekate, Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas L. Sterling. Improving the scalability of parallel N-body applications with an event driven constraint based execution model. *Accepted, The International Journal of High Performance Computing Applications*, abs/1109.5190, 2012. http://arxiv.org/abs/1109.5190.

[39] The C++ Standards Committee. ISO/IEC 14882:2011, Standard for Programming Language C++. Technical report, 2011. http://www.open-std.org/jtc1/sc22/wg21.

[40] Boost: a collection of free peer-reviewed portable C++ source libraries, 2011. http://www.boost.org/.

[41] David W. Wall. Messages as active agents. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 34–39, New York, NY, USA, 1982. ACM.

[42] Charles E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

[43] Intel. Intel Thread Building Blocks 3.0, 2010. http://www.threadingbuildingblocks.org.

[44] Microsoft. Microsoft Parallel Pattern Library, 2010. http://msdn.microsoft.com/en-us/library/dd492418.aspx.

[45] Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. In *SIGART Bull.*, pages 55–59, New York, NY, USA, August 1977. ACM.

[46] Daniel P. Friedman and David S. Wise. CONS Should Not Evaluate its Arguments. In *ICALP*, pages 257–284, 1976.

[47] Robert H. Halstead, Jr. MULTILISP: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985.

[48] Jack B. Dennis. First version of a data flow procedure language. In *Symposium on Programming*, pages 362–376, 1974.

[49] Jack B. Dennis and David Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *25 Years ISCA: Retrospectives and Reprints*, pages 125–131, 1998.

[50] Arvind and R. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture". In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE '87, Parallel Architectures and Languages Europe, Volume 2: Parallel Languages.* Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 259.

[51] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[52] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[53] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[54] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[55] PGAS. PGAS - Partitioned Global Address Space, 2011. http://www.pgas.org.

[56] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[57] Lyndon JB Nixon, Elena Simperl, Reto Krummenacher, and Francisco Martin-Recuerda. Tuplespace-based computing for the semantic web: A survey of the state-of-the-art. *Knowledge Engineering Review*, 23(2):181–212, 2008.

[58] Eric T Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces: Principles, Patterns and Practices*. Addison-Wesley Professional, 1999.

[59] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel Alexander Ford. T spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[60] GigaSpaces Platform. White paper,. 2002, new york, ny, usa: Gigaspaces technologies ltd. 14.

[61] Niklas Gustafsson and Artur Laksberg and Herb Sutter and Sana Mithani. Improvements to std::future¡T¿ and Related APIs. Technical report, 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3634.pdf.

[62] Peter Dimov. Additional std::async Launch Policies. Technical report, 2013. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3632.html.

[63] Center for Computation & Technology. `http://cct.lsu.edu`.

[64] Louisiana State University. `http://www.lsu.edu`.

[65] The Palmetto Cluster. `http://citi.clemson.edu`.

[66] Yang Wu. A Study for Scalable Directory in Parallel File Systems. Master's thesis, Clemson University, Clemson, SC, USA, July 2009.

[67] Swapnil Patil and Garth Gibson. GIGA+: Scalable Directories for Shared File Systems. Technical Report CMU-PDL-08-110, Carnegie Mellon University Parallel Data Lab, October 2008.

[68] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[69] S. A. Weil, S. A. Brandt, and E. L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'04)*, November 2004.

[70] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, 1999.

[71] Bradley W. Settlemyer. *A Study of Client-Based Caching for Parallel I/O*. PhD thesis, Clemson University, Clemson, SC, USA, August 2009.

[72] UCAR Metarates Benchmark. `http://www.cisl.ucar.edu/css/software/metarates`.

Vita

Shuangyang Yang was born on January 1985, in Taizhou, China. He finished his undergraduate studies at Zhejiang University, Hangzhou China, July 2006. He earned a master of science degree from University of Dayton, OH in Dec 2008. In January 2012 he came to Louisiana State University to pursue graduate studies in computer science. He is currently a candidate for the degree of Doctor of Philosophy in computer science, which will be awarded in December 2014.