THE UNIVERSITY OF CHICAGO

ENABLING EFFICIENT PARALLEL SCRIPTING ON LARGE-SCALE COMPUTERS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

ZHAO ZHANG

CHICAGO, ILLINOIS
JUNE, 2014
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xi</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 A Review of Many-task Applications</td>
<td>2</td>
</tr>
<tr>
<td>1.2 An Overview of Large-scale Computers</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Existing Approaches to Running Many-task Applications</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Montage Performance: A Case Study</td>
<td>5</td>
</tr>
<tr>
<td>1.5 The Challenges of Efficient Data Management at Large Scale</td>
<td>7</td>
</tr>
<tr>
<td>1.6 An Overview of the Solution</td>
<td>8</td>
</tr>
<tr>
<td>1.7 Advantages and Limitations of the Solution</td>
<td>9</td>
</tr>
<tr>
<td>1.8 Dissertation Roadmap</td>
<td>11</td>
</tr>
<tr>
<td>2 BACKGROUND INFORMATION AND RELATED WORK</td>
<td>12</td>
</tr>
<tr>
<td>2.1 Representative MTC Applications</td>
<td>12</td>
</tr>
<tr>
<td>2.1.1 Montage</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2 BLAST</td>
<td>14</td>
</tr>
<tr>
<td>2.1.3 CyberShake</td>
<td>16</td>
</tr>
<tr>
<td>2.1.4 PageRank</td>
<td>17</td>
</tr>
<tr>
<td>2.2 Parallel Scripting and Related Applications</td>
<td>18</td>
</tr>
<tr>
<td>2.3 Shared File Systems and Their Benchmarks</td>
<td>19</td>
</tr>
<tr>
<td>2.4 Data Flow Patterns and the Pattern Detection</td>
<td>21</td>
</tr>
<tr>
<td>3 UNDERSTANDING THE BASELINE</td>
<td>24</td>
</tr>
<tr>
<td>3.1 Understanding the I/O of Many-task Applications</td>
<td>26</td>
</tr>
<tr>
<td>3.1.1 Application Configuration</td>
<td>26</td>
</tr>
<tr>
<td>3.1.2 Application Profiling Methodology</td>
<td>28</td>
</tr>
<tr>
<td>3.1.3 Metadata Operation Distribution</td>
<td>29</td>
</tr>
<tr>
<td>3.1.4 I/O Distribution</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Defining MTC Envelope</td>
<td>36</td>
</tr>
<tr>
<td>3.3 MTC Envelope Measurements</td>
<td>37</td>
</tr>
<tr>
<td>3.3.1 Metadata Operation Throughput</td>
<td>38</td>
</tr>
<tr>
<td>3.3.2 1-to-1 Read Performance</td>
<td>39</td>
</tr>
<tr>
<td>3.3.3 N-to-1 Read Performance</td>
<td>41</td>
</tr>
<tr>
<td>3.3.4 Write Performance</td>
<td>43</td>
</tr>
<tr>
<td>3.3.5 Envelope Summary</td>
<td>44</td>
</tr>
<tr>
<td>3.4 Putting the Applications in the Envelope</td>
<td>45</td>
</tr>
<tr>
<td>3.4.1 I/O Time Consumption</td>
<td>45</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>3.4.2</td>
<td>I/O Performance Bounding Factors</td>
</tr>
<tr>
<td>3.5</td>
<td>A Review of Montage Performance</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>4</td>
<td>MANAGING RUNTIME DATA</td>
</tr>
<tr>
<td>4.1</td>
<td>AMFORA File System Architecture</td>
</tr>
<tr>
<td>4.2</td>
<td>Scalable Data Caching</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Directory Metadata Management</td>
</tr>
<tr>
<td>4.2.2</td>
<td>File Metadata Management</td>
</tr>
<tr>
<td>4.2.3</td>
<td>File Data Management and Access Consistency</td>
</tr>
<tr>
<td>4.3</td>
<td>Collective Data Movement</td>
</tr>
<tr>
<td>4.4</td>
<td>Functional Data Transformation</td>
</tr>
<tr>
<td>4.5</td>
<td>AMFORA File System Performance</td>
</tr>
<tr>
<td>4.5.1</td>
<td>MTC Envelope Benchmark Performance</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Collective and Functional Data Management Performance</td>
</tr>
<tr>
<td>4.6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>5</td>
<td>INTERACTING WITH DATA MANAGEMENT SYSTEM</td>
</tr>
<tr>
<td>5.1</td>
<td>Designing Interface between AMFORA Components</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Script - Task Engine</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Script - File System</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Task Engine - File System</td>
</tr>
<tr>
<td>5.2</td>
<td>Dispatching Tasks at Large Scale</td>
</tr>
<tr>
<td>5.3</td>
<td>Assuring Data Locality without File Usage Information</td>
</tr>
<tr>
<td>5.4</td>
<td>Scalable Task Dependency Resolving via Asynchronous File Access</td>
</tr>
<tr>
<td>5.5</td>
<td>An Early Task Engine with File-usage Aware Task Format</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Data Flow Pattern Detection with File Usage Information</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Asynchronous File Transfer</td>
</tr>
<tr>
<td>5.6</td>
<td>A Summary of Task Formats</td>
</tr>
<tr>
<td>6</td>
<td>COMPOSING PARALLEL SCRIPTS</td>
</tr>
<tr>
<td>6.1</td>
<td>AMFORA Shell Programming Model</td>
</tr>
<tr>
<td>6.2</td>
<td>AMFORA Shell Examples</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Montage</td>
</tr>
<tr>
<td>6.2.2</td>
<td>PageRank</td>
</tr>
<tr>
<td>6.2.3</td>
<td>KMeans</td>
</tr>
<tr>
<td>6.2.4</td>
<td>MonteCarlo</td>
</tr>
<tr>
<td>7</td>
<td>ACCELERATING MANY-TASK APPLICATIONS ON LARGE-SCALE COMPUTERS</td>
</tr>
<tr>
<td>7.1</td>
<td>Montage</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Data Cache</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Data-Aware Scheduling</td>
</tr>
<tr>
<td>7.1.3</td>
<td>Collective Gather</td>
</tr>
<tr>
<td>7.2</td>
<td>BLAST</td>
</tr>
</tbody>
</table>
7.2.1 Load Balancing ........................................ 91
7.2.2 Asynchronous Gather ................................ 92
7.2.3 Comparison with mpiBLAST .......................... 92
7.3 CyberShake-PostProcessing .............................. 93
7.4 PageRank, KMeans, and MonteCarlo ................. 94
  7.4.1 PageRank ............................................. 94
  7.4.2 KMeans ............................................... 95
  7.4.3 MonteCarlo .......................................... 96
7.5 Analysis .................................................. 97

8 CONCLUSIONS, CONTRIBUTIONS, AND FUTURE RESEARCH DIRECTIONS 99
  8.1 Conclusions ............................................ 99
  8.2 Contributions .......................................... 100
  8.3 Future Research Directions ............................ 102

REFERENCES .................................................. 105
LIST OF FIGURES

1.1 Parallel scripting’s position in a two-dimensional landscape of scalability and computation granularity. ................................................................. 5
1.2 Time distribution of generating a 6x6 degree 2MASS mosaic with Montage on 64 IBM BG/P processors, using GPFS. ................................. 6
1.3 AMFORA components. ................................................................................................................................. 9
1.4 AMFORA’s position in a two-dimensional landscape of scalability and computation granularity. ................................................................. 10

2.1 Montage Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production. Arrows with solid lines from boxes to ovals are file transfer. Dashed lines show dependencies that are not defined in terms of POSIX files. ................................................................. 14
2.2 Parallel BLAST Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production, and arrows with solid lines from boxes to ovals are file transfer. ................................................................. 15
2.3 CyberShake PostProcessing Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production, and arrows with solid lines from boxes to ovals are file transfer. ................................................................. 17
2.4 PageRank Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production, and arrows with solid lines from boxes to ovals are file transfer. ................................................................. 19
2.5 Data Flow Patterns ........................................................................................................................................ 23

3.1 peakGM Metadata Operations VS. Time. Operations traces are aggregated on a per second basis. ................................................................. 29
3.2 peakGM Metadata Operations vs. System Call Sequence ................................................................................. 29
3.3 mProjectPP Metadata Operations VS. System Call Sequence ........................................................................ 30
3.4 mProjectPP I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively. ................................................................................ 33
3.5 mDiff I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively. ................................................................. 34
3.6 mBackground I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively. ................................................................. 34
3.7 formatdb I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively. ................................................................. 35
3.8 blastp I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively. ................................................................. 35
3.9 merge I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively. ................................................................. 36
3.10 Metadata operation throughput. Performance increases linearly from one client to eight clients, then the increase slows down from eight clients to 32 clients. The performance starts to decrease from 32 clients dramatically.

3.11 1-to-1 Read Operation Throughput

3.12 1-to-1 Read Data Bandwidth

3.13 N-to-1 Read Operation Throughput

3.14 N-to-1 Read Data Bandwidth

3.15 Write Operation Throughput

3.16 Write Data Bandwidth

3.17 MTC Envelope (log scale). The performance on each metric (0.5 - 500) is relative to that on one client. The creation and open throughput reaches the peak first then starts to decrease, followed by write throughput, 1-to-1 and N-to-1 read throughput. The write, 1-to-1, and N-to-1 read bandwidth has a better scalability.

3.18 Heat map of write bandwidth

3.19 Heat map of write throughput

3.20 Projected I/O time distribution of generating a 6x6 degree mosaic with Montage on 64 IBM BG/P processors, using GPFS.

4.1 AMFORA File System Daemon Architecture. Directory metadata are spread and synchronized on all daemons. File metadata are spread across all daemons with a hash function. File data are primarily located on the daemon where it is produced.

4.2 Shuffle example on two AMFORA FS daemons

4.3 MTC Envelope improvements

4.4 Collective file movement performance

4.5 Shuffle performance

7.1 Montage workload time-to-solution ratio compared with the GPFS base case for the parallel stages.

7.2 Work-stealing improvement in mtcBLAST’s blastp. The vertical bars show the time-to-solution in seconds and correspond to the left Y axis. The dots show the performance improvements and correspond to the right Y axis.

7.3 Performance comparison of mtcBLAST and mpiBLAST

7.4 PageRank Performance

7.5 KMeans Performance

7.6 MonteCarlo Performance
<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Resource usage</td>
</tr>
<tr>
<td>2.2</td>
<td>Montage task names and descriptions</td>
</tr>
<tr>
<td>2.3</td>
<td>BLAST Tasks and Descriptions</td>
</tr>
<tr>
<td>2.4</td>
<td>CyberShake PostProcessing Tasks and Descriptions</td>
</tr>
<tr>
<td>2.5</td>
<td>PageRank Tasks and Descriptions</td>
</tr>
<tr>
<td>3.1</td>
<td>Basic Montage Stage Statistics</td>
</tr>
<tr>
<td>3.2</td>
<td>Basic BLAST Stage Statistics</td>
</tr>
<tr>
<td>3.3</td>
<td>Basic CyberShake PostProcessing Stage Statistics</td>
</tr>
<tr>
<td>3.4</td>
<td>Montage Metadata Operation Stats</td>
</tr>
<tr>
<td>3.5</td>
<td>BLAST Metadata Operation Stats</td>
</tr>
<tr>
<td>3.6</td>
<td>CyberShake PostProcessing Metadata Operation Stats</td>
</tr>
<tr>
<td>3.7</td>
<td>Montage I/O Stats. Trace length</td>
</tr>
<tr>
<td>3.8</td>
<td>BLAST I/O Stats. Trace length</td>
</tr>
<tr>
<td>3.9</td>
<td>CyberShake PostProcessing I/O Stats</td>
</tr>
<tr>
<td>3.10</td>
<td>Montage I/O Performance Decomposition and Bounding Factor(s) Classification</td>
</tr>
<tr>
<td>3.11</td>
<td>BLAST I/O Performance Decomposition and Bounding Factor(s) Classification</td>
</tr>
<tr>
<td>3.12</td>
<td>CyberShake I/O Performance Decomposition and Bounding Factor(s) Classification</td>
</tr>
<tr>
<td>4.1</td>
<td>AMFORA File System MTC Envelope Scalability</td>
</tr>
<tr>
<td>4.2</td>
<td>AMFORA File System MTC Envelope Scalability</td>
</tr>
<tr>
<td>5.1</td>
<td>Stage-InFile-Table</td>
</tr>
<tr>
<td>5.2</td>
<td>Subtable with stage b</td>
</tr>
<tr>
<td>5.3</td>
<td>Potential Performance Improvement Opportunities Comparison between File-usage Aware Format and File-usage Blind Format</td>
</tr>
<tr>
<td>7.1</td>
<td>Time (s) distribution in gather pattern</td>
</tr>
<tr>
<td>7.2</td>
<td>BLAST stage tasks, inputs, outputs, and input and output size</td>
</tr>
</tbody>
</table>
7.3 Number of tasks, inputs, and outputs, and input and output size, for each CyberShake PostProcessing stage
ACKNOWLEDGMENTS

My journey to this PhD is not an ordinary one. With too much time wasted during college, I was not an obvious outstanding candidate eight years ago when I first arrived in Chicago. I have a lot of appreciation for my advisor, Ian Foster, for giving me an internship opportunity when I was in the master program. My thesis research was made possible through Ian’s great vision of computing and his patient guidance. Ian has supported me on every aspect from research, funding to job search.

To Daniel S. Katz, member of my PhD committee. Our collaboration started in the summer of 2010. Ever since then, the collaboration between Dan and I has been one of the best parts of my PhD career. I am grateful with every timely response he has for my random ideas regardless if the idea is good or bad. The technical insights Dan provided helped positioning the system we developed together. My written English was significantly improved by his every correction on spelling, formatting, and grammar error.

Special thanks to my wife Kun Ma. This is the twelfth year we have been together. She agreed to come to Chicago simply because of me, though she had better opportunities elsewhere. Thanks for her understanding and accompanying, which filled these years with a lot of beautiful memories.

To my parents Fuyue Zhang and Guifang Wang. Without their generous support, it would not be possible for me to study abroad in the US. I was able to think as a kid, that is, not to filter ideas with so-called common sense. I have the strength to carry on when experiencing a failure. Many of these personal characteristics are the results of their education.

Many thanks for many people that helped me in the research and life. Rick Stevens inspired the initial version of AMFORA data management in his architecture class. Mike Wilde has tirelessly supported me on technical aspects of Swift and Argonne computing resources. Justin Wozniak, Tim Armstrong, and Allan Espinosa have provided valuable feedback in the process of designing AMFORA. Ioan Raicu, who I first worked with during
the summer internship, opened the door to research for me by giving the chance to port the Falkon system to the IBM Blue Gene/P system.

Development of this dissertation was supported by the University of Chicago’s Department of Computer Science, the Computation Institute of the University of Chicago and Argonne National Laboratory, and the U.S. Department of Energy under the ASCR X-Stack program (DE-SC0005380) and DE-AC02-06CH11357. Computing resources were provided by Argonne Leadership Computing Facility.
ABSTRACT

Many-task computing (MTC) applications assemble existing sequential (or parallel) programs, using POSIX files for intermediate data. The parallelism of such applications often comes from data parallelism. MTC applications can be grouped into stages, and dependencies between tasks in different stages can be in the form of file production and consumption. The computation stage of MTC applications can have a large number of tasks, thus it can have a large amount of I/O traffic (metadata traffic and I/O traffic) which is also highly concurrent. Some MTC applications are iterative, where the computation iterates over a dataset and exit when some condition(s) are reached. Some MTC applications are interactive, where the application requires human action between computation stages.

MTC applications have been seen in many scientific research domains: astronomy, biochemistry, bioinformatics, psychology, economics, climate science, physical chemistry, and neuroscience. These applications cover a wide range of methodologies, such as rational design, uncertainty quantification, parameter estimation, massive dynamic graph pruning, Monte-Carlo-based iterative fixing, and inverse modeling. Scientists have used numerous parallel programming languages/models (for example, MPI, Hadoop, Swift, Makeflow, and Pegasus) to execute the MTC applications on large-scale computers. However, for some of these languages/models, such as MPI and Hadoop, the mismatch between the languages and the applications results in a loss of conciseness, as programmers work on in-memory variables while the applications operate on files and directories. In some cases, programmers need to implement complex data management logic inside the program to achieve high efficiency, and the solution is often ad-hoc. Out of these approaches, Swift and Makeflow are functional parallel scripting languages. They fit MTC applications naturally, but they lack efficient in-memory data management since they originated from a distributed environment (grid computing). So the execution of these parallel scripting programs is usually inefficient, due to the sheer amount of application I/O and the underoptimized shared file system.

Thus, there is not a single parallel scripting framework that simultaneously provides a
concise programming interface, general efficient execution, and scalable performance. More specifically, a concise programming interface could avoid changes to the original application code and preserve the way the application is invoked. General efficient execution requires an application-independent performance improvement scheme. And the parallel scripting framework should be able to work on thousands of compute nodes with acceptable overhead. Providing these three features at the same time is challenging. For example, one common approach to improve execution performance is through data-aware scheduling, where the scheduling system needs the file usage information (input/output), while the UNIX/Linux command line parameters are usage-blind. Caching is another common approach to improve execution performance. To preserve the original applications’ POSIX interface, the parallel scripting framework should be able to cache the data in RAM with the data being accessible with POSIX interface (an in-RAM file system). However, at the time this thesis research was conducted, mainstream file systems usually had one metadata server, which is insufficient for the I/O traffic of some MTC applications. (There have been later developments of distributed metadata servers, which balances the metadata over multiple nodes, but this is sub-optimal for MTC applications.)

In this dissertation we develop a complete parallel scripting framework called AMFORA, which has a shared in-RAM file system and task execution engine. It implements the multi-read single-write consistency model, preserves the POSIX interface for original applications, and provides an interface for collective data movement and functional data transformation. It is interoperable with many existing serial scripting languages (e.g., Bash, Python). AMFORA runs on thousands of compute nodes on an IBM BG/P supercomputer. It also runs on cloud environments such as Amazon EC2 and Google Compute Engine. To understand the baseline MTC application performance on large-scale computers, we define MTC Envelope, which is a file system benchmark to measure the capacity of a given software/hardware stack in the context of MTC applications.

The main contributions of this dissertation are:
1. A system independent approach to profile and understand the concurrency of MTC applications’ I/O behavior

2. A benchmark definition that measures the file system’s capacity for MTC applications

3. A theoretical model to estimate the I/O overhead of MTC applications on large-scale computers

4. A scalable distributed file system design, with no centralized component, that achieves good scalability

5. A collective file system management toolkit to enable fast data movement

6. A functional file system management toolkit to enable fast file content transformation

7. A new parallel scripting programming model that extends a scripting language (e.g., Bash)

8. A novel file system access interface design that combines both POSIX and non-POSIX interfaces to ease programming without loss of efficiency

9. An automated method for identifying data flow patterns that are amenable to collective optimizations at runtime

10. The open source implementation of the entire framework to enable MTC applications on large-scale computers
CHAPTER 1
INTRODUCTION

Many scientific applications are constructed by gluing together legacy sequential or parallel programs via a file system abstraction. Task dependencies can be represented by file production and consumption. These application can have a huge number of tasks, thus with a large amount of I/O that is also highly concurrent. We call these applications many-task computing (MTC) applications (through this dissertation MTC applications and parallel scripting applications are used interchangeably). Domain scientists have used many approaches to run MTC applications on large-scale computers, as their problem sizes increase and large-scale computers offer opportunities for massive parallel execution. Some of the approaches can achieve high efficiency, but the solutions are application specific and maintaining these application-specific parallel framework can be labor intensive. Parallel scripting is a powerful and convenient tool to construct MTC applications without modifying the original programs. But each such system introduces the need to adopt a somewhat specialized programming model, and the lack of efficient data management results in low efficiency.

In this dissertation, we seek to enable concise, fast, and scalable parallel scripting execution on large-scale computers. Specifically, by profiling a group of MTC applications, we identify that the MTC application performance bottleneck on large-scale computers lies in the lack of efficient data management. We also show that the problem can be overcome through the use of scalable algorithms for data caching, movement, and transformation. By exposing the underlying data management functionality through a POSIX compatible command line interface, programmers are able to compose concise parallel scripts using serial scripting languages (e.g., Bash, Python, Ruby) with slight changes and tags added to the serial scripts. This introduction provides an overview of the research and a roadmap of my dissertation.

This dissertation is a fusion of my previous work, including an application survey [43], application and file system profiling [81], data management [82], and a parallel scripting
1.1 A Review of Many-task Applications

Many-task Computing was first defined in Raicu’s Ph.D thesis [61], where MTC was described as high-performance computations that comprise multiple distinct activities, coupled via file system operations or message passing. Tasks can be either small or large (in terms of task length), serial or parallel, and compute-intensive or data-intensive. The set of tasks can be static or dynamic, homogeneous or heterogeneous, loosely coupled or tightly coupled. Often, the number of tasks, quantity of computing, and volume of data can be extremely large.

MTC applications have been widely used by scientists in fields of astronomy [62, 42, 44], biological science [63, 58, 4, 47], chemistry [39], earth science [46, 18, 33], economics [28], material science [25], and many others. And researchers have enabled such applications on many types of platforms such as clusters, clouds, supercomputers, and grids [86, 2, 6, 48, 75].

In this dissertation, We focus on the MTC applications that are composed by serial tasks, where each task requires exactly one processor. In addition to the properties defined previously, there are some additional MTC application properties regarding data flow and control flow:

- Besides the sheer amount of I/O produced and consumed by tasks, the I/O traffic is also highly concurrent.

- The data flow between application stages has patterns of pipeline, multicast, gather, scatter, allgather, and alltoall.

- MTC applications can be iterative and interactive.

All of these properties imply potential challenges for fast and scalable execution of MTC applications on large-scale computers.
1.2 An Overview of Large-scale Computers

Today’s most powerful large-scale computers (e.g., IBM Blue Genes and Crays) offer a massive parallel execution capability, which meets the MTC applications’ requirement of a huge amount of computing resources. These machines have a large number of compute nodes (∼10,000 at the time that this dissertation is being written). Each compute node is equipped with a multi-core processor and limited amount of RAM. There is no local disk on the compute nodes. The compute nodes are connected by at least one low-latency, high-bandwidth network, laid out in system-specific topologies. Some machines have multiple networks, for example, for communication, I/O, and synchronization. Large-scale computers may have one or more shared file systems (e.g. GPFS [66], PVFS [19], and Lustre [26]), and the shared file systems are treated as peripheral devices.

1.3 Existing Approaches to Running Many-task Applications

There are numerous parallel programming languages/models (MPI, Hadoop, Swift, Makeflow, Pegasus and others) that domain scientists can use to execute MTC applications on large-scale computers.

One approach to parallelizing an MTC application is to rewrite it as a monolithic program using a parallel library or language such as MPI or PGAS. Communications that originally occurred via file system operations then occur via messaging. However, this approach can be labor intensive and can lead to code maintenance problems, if the original program’s authors continue to evolve it independently of the parallelization. For example, while the popular mpiBLAST [45] rewrite of BLAST has good parallel scalability and high efficiency, the substantial effort required to re-engineer BLAST for MPI execution means that its latest version was built from NCBI BLAST 2.2.20, while BLAST itself has evolved to version 2.2.26 as of April 2012. Moreover, the mismatch between the parallel library or language and the applications results in a loss of conciseness, as the programmers work on in-memory variables.
while the applications operate on files and directories. In some cases, programmers need to implement complex data management logic inside the parallel framework to achieve high efficiency, and such solutions are often ad-hoc and specific to one application.

The other approach for such applications is through a parallel scripting language, such as Swift [76] and Makeflow [2]. The user represents the task graph by a script, which is compiled to tasks when running. Tasks are distributed to computing resources via the runtime scheduler. Files are used for intertask communication, and are stored on shared file system for further access. With this approach, the users do not have to change the original serial application code, and substituting one version of a program for another in a script is a trivial task, unless the program’s interface changes substantially. However, we can encounter scaling problems in the scripting paradigm, due to the bottleneck inherent in having all interprocess communication occur via file system operations. Even if a parallel computer provides a shared file system accessible from all nodes, the volume and concurrency of file system operations generated by an MTC application will often be overwhelming [83] and will often result in inefficient execution.

Figure 1.1 shows the position of parallel scripting in a parallel system space that has two dimensions: scalability and computation granularity. Legacy programming languages (e.g. C, C++, Fortran, Java) are often successfully used to solve most problems on a single processor, particularly when the computation has a fine granularity. Shared-memory based programming languages such as Pthreads [17] and OpenMP [21] can be used on a compute node with multiple processors. In the situation where there are multiple compute nodes, one might consider a shared-memory programming language such as a PGAS [10] language or a message-passing programming library such as MPI [34]. MapReduce [22] can be applied when the applications are based on the key-value pair data structure and require a large number of compute nodes. When the computation granularity is coarse, scripting languages such as Bash are often chosen to work on a single node. Parallel scripting languages can be used to compose applications that have coarse computation granularity and are required to
execute at large-scale.

1.4 Montage Performance: A Case Study

The Montage [42] astronomy image processing application assembles large mosaics from multiple small images obtained from telescopes, while preserving the amount and position of the energy. It is a typical MTC application, in that it comprises many (in this case, $O(10,000)$) tasks that are loosely coupled via POSIX read and write operations on a shared file system.

To understand the present capability of MTC applications with the existing software-hardware stack on large-scale computers, we run a Montage benchmark problem on 64 processors of an IBM BG/P with intermediate results stored on GPFS. Figure 1.2 shows the CPU-time distribution of the stages that can run in parallel and those that perform
large amounts of I/O. On this small number of processors: 44.3% is consumed by I/O; 35.5% is CPU Time; 3.1% is the idle time caused by the sequential part of the program, where there is only one task running while other processors are sitting idle; 16% of is idle time caused by a particular data flow pattern (gather in this case), in which all but one processor sit idle while data is fetched from GPFS; 1.1% of is wasted by the trailing task of each stage, where there are not enough tasks to feed all processors.

From this profiling, we see that data management is a performance bottleneck in the Montage application’s parallel execution framework, and it is generally the bottleneck for MTC applications on large-scale computers. We observe two common overhead sources:

- A great portion of the I/O time consumption in MTC applications is from unnecessary data transfer between compute nodes and the persistent storage.

- Some idle time are caused by particular data flow patterns such as gather.

Thus we see a great opportunity for an efficient data management solution that can significantly improve overall application performance. This solution needs to tackle two

![Time distribution of generating a 6x6 degree 2MASS mosaic with Montage on 64 IBM BG/P processors, using GPFS.](image)

Figure 1.2: Time distribution of generating a 6x6 degree 2MASS mosaic with Montage on 64 IBM BG/P processors, using GPFS.
sub-problems: scalable data caching, and fast data flow pattern specific movement.

1.5 The Challenges of Efficient Data Management at Large Scale

**Challenge 1:** The first challenge that we address in this dissertation is to understand the baseline performance of MTC applications on the existing software-hardware stack. This challenge can be decomposed into several sub-questions: What is the I/O behavior of MTC applications in terms of I/O quantity, I/O concurrency, I/O pattern, and data flow patterns? How does each of such behavior quantitatively map to the existing software-hardware stack? What are the factors that slow MTC application performance and limit MTC application scalability on large-scale computers? What is a representative performance benchmark suite to measure the capacity of MTC applications on large-scale computers?

**Challenge 2:** Caching should be able to improve MTC application performance since it can eliminate unnecessary data movement between compute nodes and the shared file system. However, enabling data caching at large scale is challenging. The huge number of intermediate files in MTC applications requires a scalable data caching mechanism so that the producers can update the metadata (location, size) of the intermediate files and the consumers can access the data.

**Challenge 3:** Data movement at large scale is another challenge. We have identified a group of primitive data flow patterns in MTC applications: multicast, scatter, gather, allgather, and alltoall. For example, a sequential multicast algorithm on 1,024 compute nodes uses 1,024 connections, which incurs a significant amount of latency overhead. A minimum-spanning tree (MST) algorithm is able to complete the multicast on 1,024 compute nodes in 10 sequential transfers when the transfer is latency overhead dominated. Thus, we need to investigate which algorithm is faster under which conditions and provide an interface to programmers.

**Challenge 4:** With the performance improvements from data caching and fast movement, how to expose these system improvements to programmers is a fourth challenge. We
could expose the interface so that programmers explicitly call those primitives in their scripts. A second solution is to build the runtime system with the logic to detect data flow patterns and call the primitives automatically. The second approach relieves the programmers from determining the data flow pattern, but automated data flow pattern detection might be limited by the information the task description conveys to the runtime.

**Challenge 5**: MTC application tasks often use a POSIX-compatible interface to access the file system. So the last challenge that we address is to define a set of commands that can be inserted into serial scripts while preserving the semantics of the original application’s POSIX interface, and which can be implemented in a manner that permits efficient execution and data movement (e.g., data-aware scheduling and collective data movement).

### 1.6 An Overview of the Solution

Throughout this dissertation, we will present the AMFORA system as the solution to enable concise, fast, and scalable parallel scripting on large-scale computers through efficient data management. The AMFORA system has three major components: the parallel scripts, the AMFORA Task Engine and the AMFORA File System, as shown in Figure 1.3.

The AMFORA File System is a shared in-RAM file system, so the MTC applications’ data can be cached in RAM. The AMFORA File System is POSIX compatible, so the original serial applications do not have to be changed. The AMFORA File System implements multi-read single-write consistency, which meets MTC applications’ I/O consistency requirement. The AMFORA File System provides interfaces for collective data movement and functional data transformation. The AMFORA File System’s performance scales to thousands of compute nodes. The AMFORA Task Engine can queue tasks submitted from the parallel scripts and dispatch all tasks to all available resources upon instruction. Users can use many existing serial scripting languages (e.g., Bash, Python) with the AMFORA interface (referred as AMFORA Shell in the rest of this dissertation) to compose parallel scripts. The AMFORA system runs on supercomputers, clouds, and commodity clusters.
Figure 1.3: AMFORA components.

Figure 1.4 presents the positioning of AMFORA in a two-dimensional parallel programming tool space of scalability and computation granularity. AMFORA fills the role of a parallel programming tool with coarse computation granularity at large scale.

1.7 Advantages and Limitations of the Solution

Serial Scripting languages (such as Bash) with AMFORA Shell can express parallel scripting applications that feature simple data flow, iterative computation, and interactive analysis. It provides efficient execution because:

- AMFORA caches files in memory;
- AMFORA’s scalable file access makes scripting performance scalable;
- the AMFORA execution engine interface simplifies the parallelization of Bash scripts;
- the collective data management interface and the underlying system support moving data efficiently and scalably; and
Figure 1.4: AMFORA’s position in a two-dimensional landscape of scalability and computation granularity.

- the AMFORA interface to persistent storage enables checkpointing by copying or moving the files from AMFORA to a persistent storage location.

AMFORA has limitations: it requires that all data fit in the compute node’s memory. It can not launch tasks that need multiple compute nodes. The application shell commands are not declarative, so the AMFORA Task Engine doesn’t know which files in the command line are inputs or outputs. Optimization based on declaring file usage (such as data aware scheduling) does not work with AMFORA Shell. AMFORA prefers longer tasks than shorter tasks. Finer-grained data flow based applications can be better addressed by a data flow based parallel programming language, such as Swift/T [78] or DAGuE [14].
1.8 Dissertation Roadmap

The rest of dissertation is organized as follows, with each chapter presenting part of my work in greater detail:

**Chapter 2: Background Information and Related Work** surveys the previous work on parallel scripting and related applications, collective data management schemes, shared file system benchmarks, and data flow patterns and pattern identification.

**Chapter 3: Understanding the Baseline** presents work profiling three representative MTC applications’ I/O behavior on large computers, where we summarize the common I/O traffic, data flow patterns, and I/O pattern of MTC applications. MTC Envelope is defined in this chapter as the benchmark of a large-scale computer’s capacity for MTC applications. This chapter addresses **Challenge 1**.

**Chapter 4: Managing Runtime Data** addresses data caching, data movement, and transformation at large scale. This chapter presents the transfer and transformation algorithms, data flow pattern detection with file usage information, and the technical approaches to ensure data locality in situations with and without file usage information. This chapter addresses **Challenge 2** and **3**.

**Chapter 6: Composing Parallel Scripts** explains how the AMFORA programming interface is designed, and the interoperation of AMFORA Shell and other serial scripting languages to compose parallel scripts. This chapter addresses **Challenge 4** and **5**.

**Chapter 7: Accelerating Many-task Applications on Large-scale Computers** shows the performance improvements both from the benchmark perspective and the overall application perspective.

**Chapter 8: Conclusions, Contributions and Future Research Directions** re-states the contributions of this thesis, draws conclusions, and provides future research directions in the context of parallel scripting.
CHAPTER 2
BACKGROUND INFORMATION AND RELATED WORK

This chapter presents the four representative MTC applications used in the research, existing work of parallel scripting, and relevant work of MTC Envelope and AMFORA.

2.1 Representative MTC Applications

In previous work [43], my colleagues and I studied 12 applications that feature MTC characteristics: AstroPortal [62], PTMap [20], OOPS [39], DOCK6 [63, 58], Montage [42, 44], Social Learning Strategy [64], BLAST [4, 45, 47], RDCEP [28], SYNAPPS [53], Deem’s Database of Hypothetical Zeolite Structures [25], fMRI [49, 32], and Model SEED: Genomescale Metabolic Models [38, 37]. Those applications cover the research domain of astronomy, biochemistry, bioinformatics, psychology, economics, climate science, physical chemistry, and neuroscience. We have chosen four applications from above: Montage, BLAST, CyberShake PostProcessing [46, 18, 33], and PageRank. These four representative applications will be used to study the baseline performance of existing software-hardware stack, the programmability of the new programming model, and the performance improvements of the data management system. These four applications span a range of CPU, memory, and disk I/O needs, as shown in Table 2.1.

<table>
<thead>
<tr>
<th>Application</th>
<th>CPU</th>
<th>Memory</th>
<th>Disk I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Montage</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>CyberShake</td>
<td>Mid</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>BLAST</td>
<td>High</td>
<td>Mid</td>
<td>Mid</td>
</tr>
<tr>
<td>PageRank</td>
<td>Mid</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>
2.1.1 Montage

The Montage [42] astronomy image processing application builds mosaics from small images obtained from telescopes while preserving the amount and the position of the energy between the input and output images. Initial parallel implementations were constructed in MPI and Pegasus [44]; other systems, including Swift [87, 51], have subsequently been used to develop additional versions.

In the MTC version of Montage, we divide the code into eight stages. The name and detailed explanation of the stages are shown in Table 2.2.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mProject</td>
<td>inputs raw image files, outputs reprojected images, pulls one line out from output image that will be consumed by mImgtbl</td>
</tr>
<tr>
<td>mImgtbl</td>
<td>takes the one line output from mProject and concatenates them into one file</td>
</tr>
<tr>
<td>mOverlaps</td>
<td>analyses the image table, produces a metadata table describing which images overlap along with a task list of mDiffFit tasks (one for each pair of overlapping images)</td>
</tr>
<tr>
<td>mDiffFit</td>
<td>inputs two overlapping output files from mProject, fits a plane to the overlap region</td>
</tr>
<tr>
<td>mConcatFit</td>
<td>gathers all output data from the previous stage (coefficients of the planes), and summarizes them into one file</td>
</tr>
<tr>
<td>mBgModel</td>
<td>analyses the metadata from mImgtbl and the data from mConcatFit, creating a set of background rectification coefficients for each image, then generates a mBackground task list</td>
</tr>
<tr>
<td>mBackground</td>
<td>applies coefficients to the reprojected images</td>
</tr>
<tr>
<td>mAdd</td>
<td>reads output files from mBackground, and writes an aggregated mosaic file</td>
</tr>
</tbody>
</table>

As shown in Figure 2.1, the mImgtbl stage involves a single task that gathers the data output by the processes involved in the mProject stage. The mImgtbl task produces a single file that is then consumed by the mOverlaps task. mOverlaps produces the task list for the next stage. From the mProject stage to the mDiffFit stage, each task in mDiffFit fetches two files from mProject. Then from the mDiffFit stage to the mConcatFit stage is a Gather
pattern 2: \texttt{mConcatFit} will process all outputs from \texttt{mDiffFit} and summarize them into one file. There is a producer-consumer from \texttt{mConcatFit} to \texttt{mBgModel}: \texttt{mBgModel} produces a task list for the next stage of computation. Again, from \texttt{mProject} to \texttt{mBackground}, there is a parallel producer-consumer relation: each of the \texttt{mBackground} tasks processes exactly one output of \texttt{mProject}. Finally, the data flow between \texttt{mProject} and \texttt{mAdd} features a Gather pattern 3.

![Montage Data Flow](image)

Figure 2.1: Montage Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production. Arrows with solid lines from boxes to ovals are file transfer. Dashed lines show dependencies that are not defined in terms of POSIX files.

### 2.1.2 BLAST

BLAST (The Basic Local Alignment Search Tool) searches one or more nucleotide or protein sequences against a sequence database, and calculates similarities. It has been parallelized with different frameworks. For example, mpiBLAST [45] wraps BLAST as a library within an MPI framework, CloudBLAST [48] uses MapReduce as the wrapper, and Parallel...
BLAST [47] uses PVM as the parallel framework and maintains a POSIX file interface among stages. We start with Parallel BLAST as our base case, and scale this implementation with our runtime and data management system.

Figure 2.2 shows the data flow patterns in Parallel BLAST. Table 2.3 explains the tasks in each stage of Parallel BLAST.

Figure 2.2: Parallel BLAST Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production, and arrows with solid lines from boxes to ovals are file transfer.
Table 2.3: BLAST Tasks and Descriptions

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastasplit</td>
<td>splits the database into several slices</td>
</tr>
<tr>
<td>formatdb</td>
<td>formats each database slice</td>
</tr>
<tr>
<td>blastp</td>
<td>searches protein sequence against database slice</td>
</tr>
<tr>
<td>merge</td>
<td>merges the results for one each query</td>
</tr>
</tbody>
</table>

Table 2.4: CyberShake PostProcessing Tasks and Descriptions

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract</td>
<td>extracts a subset of the SGT files, produces sub-SGT files</td>
</tr>
<tr>
<td>seis</td>
<td>takes one subSGT file from extract and one variation file, computes the seismic reciprocity, writes result in a 24-KB file</td>
</tr>
<tr>
<td>peak</td>
<td>takes the output from seis stage, computes the peak spectral acceleration, writes a 214-B file.</td>
</tr>
</tbody>
</table>

2.1.3 CyberShake

CyberShake [46, 18, 33], an application from the Southern California Earthquake Center (SCEC) that finds the peak ground movement at a physical site caused by a set of possible earthquakes, has two major steps. The first generates Strain Green Tensors (SGTs). This runs two MPI calculations that produce two SGT files for the physical site. The second step is PostProcessing, which performs a parameter sweep over two dimensions: ruptures that could affect the site and the variations of every rupture. For each rupture, PostProcessing extracts a subset of the SGT files, and then for each variation of that rupture, PostProcessing calculates a computational seismogram then finds the peak ground motion in the seismogram. Table 2.4 describes each stage in detail.

Figure 2.3 shows the data flow of the CyberShake PostProcessing application. There is a multicast data flow pattern from the extract stage to the seis stage. Then there is a pipeline pattern from the seis stage to the peak stage. One thing to notice is that in peak stage, the input file usage varies significantly. A simplistic data-aware scheduling will incur severe load imbalance.
Figure 2.3: CyberShake PostProcessing Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production, and arrows with solid lines from boxes to ovals are file transfer.

2.1.4 PageRank

PageRank [54, 15] is a widely used algorithm for object ordering. And it is one of the several motivating application of the Hadoop [7] distributed computing framework. In my dissertation work, we revisited the PageRank algorithm, and implemented a parallel scripting version PageRank with POSIX files as intermediate data format. Table 2.5 explains each stage in details.

Figure 2.4 presents the data flows in PageRank algorithm. The initial file score is consume by all Distribution tasks, thus it is a multicast pattern. It is the pipeline pattern between Distribution and Partition stage, Concat2 and Sum stage respectively. The data flow between the Partition and Concat1 stages is an alltoall data flow pattern. It is a gather pattern between the Sum stage and the Concat stage.
Table 2.5: PageRank Tasks and Descriptions

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distribution</td>
<td>distributes the initial score from the source to the targets, writes target and score to files</td>
</tr>
<tr>
<td>Partition</td>
<td>partitions the output files of previous stage based on the hash value of the target, groups the key-value pairs into a set of output files</td>
</tr>
<tr>
<td>Concat1</td>
<td>reads the previous output files with the same hash value of the keys, concatenates them into a single file</td>
</tr>
<tr>
<td>Sum</td>
<td>add together the values in previous output file, the sum is the new score of the target page</td>
</tr>
<tr>
<td>Concat2</td>
<td>concatenates all previous output files into one single output file and sort it base on the target page name</td>
</tr>
</tbody>
</table>

2.2 Parallel Scripting and Related Applications

Swift [76] and Makeflow [3] use a functional programming approach to compose and execute parallel scripting applications. Programmers can declare application tasks as functions and compose execution logic with control flow.

Many parallel scripting applications leverage the parallelism that MapReduce offers, including for example CloudBLAST [48], which runs the BLAST sequence alignment computation on clouds using Hadoop [7]. Hadoop, Amazon EMR [5], and Azure MapReduce [50] have been evaluated [35] as parallel frameworks for CAP3 [40], a sequence assembly application. High energy physics text processing [35] is run in parallel with MapReduce.

Twister [27] and HaLoop [16] incorporate iteration support to accommodate iterative applications. Hadoop Streaming has been used for parallel execution of the parallel scripting applications, because it has the flexibility to run user-defined mapper and reducer functions as executables regardless of implementation language. However, this approach requires that all communications occur via standard input and standard output. MARISSA [23] runs Hadoop on a POSIX file system to eliminate this limitation.

Bash over shell pipes [74] has been studied to enable parallel execution of Bash scripts.
Figure 2.4: PageRank Data Flow. Ovals represent tasks and boxes represent files. Arrows with solid lines from ovals to boxes are file production, and arrows with solid lines from boxes to ovals are file transfer.

The programming abstraction here is key-value pairs. Computation stages exchange information through key-value pair aggregation.

2.3 Shared File Systems and Their Benchmarks

There are many well known and deployed shared file system such as GPFS [66], PVFS [19], and Lustre [26]. GPFS and Lustre process metadata operations with segregated servers, so that the bulk I/O operations can be distinct from the metadata operations. The latest GPFS and PVFS versions support distributed metadata servers and dynamic metadata balancing.
with similar techniques to GIGA+ [57, 56]. GPFS and Lustre use locking mechanisms to preserve file system consistency, while PVFS solves the issue by a defined sequence of data and metadata operations without locking. HDFS [13], in the context of MapReduce [22], is a special case, since it is a shared persistent storage as well as a runtime file system. HDFS’s characteristics fit the MTC runtime file system requirements to some extent. However, HDFS has a single metadata server, which limits the scalability of the system, and it doesn’t support data flow pattern optimization, since it is dedicated to MapReduce paradigm.

We have also seen work that aggregates RAM to enhance a clusters’ capability to process some type of applications. MosaStore [1] can aggregate the compute nodes’ RAM disk and provide a POSIX interface to the compute nodes. The original MosaStore can not address the MTC runtime file system issue at a large scale, because its metadata is managed on a single node. RamDisk [30] and dRamDisk [65] aggregate local and remote RAM and provide a uniform memory address space. DASH [36, 52] does a similar thing on clusters. Though these systems could transparently provide a larger memory space, they can not solve the MTC runtime file system issue because MTC relies on data in the format of POSIX files.

Derived from the cosmic microwave background application [11], the MADbench2 [12] benchmark can measure the system’s unique (one-file-per-process) and shared file (one-file-for-all-processes) access performance. It could also measure the improvement of the asynchronous I/O with computation and I/O overlapped on the timeline. The MADCAP application can be viewed as an MTC application due to its inter-process communication and data flow pattern. Thus the MADbench2 benchmark provides some insights into the ability to support MTC applications. However, as the traditional parallel file system does not expose data locality information, there is no way to find out the capacity of locality exposure with this benchmark. The Effective I/O Benchmark [60] is a purely synthetic benchmark with predefined configurations and runs for a limited time to output a score as the performance metric. It is difficult to reason about the real application performance with the Effective I/O Benchmark performance. The SPIObench [69] and PIORAW [72] benchmark are synthetic
benchmarks to measure the concurrent read/write operations on unique files. They could address the pipeline pattern for MTC applications, but not for Gather, Scatter, or Multicast patterns. The IOR [41] benchmark is a highly-parametrized HPC storage benchmark, and previous work [67] has used it to model and predict the HPC application I/O performance with specified parameters. However, it lacks the measurement of the locality exposure as locality is not the concern of the traditional HPC storage systems.

2.4 Data Flow Patterns and the Pattern Detection

Data flow patterns have been well studied in previous papers [79, 75]. Building on our previous work [43], we define six primitive data flow patterns, as shown in Figure 2.5:

- **Scatter**: A set of data must be distributed to a set of subsequent tasks. The data to be distributed could be the output of a previous task, or it could be data from the global file system. For example, a single task might extract a group of N pieces from a big database, and send one piece to each of N subsequent tasks for further processing.

- **Multicast**: One piece of data is consumed by more than one subsequent task. An example is a database processed by a group of tasks on a set of nodes.

- **Gather**: A small number of tasks take as input the output from a large number of tasks. Examples include a task that checks the results of previous tasks for a convergence criterion, or a task that calculates summary statistics from the output of many tasks.

- **Allgather**: A number of tasks take as input all the output from immediate previous tasks. Examples include a number of tasks that run distinct computation on a unique set of output of many tasks.

- **Pipeline**: A set of tasks operate on given data in sequence, with the output of one task becoming the input of the next. Examples include preprocessing the input file, then apply the analysis to the preprocessed file.
• Alltoall: A set of tasks read a partition of every output from previous tasks, and reorganize the partitions into a new set of output. Examples include the key-value pair based applications such as WordCount, Distributed Grep, and PageRank [15].

Data flow patterns can be identified by the user, the compiler, or an optimizer.

**User Input:** MPI [71, 70] and PGAS [55] programming languages require users to explicitly declare when and where to invoke collective primitives. It is then the user’s responsibility to guarantee the correctness of synchronization among processes. Recent work in MosaStore [73] includes pattern identification, based on Swift’s collective data management (CDM) [77]; both require users to explicitly define data flow patterns based on a regular expression of the file names.

**Compiler Analysis:** Pegasus [68] uses label-based task clustering to optimize data locality in the pipeline data flow pattern for inter-site task scheduling. This idea can also be applied to intra-site scenarios. Nevertheless, this work employs a static approach at compile time, while our solution is based on runtime analysis.

**Application Profiling:** Preissl et al. [59] propose to identify hand crafted collective operations in MPI by profiling the execution trace. The issue solved by this work is conceptually equivalent to ours. But in the parallel scripting scenario, there is no explicitly annotated data transfer using send() or recv(). Instead, the data transfer is shown by a file being an input or an output.
Figure 2.5: Data Flow Patterns

- Scatter
- Gather
- Multicast
- Allgather
- Pipeline
- Alltoall

Legend:
- Job
- Data
- Data Generation
- Data Transfer

23
CHAPTER 3
UNDERSTANDING THE BASELINE

Running parallel scripting applications is considered an inefficient usage of supercomputers by many users and administrators. However, there is not a quantitative study showing how slow the performance is or how inefficient it is. Thus, this chapter’s goal is to quantitatively evaluate supercomputers’ capacity for running parallel scripting applications and to investigate the root cause of the slow performance. The parallel scripting application capacity of supercomputers are measured with an I/O benchmark suite called MTC Envelope, which consists of eight I/O performance metrics that represent the parallel scripting applications’ I/O behavior.

We first profile the applications’ I/O behavior on large-scale computers, and summarize the common I/O patterns such as operations, consistency, and concurrency, and the performance metrics that can characterize applications’ I/O performance. Secondly, by measuring those performance metrics on existing large-scale systems, we can determine the MTC Envelope at various scale and see the scalability curves of the metrics. The third step is to map the applications’ I/O behavior on the measured envelope so that we can determine the limiting factor of efficient performance at large scale. Finally, we compile the measured envelopes into a user guide, so that programmers can estimate the rough I/O time consumption of their application, given the concurrency and I/O traffic characteristics of the applications.

We select three representative parallel scripting applications: Montage [42], BLAST [4] and CyberShake PostProcessing [46]. We run all stages (fourteen in total) of these applications and trace the file system related system calls, then align the trace of the tasks in one stage with the system call sequence order from the profile of that stage. We identify four common concurrent file system accesses from those application profiles: file open, file create, read, and write. Parallel scripting applications feature a multi-read-single-write I/O pattern, where a file can be read many times, but only written once. Within the multi-read pattern, we further classify read access as 1-to-1 read and N-to-1 read. 1-to-1 read refers to
the case where each task reads one distinct file, while N-to-1 read refers to the case where many tasks read one common file.

We therefore define the MTC Envelope as eight curves, showing

- file open operation throughput,
- file creation operation throughput,
- 1-to-1 read data throughput,
- 1-to-1 read data bandwidth,
- N-to-1 read data throughput,
- N-to-1 read data bandwidth,
- write data throughput, and
- write data bandwidth

on the $y$ axis, versus the number of shared file system clients on the $x$ axis.

With these eight curves measured on the GPFS system of the Intrepid BG/P supercomputer, we are able to classify the application stages by the factors that bound them. An application can be metadata bound or I/O bound. In the I/O bound case, they can further be classified as operation throughput bound, data bandwidth bound or concurrency bound. To guide users programming parallel scripting applications on systems with shared file systems, we compile benchmark performance data on operation concurrency and amount of I/O into two heat maps of projected I/O throughput and bandwidth for each file system. This allows the programmer to roughly estimate if the application can run efficiently on a given computer, what is an efficient scale for the application, and how much time the I/O operations will take.

In this first attempt to characterize the MTC Envelope of large-scale systems we assume two realistic constraints: we evaluate a set of existing, unmodified application scripts as they are currently implemented by their science communities (but transcribed into Swift for ease of testing); and we measure the behavior of the current production BG/P file system with its standard configuration, typical background workload, and performance limitations. Note
that we run many of our performance measurements on I/O nodes (referred to as GPFS clients). The reader should consider that each of these I/O nodes is associated with 64 compute nodes, or 256 compute cores.

Regarding definitions, throughout this chapter, we refer to the entire parallel script used to execute a scientific task as the application, and examine in detail the various stages of processing within these application scripts. A task within these stages refers to the invocation of a single application program by the parallel script. Further, we model and examine the throughput of the application script in terms of the rate at which it performs I/O operations, and the bandwidth in terms of the rate of data transfer in units of (scaled) bytes per second.

The contributions of this work include a novel platform-independent approach to understand the concurrency of parallel scripting applications’ I/O behavior, a suite of performance metrics that characterize parallel scripting applications’ I/O behavior and measure later system improvements, and a guide for parallel scripting application writers to make better use of the existing hardware-software stack.

### 3.1 Understanding the I/O of Many-task Applications

In this section, we seek to profile the I/O behavior of the parallel stages of Montage, BLAST, and CyberShake (ten stages total). We are particularly interested in the I/O traffic distribution (both frequency and quantity) and metadata traffic distribution, which will lay out the foundation to study the relationship between the I/O behavior and system performance in Section 3.3 and Section 3.4.

#### 3.1.1 Application Configuration

For Montage, we run a 6x6 degree 2MASS mosaic centered at galaxy m101. Montage has four parallel stages: mProject, MDiff, mFit, and mBackground. For BLAST, we search the first 256 sequences in the NCBI nr database against the database itself. BLAST has three
parallel stages: formatdb, blastp, and merge. For CyberShake PostProcessing, we run the post-processing workload against the geographic site TEST. CyberShake PostProcessing has three parallel stages: extract, seis, and peakGM.

Table 3.1, 3.2 and 3.3 show the basic statistics of Montage, BLAST, and CyberShake PostProcessing respectively: number of tasks, number of input/output files, and total input/output amounts. Note that the ratio between the input amount and the number of input files denotes how many bytes are read from each input file, while a file or part of a file can be read many times.

Table 3.1: Basic Montage Stage Statistics. Tasks, input files, output files, input amount, output amount refer to number of tasks, number of input files, number of output files, total amount of input and total amount of output respectively. Input amount and output amount are measured in GB.

<table>
<thead>
<tr>
<th>stage</th>
<th>mProject</th>
<th>mDiff</th>
<th>mFit</th>
<th>mBackground</th>
</tr>
</thead>
<tbody>
<tr>
<td>tasks</td>
<td>1319</td>
<td>3883</td>
<td>3883</td>
<td>1297</td>
</tr>
<tr>
<td>input files</td>
<td>2638</td>
<td>7766</td>
<td>3883</td>
<td>1297</td>
</tr>
<tr>
<td>output files</td>
<td>2594</td>
<td>3883</td>
<td>3883</td>
<td>1297</td>
</tr>
<tr>
<td>input amount</td>
<td>3.2</td>
<td>35.8</td>
<td>2.2</td>
<td>5.8</td>
</tr>
<tr>
<td>output amount</td>
<td>10.9</td>
<td>3.6</td>
<td>0.001</td>
<td>5.4</td>
</tr>
</tbody>
</table>

Table 3.2: Basic BLAST Stage Statistics. Tasks, input files, output files, input amount, output amount refer to number of tasks, number of input files, number of output files, total amount of input and total amount of output respectively. Input amount and output amount are measured in GB.

<table>
<thead>
<tr>
<th>stage</th>
<th>formatdb</th>
<th>blastp</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>tasks</td>
<td>63</td>
<td>1008</td>
<td>16</td>
</tr>
<tr>
<td>input files</td>
<td>63</td>
<td>4032</td>
<td>1008</td>
</tr>
<tr>
<td>output files</td>
<td>252</td>
<td>1008</td>
<td>16</td>
</tr>
<tr>
<td>input amount</td>
<td>4.0</td>
<td>64.5</td>
<td>0.9</td>
</tr>
<tr>
<td>output amount</td>
<td>4.7</td>
<td>0.9</td>
<td>0.06</td>
</tr>
</tbody>
</table>
Table 3.3: Basic CyberShake PostProcessing Stage Statistics. Tasks, input files, output files, input amount, output amount refer to number of tasks, number of input files, number of output files, total amount of input and total amount of output respectively. Input amount and output amount are measured in GB.

<table>
<thead>
<tr>
<th>stage</th>
<th>extract</th>
<th>seis</th>
<th>peakGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>tasks</td>
<td>393</td>
<td>11868</td>
<td>11868</td>
</tr>
<tr>
<td>input files</td>
<td>1179</td>
<td>35604</td>
<td>11868</td>
</tr>
<tr>
<td>output files</td>
<td>786</td>
<td>11868</td>
<td>11868</td>
</tr>
<tr>
<td>input amount</td>
<td>41.3</td>
<td>2655.2</td>
<td>0.4</td>
</tr>
<tr>
<td>output amount</td>
<td>40.1</td>
<td>0.3</td>
<td>0.03</td>
</tr>
</tbody>
</table>

3.1.2 Application Profiling Methodology

For each application stage, we run all tasks in parallel and trace all system calls during execution. We initially attempted to align the execution trace with the absolute time, as shown in Figure 3.1. We found two problems. First, run time is dependent on the machine where the application stage is run, so the profile on Machine A is not necessarily valid on Machine B. Second, the file system is usually shared by multiple users, so if we align the execution trace by time, the shared file system access delay from other users will add noise, and we won’t be able to observe the spike of operations that would otherwise be concurrent. Note that we aggregated file creation and file open over one second intervals. Most of the time, the frequency of open is identical to the frequency of create, as in most of the traces for peakGM, the open and creation occur in the same second.

We therefore decided to align the system call trace with the system call sequence, rather than with time. A significant benefit of this approach is that the profile represents the characteristics of the application, independent of any machine, and the time can be projected when we map the file system operation onto specific machines. We are thus able to see spikes of concurrent metadata operations in opens and creates, as shown in Figure 3.2.

28
Figure 3.1: peakGM Metadata Operations VS. Time. Operations traces are aggregated on a per second basis.

Figure 3.2: peakGM Metadata Operations vs. System Call Sequence

3.1.3 Metadata Operation Distribution

Among the ten application stages, we see two major patterns of metadata operation distribution. Figure 3.2 shows file creation and open distribution over the system call sequence:
file creation and file open operations are two spikes. When those two sets of operations are executed on a system with a shared file system, they will first be limited by the concurrency bound, then they will be limited by throughput at a particular scale. A second pattern is shown in Figure 3.3, where the file creations are spread across a time range, which means that at any given time, the concurrency bound of file creation in mProjectPP is 303, though there are 2,594 file creations in total.

![Figure 3.3: mProjectPP Metadata Operations VS. System Call Sequence](image)

In those reads, we see three access patterns for input files: some files are read by only one task in a stage, some files are read by all tasks in a stage, and some files are read by a number of tasks less than the total number of tasks in a stage. We refer those three cases as 1-to-1, N-to-1, and Few-to-1 respectively. Table 3.4, 3.5 and 3.6 shows the detailed statistics of the metadata operation spikes as well as the maximum concurrency. These access patterns suggest opportunities for potential read optimization in I/O middleware. For example, the N-to-1 reads can be replaced by a single read by one of the tasks, followed by a broadcast to all other tasks with a parallel algorithm. MPI-collective functions [71] can apply such optimizations, but this breaks the MTC paradigm of independent tasks.
Table 3.4: Montage Metadata Operation Stats

<table>
<thead>
<tr>
<th>stage</th>
<th>mProject</th>
<th>mDiff</th>
<th>mFit</th>
<th>mBackground</th>
</tr>
</thead>
<tbody>
<tr>
<td>open spikes</td>
<td>8</td>
<td>10</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Max. open concurrency</td>
<td>1319</td>
<td>3883</td>
<td>3883</td>
<td>1297</td>
</tr>
<tr>
<td>create spikes</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Max. create concurrency</td>
<td>303</td>
<td>445</td>
<td>3883</td>
<td>113</td>
</tr>
<tr>
<td>1-to-1 read spikes</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>N-to-1 read spikes</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Few-to-1 read spikes</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.5: BLAST Metadata Operation Stats

<table>
<thead>
<tr>
<th>stage</th>
<th>formatdb</th>
<th>blastp</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>open spikes</td>
<td>1</td>
<td>4</td>
<td>63</td>
</tr>
<tr>
<td>Max. open concurrency</td>
<td>63</td>
<td>1008</td>
<td>16</td>
</tr>
<tr>
<td>create spikes</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max. create concurrency</td>
<td>63</td>
<td>1008</td>
<td>16</td>
</tr>
<tr>
<td>1-to-1 read spikes</td>
<td>1</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>N-to-1 read spikes</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Few-to-1 read spikes</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.6: CyberShake PostProcessing Metadata Operation Stats

<table>
<thead>
<tr>
<th>stage</th>
<th>extract</th>
<th>seis</th>
<th>peakGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>open spikes</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max. open concurrency</td>
<td>393</td>
<td>11868</td>
<td>11868</td>
</tr>
<tr>
<td>create spikes</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Max. create concurrency</td>
<td>81</td>
<td>768</td>
<td>11868</td>
</tr>
<tr>
<td>1-to-1 read spikes</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>N-to-1 read spikes</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Few-to-1 read spikes</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

3.1.4 I/O Distribution

To show the I/O traffic of each application stage, we record the number of bytes of each read() and write() system call. Then we align this I/O traffic to the system call sequence to determine its concurrency. In Figures 3.4-3.9, the vertical lines show the total number of bytes read/written at any given system call (left Y axis), and the points show the average I/O amount (right Y axis). Table 3.7, 3.8, and 3.9 show the trace length (number of system
Table 3.7: Montage I/O Stats. Trace length refers to the total number of system calls in the strace log.

<table>
<thead>
<tr>
<th>stage</th>
<th>mProject</th>
<th>mDiff</th>
<th>mFit</th>
<th>mBackground</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace length</td>
<td>388</td>
<td>328</td>
<td>80</td>
<td>201</td>
</tr>
<tr>
<td>Max. read concurrency</td>
<td>1319</td>
<td>3883</td>
<td>3883</td>
<td>1297</td>
</tr>
<tr>
<td>Max. write concurrency</td>
<td>1246</td>
<td>1931</td>
<td>913</td>
<td>1216</td>
</tr>
<tr>
<td>Max. bytes read/call (MB)</td>
<td>86.4</td>
<td>254.5</td>
<td>250.3</td>
<td>84.9</td>
</tr>
<tr>
<td>Max. bytes written/call (MB)</td>
<td>81.5</td>
<td>110.2</td>
<td>0.3</td>
<td>79.6</td>
</tr>
</tbody>
</table>

Table 3.8: BLAST I/O Stats. Trace length refers to the total number of system calls in the strace log.

<table>
<thead>
<tr>
<th>stage</th>
<th>formatdb</th>
<th>blastp</th>
<th>merge</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace length</td>
<td>488145</td>
<td>17977</td>
<td>3206</td>
</tr>
<tr>
<td>Max. read concurrency</td>
<td>63</td>
<td>1008</td>
<td>63</td>
</tr>
<tr>
<td>Max. write concurrency</td>
<td>63</td>
<td>1008</td>
<td>16</td>
</tr>
<tr>
<td>Max. bytes read/call (MB)</td>
<td>4.1</td>
<td>6.6</td>
<td>1</td>
</tr>
<tr>
<td>Max. bytes written/call (MB)</td>
<td>2.6</td>
<td>0.2</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 3.9: CyberShake PostProcessing I/O Stats. Trace length refers to the total number of system calls in the strace log.

<table>
<thead>
<tr>
<th>stage</th>
<th>extract</th>
<th>seis</th>
<th>peakGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace length</td>
<td>35048</td>
<td>393</td>
<td>11868</td>
</tr>
<tr>
<td>Max. read concurrency</td>
<td>16</td>
<td>393</td>
<td>11868</td>
</tr>
<tr>
<td>Max. write concurrency</td>
<td>16</td>
<td>183</td>
<td>768</td>
</tr>
<tr>
<td>Max. bytes read/call (MB)</td>
<td>1.0</td>
<td>129.0</td>
<td>777.8</td>
</tr>
<tr>
<td>Max. bytes written/call (MB)</td>
<td>1.0</td>
<td>6.0</td>
<td>18.4</td>
</tr>
</tbody>
</table>

calls), the maximum read/write concurrency, and the maximum I/O traffic.

One interesting I/O pattern in the mProjectPP, mDiff, and mBackground stages is shown in Figures 3.4, 3.5, and 3.6, where the reads and writes tend to transfer the same amount of I/O traffic per system call, and the number is close to 64 KBytes. It is likely that a buffer of size 64 KBytes is used for file access. We see the same pattern for reads in the formatdb, blastp, and merge stages, shown in Figures 3.7, 3.8, and 3.9. However, the write traffic in formatdb and blastp comprises many small writes of hundreds of bytes. The reason is that these two stages process the protein sequence in a sequential order, and each sequence has
hundreds of bytes. The I/O traffic patterns in the extract, seis and peakGM stages are all unique, yet they feature high concurrency and are data-intensive.

Figure 3.4: mProjectPP I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively.
Figure 3.5: mDiff I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively.

Figure 3.6: mBackground I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively.
Figure 3.7: formatdb I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively.

Figure 3.8: blastp I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively.
Figure 3.9: merge I/O Traffic Profile. Vertical bars (left Y axis) and dots (right Y axis) show the aggregated and average traffic of I/O operations at any given call respectively.

### 3.2 Defining MTC Envelope

The goal of the MTC Envelope is to quantify the capacity of large-scale computers in the context of parallel scripting applications so that large-scale computer users can use the benchmark (envelope) measurement to obtain a better expectation of their MTC applications’ performance. Section 3.1 showed us that the I/O performance of an application stage is first bound by concurrency, which determines the metadata throughput, I/O throughput and bandwidth at a certain scale. Then the I/O performance is bound by metadata throughput. For either read or write traffic, the stage will query the metadata to find out the location of the actual files or create entries in metadata servers if the files do not exist in the write case. Finally, the I/O performance is bound by I/O throughput for small files and by I/O bandwidth for large files.

We define the MTC Envelope for a fixed number of computing resources as a set of eight performance metrics:
• file open operation throughput,
• file creation operation throughput,
• 1-to-1 read data throughput,
• 1-to-1 read data bandwidth,
• N-to-1 read data throughput,
• N-to-1 read data bandwidth,
• write data throughput, and
• write data bandwidth

In §3.1, we also discussed the Few-to-1 case, but here we simplify the approach by merging Few-to-1 into 1-to-1.

A system will exhibit different MTC Envelopes at different scales. Its throughput and bandwidth will change as the level of I/O concurrency changes. Therefore, we further define the MTC Envelope for a given large-scale system as eight curves (rather than eight points), where each curve reflects the change of one of the performance metric along the scales.

### 3.3 MTC Envelope Measurements

In this chapter, we measure the MTC Envelope of the GPFS deployment on the Intrepid BG/P supercomputer at Argonne National Laboratory as an example. This GPFS deployment has multiple metadata server and 128 IBM x3545 file servers, each with two 2.6-GHz Dual Core CPU and 8 GB RAM. The multiple metadata server design in this GPFS deployment is used for redundancy rather than performance. The compute nodes reach the shared file system through I/O nodes, where each I/O node is associated with 64 compute nodes.

We run the I/O benchmark workloads on the I/O nodes, where each I/O node is a GPFS client. The performance test space for metadata operations is \{create, open\}×\{1, 2, 4, 8, 16, 32, 64, 128, 256\} clients, while the performance test space for I/O is \{read, write\}×\{1 KB, 128 KB, 1 MB, 16 MB\}×\{1, 2, 4, 8, 16, 32, 64, 128, 256\} clients. The total number of
performance test jobs is 630, and the whole test takes about 2.2 million core hours on the BG/P. Note that we used a shell script as our test framework to represent a typical parallel scripting application, and the BG/P supercomputer guarantees launching the script on all I/O nodes simultaneously. The BG/P supercomputer is a multi-user system, not a dedicated system. Therefore, we are not measuring the peak performance of the I/O system, but rather the performance that parallel scripting applications experience when sharing the file system with other users.

### 3.3.1 Metadata Operation Throughput

The file creation throughput is measured by asking all GPFS clients to create independent files in one single directory. In all test cases, the total number of file creations is 8,192. Using a similar strategy, we measure file open throughput by first creating 8,192 files in one directory, then ask all GPFS clients to query the files, with each file only queried once.

Figure 3.10 shows the scaling of throughput of file creation and open in a single directory. Both creation and open throughput increase linearly to eight GPFS clients. Then the rate of increase slows down from eight clients to 32 clients. Above 32 clients, throughput starts to slow down. The reason for the creation slowdown is that the GPFS metadata server uses a locking mechanism to resolve concurrent create operations in one directory, and 64 and more clients triggers the concurrency bottleneck of the locking mechanism. The query slowdown is due to the saturated single metadata server of the tested GPFS deployment.
Figure 3.10: Metadata operation throughput. Performance increases linearly from one client to eight clients, then the increase slows down from eight clients to 32 clients. The performance starts to decrease from 32 clients dramatically.

3.3.2 1-to-1 Read Performance

In the 1-to-1 read benchmark measurement, we first create 8,192 files with size of \{1 KB, 128 KB, 1 MB, 16 MB\} in one directory. Then we start a number of GPFS clients to simultaneously read those files. With \(N\) clients, each client reads \(8,192/N\) distinct files by copying the file from GPFS to its local RAM disk. Figures 3.11 and 3.12 show the 1-to-1 read operation throughput and data bandwidth respectively.
In Figure 3.11, the 1-KB and 128-KB 1-to-1 reads are dominated by latency, as seen by the closeness of the two curves. For these two reads, the throughput increases up to
32 GPFS clients, then starts to slow down. This slowdown is similar to that of the open operation in Section §3.3.1, and is likely due to the same cause. As the cost of the small file read is mostly that of determining the file’s location by querying the metadata server, the throughput of read is about 25% lower than that of open due to the extra overhead that comes from the actual data transfer. 1-MB reads reach the throughput bound at a larger scale than 1-KB and 128-KB reads because the metadata traffic congestion on the metadata server is alleviated by the actual data transfer. With larger file size, the 16-MB reads have not become throughput bound with 256 clients.

Peak GPFS read performance has been documented at 62,000 MB/s [8]. Figure 3.12 (log scale) does not manage to reach this limit; the peak bandwidth for 16-MB reads is 6,880 MB/s. For the bandwidth dominated 1-MB and 16-MB read tests, the performance scales up nearly linearly from one to 128 clients, then the 1-MB read reaches its ceiling, while the 16-MB read performance is still increasing.

3.3.3 N-to-1 Read Performance

To measure the N-to-1 read performance, we first create a single file with size \{1 KB, 128 KB, 1 MB, 16 MB\}. Then we let a number \{1, 2, 4, 8, 16, 32, 64, 128, and 256\} of GPFS clients concurrently read the file. With \(N\) clients, each client will read the same file \(8,192/N\) times.

Figures 3.13 and 3.14 show the N-to-1 read operation throughput and data bandwidth respectively. 1 KB and 128 KB reads reach peak throughput with 64 clients, which is twice as many clients as where the 1-to-1 read throughput peaks. One potential explanation of the improvement is metadata caching in the GPFS client, where when file is read multiple times, the client does not have to re-query the metadata server. However, with 128 clients, the performance is bound by the read concurrency of the shared file system.
Figure 3.13: N-to-1 Read Operation Throughput

Figure 3.14: N-to-1 Read Data Bandwidth
3.3.4 Write Performance

We run the write performance benchmark in the space of \{1 KB, 128 KB, 1 MB, 16 MB\} × \{1, 2, 4, 8, 16, 32, 64, 128, 256\} clients. To focus only on write performance, we initially create 8,192 empty files in one directory, and then each client writes to a mutually exclusive group of the files. With \( N \) clients, each client writes to \( 8,192/N \) distinct files.

As shown in Figure 3.15, the 1-KB, 128-KB, and 1-MB writes reach the operations throughput bound at 32 clients, which is identical to the peak of 1-to-1 reads. In Figure 3.16, the 16-MB write reaches the data bandwidth bound at 64 clients, yielding a data bandwidth of 3,497 MB/s.

One interesting observation of Figures 3.11 and 3.15 is that the operations throughput is independent of the file size. This suggests that 256 clients limit the throughput of GPFS to approximately 470 Op/s for read and 280 Op/s for write. This is a significant characteristic and a limiting factor of the envelope.

Figure 3.15: Write Operation Throughput
3.3.5 Envelope Summary

We have defined the MTC Envelope as eight performance metrics of file open throughput, file creation throughput, 1-to-1 read throughput, 1-to-1 read bandwidth, N-to-1 read throughput, N-to-1 read bandwidth, write throughput, and write bandwidth. We show the envelope on 1, 2, 4, 8, 16, 32, 64, 128, and 256 clients as a Kiviat diagram in Figure 3.17. This shows the performance on each metric relative to that on one client.

Scaling up to 16 clients shows generally good performance. Overall, most of the metrics reach a peak at some number of clients and then decrease, with the exception of N-to-1 read bandwidth, which has not yet reached a peak using 256 clients. The throughput metrics, for both metadata and transfer, appear to have the worst scalability. 1-to-1 read bandwidth and write bandwidth appear to have reached their peak with 256 clients. Our characterization of the MTC Envelope indicates that while the BG/P I/O system can sustain a very large I/O data bandwidth, its peaks in operation throughput per second remain a limiter of MTC application performance.
Figure 3.17: MTC Envelope (log scale). The performance on each metric (0.5 - 500) is relative to that on one client. The creation and open throughput reaches the peak first then starts to decrease, followed by write throughput, 1-to-1 and N-to-1 read throughput. The write, 1-to-1, and N-to-1 read bandwidth has a better scalability.

3.4 Putting the Applications in the Envelope

So far, we have measured the performance metric suite at multiple scales, which is enough to characterize the I/O performance of the system for running parallel scripting applications. Now, we examine predicting I/O time consumption and I/O performance bounding factors for the parallel scripting applications themselves.

3.4.1 I/O Time Consumption

To ease the work of predicting I/O time consumption of parallel scripting applications on the shared file system, we first examine write on GPFS.
Assume one stage of an application has $N$ tasks, with each writing one output file of size $D$ bytes at the same time, and that the stage runs on a group of computing resources that has $C$ shared file system clients. To evaluate the I/O consumption of this stage, we first find the I/O bandwidth $B$, throughput $T_w$, and metadata creation throughput $T_m$ at scale $C$. All writes will come as $\lceil N/C \rceil$ rounds. In each round, metadata creation takes $C/T_m$ time, and the write time consumption is $C/T$ when throughput dominates or $C \times D/B$ when bandwidth dominates. So the write time consumption can be approximately expressed as

$$Time = \lceil N/C \rceil \times \left( C/T_m + C \times D/B \right)$$ \hspace{1cm} (3.1)$$

when the write is bandwidth bound, or as

$$Time = \lceil N/C \rceil \times \left( C/T_m + C/T_w \right)$$ \hspace{1cm} (3.2)$$

when the write is throughput bound. Higher throughput and bandwidth are desired for parallel scripting applications in order to reduce runtime.

We plot two heat maps of concurrent write bandwidth and throughput with our benchmark numbers and Equations 3.1 and 3.2. Figure 3.18 shows the bandwidth distribution with various file size on multiple scales, while Figure 3.19 shows the throughput distribution with the same input as Figure 3.18. When used to predict write performance, the two heat maps yield the same predictions, however, in Figure 3.18, the difference among the file size of 1 KB, 128 KB and 1 MB is barely discernible.

For a read workload, we denote $N$ as the number of tasks, with each reading a file of size $D_1$ bytes at the same time and a common input file of size $D_N$ shared among all tasks. On a group of computing resources that has $C$ shared filesystem clients, the MTC Envelope delivers 1-to-1 read bandwidth of $B_1$, throughput of $T_1$, N-to-1 read bandwidth of $B_N$ and
throughput of $T_N$. The read time consumption can be approximately expressed as

$$\text{Time} = \lceil N/C \rceil \times (\max(C/T_1, C \times D_1/B_1) + \max(C/T_N, C \times D_N/B_N))$$  \hspace{1cm} (3.3)$$

We do not compile the 1-to-1 read and N-to-1 read performance into heat maps because the benchmark numbers are measured directly. Users can refer to Figures 3.11, 3.12, 3.13, 3.14 for numbers to project read time consumption.
Using such information, a parallel scripting application programmer could obtain a rough picture of how much time was taken by I/O in the execution. For example, assuming we have an application stage with 8,192 tasks, each writing one output file of size 16 MB, and running on 64 GPFS clients, then the write consumption can be estimated by the following steps. 16-MB writes are bandwidth dominated, so we look at the bandwidth heat map, and find that the bandwidth is about 3500 MB/s. The write has 8,192*16 MB of data to write, so it will take 8,192*16/3,500 = 37 seconds. Alternatively, a programmer with a requirement for the I/O time might first determine what file size and scale combinations for the I/O time are valid on the shared file system, and then could choose from the candidate combinations.

### 3.4.2 I/O Performance Bounding Factors

To begin to investigate the I/O performance bounding factor(s) of the example applications, we map the applications’ I/O behavior to the MTC Envelope that we measured on GPFS on 32 clients. Table 3.10, 3.11, 3.12 illustrate the time consumption decomposition and bounding factors for each applications stage. The calculations were done based on the profile of each application and Equations 3.1 and 3.2. The MTC Envelope on 32 GPFS clients shows file creation throughput of 1,367 Op/s, file open throughput of 1,974 Op/s, 1-to-1 read performance of 1,624 reads/s and 2,123 MB/s, N-to-1 read performance of 3,489 reads/s and 4,118 MB/s, and write performance of 1,481 writes/s and 3,429 MB/s.

<table>
<thead>
<tr>
<th></th>
<th>mProject</th>
<th>mDiff</th>
<th>mFit</th>
<th>mBackground</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metadata</td>
<td>69.7%</td>
<td>82.0%</td>
<td>23.3%</td>
<td>24.1%</td>
</tr>
<tr>
<td>I/O</td>
<td>30.3%</td>
<td>18.0%</td>
<td>76.7%</td>
<td>75.9%</td>
</tr>
<tr>
<td>Bound. Fact.</td>
<td>bw</td>
<td>tput</td>
<td>bw</td>
<td>bw</td>
</tr>
<tr>
<td>Write</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metadata</td>
<td>37.4%</td>
<td>52.0%</td>
<td>52.0%</td>
<td>37.7%</td>
</tr>
<tr>
<td>I/O</td>
<td>62.6%</td>
<td>48.0%</td>
<td>48.0%</td>
<td>62.3%</td>
</tr>
<tr>
<td>Bound. Fact.</td>
<td>bw</td>
<td>tput</td>
<td>tput</td>
<td>bw</td>
</tr>
</tbody>
</table>
Table 3.11: BLAST I/O Performance Decomposition and Bounding Factor(s) Classification (bw=bandwidth, tput=throughput) on 32 GPFS Clients

<table>
<thead>
<tr>
<th></th>
<th>Metadata</th>
<th>I/O</th>
<th>Bound. Fact.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Metadata</td>
<td>1.7% 6.0% 1.9%</td>
<td>bw bw bw</td>
</tr>
<tr>
<td></td>
<td>I/O</td>
<td>98.3% 94.0% 98.1%</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Metadata</td>
<td>11.9% 52.0% 40.8%</td>
<td>bw tput tput</td>
</tr>
<tr>
<td></td>
<td>I/O</td>
<td>88.1% 48.0% 59.2%</td>
<td></td>
</tr>
<tr>
<td>Bound. Fact.</td>
<td>bw</td>
<td>tput</td>
<td>bw</td>
</tr>
</tbody>
</table>

Table 3.12: CyberShake I/O Performance Decomposition and Bounding Factor(s) Classification (bw=bandwidth, tput=throughput) on 32 GPFS Clients

<table>
<thead>
<tr>
<th></th>
<th>extract</th>
<th>seis</th>
<th>peakGM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Metadata</td>
<td>3.1% 1.4% 82.4%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I/O</td>
<td>96.9% 98.6% 17.6%</td>
<td></td>
</tr>
<tr>
<td>Bound. Fact.</td>
<td>bw</td>
<td>bw tput</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>Metadata</td>
<td>4.7% 52.0% 52.0%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>I/O</td>
<td>95.3% 48.0% 48.0%</td>
<td></td>
</tr>
<tr>
<td>Bound. Fact.</td>
<td>bw</td>
<td>tput</td>
<td></td>
</tr>
</tbody>
</table>

Given an application stage’s profile, we compute the read I/O consumption as the difference between the time consumption of reads and metadata operations. Write I/O consumption can be measured directly as shown in §3.3.4. We also compute the time consumption numbers from Equations 3.1 and 3.2 and determine which is larger to decide if an operation is bandwidth bound or throughput bound.

We see from the read time distribution for mProject and mDiff in Table 3.10 that for small file I/O, even though the metadata processing takes more time than the I/O, we can determine if the I/O is bandwidth-bound or throughput-bound. If we perform the same calculation on the MTC Envelope on 64 clients, we would see a dramatical decrease of metadata and I/O throughput, which is the concurrency bounding factor of the applications’ I/O behavior.
3.5 A Review of Montage Performance

We now review the time distribution of the Montage test case we have shown in Figure 1.2 of §1.4. We want to decompose the 44.3% of CPU-time to understand how much of the time was consumed by metadata processing and how much was consumed by actual data movement.

The Montage application test case has eight stages: mProject, mImgtbl, mOverlaps, mDiffFit, mConcatFit, mBgModel, mBackground and mAdd. For the parallel stages of mProject, mDiffFit, mBackground, we use the data in Table 3.10 to estimate the metadata and actual data processing time. For the stages of mImgtbl, mConcatFit, and mAdd, there is only one task in these stages, but the only tasks has thousands of input files. So we estimate the metadata processing time as there is only one client with throughput shown in Figure 3.19. The actual data transfer overhead is estimated in a similar way. We did not take into account the stages of mOverlaps and mBgModel, since there is the only one task of each stage has only a few input/output files and they can be run on other computers (e.g., login nodes of the supercomputer) without wasting the supercomputer CPU-time.

Since we estimate the metadata concurrency overhead in the worst case, the projected decomposition of the 44.3% CPU-time is the upper bound of metadata processing. In the real application, the metadata processing can only be faster than the estimation as the concurrency of the file creation and write phase can be diminished by the irregularity of the task time-to-solution in each stage.

Figure 3.20 shows the further I/O time decomposition. We can tell that over all stages of Montage, 25.9% of the I/O time was used for metadata processing (11.5% in total time-to-solution) and 74.1% of the I/O time was used for actual data I/O (32.9% in total time-to-solution).

Early in Section 1.4, we concluded that there was a great opportunity for an efficient data management solution that can improve the overall application performance significantly, and this solution needs to tackle two sub-problems: scalable data caching and fast data flow
Figure 3.20: Projected I/O time distribution of generating a 6x6 degree mosaic with Montage on 64 IBM BG/P processors, using GPFS.

pattern specific movement. Now we can further extend the first sub-problem as: an efficient data management solution needs to provide a scalable data caching mechanism with both fast metadata access and data access.

3.6 Conclusion

In this chapter, we first studied the I/O behavior of parallel scripting applications that use a shared file system abstraction for inter-task communication on large-scale computers. Then we defined the MTC Envelope as a set of eight performance metrics: metadata query throughput, metadata creation throughput, 1-to-1 read throughput, 1-to-1 read bandwidth, N-to-1 read throughput, N-to-1 read bandwidth, write throughput, and write bandwidth. We also showed that the MTC Envelope is sufficient to characterize the I/O behavior of parallel scripting application. Taking GPFS on BG/P as an example, we benchmarked the MTC Envelope of GPFS at multiple scales, and studied its scalability. Based on the quantitative analysis of the MTC applications and the MTC Envelope on GPFS, we found
out that highly concurrent metadata access is one bottleneck of GPFS in the context of MTC applications. For the same reason, the metadata processing dominated benchmark metrics (small read, small write) are also performance bottlenecks with poor scalability. We believe the envelope model may be useful as a guide for aggregating many smaller operations into fewer larger ones, replacing shared file system accesses with scalable local file system usage, and spreading shared file system accesses in a manner that reduces locking and other forms of resource contention. Finally, we presented a way to convert performance measurements into heat maps that can guide programmers to make better use of existing shared filesystems for parallel scripting applications.

We believe this work can also benefit other types of applications such as HPC and MapReduce, as the same approach can be applied to those applications and hardware-software stacks. We are in the era of Big Data where many interesting applications are inevitably data-intensive, so such shared file system studies in other communities will help them use the current I/O and storage in a more efficient way.

The projected I/O time decomposition gives us clear requirements for an efficient data management system: the system should provide fast metadata access (query and creation); fast data access (read and write); and it should transfer data flow specific movement with faster algorithms.
CHAPTER 4
MANAGING RUNTIME DATA

The goal of runtime data management is to enable fast execution of MTC applications on large-scale computers. As discussed in §1.5 and §3.5: one sub-problem is to provide scalable caching and fast access of intermediate data across the RAM of the compute nodes; the other sub-problem is to expedite the data flow pattern specific movement. Since MTC applications’ data are in the form of POSIX files, the POSIX file format has to be preserved in the data management system. The caching and movement algorithms need to be scalable to work at large scale.

We present the AMFORA File System (referred as AMFORA FS in the rest of this chapter) as the data management solution. AMFORA FS is a shared in-RAM file system with a fully distributed metadata and data management design. AMFORA FS implements the multi-read single-write I/O pattern, which suffices the MTC applications’ I/O requirement. AMFORA FS preserves the POSIX interface so the original applications do not have to be changed. AMFORA FS provides interface for collective data movement and functional data transformation. AMFORA FS’s performance scales to thousands of compute nodes.

In this chapter, we will first introduce the AMFORA FS’s architecture in a nutshell, then explain the technical details of how fast access and scalability challenges are tackled.

4.1 AMFORA File System Architecture

The AMFORA implementation places an AMFORA daemon instance (the file system thread is referred as the AMFORA FS daemon in the rest of this dissertation) on every compute node involved in a computation. Each compute node is thus both an I/O server and a metadata server. Because parallel scripting applications have a multi-read single-write I/O pattern, we further define the AMFORA FS I/O routines to be local-read-if-possible and local-write. AMFORA FS places file locality information in the metadata of a file. That
is, AMFORA tries its best to place a task that reads a file on the node where that file is located, but can not guarantee that all input files are local to the task. All writes are local to the task to maximize write performance.

Figure 4.1 shows the architecture of the AMFORA FS daemon. Each AMFORA FS daemon manages three types of data: directory metadata, file metadata, and file data.

![AMFORA File System Daemon Architecture](image)

Figure 4.1: AMFORA File System Daemon Architecture. Directory metadata are spread and synchronized on all daemons. File metadata are spread across all daemons with a hash function. File data are primarily located on the daemon where it is produced.

### 4.2 Scalable Data Caching

AMFORA FS uses different policies to manage these three types of data. Directory metadata is synchronized and consistent across all the daemons. File metadata are spread across all the daemons using a hash function with file name and the daemon list as input parameters. The file data are primarily stored on the compute nodes where it is produced.
4.2.1 Directory Metadata Management

AMFORA FS maintains directory metadata consistently across all nodes. Upon the creation of a directory, a create message is broadcast to all AMFORA FS daemons, and the initiating daemon blocks until all daemons return. Every AMFORA FS daemon creates an entry for this directory. The same logic is used to remove a directory. In a directory read operation, the initiating daemon broadcasts a read message to all AMFORA FS daemons, and along with the acknowledgement, each AMFORA FS daemon sends all file metadata it stores locally. The initiating daemon then adds the subdirectory information to the file information, and returns. Directory metadata operations are optimized using a minimum spanning tree (MST) algorithm. The time complexity of such directory operations is $O(logN)$, where $N$ is the number of AMFORA FS daemons. Since directory operations require synchronization among all AMFORA FS daemons, they do not work well when there are multiple concurrent directory metadata operations or directory-intensive operations.

4.2.2 File Metadata Management

File metadata are spread across AMFORA FS daemons based on the hash value of the file name and the scale of AMFORA FS instances. This metadata includes both regular information about a file, a list of the location(s) that store the actual file data, and a list of AMFORA FS daemons that have tried to access this file before it is produced. Once the file is produced, the AMFORA FS daemon will broadcast the file’s location to all daemons that have tried to access it. This feature resolves the data dependencies in a fully distributed way. Distributing file metadata over AMFORA FS daemons is important to balance the many file metadata access operations that can occur in parallel scripting applications.
4.2.3 File Data Management and Access Consistency

Parallel scripting applications have a multi-read single-write I/O property, where each file can be read multiple times but only written once. There are no concurrent writes to a single file at any given time. AMFORA FS maintains four hash tables to ensure the file consistency. The hash tables are: cache meta (cmeta), meta (meta), cache data (cdata), and data (data). The cmeta table stores the metadata of the files that are locally created. The cdata table keeps the actual data of the files that are locally written. The meta table stores the metadata of the files whose hash result points to this AMFORA FS daemon. The data table stores the files that are created and written locally as well as the data that is replicated from remote peers. The cmeta and cdata tables are only accessible by local AMFORA FS daemon, and can not be seen or accessed by peer daemons. The meta and data tables are globally accessible.

A file’s contents are always stored where the file is first produced with a mutable byte array data structure. Upon a creation of a file, the local AMFORA FS daemon first initializes an entry in cmeta table and cdata table respectively. The writes flow to the correspondent entry in cdata table. During the writes, the file is not visible in the file system as it is stored in cmeta and cdata. Once the file is released (In POSIX, it means the file is closed), AMFORA FS replicates the file from cdata to data and updates the metadata in cmeta table to the nodes that should store the metadata of this file (calculated by the hash function).

In the scenario of a local read, the file contents is returned with the contents stored in the data table. In the remote read case, the whole file will be replicated to the daemon that sends the request, then the file contents are returned from the data table on that daemon. The appropriate metadata is also updated to record the new replica.

This approach implements the multi-read single-write I/O pattern. The invisible-until-release write policy can guarantee that once the file is visible in the file system, the task that has a dependency on it can be launched immediately.
AMFORA FS supports four types of collective data movement: Multicast, Gather, All-gather, and Scatter. Multicast, Gather, and Scatter are implemented with two algorithms: sequential and minimum-spanning tree. As many have previously discussed (e.g., [71, 70]), MST performs significantly faster for small files, as the data movement is latency bound and MST only invokes $O(\log N)$ file transfers on $N$ servers. When the file size is large, the two algorithms deliver similar performance, as the data movement is bandwidth bound. Programmers can select the sequential algorithm where the number of concurrent connections to a server is always one, if the network is congestion sensitive, compared to $\log N$ connections in the MST algorithm. The AMFORA FS collective data management interface allows programmers to specify which algorithm to use. Allgather is implemented as a gather followed by a multicast. The gather involves $O(\log N)$ inbound file transfers and the multicast another $O(\log N)$ outbound file transfers. §4.5 provides a model for collective data management.

The sequential data movement scheme is also considered in the context. We use the gather pattern movement as an example to explain the pros and cons of collective (collective gather) and sequential scheme (asynchronous gather). As shown by Equations 4.1 and 4.2, the collective gather performs faster than the sequential algorithm only when files are so small that file transfer time is dominated by network latency. Otherwise, the two algorithms should perform similar to each other, when files are sufficiently large that file transfer time is dominated by network bandwidth. On the other hand, the collective gather requires all AMFORA FS daemons to synchronize for the collective data transfer; if concurrent gathers are initiated from multiple AMFORA FS daemons, deadlock can occur. The sequential gather does not have this side effect for concurrent gather. Thus, the AMFORA FS supports the collective gather through the basic primitive data management call. AMFORA FS also supports the sequential gather (asynchronous gather) based on asynchronous file access described by §5.4, where a fraction of the files can be transferred as long as they are produced, and the transfer can be overlapped with the remained task execution. In terms of
performance improvements, collective gather benefits from the parallel transfer at each step, whereas asynchronous gather benefits from overlapping computation and file transfer.

\[
T = (\log_2 N) \times a + \frac{N - 1}{N} \times S \times b + (M - 1) \times c \tag{4.1}
\]

\[
T = (N - 1) \times a + \frac{N - 1}{N} \times S \times b + (M - 1) \times c \tag{4.2}
\]

where: \( T \) = time consumption, \( S \) = amount of data transferred, \( N \) = number of nodes, \( M \) = number of files, \( a \) = latency overhead/file transfer, \( b \) = bandwidth overhead/byte, and \( c \) = file system overhead/file.

### 4.4 Functional Data Transformation

The AMFORA FS functional data management interface supports load, store, and shuffle. In the case of load and store, there might or might not be a globally accessible persistent storage that all AMFORA FS daemons can access. If persistent storage is not globally accessible by all AMFORA FS daemons, the initiating AMFORA FS daemon first loads all input files into its RAM, then calls the scatter function to distribute all files to all daemons. While in the case of store, the initiating AMFORA FS daemon first calls a gather function, then stores the files to a persistent storage. If persistent storage is globally accessible by all AMFORA FS daemons, the initiating AMFORA FS daemon computes a distribution plan and passes that plan to all AMFORA FS daemons; the AMFORA FS daemons concurrently load files from persistent storage. The same algorithm is used for AMFORA_store.

AMFORA_shuffle is implemented as a four-step procedure. First, the initiating AMFORA FS daemon notifies all daemons via MST and returns once all daemons have received the message. Second, all daemons process the files in the specified directory locally and partition the files based on the hash value of the first column into hashmaps that are indexed according to which AMFORA FS daemon is addressed. Third (in a alltoall pattern), every
node starts exchanging data with its immediate right-hand neighbor (using a virtual ring overlay), followed by its second nearest right-hand neighbor, etc., until it has communicated with all other AMFORA FS daemons. Upon receiving all partitions from all other AMFORA FS daemons, every daemon will concatenate the partitions into one single file. Fourth, the AMFORA FS daemons notify their parent server in the MST upon completion.

The partition and exchange step of a simple shuffle on two AMFORA FS daemons is shown in Figure 4.2.

![Figure 4.2: Shuffle example on two AMFORA FS daemons](image)

4.5 AMFORA File System Performance

This section presents the AMFORA File System performance with MTC Envelope measurements and the collective/functional data management performance. Both sets of perfor-
4.5.1 MTC Envelope Benchmark Performance

We measured the MTC Envelope of an IBM Blue Gene/P with the ZeptoOS operating system and the GPFS shared file system on \{64, 128, 256, 512, 1024, 2048\} compute nodes. Then we measured the same performance metrics with AMFORA replacing GPFS, as shown in Tables 4.1 and 4.2. Figure 4.3 shows the comparison, where numbers greater than one show improvements, and numbers less than one show slowdown.

Most of metrics show improvements with AMFORA Shell, as the file system I/O is cached in memory. Exceptions are open throughput on 64 compute nodes, and N-1 read throughput and bandwidth in general. On 64 compute nodes, though we spread the metadata across all compute nodes, the aggregated throughput is still lower than that of the GPFS metadata server, as the compute node is 850 MHz while the GPFS server is 2.6 GHz. The N-1 read is a two-step procedure with AMFORA Shell: a multicast followed by a group of synchronized concurrent reads to the local data replica. Since the concurrent local reads and the 1-1 read both access local memory, the N-1 read performance can be estimated as the sum of the multicast time and a group of concurrent 1-1 reads. Thus 1-1 read performance is an upper bound of N-1 read performance.

Examining Tables 4.1 and 4.2, we see that open throughput, 1-1 read throughput, 1-1 read bandwidth increase close to linearly as the number of compute nodes increases. This linear scalability is the result of the distributed metadata server topology. Create throughput and write throughput scale similarly to each other, though less well than open throughput, etc., as the file name distribution over all metadata servers is not uniform and both of the metrics are metadata operation dominated. Comparing write bandwidth with write throughput, the former scales better than the latter as it has more data to write in local RAM while other overhead should be identical. The N-1 read is a two-step operation: multicast followed by
read. The speedup of N-1 read throughput and bandwidth is a mix of the scalability of the multicast and read. Thus when the multicast stage dominates the N-1 read, the scalability of N-1 read is close to the scalability of multicast, otherwise it is close to the scalability of 1-1 read.

Table 4.1: AMFORA File System MTC Envelope Scalability. tput is throughput (in op/s).

<table>
<thead>
<tr>
<th>Scale</th>
<th>Create tput</th>
<th>Open tput</th>
<th>1-1 read tput</th>
<th>N-1 read tput</th>
<th>Write tput</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>255.5</td>
<td>170.8</td>
<td>174.6</td>
<td>139.7</td>
<td>248.9</td>
</tr>
<tr>
<td>128</td>
<td>1.5x</td>
<td>2.0x</td>
<td>2.2x</td>
<td>2.1x</td>
<td>1.4x</td>
</tr>
<tr>
<td>256</td>
<td>2.9x</td>
<td>4.1x</td>
<td>4.5x</td>
<td>4.2x</td>
<td>2.7x</td>
</tr>
<tr>
<td>512</td>
<td>6.8x</td>
<td>9.8x</td>
<td>10.1x</td>
<td>8.6x</td>
<td>4.5x</td>
</tr>
<tr>
<td>1024</td>
<td>11.6x</td>
<td>15.7x</td>
<td>17.3x</td>
<td>14.9x</td>
<td>9.5x</td>
</tr>
<tr>
<td>2048</td>
<td>19.4x</td>
<td>29.4x</td>
<td>29.6x</td>
<td>26.6x</td>
<td>20.2x</td>
</tr>
</tbody>
</table>

4.5.2 Collective and Functional Data Management Performance

The time consumption for the data functions are:

- Multicast: $T = (\log_2 N) \ast (a + b \ast S)$
Table 4.2: AMFORA File System MTC Envelope Scalability. bw is bandwidth (in Mb/s).

<table>
<thead>
<tr>
<th>Scale</th>
<th>1-1 read bw</th>
<th>N-1 read bw</th>
<th>Write bw</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>2215.3</td>
<td>537.3</td>
<td>3553.2</td>
</tr>
<tr>
<td>128</td>
<td>2.2x</td>
<td>1.8x</td>
<td>1.9x</td>
</tr>
<tr>
<td>256</td>
<td>4.8x</td>
<td>3.3x</td>
<td>3.6x</td>
</tr>
<tr>
<td>512</td>
<td>8.8x</td>
<td>5.9x</td>
<td>6.8x</td>
</tr>
<tr>
<td>1024</td>
<td>16.9x</td>
<td>11.0x</td>
<td>12.5x</td>
</tr>
<tr>
<td>2048</td>
<td>27.9x</td>
<td>20.0x</td>
<td>24.2x</td>
</tr>
</tbody>
</table>

- Gather: \( T = (\log_2 N) \cdot a + \frac{N-1}{N} \cdot S \cdot b + (M - 1) \cdot c \)
- Scatter: \( T = (\log_2 N) \cdot a + \frac{N-1}{N} \cdot S \cdot b + (M - 1) \cdot c \)
- Allgather: \( T = 2 \cdot (\log_2 N) \cdot a + ((\log_2 N) + \frac{N-1}{N}) \cdot S \cdot b + (M - 1) \cdot c \)
- Shuffle: \( T = (N - 1) \cdot a + \frac{S}{N} \cdot b + \frac{S}{N} \cdot d \)

where: \( T \) = time consumption, \( S \) = amount of data transferred, \( N \) = number of nodes, \( M \) = number of files, \( m \) = memory on each node, \( a \) = latency overhead/file transfer, \( b \) = bandwidth overhead/byte, \( c \) = file system overhead/file, and \( d \) = processing overhead/byte.

Figure 4.4 shows AMFORA’s collective file movement performance. As we discussed previously [82], Multicast shows \( O(\log(N)) \) scalability regardless of file size. Gather, Scatter, and Allgather performance show \( O(\log(N)) \) scalability when the data size is small; they are dominated by latency. For other sizes, Gather, Scatter, and Allgather show linear scalability.

The time consumption of Shuffle has three elements, due to latency overhead \( (N - 1)\cdot a \), bandwidth \( \frac{S}{N} \cdot b \), and processing \( \frac{S}{N} \cdot d \). Since Shuffle requires the total input storage, processing, and output storage data size on each node to be less than its available memory, it requires \( N \geq \frac{3 \cdot S}{m} \) compute nodes.

On the other hand, for large \( N \), \( N - 1 \) approaches \( N \) and the time consumption can be estimated as \( T = (N) \cdot a + \frac{S}{N} \cdot (b + d) \). The minimum value of this equation is \( 2 \cdot \sqrt{S \cdot a \cdot (b + d)} \), when \( N = \sqrt{\frac{S \cdot (b + d)}{a}} \). In other words, using more than \( \sqrt{\frac{S \cdot (b + d)}{a}} \) compute nodes for a shuffle of files of size \( S \) won’t improve performance. These two inequalities
indicate that to enable an efficient shuffle operation on files of size $S$, the number of compute nodes has to meet the following condition:

$$\frac{3 * S}{m} \leq N \leq \sqrt{\frac{S * (b + d)}{a}} \quad (4.3)$$

To validate AMFORA shuffle, we fixed the data size of the shuffle operation at 4 GB, and ran the shuffle function on \{64, 128, 256, 512, 1024\} AMFORA FS daemons, as shown in Figure 4.5. As discussed in §4.4, a shuffle operation has four steps, and the partition and transfer steps dominate the time consumption. With fixed data size, increasing the number of servers results in a shorter partition time as the data size for each server decreases. On the other hand, the transfer step consumes more time due to the time complexity of the ring algorithm. We use the measured performance at 64 and 128 servers to regress the parameters in a time consumption model. Equation 4.3 predicts the optimal number of servers is 231 servers, while the best performance in our measurements runs on 256 servers. We see that the real system performance is far from the prediction on 1024 servers, with two potential
reasons. First, transfer congestion is more likely as the scale increases, which results in unmodeled delay. Secondly, with the same dataset, more compute nodes can lead to more unbalanced traffic in the alltoall data transfer phase, which in turn results in unmodeled delay.

4.6 Conclusion

The AMFORA FS successfully tackles the two sub-problems we presented in the beginning of this chapter: to provide scalable caching and fast access of intermediate data across RAMs and to expedite the data flow pattern specific movement. The AMFORA FS performs 10-100x faster for all performance metrics of MTC Envelope, and it shows good scalability from small to large scale on the test machine. The collective data movement algorithms can move the data faster (from $O(N)$ to $O(\log(N))$) in many cases and the correctness of the implementation is shown by the overlapped curve of model prediction and the real measurements. The improvements are shown by benchmark measurements. The overall
application improvements will be shown in §7.
CHAPTER 5
INTERACTING WITH DATA MANAGEMENT SYSTEM

This chapter discusses the design and implementation of the interface between the AMFORA components. The discussion focuses on programmability and performance. Section 5.1 qualitatively evaluates the options in designing the three interfaces between the AMFORA components, as shown in Figure 1.3. Section 5.3 explains how to address task locality in the file-usage blind format. Section 5.4 demonstrates asynchronous file access, with which we can enable fully distributed task dependency resolution.

We also present an early version of AMFORA Task Engine called AME [85], which is a hierarchical task engine that accepts the file-usage aware task format. We discuss the task dispatching, data flow pattern detection, automatic data replication, and asynchronous file transfer in §5.5.

5.1 Designing Interface between AMFORA Components

As shown in Figure 1.3, AMFORA has three components: the script, the in-RAM file system, and the task engine. This section discusses various design options between those components.

5.1.1 Script - Task Engine

In parallel scripting languages such as Swift [76], programmers declare inputs and outputs when defining applications tasks; Makeflow [3] uses a similar strategy by defining tasks with make file rules, so the interpreter knows how files are used. In addition to the commonly used input and output definitions, Pegasus [24] also uses ‘intermediate’ to mark files that do not need to be saved after they are read, to eliminate unnecessary data movement.

Intuitively, a programmer can express a task as a command line in the exact format in which the task is executed.

File-usage blind: \texttt{bin/exec input.txt output.txt}
Here, the engine does not know which files are inputs and which files are outputs. Alternatively, the programmer can provide file usage information on the command line:

File-usage aware: \texttt{bin/exec input.txt output.txt -if input.txt -of output.txt}

The file-usage aware format could enable the execution engine’s logic to use techniques such as data-aware scheduling, automatic data replication, and smart data placement. On the other hand, the file-usage aware format is not intuitive. It requires the knowledge of file usage in addition to the command line that will execute the task.

\textbf{5.1.2 Script - File System}

The design options for the interface between the scripts and the file system are POSIX and non-POSIX. When a script needs to access a directory to find the file names in that directory (e.g., line 6 in Listing 6.2), the programmer can use \texttt{ls}. This is more convenient for programmers than a customized library’s remote procedure call. On the other hand, collective file movement in this runtime file system design is not supported by the POSIX standard, so the only option is to implement the interface as customized command lines that are callable inside a script.

In AMFORA, the interface between the script and the runtime file system is a mix of POSIX compatible operations and non-POSIX commands. The POSIX interface allows a programmer to access files that are spread across multiple nodes, while the non-POSIX commands give performance improvements (AMFORA_multicast, AMFORA_gather, AMFORA_scatter, AMFORA_allgather), or build in support for particular file formats (AMFORA_shuffle). This mixed design preserves the ease of programming without loss of potential performance improvements, though it requires the programmer to have knowledge of the data flow patterns.
5.1.3 Task Engine - File System

An execution engine communicates with the runtime file system for file location lookup and task execution. File locations can be exposed either by a customized remote procedure call or they can be embedded in the POSIX standard’s additional attributes. In either case, extra execution engine logic is required to process the information. We chose a customized remote procedure call for AMFORA Shell, as it is more flexible than enabling asynchronous file access.

When a task is executed, a POSIX-compatible file system can trivially provide file access. Since many programmers lack root access on the machines where they run, FUSE [31] is often used as a POSIX interface. However, the use of FUSE can limit I/O performance because the FUSE write buffer is at most 128 KB. A second solution is using a customized file system interface, which is more efficient but needs the task to be in file-usage aware format and requires a translation layer between the file system and the execution engine. On the other hand, many applications can not benefit from the higher bandwidth of the non-POSIX interface, as the I/O is limited by the application tasks themselves. In such cases, a POSIX interface makes the communication between execution engine and runtime file system much easier.

AMFORA implements the POSIX interface for task execution, since we previously decided that the task format would be file-usage blind.

5.2 Dispatching Tasks at Large Scale

AMFORA’s task management scheme uses the Minimum Spanning Tree (MST) algorithm to evenly spread all queued tasks to all AMFORA daemons. Each AMFORA daemon maintains an IP address list of all daemons, with each list in identical order. When the execute command is launched on one daemon (the launching daemon), the launching daemon will recursively pop half of the addresses from the IP address list (the popped addresses compose
the target list), and send half of the queued tasks to the first address on the target list. The leading address of the target list then become the immediate child of the launching daemon. The send operation terminates when the IP address list is empty. The launching daemon then starts to execute the remaining tasks in the queue.

When a daemon completes all of its tasks, it will collect the information about those tasks information (execution time, exit code, standard error and others) and check if all its immediate children have finished. If all its children have returned, then it acknowledges back to its parent. In a recursive way, the completed task information thus finally reaches the compute nodes where the execution command was initially launched.

### 5.3 Assuring Data Locality without File Usage Information

In §5.1.1, we discussed the file-usage blind task format. The advantages of this format is that the command line for a task is exactly the same as the command line the user would use to run the task on a single computer. If users can run the tasks in parallel in the file-usage blind format, they are not required to have knowledge of the tasks’ file usage. This gives the runtime system no information about file usage; thus it is not feasible for the runtime system to do data-aware scheduling. And the data flow pattern detection algorithm presented in §5.5.1 does not apply to this situation.

However, dispatching tasks to the compute nodes that host the input files is a key for application execution performance. We solve this problem by coordinating the task queuing in AMFORA using the file order returned by regular list directory operation. The details is explained in next paragraph.

Our solution is meant to satisfy the following requirements:

- The task format is file-usage blind
- AMFORA Task Engine supports queuing individual task
• AMFORA Task Engine can dispatch all tasks to all AMFORA daemons when instructed

An AMFORA Shell code segment shown in Listing 5.1 explains how the locality is assured. The code segments replicates the files in the inputdir/ to the files in outputdir/. Line 3 loads the inputdir/ directory from the shared file system to AMFORA File System in parallel. It also creates the inputdir/ in AMFORA File System before the parallel load. Line 5-7, the code iterates over the files in inputdir/ and queues a task that consumes the input file. Upon calling AMFORA_execute in Line 8, all tasks are dispatched to the AMFORA daemons to be executed.

Listing 5.1: Assuring Data Locality without File Usage Information

```bash
#!/bin/bash

AMFORA load inputdir/
mkdir outputdir/
for file in `ls inputdir/`
do
  AMFORA_queue cat inputdir/$file > outputdir/$file
done
AMFORA_execute
```

AMFORA addresses the data locality issue by preserving the file name order and task order in the AMFORA_load, ls, and AMFORA_execute commands. For example in Line 3, the AMFORA_load contains an implicit ls operation, where it first runs an operation on the inputdir/ and gets the file list in alphabetical order. Then AMFORA_load will do a static mapping between the files and AMFORA FS daemons, so each AMFORA FS daemon copies an distinct set of files from shared file system. In Line 5, AMFORA iterates over the inputdir/ and queues the task in exactly the same order as the file names returned by the explicit ls operation. When AMFORA_execute is launched, AMFORA will do the mapping between tasks and AMFORA FS daemons, so that each AMOFRA daemon runs
for a distinct set of tasks. So we see that if the file name list in AMFORA_load is the same as the one returned by the explicit \texttt{ls} operation, we can always guarantee that the tasks will each be sent to the AMFORA FS daemon that cached the corresponding input file.

5.4 Scalable Task Dependency Resolving via Asynchronous File Access

AMFORA allows for asynchronous file access: when a task execution fails because a file does not exist, the execution engine temporarily puts the failed task in a task hashed data structure, using the missing file as the key and the task as the value. At the same time, the AMFORA FS daemon that will eventually hold the metadata of the file associates the requester’s address with this file. When the file is later produced, the metadata entry is created on the AMFORA FS daemon. The AMFORA FS daemon then notifies the requester. The requester replicates the file from its producer and then retries the previously failed task. If a task has multiple input files, this process will repeat if other files are unavailable.

With asynchronous file access, task dependencies can be resolved at runtime in a fully distributed manner. The task dependencies are resolved through the file availability status, and are spread across all AMFORA FS daemons with an identical distribution to that of the file metadata in the AMFORA File System.

5.5 An Early Task Engine with File-usage Aware Task Format

In an early version of AMFORA, we implemented a hierarchical task engine [85] (referred to as AME in the rest of this chapter) that uses a file-usage aware task format. In this implementation, we partition all the compute nodes into groups and select one compute node of each group as a dedicated task dispatcher while other compute nodes are workers. The application performance we show in §7.1, §7.2 and §7.3 are measured with this task engine implementation.
We qualitatively evaluated the pros and cons of the file-usage aware task format and the file-usage blind task format in §5.1.1. The file-usage aware task format has the opportunity to enable the following features that would be infeasible or difficult to implement with the file-usage blind task format:

- Data-aware scheduling
- Data flow pattern identification at runtime
- Automatic data replications for load balancing purpose
- Asynchronous file transfer based on file availability

Data-aware scheduling is trivial with the file-usage aware task format, as the task engine can simply check the input file location, then send the task to where the input file is. §5.5.1 introduces a data flow pattern identification algorithm with file-usage format information. §5.5.2 explains how the file-usage aware task format enables asynchronous file transfer (asynchronous gather).

### 5.5.1 Data Flow Pattern Detection with File Usage Information

This section presents a new interactive design between AME and the AMFORA File System to enable data flow pattern detection and corresponding data placement at runtime.

This system must meet several technical requirements:

- The task format is file-usage aware
- The AMFORA File System metadata server can expose file locality upon queries
- AME is able to send a task to a specific address
- The applications stages can be differentiated by the executables’ names
- AME will check the availability of input files of the tasks before dispatching them, and keeps all file availability statuses.
We first explain the multicast pattern detection and data placement in detail. Then we briefly introduce the identification of other patterns. In our current implementation, we do not have an effective algorithm to detect the scatter data flow pattern.

**Multicast Handling**

There three sub-problems to solve for multicast: (1) identify the data flow pattern, (2) determine the number of replicas, and (3) decide where to place the replicas for load balancing. We present the details of multicast as an example of the collective operations because it is the most complicated and because it involves pattern identification as well as quantitative decisions. Initially we consider a simple example, with a centralized AME dispatcher and 200 AME workers. Two stages of computation are involved: a and b. We have 100 tasks in stage a, and 1,000 tasks in stage b. The tasks in stage a do not have input files, and each produces one output file, which will be consumed by 10 tasks in stage b. Each stage b task has only one input file.

**Pattern Identification**

AME pre-processes all the tasks and produces two hash tables. The first one has the keys being the name of the stage, which could be the application executable names. The values are a list of input file names associated with all the stage’s tasks. So in our simple example, the table, shown in Table 5.5.1, has two records. The dispatcher also produces another nested table with stage name as key; the value is a sub-table, with keys being the input file names and values as lists of tasks that will consume each input file, as shown in Table 5.5.1. Using this table, an AME dispatcher identifies a multicast pattern by comparing the number of subsequent tasks of a file with some predefined threshold.

**Replica Number Decision**

Once a file is produced and the AME dispatcher is notified of the address of that file, we compute the fraction of this file’s consumption among its peer files in the same stage. In this case, file1 will be consumed by ten stage b tasks. The
consumption of file1 is 1% of the total file consumption for input files in stage b. Thus we need to multicast file1 to 1% of the AMFORA FS daemons, which is two. Since the file is already on one AMFORA FS daemon, we replicate the file only once.

**Replica Placement** To distribute the input files within a single stage in a balanced way, we need the exact AMFORA FS daemon and data mapping. This information is stored in a table with storage server address as key and the file list as value. The table can be initialized when stage a tasks are dispatched to AMFORA FS daemon. In this case, the future output file will be produced by the task on the same node, so we put the daemon’s address and output file name into the table. Then we select the AMFORA FS daemons with the least number of input files in this stage to hold the replicas.

**Extending the Solution** Multiple AMFORA File System metadata servers will not make it more difficult to solve the issues above, since each metadata server will be contacted by only the centralized AME dispatcher in order to obtain file state update and location information. But a hierarchical AME dispatcher architecture does raise some issues. By this, we mean that we would group the compute nodes into partitions and appoint one compute node in each partition as the second level scheduler. The compute node where the user submits tasks

---

**Table 5.1: Stage-InFile-Table**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>[]</td>
</tr>
<tr>
<td>b</td>
<td>[file1, file2, ..., file100]</td>
</tr>
</tbody>
</table>

**Table 5.2: Subtable with stage b**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>file1</td>
<td>[task1, task2, ..., task10]</td>
</tr>
<tr>
<td>file2</td>
<td>[task11, task12, ..., task20]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>file100</td>
<td>[task991, ..., task1000]</td>
</tr>
</tbody>
</table>
to is the first level scheduler.

While each second level scheduler could still produce the tables such as Tables 5.5.1 and 5.5.1, some of the input files of stage b will not be produced within this dispatcher’s scope. However, the dispatcher will still get an on-time file state update as long as that file is an input of some task of stage b and this specific task is in the queue of this second level scheduler. Therefore, we only need to make a small change in the above solution in order to make it work with the hierarchical scheduler. If the file is not produced within this second level scheduler’s scope, and there should be N replicas from this second level scheduler’s point of view, then we make N copies of that file within this second level scheduler’s scope. Otherwise, if the file is produced within this second level scheduler’s scope, the scheduler should make N-1 replicas.

To remedy the issue with multiple input files for each task, we introduce a new concept of file group. A file group is defined by all input files that are also output files of the previous stage for a stage b task. We replace the role of file in the Tables 5.5.1 and 5.5.1 with file group. The files are replicated based on the production of the whole group.

We also consider a potential problem when the number of stage b tasks is less than the number of AMFORA FS daemons. In this case, we simply replicate the stage b input files proportionally to their consumption.

Other Collective Operations Handling

Gather is identified by AME dispatcher based on the number of input files of a task. Scatter can be correctly but not efficiently identified, since the dispatcher needs to communicate with all AMFORA File System metadata servers in order to determine the subsequent tasks. If there is a single task where the input and output file sizes exceed the compute node’s memory limit, we run it on the login node.
5.5.2 Asynchronous File Transfer

As discussed in §5.4, a task whose input files are not immediately available can be launched based on file availability update using file state change broadcasting. Without file usage information, a task with multiple input files may be tried for an indefinite number of times before all its input files are available, since the task engine in the scenario has no idea which files are input files and the only thing it could try is to launch the task, then gets the return code from the operating system. In this AME implementation, resolving task dependency through asynchronous file access can be done without launching tasks. An AME worker that is to gather files proceeds as follows. For each of the task’s input files, it requests that file from AMFORA FS. If the file is available, then the worker copies the file. Otherwise, the worker proceeds to the next input file and uses another thread to wait for acknowledgements of all the not-yet-available files. As each file is produced, the worker is notified and copies the file. We call this approach asynchronous gather in order to differentiate it from the collective gather, which is synchronous with the task completion of the previous stage. Collective gather benefits from the parallel transfer at each step, whereas asynchronous gather benefits from overlapping computation and file transfer. We compare the asynchronous gather and collective gather with two real applications, the results can be found in §7.1 and §7.2.

5.6 A Summary of Task Formats

During the research and development process of AMFORA, we developed two implementations of task engines. The first, AME, requires a file-usage aware task format and is compatible with Swift and Pegasus. The second, AMFORA Task Engine, is compatible with the file-usage blind task format and the task descriptions are no different from the ones that users run on a computer.

Though the file-usage task format has many user experience advantages, it loses many potential performance improvement opportunities. Table 5.6 summarizes the missing or
Data-aware scheduling is trivial for the file-usage aware task format, as a task scheduler query the input file location and send the task directly to that place. For the file-usage blind task format, determining which file is an input is infeasible given only the task format, and there are also issues with data-aware scheduling. The solution presented in §5.3 uses the identical order returned by the ls command and the AMFORA load (AMFORA scatter). The file-usage blind task format also eliminates the opportunity of data flow pattern detection at runtime as well as the automatic data replication for load balancing. A task with many input files with the file-usage blind format can be retried for an indefinite number of times until all its input files are available, while with the file-usage aware format, the task engine can make sure that all input files are available before launching the task.
This chapter presents the AMFORA programming model and its interoperability with other serial scripting languages. Section 6.1 documents the AMFORA task and data management interfaces. Section 6.2 examines the AMFORA programmability, via investigations with real application examples of Montage, iterative PageRank and KMeans.

6.1 AMFORA Shell Programming Model

In the scripting programming paradigm, files and directories are often used as programmable elements much as in-memory variables are used in MPI [34] and key-value pairs are used in MapReduce [22]. A script glues programs together by specifying shared files or directories that are used to pass intermediate data. Thus, to produce or consume files and directories, applications must stick to the POSIX interface.

AMFORA Shell provides three groups of commands to programmers. The first two groups deal with data management while the third group deals with task management.

The commands in the first group are functional data management commands, meaning that the script can not correctly execute without them; they include:

- **AMFORA_load**: loads a directory from persistent storage to AMFORA; the transfer can be done in parallel if the persistent storage is globally accessible
- **AMFORA_store**: stores a directory from AMFORA to persistent storage; the transfer can be done in parallel if the persistent storage is globally accessible
- **AMFORA_shuffle**: shuffles all files in a directory to another directory based on the first column of the file

The collective data movement commands in the second group can be thought of as performance hints, since a script runs correctly without calling them, but with lower performance.
They include:

- **AMFORA multicast**: multicasts a file or a directory to all compute nodes
- **AMFORA_gather**: gathers all files in a directory to one compute node
- **AMFORA_allgather**: gathers all files in a directory to all compute nodes
- **AMFORA_scatter**: scatters all files in a directory to all compute nodes

The third group implements an interface to the execution engine’s task management system and they are functional:

- **AMFORA_queue**: push a task into the queue; user can specify the data-aware option through parameters
- **AMFORA_execute**: execute all tasks in the queue, blocked until all tasks finish

AMFORA_queue returns immediately, while AMFORA_execute returns only when all queued tasks have finished.

### 6.2 AMFORA Shell Examples

Some examples of parallel scripts that combine AMFORA Shell and Bash are shown in this section.

#### 6.2.1 Montage

Listing 6.1 is an AMFORA Shell implementation of Montage [42]. Eight stages are expressed in 38 lines of code. The mProjectPP, mDiffFit, and mBackground stages can run in parallel; mImgtbl, mConcatFit, mAdd can benefit from parallel file transfer. mOverlaps and mBgModel can run on any AMFORA daemon. In Line 19, the programmer produces the input file names for mDiffFit tasks by processing the content of diffs.tbl.
Listing 6.1: Parallel Script for Montage

```bash
#!/bin/bash

# mProjectPP
mkdir tempdir/
for file in 'ls rawdir/'
do
  AMFORA_queue mProjectPP rawdir/${file} tempdir/hdu_${file} template.hdr
done
AMFORA_execute

AMFORA_gather tempdir/
  mImgtbl tempdir/ images.tbl
mOverlaps images.tbl diffs.tbl

mkdir diffdir/
# processing diffs.tbl requires
# programmer's interaction
for filepair in 'process diffs.tbl'
do
  AMFORA_queue mDiffFit ${filepair} diffdir/
done
AMFORA_execute

AMFORA_gather diffdir/
  mConcatFit diffdir/ fits.tbl
mBgModel images.tbl fits.tbl corr.tbl

mkdir corrdir/
for file in 'process corr.tbl'
do
```

80
AMFORA_queue mBackground ${file} corrdir/${file} corr.tbl
done
AMFORA_execute

AMFORA_gather corrdir/
mAdd corrdir/ final/m101.fits

6.2.2 PageRank

Listing 6.2 implements a data-parallel PageRank [15]. The script has three explicit stages. First, lines 5 to 10 run, for each file in link/, a PageRank_Distribution task to calculate the PageRank score distribution from one page to another, and then sum the scores for the same page. As score.txt is needed for every task, the programmer uses AMFORA_multicast to broadcast score.txt to every node. Second, line 13 calls AMFORA_shuffle to reorganize the contents of all files in temp/ to target/ according to the hash value of the first column of the files in temp/. Third, lines 15 to 20 invoke a PageRank_Sum task for each file in the target/ directory. PageRank_Sum sums the score for each page, applies a damping factor to the score, and writes the result to a file in result/. The programmer then calls AMFORA_gather on result/ and redirects all files in result/ to new-score.txt. The script continues until a convergence condition (line 24) is satisfied.

Programmers require some insight into the data flow and parallel execution patterns of their script if they are to determine which AMFORA Shell commands are needed to achieve efficient parallel execution. For example, in parallel PageRank, if the AMFORA_gather is missing, files are transferred sequentially, which may be acceptable on a small computer but slow on many processors. If AMFORA_allgather, AMFORA_multicast, and AMFORA_scatter are missing, I/O traffic is congested due to the large number of concurrent connections to a single server. On large-scale computers, such congestion may be inefficient, or even compromise the stability of the parallel computer.
Listing 6.2: Parallel Script for PageRank

```bash
#!/bin/bash
while [ ${converge} -ne 0 ]; do
    AMFORA_multicast score.txt

    mkdir temp/
    for file in 'ls link/'
    do
        AMFORA_queue PageRank_Distribution link/$(file) score.txt temp/$(file).temp
    done
    AMFORA_execute

    mkdir target/
    AMFORA_shuffle temp/ target/

    mkdir result
    for file in 'ls target/'
    do
        AMFORA_queue PageRank_Sum target/$(file) result/$(file).result
    done
    AMFORA_execute

    AMFORA_gather result/
    cat result/* | sort > new-score.txt
    converge = 'diff score.txt new-score.txt | echo $?'
    mv new-score.txt score.txt
done
```

6.2.3 KMeans

Listing 6.3 shows an AMFORA Shell script for K-means. Each iteration of K-means is a three step procedure: group, shuffle, and centroid. Line 4 broadcasts the randomly
selected centroids to all AMFORA FS daemons as it is needed by all group tasks. Lines 10-15 run the group tasks in parallel. Each group task computes the distances between each vertex and all candidate centroids and writes the a pair of vertices of the closes centroid and the vertex itself. Line 20 shuffles the outputs of Group tasks and reorganizes them into a group of files, with each file contains all the vertex pairs with the same candidate centroid as the closest centroid. The centroid tasks (Lines 24-28) read the shuffled results, and calculate a new centroid for each group whose vertex pair has the same closest centroid. Lines 31 and 32 gather all new centroids together, and sort them in a text file. Line 33 checks the convergence condition (if the new centroids are the same as the previous). The whole process iterates until the centroids do not change any more.

Listing 6.3: Parallel Script for K-means

```bash
#!/bin/bash

# File content format is: v(x, y)
while [ ${converge} -ne 0 ]; do
    AMFORA multicast centroid.txt

    # calculate the distance of every node
    # to the centroids, output the closest
    # centroid in the form of vertex pair of (v c)
    mkdir temp/
    for file in `ls input/`
    do
        AMFORA_queue KMeans-Group input/${file} centroid.txt temp/${file}.temp
    done
    AMFORA_execute

    # shuffle the intermediate file
    # according to centroids
    mkdir temp/
```
6.2.4 MonteCarlo

Listing 6.4 shows an AMFORA Shell script for a Monte Carlo simulation to compute the value of $\pi$. Line 2 loads the seed files from persistent storage to AMFORA FS. Lines 3-8 generate one billion random coordinates within a square. Lines 14-19 calculate the distances between each coordinate and the centroid, then count how many coordinates fall in the circle. Lines 22-23 summarize the total number of coordinates that fall in the circle. Thus the programmer calculates the value of $\pi$ with the total number of coordinates, the number of coordinates that fall in the circle, the diameter of the circle, and the area of the square.

Listing 6.4: Parallel Script for Computing Pi with Monte Carlo Algorithm

```bash
#!/bin/bash
#load seed files into AMFORA
AMFORA_load seed/ .
```
# Generating random pairs of float numbers \((x, y)\)

```bash
mkdir random/
for file in 'ls seed/'
do
  AMFORA_queue RandomGen seed/${file} random/${file}.random
done
AMFORA_execute
```

# Calculating and write the number of pairs that meet the condition: \(x^2 + y^2 < r^2\)

```bash
mkdir result/
for file in 'ls random/'
do
  AMFORA_queue Calculate random/${file} result/${file}.result
done
AMFORA_execute
```

# Sum the contents of files in result/

```bash
AMFORA_gather result/
Sum result/ > result.txt
```
CHAPTER 7
ACCELERATING MANY-TASK APPLICATIONS ON LARGE-SCALE COMPUTERS

In §3, we measured the baseline performance of MTC applications and analyzed the bottlenecks that we found to be responsible for slow performance. Based on the results, §4 discussed technical solutions and alternatives to tackle those bottlenecks. We now examine the performance improvements contributed by each technique and present the overall performance improvements obtained relative to GPFS when using the AMFORA framework. We show in §7.1 that AMFORA improves the parallel stages of the Montage application by 83.2%, for technical reasons that we discuss. In §7.2, we show that the MTC version of BLAST (mtcBLAST) delivers similar performance to mpiBLAST, which is commonly believed as an efficient and scalable parallel BLAST implementation. In §7.3, we show that CyberShake PostProcessing performance is improved by 7.9%; we discuss the reason for this moderate improvement and the limitation that it reveals in the AMFORA framework. In §7.4, we examine the performance improvements achieved for PageRank, KMeans, and MonteCarlo implementations.

The Montage, BLAST and CyberShake PostProcessing applications are executed through an early version of the AMFORA Task Engine [85], which has a hierarchical architecture. In its deployment on the Blue Gene/P supercomputer, we designate one node out of every 64 nodes as their task dispatcher. We also run the AMFORA File System metadata server on that node too. This version of AMFORA Task Engine accepts the file-usage aware task description, thus it is capable of data aware scheduling, data flow pattern detection at runtime, and asynchronous gather.

The PageRank, KMeans, and MonteCarlo application are executed through a later version of AMFORA Task Engine [84]. In this version, every node is a task dispatcher, worker, and file system metadata server as well as a file system storage server. This version of AM-
FORA Task Engine accepts the file-usage blind task format, which preserves the original task command line in the applications.

### 7.1 Montage

Our test case of Montage is a 6x6 degree 2MASS mosaic centered at Galaxy m101. Table 3.1 shows the number of tasks, input files, and output files and the amount of I/O performed for each stage. Figure 7.1 shows the time-to-solution ration between various versions of Montage against the STAGING base case. The Montage version that we use as the STAGING base case is one in which input files are initially stored on GPFS but then staged to RAM disk for reads and writes. We use STAGING performance as the base case because the different methods used to implement the MPI and MTC versions make a stage-by-stage comparison of the MPI and MTC versions infeasible. Also, the MPI version performs all reads and writes directly on GPFS, which is extremely slow: STAGING ran in 45% of the time of the MPI implementation (with all I/O directly using the shared file system).

We studied the performance difference for the MPI version of Montage and for each of the optimization methods data cache (CACHE), data-aware scheduling (AWARE), collective gather (COGATHER), and asynchronous gather (ASGATHER), against the STAGING base case. In each case, we show the time-to-solution ratio for each parallel stage (except mAdd), the sum of the parallel stages (Sum_Para), and the sum of the whole workload (Sum_All) on 512 BG/P cores, compared with the STAGING base case. Ratios less than 1 represent improvements, whereas ratios greater than 1 are slowdowns.

For MPI Montage, we show only the sum of the parallel stages and the sum of the whole workload, since we cannot easily separate out the other stages. In the asynchronous gather column, the ratio given for mImgtbl represents both mProject and mImgtbl, since these two stages are overlapped when using the asynchronous method and thus the time taken by each cannot be separated; similarly, the ratio given mConcatFit encompasses both mDiffFit and mConcatFit.
Figure 7.1: Montage workload time-to-solution ratio compared with the GPFS base case for the parallel stages.

### 7.1.1 Data Cache

Compared with the STAGING base case, Data Cache reduces the time to solution by 39.6%, 46.7%, and 1.7%, respectively, for the mProject, mDiffFit, and mBackground stages. These improvements result from the elimination of writes to GPFS in the three stages and the replacement of the input file copies from GPFS with peer-node file transfers in mDiffFit and mBackground. The mBackground output files have to be moved to GPFS in both cases, so the 1.7% decrease is due to the different input handling scheme. Overall, the Data Cache improvements are due primarily to reducing GPFS input and output, from 53.6 GB to 8.1 GB.

Stages mImgtbl and mConcatFit are 6.2x and 4.5x slower with Data Cache than STAGING, respectively. The gather pattern of these stages causes the lower performance. The mImgtbl and mConcatFit tasks have 1,319 and 3,883 input files each, and the sequential file location lookup and transfer is less efficient than file copies from GPFS.
7.1.2 Data-Aware Scheduling

Data-aware Scheduling eliminates half of the input transfer in the mDiffFit stage and all of the input transfer in the mBackground stage. In practice, data-aware scheduling reduces run time by 40.7%, and 45.1% relative to Data Cache for mDiffFit, and mBackground, and by 68.4% and 46.1% over STAGING, respectively. Stages mImgtbl and mConcatFit perform 7.7x and 5.8x slower than with STAGING, respectively, for the same reason as for Data Cache.

An unexpected improvement is that the average mDiffFit runtime decreases from 10 s to 1.2 s. This improvement comes from a memory cache hit for an input file, since this file is consumed on the same node where it is produced with Data-aware Scheduling and thus can be read from memory cache. We see the same result for mBackground, whose average runtime decreases from 8.3 s to 6.9 s.

7.1.3 Collective Gather

To optimize file transfer in the stages that have a gather data flow pattern, we apply collective gather along with Data-aware Scheduling. We see a decrease in runtime of 71.0% and 53.6% for mImgtbl and mConcatFit, respectively, compared with the STAGING base case. Looking at just the file transfer time, Collective gather saves 77.7% and 86.6% for mImgtbl and mConcatFit. The combination of these two techniques decreases the whole workload time to solution by 83.2% compared with that of the MPI Montage implementation. Table 7.1 shows the time distribution of mImgtbl and mConcatFit with collective gather. In this table, input is the end-to-end gather time; execute is the execution time for the task; output is the time to write the output (short as there is only one output file per task); and overhead is system overhead. The system overheads for mImgtbl and mConcatFit, 0.80 s and 1.33 s, respectively, are due to the need to transfer and process the long task descriptions that enumerates all input files; these descriptions are 21.3 KB and 147.92 KB in size, respectively. Our alternative Asynchronous Gather solution does not perform as well as collective gather in this case, since the task runtime is not irregular enough to mask the data movement overhead.
Combining the data-aware scheduling and collective gather techniques, the time-to-solution of Montage parallel stages are improved by 83.2% compared to the case with all data read and written on GPFS. The overall performance of all stages is improved by 59.3%.

### 7.2 BLAST

Our parallel BLAST (which we refer to as mtcBLAST) is based on Parallel BLAST [47]. In our experiments, we match a random set of sequences selected from \( nr \), a non-redundant protein database maintained at NCBI, against the full \( nr \) database. On \( n \) (256, 1024, 4096, 16384, and 32768) cores, we randomly choose \( n \) sequences from \( nr \). We aggregate these sequences into \( n/16 \) query files, each containing 16 sequences; thus, our scheduling granularity is 16 sequences. Table 7.2 shows the number of tasks, the number of distinct inputs and outputs, and the total input and output sizes. \( N \) is defined as the number of fragments of the \( nr \) database, while \( M \) is defined as the number of query files, each of which contains 16 sequences.

<table>
<thead>
<tr>
<th>Stage</th>
<th># Tasks</th>
<th># In</th>
<th># Out</th>
<th>In (MB)</th>
<th>Out (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fastasplitn</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>4039</td>
<td>4039</td>
</tr>
<tr>
<td>formatdb</td>
<td>N</td>
<td>N</td>
<td>3N</td>
<td>4039</td>
<td>4400</td>
</tr>
<tr>
<td>blast</td>
<td>N(^*M)</td>
<td>N+M</td>
<td>N(^*M)</td>
<td>73(^*N)(^*M)</td>
<td>2.4(^*N)(^*M)</td>
</tr>
<tr>
<td>merge</td>
<td>M</td>
<td>N(^*M)</td>
<td>M</td>
<td>2.4(^*N)(^*M)</td>
<td>4.8(^*M)</td>
</tr>
</tbody>
</table>

Normally, users simply run fastasplitn and formatdb once and reuse the formatted database slices. We also run fastasplitn once, since it just does file fragmentation based on text; but we put formatdb in our mtcBLAST workload because it is an intriguing example of the multicast data flow pattern and we can run it in parallel. We partition the \( nr \) database
into 63 slices; formatting those slices on 63 nodes takes 56 s. (The number 63 is chosen because we used a hierarchical AMFORA deployment for this experiment. For every 64 compute nodes, we dedicate one compute node as the AMFORA Task Engine as well as the AMFORA File System metadata server; the other 63 nodes are workers.)

7.2.1 Load Balancing

The blastp task lengths are not related to the query sequence length [45], and we saw several starving situations in our small-scale test on 256 cores. Therefore, we apply a simple work-stealing mechanism in the AMFORA framework. For scalability, we limit stealing to a compute node’s neighbors (on BG/P, we choose neighbors on the 3D torus network). While this limited stealing is not a globally optimal solution, it delivered a 3% to 9% decrease in time to solution for the blastp stage, as shown in Figure 7.2.

![Figure 7.2: Work-stealing improvement in mtcBLAST’s blastp. The vertical bars show the time-to-solution in seconds and correspond to the left Y axis. The dots show the performance improvements and correspond to the right Y axis.](image-url)
7.2.2 Asynchronous Gather

To evaluate the choice between Collective Gather and Asynchronous Gather, we run each on the same query workload. Collective Gather takes 143.3 s to finish the blastp and merge stages, while Asynchronous Gather takes 139.6 s, because of overlapping the computation of blastp and the input data transfer of merge. We therefore choose Asynchronous Gather in mtcBLAST.

7.2.3 Comparison with mpiBLAST

We compare mtcBLAST with mpiBLAST on 256 to 32,768 cores, with the same queries run against the nr database in both cases. For a fair comparison, mtcBLAST uses the NCBI legacy engine. Figure 7.3 shows that mtcBLAST is somewhat faster than mpiBLAST at each scale. This improvement is due to AMFORA support for large-scale data-aware scheduling, data replication, gather data flow optimization, and workstealing.

![Figure 7.3: Performance comparison of mtcBLAST and mpiBLAST](image-url)

Figure 7.3: Performance comparison of mtcBLAST and mpiBLAST
7.3 CyberShake-PostProcessing

We run the PostProcessing workload for the CyberShake TEST site (a small benchmark problem) on 2048 cores on IBM BG/P. Each core has 512 MB available memory. Table 7.3 shows the stages of the application, the related number of tasks, the number of distinct inputs and outputs, and the total amount data transferred between the shared file system and the compute nodes.

Table 7.3: Number of tasks, inputs, and outputs, and input and output size, for each CyberShake PostProcessing stage

<table>
<thead>
<tr>
<th>Stage</th>
<th># Tasks</th>
<th># In</th>
<th># Out</th>
<th>In (MB)</th>
<th>Out (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>extract</td>
<td>393</td>
<td>395</td>
<td>786</td>
<td>212200</td>
<td>65600</td>
</tr>
<tr>
<td>seis</td>
<td>11868</td>
<td>1179</td>
<td>11868</td>
<td>65600</td>
<td>285</td>
</tr>
<tr>
<td>peakGM</td>
<td>11868</td>
<td>11868</td>
<td>11868</td>
<td>285</td>
<td>4</td>
</tr>
</tbody>
</table>

Each extract task reads in two common large (2.7-GB) input files along with a small file that is unique for each task. Each task produces two output files, with sizes that vary from 12 MB to 168 MB. The total amount of the output is 65.6 GB. The input and output are far beyond the capacity of the compute node’s RAM disk, so we don’t use the Data Cache scheme for this stage. As the tasks are also memory intensive, we configure AMFORA with one execution thread on each worker. The seis tasks read the output files from the extract stage and one additional file as inputs, and they each produce a single 24-KB output file. Finally the peakGM tasks read the 24-KB file as input, and produce a 360-B output file. Please note, even though the total amount of peakGM output data is not significantly large, it requires 1.5 GB on shared file system due to the chunk configuration.

AMFORA will not improve the performance of the extract stage, as the I/O size far exceeds the capacity of the compute node. The Data Cache and Data-aware Scheduling techniques decrease the running time of the seis stage from 829.3 seconds to 757.4 seconds, as the output files are cached on the RAM disk of compute nodes. The time-to-solution of peakGM stage is reduced from 59.1 seconds to 56.7, as the tasks do not access input data on shared file system. Combining the two techniques, the AMFORA framework reduces the
complete workload time-to-solution by 7.9%.

7.4 PageRank, KMeans, and MonteCarlo

In this section, the application (PageRank, KMeans, and MonteCarlo) performance comparisons between the AMFORA and GPFS are presented. We run the PageRank application with the Wikipedia 5.7m-page page link dataset. In K-means, we randomly generate one million coordinates, and cluster them into 2,048 groups. In Monte Carlo, we compute $\pi$ by counting random coordinates in a square that fall in the circle with diameter equal to the square’s width.

7.4.1 PageRank

We run with the Wikipedia 5.7m-page page link dataset (1.1 GB size), to produce an output of 120 MB, with 4 GB of intermediate files. PageRank-Distribution distributes the score of the current page to the pages it links to, and produces output files with the link and its score. PageRank-Shuffle reorganizes those files so that all records of the same page are in a single file. PageRank-Sum adds the scores for each page to produce a new score for it.

In Figure 7.4, we first fixed each problem size, and ran the workloads on an increasing number of AMFORA daemons (one daemon on one compute node) to study the scalability of the application using AMFORA. We compare each application stage’s performance on AMFORA against that on GPFS, except the shuffle stages, as a file based shuffle on $N$ compute nodes results in $N^2$ intermediate files. Writing this large number of intermediate files to GPFS can dramatically slow GPFS, which is shared among many users.
In K-means, we randomly generate one million coordinates, and cluster them into 2,048 groups. This test has 19 MB of input, 38 MB of output and 76 MB of intermediate data. KMeans-Group first computes each point’s distance to all candidate centroids, and writes a pair of points with the nearest centroid in the first column. KMeans-Shuffle then reorganizes those output files based on the candidate centroids. KMeans-Centroid reads all point pairs with the same candidate centroid as key and calculates the new candidate centroid.
7.4.3 MonteCarlo

In Monte Carlo, we compute $\pi$ by counting random coordinates in a square that fall in the circle with diameter equal to the square’s width. MonteCarlo-Sim generates one billion random coordinates within a square, counts those that fall in the target circle, and writes this to an output file. MonteCarlo-Sum reads those files and adds them together.
7.5 Analysis

AMFORA both performs well and scales well for application stages where tasks access distinct files, such as Montage-mProjectPP, Montage-mBackground, and KMeans-Centroid. Stages such as PageRank-Distribution, PageRank-Sum, KMeans-Group and MonteCarlo-Sim also have this pattern, but the AMFORA improvement in these stages is marginal, because the long task runtimes hide the improvement in the small file I/O.

AMFORA works well when the I/O size is large but does not exceed the memory space of any AMOFRA daemon. Montage-mDiffFit’s tasks each read in two 4 MB input file, and write a 1 MB output file. Our test case has 3,883 tasks. With AMFORA, all writes are local to memory, and data-aware scheduling guarantees at least one read file is in memory. That results in a 3.6x speedup of this stage.

For computations that involve the shuffle operation, larger scale does not imply shorter time-to-solution. The optimal number of compute nodes for the problems size of our PageR-
ank and K-means cases are 256 and 512, respectively.

AMFORA eliminates GPFS I/O traffic by caching intermediate files in memory. This not only offers better performance, but also removes load on the shared resources (GPFS) that might be used by other applications on the same system.

AMFORA is slower than GPFS on \{64, 128, 256, 512\} compute nodes for MonteCarlo-Sum, which has a single task that reads many input files. Even though we move the input files into one node using AMFORA_Gather, the lower CPU frequency on the compute nodes than on the I/O nodes and the FUSE latency make AMFORA slower than GPFS on less than 1,024 nodes. However, we see that on 1024 nodes, AMFORA outperforms GPFS, as the task hits the GPFS concurrency wall, while the AMFORA’s performance is almost consistent (O(log(N)) time, where N is the number of nodes.)

Overall, we have shown that an intelligent runtime data management system can significantly improve the performance of many-task applications in situations in which shared filesystem access dominates execution time. Our methods as implemented in AMFORA reduce the time to solution for parallel stages in Montage by a factor of 83.2% (59.3% in all stages) and reduce the time to solution of CyberShake PostProcessing by 7.9%. Our mtcBLAST achieves performance comparable to that of mpiBLAST while preserving the flexibility of the core NCBI BLAST routines. For the computation intensive stages of the applications, e.g., PageRank-Distribution, KMeans-Group, MonteCarlo-Sim, AMFORA delivers similar performance to GPFS since the execution time of these stages is dominated by computation. Though AMFORA is slower than GPFS for the PageRank-Sum, KMeans-Centeroid, and MonteCarlo-Sum stages at small scale, our measurements show that AMFORA can run faster than GPFS at larger scale with the same problem size.
CHAPTER 8
CONCLUSIONS, CONTRIBUTIONS, AND FUTURE RESEARCH DIRECTIONS

My dissertation work has focused on data management in distributed systems. It has enabled concise, fast, and scalable parallel scripting on large-scale computers through efficient data management. The work also showcased how novel data management techniques allow specific scientific applications to be solved at large scale with pleasant programming and execution experience. This chapter concludes my dissertation work, states the contributions, and discusses future research directions.

8.1 Conclusions

We profiled and analyzed the I/O behavior of several typical MTC applications on the IBM BG/P supercomputers and identified several common I/O characteristics that slow down the system performance:

- Highly concurrent metadata traffic
- Highly concurrent I/O traffic
- Data flow patterns of multicast, gather, scatter, allgather and alltoall

The AMFORA system reduces the time to solution of the Montage application’s parallel stages by a factor of 83.2% (59.3% in all stages) and reduces the time to solution of CyberShake PostProcessing by 7.9%. The AMFORA based BLAST (mtcBLAST) performance is slight faster than mpiBLAST (mpiBLAST is commonly believed to be an efficient and scalable version of parallel BLAST on supercomputers) from 64 compute nodes to 8,192 compute nodes. The MTC Envelope benchmarking shows that the AMFORA file system runs 10x-100x faster than GPFS for all eight performance metrics and AMFORA shows good scalability up to 2,048 compute nodes (not tested on larger scale).
The AMFORA programming interface allows users to interact with the system using serial scripting languages such as Bash and Python. The Montage, Iterative PageRank, KMeans, and MonteCarlo scripts shown in Section 6 are concise and readable.

8.2 Contributions

Parallel scripting is a powerful and convenient tool to construct MTC applications, by gluing together existing programs without modifying them. Many interesting applications can be expressed with parallel scripts and executed on distributed or parallel platforms, such as clusters, grids, clouds and supercomputers. MTC applications exist across a wide range of domains that includes (but is not limited to): astronomy, biochemistry, bioinformatics, psychology, economics, climate science, physical chemistry, and neuroscience. However, we often see poor performance when running these applications on large-scale computers due to the applications’ rate and amount of file system I/O and the absence of appropriate optimizations for such parallel scripting applications. Domain scientists often seek to compose MTC applications with existing parallel programming frameworks: e.g. MPI and MapReduce. The mismatch between the programming frameworks and the applications’ basic data structures lead to unnecessary program verbosity and poor readability.

To understand the reasons for the poor performance of MTC application on large-scale computers, we first profiled several typical MTC applications to understand their common I/O operations, I/O volume, I/O concurrency, I/O pattern, and data flow pattern. Then, we defined the MTC Envelope as a file system benchmark that shows how a large-scale computer with a shared file system accommodates MTC applications. Taking GPFS on IBM Blue Gene/P supercomputer as an example, we measured and quantitatively studied the performance and scalability of the system. The results show that the MTC performance can be limited by the throughput and bandwidth of the shared file system and the performance scalability is poor due to GPFS’s single metadata server design, especially when the I/O traffic is metadata processing dominated. On the other hand, the data flow pattern
specific movement can be slow and inefficient, such as when a single compute node reads a
large amount of data while the other compute nodes sit idle. The two observations implied
two opportunities for performance improvements: data caching and faster collective data
movement.

Thus, the first goal of my dissertation was to design and build a data management
system on large-scale computers in the context of parallel scripting. The data management
system (AMFORA File System) have been implemented as a in-RAM shared file system
on large-scale computers. The AMFORA File System implements multi-read single-write
I/O pattern. It has a fully distributed metadata server design, where highly-concurrent
metadata traffic within a single directory are spread across as many servers as possible. In
addition to the regular POSIX interface, it also supports asynchronous file access; the file
system keeps records of file access requests before the file is created, and broadcasts the file
data location upon future file creation. The AMFORA File System also supports collective
data management, with which users can move data in specific data flow patterns faster
(than sequential transfer) at large scale. The functional data management techniques allow
file content reorganization based on the key value pair abstraction. The shuffle function
facilitates many parallel scripting applications that have previously been implemented as
MapReduce applications on large-scale computers. The functional data management also
allows loading data from the shared file system to the AMFORA File System in parallel, and
storing the data from AMFORA File system to the shared file system in the same manner.
The performance measurements show that AMFORA File System performs 10x-100x faster
than GPFS on the eight performance metrics we defined in MTC Envelope. The AMFORA
File System shows good performance scalability from 64 compute nodes to 2048 compute
nodes. Based on the layout of the AMFORA File System design, we believe the AMFORA
File System can continue scaling beyond 2048 compute nodes.

The second goal of this dissertation was to enable concise parallel scripting by exposing
the system performance improvements to the programming languages. We define the AM-
FORA Shell as the programming interface of the AMFORA File System. The AMFORA Shell’s interface can be divided into four categories: a regular POSIX interface, a collective data movement interface, a functional data management interface, and a task management interface. The four interface categories are exposed to users as POSIX-compatible command lines. Programmers can choose the serial scripting language (e.g., Bash and Python) to work with AMFORA Shell, as long as the serial scripting language is capable of launching Linux commands. Using Bash as an example language, we show real parallel scripts for Montage, iterative PageRank, KMeans, and MonteCarlo. These parallel scripts can be easily transformed from existing serial scripts by inserting AMFORA data management commands and prefixing the tasks with AMFORA task management commands. The parallel scripts composed with Bash and AMFORA Shell are concise and readable, and this relieves the programmers from having to change the original (task) programs.

To summarize, my dissertation work has identified the lack of data management as the key bottleneck of parallel scripting performance on large-scale computers. The AMFORA framework enables efficient data management, including scalable data caching, collective data movement, and functional data management while preserving the POSIX interface. AMFORA exposes these underlying system improvements through a POSIX-compatible Linux command line format, which enables interoperability with other serial scripting language and concise parallel scripts programming. Thus I have proven my thesis statement: Efficient data management can enable concise, fast and scalable parallel scripting on large-scale computers.

8.3 Future Research Directions

My future work will continue focusing on parallel scripting on large-scale computers, covering: 1) enabling more scientific applications, 2) improving the resilience of the overall system, and 3) adapting parallel scripting to future computer storage architecture changes.

At the time this dissertation is being written, many believe that advances in future science
research will be driven by big data processing and analysis. The data can be from large-scale simulations, a huge number of sensors, genome sequencing, internet logs, large-scale sky surveys, etc. Processing and analysis of the big data in a timely manner is key to scientific success of the research. I believe parallel scripting can solve part of (if not all) the problem, as many science domains already use serial codes to analyze and process simulation data. Parallel scripting can provide an easy, fast, and scalable way to parallelize this analysis and processing.

Parallel scripting, as a relatively new programming paradigm, is not as well known as other parallel programming paradigms such as message passing or PGAS. Education in parallel scripting is often in the form of boot camps or online tutorials. Today, enabling a new scientific application on large-scale computers using parallel scripting often requires parallel scripting designers who deeply understand the programming interface and underlying system working closely with domain scientists. This approach limits the number of users and the popularity of parallel scripting in many science domains.

The scale of the large computers is increasing at high speed (e.g., today’s IBM Blue Gene/Q supercomputer at Argonne National Laboratory has ∼750,000 CPU cores). This enormous amount of hardware significantly reduces the mean time to failure (MTTF) for the overall machine. This shortened MTTF implies that middleware development on large-scale computers should take failure as a norm rather than an exception. Many existing production or research software on large-scale computers use techniques such as checkpointing and restarting. Some systems implement resilience using spatial replication, such as HDFS [13]. Some systems implement resilience with temporal replication, such as Spark [80]. The AM-FORA framework should also take resilience into account, to accommodate the increasing scale of hardware. At the time this dissertation is being written, I and my collaborators are working on a resilience solution in the context of parallel scripting that combines spatial and temporal replication.

We have seen storage techniques change rapidly over the past several years. Non-volatile
RAM has entered computer users’ daily life as well as the storage system (both temporary and persistent) of large-scale computers. The Gordon cluster [36, 52] has installed NV-RAM on its I/O nodes as temporary storage for in-situ data analysis. IBM’s Active Storage [29] also exploits NV-RAM to provide fast data analysis on large-scale supercomputers. These improvements imply that the computation should be closer to data than ever before for performance, efficiency, power conservation, and many other reasons. The AMFORA File System perfectly fits a computer cluster with NV-RAM as its storage media, as it maximizes the I/O performance provided by NV-RAM by localizing the writes and distributes metadata operations to all computing resources.

The NV-RAM technique may replace both the RAM and rotating disk we are seeing today. And systems might have a flat storage layer, which is persistent. Middleware such as AMFORA needs to continue to evolve to address such changes, for example by integrating techniques such as byte addressable file systems or other new methods that can not be foreseen today.
REFERENCES


