University Politehnica of Bucharest

Bachelor Thesis

Benchmarking Distributed Filesystems for Many-Task Computing

Author:
Mihnea Dobrescu-Balaur

Supervisors:
Dr.-Ing. habil. Thilo Kielmann
Prof. Dr.-Ing. Nicolae Tapus
M.Sc. Alexandru Uta

A thesis submitted in fulfilment of the requirements
for the degree of Bachelor of Computer Science

in the

Faculty of Automatic Control and Computers

June 2014

Declaration of Authorship

I, Mihnea Dobrescu-Balaur, declare that this thesis titled, 'Benchmarking Distributed Filesystems for Many-Task Computing' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Testing can be used to show the presence of bugs, but never to show their absence!"

Edsger W. Dijkstra

UNIVERSITY POLITEHNICA OF BUCHAREST

Abstract

Faculty of Automatic Control and Computers

Bachelor of Computer Science

Benchmarking Distributed Filesystems for Many-Task Computing

by Mihnea Dobrescu-Balaur

The Many-Task Computing (MTC) paradigm is used throughout scientific applications today. One of its technical particularities is the use of shared files that compute nodes use to process results between successive steps of computation. This favors the use of shared, distributed filesystems. Development of such filesystems is an open problem, and proper benchmarks are key to making progress. The MTC Envelope, a microbenchmark for MTC workloads, specifies what metrics are relevant for MTC filesystems. However, current filesystem benchmarks are targeted toward single-node usage. While this behavior is fine for regular filesystems, the relevant MTC-specific measurements include test cases such as N-to-1 reads, where all nodes part of a computation try to read a file that is stored on a single node. In order to address this shortcoming, we have introduced a new distributed benchmark that supports distributed MTC filesystems natively. Our solution builds upon state-of-the-art single-node filesystem benchmarks and runs them in a coordinated manner in order to compute MTC-specific metrics. By using a simple JSON file, the user can specify complex test-cases. Our benchmark handles synchronization and results aggregation, providing structured, machine-readable metrics after test runs.

Acknowledgements

I would like to express my deep gratitude to my research supervisors, Professor Thilo Kielmann and Professor Nicolae Tapus, for their patient guidance, encouragement and constructive feedback. Special thanks go to Alexandru Uta, my daily supervisor, for his incredible support as well as for his great vision throughout this work. Last but not least, I would like to thank my family who, in many ways, made this work possible and to whom this work is dedicated.

Contents

List of Figures

Abbreviations

MTC	**M**any **T**ask **C**omputing
JSON	**J**ava **S**cript **O**bject **N**otation
MBps	**M**ega **B**ytes **p**er **s**econd
GBps	**G**iga **B**ytes **p**er **s**econd
RAM	**R**andom **A**ccess **M**emory
MPI	**M**essage **P**assing **I**nterface
I/O	**I**nput **O**utput
ANSI	**A**merican **N**ational **S**tandards **I**nstitute
API	**A**pplication **P**rogramming **I**nterface
POSIX	**P**ortable **O**perating **S**ystem **I**nterface
CPU	**C**entral **P**rocessing **U**nit
SSH	**S**ecure **SH**ell

To my Mother, Father and Sister...

Chapter 1

Introduction

Many-Task Computing (MTC)[1] is a new computing paradigm that attempts to place itself in-between High-Throughput Computing (HTC) and High-Performance Computing (HPC). The core idea of MTC is that it uses large amounts of computing resources over short periods of time, in order to complete multiple tasks. While a common metric in HTC is the number of tasks fulfilled per month, MTC is usually quantified in tasks per second. Also, as opposed to other paradigms that use message passing as a way to communicate between successive steps in a computation, MTC couples its compute phases via files. Because of this, another common metric of MTC is megabytes per second (MB/s).

Any computing task that is comprised of multiple small parallel jobs is a good fit for MTC. Multiple applications have been found good fits for MTC[1], spanning domains like astronomy, astrophysics, economic modelling, pharmaceutics, chemistry, bioinformatics, neuroscience, data analytics, data mining and biometrics.

MTC exhibits a specific set of technical particularities which need to be properly supported in order to obtain good performance. The high number of parallel tasks that need to be executed underlines the importance of a streamlined scheduling system. Besides that, the way the tasks use files to communicate makes the overall performance largely dependant on the underlying filesystem performance. For the purpose of our subject, we will focus on the filesystem aspect.

Figure 1.1 shows an example workflow of Montage[2], a software toolkit for astronomical image mosaicking. It shows the outline of a MTC workload, as well as the specific access patterns that emerge, such as data partitioning and aggregation.

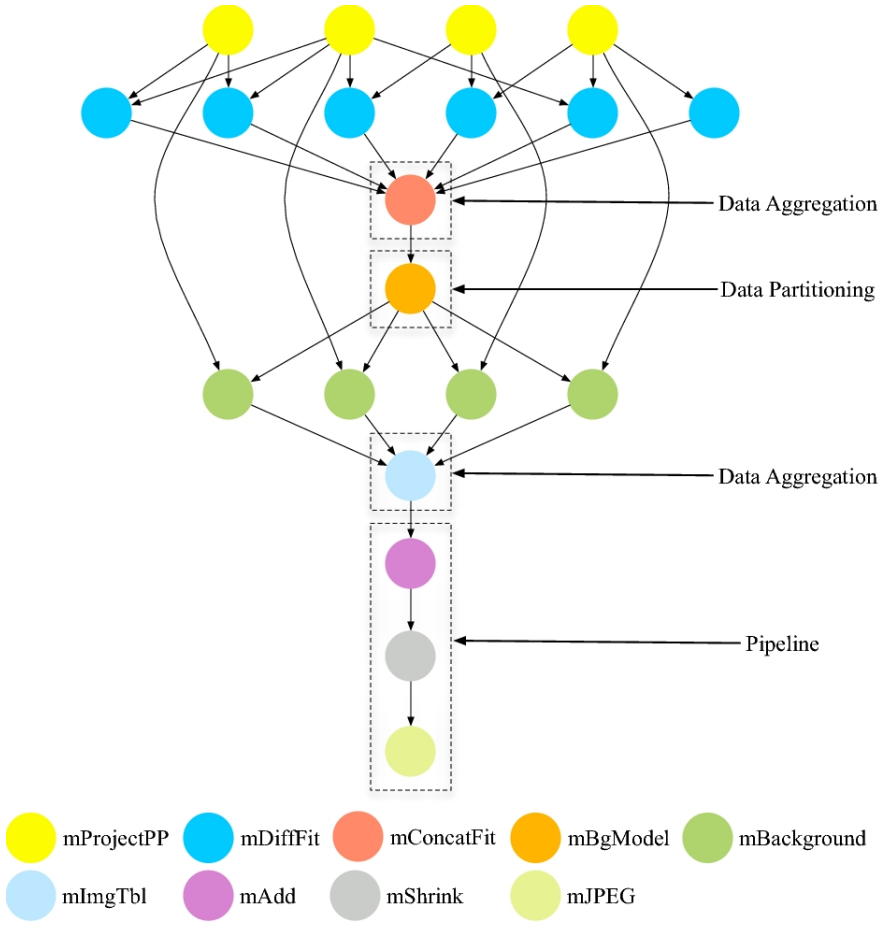

FIGURE 1.1: Montage workflow

Typically, MTC jobs are run on clusters of compute nodes. These compute nodes need to share data files between them in order to complete the tasks that are assigned to them. Usually, files are locally stored on disk drives (either mechanical disk drives or solid state drives), and are sent over the network when a different node needs access. However, even with the fastest drives commercially available today, the transfer rates of disk drives are inferior to the bandwidth of InfiniBand connections that are used in clusters, considering the fact that a SATA link tops out at 3 GBps while InfiniBand 12X supports a 30 GBps data link. This shows that the bottleneck is not the network, but the storage technology. To overcome this bottleneck, RAM can be used as a storage medium. Obviously it cannot achieve the same storage capacity as regular disk drives, but that is not a problem for the purposes of MTC, as the data files used are not so large in size.

In order to make use of the high speed RAM and network connections that compute nodes in today's clusters have, research groups have started work on developing distributed, in-memory filesystems. These filesystems are specifically designed to support

MTC workloads and yield better results than common network filesystems like NFS.

To illustrate how such filesystems work, we will describe two implementations - AMFS[3] and MemFS[4].

1.1 MemFS

MemFS[4] is a distributed, in-memory filesystem for MTC applications. Its motivating idea is that data locality is hard to preserve in MTC applications, and so a filesystem that is locality-agnostic could take advantage of the low-latency, high-bandwidth network connections that we have today and provide good performance for MTC applications. It has three main design goals:

1. maximize read and write throughput/bandwidth trying to achieve line-speed
2. achieve close-to-linear I/O operations scalability when increasing the number of compute nodes
3. reduce latency by storing data in memory and by improving the metadata protocol

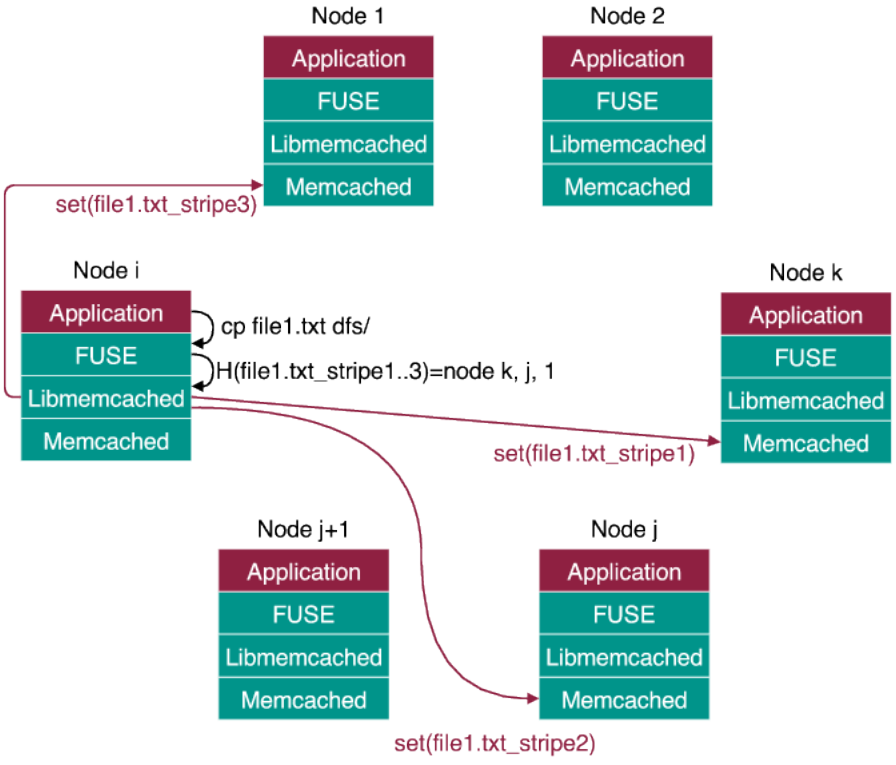

FIGURE 1.2: MemFS design

Its three main components are Memcached[5], a distributed cache system that is used in the industry, libmemcached[6], a client for interfacing with the Memcached server(s) and a FUSE[7] filesystem that exposes the data storage mechanism to the MTC applications that run on top of MemFS.

1.2 AMFS

AMFS[3] is another distributed filesystem for MTC applications. In contrast to MemFS, it also includes an execution engine. This execution engine exposes a shell programming environment which application developers have to use to access distributed programming features, such as distributing the files in a folder across multiple nodes.

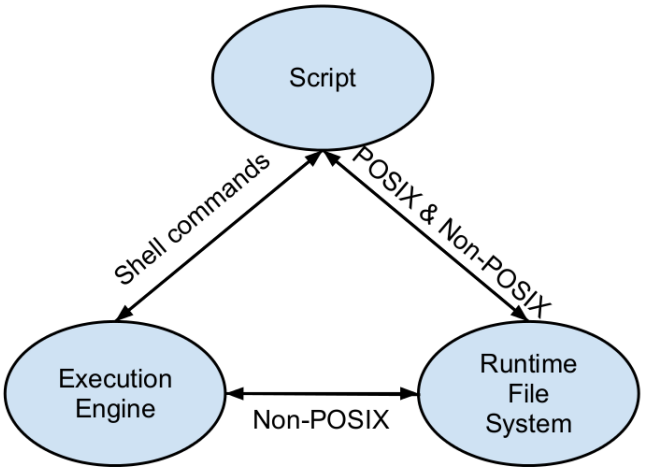

FIGURE 1.3: AMFS design

Another particularity of AMFS is that it tries to exploit data-locality. Its execution engine will try to run tasks on the nodes that already have the needed files, and only if this is not possible will it copy the needed files over to another node.

AMFS also stores file information in memory, but it decouples the medatada from the actual file blobs. Thus, it contains a metadata engine that keeps track of file properties and where the actual binary data is stored within the nodes.

1.3 Benchmarking MTC filesystems

Besides the different storage medium (RAM), MTC-specific filesystems also target different access patterns. First, there is the usual 1-to-1 read pattern, where a node needs

access to one file. In MTC workloads, it is probable that the file will reside on a different node's local storage. Second, there is the more complicated N-to-1 read pattern, where multiple nodes need to access the exact same file that is stored on a single node's local storage. In order for the MTC workload to achieve maximum performance, the underlying filesystem should make the most out of the network bandwidth, without creating bottlenecks because of contention.

In order to properly benchmark these filesystems, we need a tool that can simulate the MTC specific access patterns. This tool should be able to support a variable number of nodes, such that the scalability of the filesystem can be measured. Furthermore, the benchmark should be able to coordinate the nodes in a deterministic and synchronized manner in order to correctly reproduce access patterns such as N-to-1 reads. Finally, the results aggregation step should be built in, to facilitate tracking performance improvements between different test runs. Our solution provides all these features and, in addition, it uses a simple JSON[8] file to specify complex test cases. We will detail all features later, in this study.

The rest of this study is structured as follows: Chapter 2 provides more context on benchmarking distributed MTC filesystems and related work. In Chapter 3 we discuss our implementation, as well as the design decisions we have made. Then, in Chapter 4 we evaluate the performance of our solution. Finally, Chapter 5 presents our conclusions.

Chapter 2

Background and Related Work

As we outlined in the Introduction, distributed MTC filesystems are inherently different when compared to usual filesystems. Thus, they have to be benchmarked differently. The relevant set of metrics that needs to be measured is defined in the MTC Envelope[9]. It includes the following:

- 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
- write data bandwidth

In the above list, we can observe two special access patterns, specific to MTC filesystems: 1-to-1 and N-to-1 reads.

The 1-to-1 access pattern can be described as follows. There is created a number N of files. Then, the M nodes each read N/M of the files. Thus, every node has to do with its own set of files and there is no concurrent access over any one file.

The N-to-1 access pattern is a bit more involved. There is only one file created, and then all M nodes read it, concurrently. This pattern generates contention and is more penalizing on the performance of the filesystem.

2.1 Why current benchmarks are not enough

Overall, there are already a significant number of filesystem benchmarks available[10]. However, none of them is a good fit on its own for benchmarking MTC filesystems. We outline a few sample reasons.

Most filesystem benchmarks target disk-based filesystems. This does not fit MTC filesystems because they use RAM as datastore, for performance reasons. The differences between RAM and disks make regular benchmarks unreliable. For example, memory is random access, while disk drives favor sequential access. The difference in access patterns yields less relevant results overall.

Generally, filesystem benchmarks are local - they run on a single node, just like the filesystem implementations they commonly test. This means that in order to run a benchmark on a distributed filesystem, one has to copy the executable on all nodes, run it simultaneously and then aggregate all the results. This approach is tedious, requiring a lot of extra work that is unrelated to the research problem. Furthermore, this way of running tests is not necessarily correct. For example, if an N-to-1 pattern is tested, the N readers should access the file all at the same time. To obtain this, a global synchronization mehtod, such a barrier, should be used. This synchronization cannot be reached by just using a shell script. This means that different test runs will output different results.

However, we can build upon such existing benchmarks. For our solution, we chose to use *IOzone*[11] and *mdtest*[12].

2.2 IOzone

IOzone is a benchmark for measuring I/O performance. It can generate multiple types of file operations, as well as measure their performance. The main operations that are useful for our purposes are:

- read
- write
- re-read
- re-write

Other operations such as random read and random write are supported as well, but they are not relevant for memory-based filesystems.

A big advantage of IOzone is that it is written in ANSI C, using POSIX APIs. Because of this, IOzone has been ported to run on multiple computer platforms. It also supports large files in its test runs, which helps in simulating MTC-sized data files.

Another useful feature of IOzone that made embedding it into our benchmark easier is the fact that it can output its results in tab-separated columns, making the results easy to parse.

Below are a subset of the configuration options IOzone supports.

- cache line size
- total cache size
- record size
- file size
- including the *close* operation in the timing
- including flush operations (*fsync*, *fflush*) in the timing
- performing synchronized writes
- give results in operations per second
- purging the CPU cache before each operation

2.3 mdtest

While IOzone is a great tool for benchmarking the throughput of reads and writes, we turned to mdtest for benchmarking metadata operations. *mdtest*[12] is a filesystem metadata benchmark that works as follows - first, it creates a file structure (directories and trees) and then it measures performance of operations such as *open*, *stat* and *close* on that file structure.

mdtest is highly configurable, supporting options like:

- branching factor of the created file tree
- number of iterations

- number of files created within a directory
- restrict to only measuring one operation (i.e. read)

On top of these, mdtest is already a distributed benchmark through the use of MPI. This means that we can use mdtest out of the box to benchmark metadata operations on MTC filesystems. However, there is still room for improvement, especially in the results aggregation and plotting. The benchmark's results are not presented in a machine-readable format, and that means there is no easy way to feed them into a plotting tool. To overcome this, one can use a parser that transforms mdtest output into structured data, like a JSON[8] file.

Chapter 3

A Benchmark for MTC Filesystems

In the previous two chapters we have discussed about Many-Task Computing, how it is relevant for today's computations and why it needs special filesystems. We looked at how those filesystems differ from regular disk-based ones, as well as why they need to be benchmarked differently. We also outlined the shortcomings of current benchmarks and how they can be surpassed. In this chapter, we focus on our approach to building an MTC-specific filesystem benchmark, the architecture behind it and the ways in which it can be used.

3.1 Running coordinated commands across multiple nodes

As we previously explained, in order to correctly benchmark access patterns on distributed filesystems, all nodes have to start running any given operation simultaneously. To solve this, we need to have a way of synchronizing nodes. Fortunately, synchronization is a common problem with multiple solutions in distributed programming frameworks.

To solve this problem, we used OpenMPI[13] barriers. We developed an MPI application that executes any given command on a set of available nodes. To do so, we spawn MPI workers in the default shared MPI communicator, we then synchronize them with the use of a barrier, and then the workers run the given command. By using the barrier, we can be sure that the command is run in a coordinated manner, at the same time.

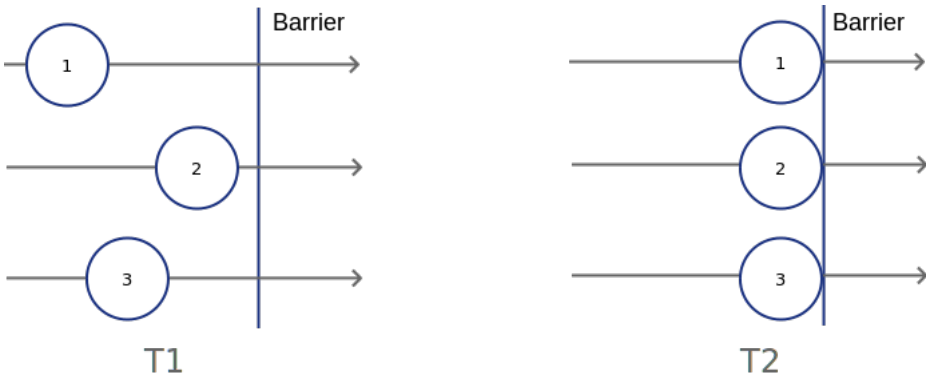

FIGURE 3.1: Synchronizing processes with a barrier

3.2 Why OpenMPI

For our solution, we chose to use the MPI framework and, specifically, the OpenMPI[13] implementation. In this subsection we argument why.

Message Passing Interface (MPI) is a standardized communication protocol that is used to implement parallel computations. It defines an API that supports both collective (broadcast, scatter, gather) and point-to-point (send, receive) operations. Our solution makes use of both types of operations. We use a barrier to synchronize the nodes before running the core of the benchmark, and then we use point-to-point operations to send back the output from individual nodes to the master node, which aggregates the results. From our point of view, using MPI has two big advantages.

First, there are implementations available for most of the existing computer architectures existent today[14]. This means that by using standard POSIX and MPI in a program, it can run with no extra porting work on multiple architectures. Since we want our benchmark to be accessible to everybody, portability is a great concern. To achieve this, we made sure to only use standard APIs and to not rely on any implementation-specific behavior in our code.

Second, MPI is built to scale natively. This means that we can write the code for a distributed application once and then run it on any number of nodes without having to change anything. The way this works at the API level is fairly simple - all instances are together in a container (called communicator) and within that container they all have a rank (which is an integer identifier).

Because MPI was implemented for a multitude of different architectures, it is generally supported on computer clusters. We developed our solution on the DAS4[15] cluster which has support for multiple MPI implementations:

- OpenMPI
- MPICH
- MPICH2
- Intel MPI

  We decided to use OpenMPI mainly because of its open source and vendor-independent nature. However, if OpenMPI is not available for some users of our benchmark, they can use any other implementation without problems, as our code relies only on standard behavior.

3.3 Aggregating the outputs of multiple nodes

Having an MPI program that runs any given command across multiple nodes, we need a way to capture the outputs and then aggregate them. This is needed because benchmarks like IOzone and mdtest display the performance statistics on the standard output (stdout). Our distributed program has every slave node capture its own output then send it back to the master node, which in turn aggregates all outputs and formats them accordingly.

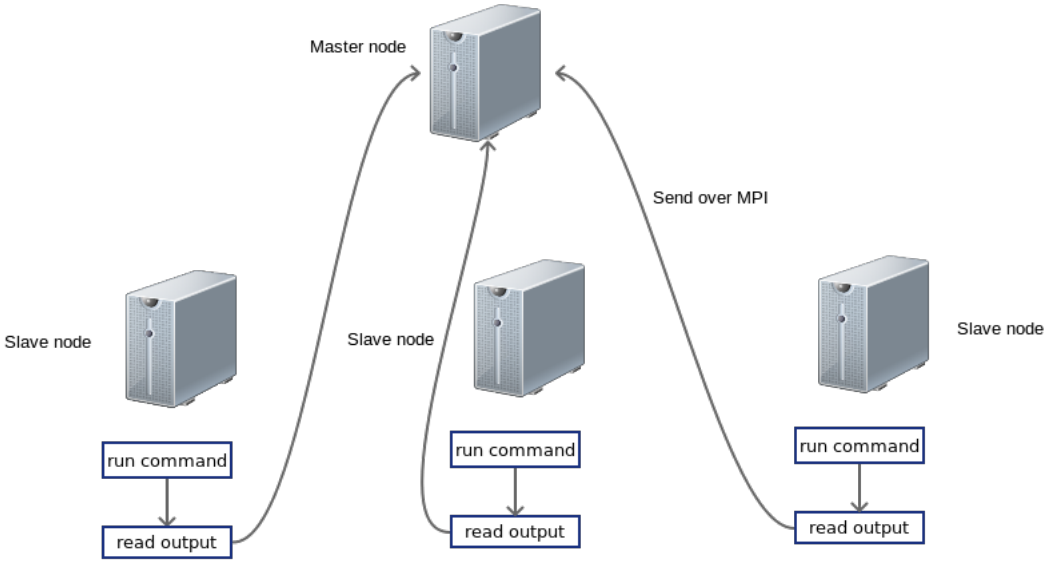

FIGURE 3.2: Slave nodes sending output to master node

To solve this problem, our initial solution was to use the *popen*[16] system call. This runs a given command and captures its output. Then, that output can be sent over MPI to the master node for aggregation. While this approach worked when testing locally using threads, when we tested across distributed nodes using the network stack for communication, the process failed. We discovered that this is due to the fact that *popen* uses *fork* and pipes, which MPI does not fully support.

Since using *popen* was not possible, we found that we could run commands with *system*[17]. However, *system* does not provide a way to capture output, so we had to overcome this new limitation. We did so by making use of output redirection. This means that when we call an external command, we redirect its output to a temporary file that is local to the node, read the output from there and then delete that file.

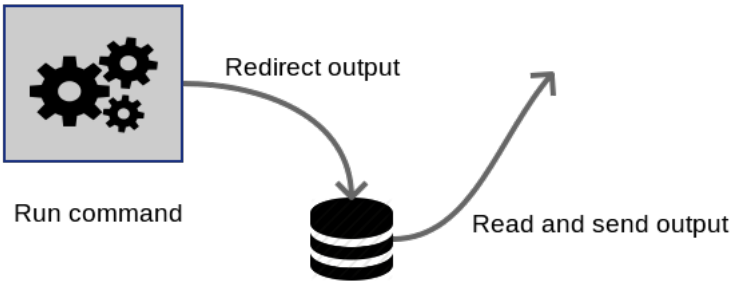

FIGURE 3.3: Local execution on the slave node

3.4 Varying the number of nodes

An important aspect of distributed software of any kind is scalability - what is the relation between performance and number of nodes? Naturally, this is relevant for MTC filesystems as well. To measure this, we need to run the same benchmark on different numbers of nodes and then compare the results.

Since this is a given in MTC filesystem benchmarking, our solution handles this by default. The user can configure on what series of node numbers they want the test to run, and the benchmark will measure scalability automatically. To achieve this, we re-run the benchmark with more and more of the available nodes and keep track of the results.

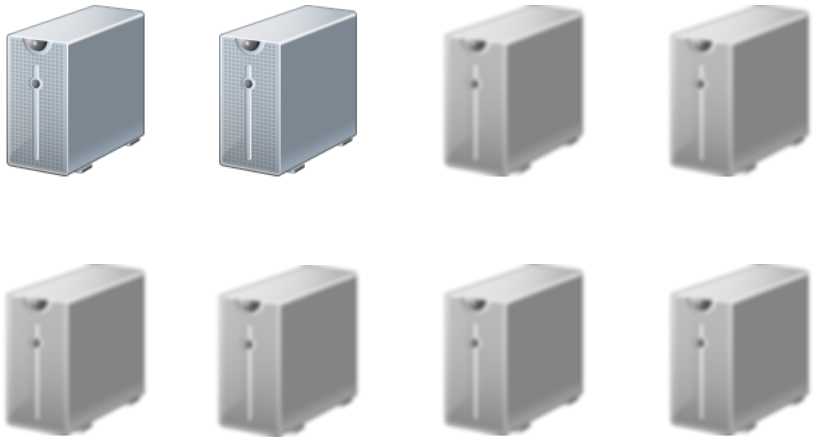

FIGURE 3.4: Varying the number of nodes

The implementation consists of a Python wrapper over the MPI code mentioned in Section 3.1. It iterates over an array of integers representing node counts and runs the benchmark with the given amount of nodes. Because it uses the MPI code, it needs to tell MPI exactly what nodes to use. This is done using a machinefile - a text file that contains node IPs, one per line. The MPI program always uses the same machinefile, and the Python wrapper updates it between runs as necessary, increasing the number of nodes.

Because clusters usually have a reservation process that a user has to go through before having access to a set of nodes, we added support for that to our solution as well. For this, we provide a *reserve.sh* script that takes a number of nodes as argument and reserves that many nodes on the cluster. It also checks if the user already has reserved nodes and if so, it reuses those. The Python wrapper uses this script to reserve the maximum amount of nodes needed at the beginning of the run, and then uses them increasingly as the tests advance. So, if the user specified the array *[1, 2, 4, 8, 16]* for the amounts of test nodes, the benchmark would reserve 16 nodes from the start and then run the tests with 1, 2, 4 (and so on) nodes.

The reason for which we kept the node reservation process in a separate script is that other clusters might require different commands to reserve nodes. We want our solution to be accessible to as many users as possible, and running on a different cluster shouldn't imply changing the benchmark's code. This way, the only necessary change is to adapt the *reserve.sh* script.

In order to properly support multiple nodes in tests, there also needs to be a way to parametrize the given commands by nodes. For example, maybe the user wants node 3 to run IOzone against `test_file_3`. To achieve this, our tool sets an environment

variable to the ID of every node, such that examining `${NODENAME}` on node 3 will yield `3`.

3.5 Leveraging existing benchmarks

As discussed in Chapter 2, our solution builds on top of existing benchmarks, adding only what is lacking in order to provide a complete MTC filesystem benchmark. Thus, we can decompose it into 5 distinct components:

- IOzone (3rd party benchmark)
- mdtest (3rd party benchmark)
- the MPI coordination program
- a set of parsers for IOzone and mdtest ouputs
- a set of plotters for IOzone and mdtest results

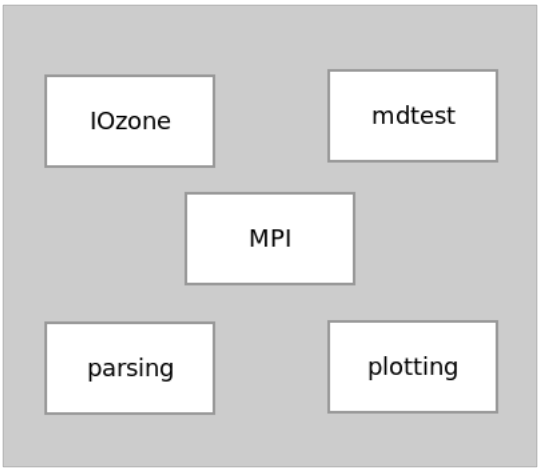

FIGURE 3.5: Components

Having already discussed about IOzone and mdtest in Chapter 2, this section explains how our solution makes use of them, as well as how it processes their results, from unstructured blobs of output to structured data, and then to plots.

3.5.1 IOzone

IOzone's main use is for measuring performance on reads and writes. Our project uses it for the read and write throughput measuring that is specified by the MTC Envelope.

For the 1-to-1 test cases, we let IOzone create the test files. However, for the N-to-1 test cases, we create a file using *dd* before running IOzone, and then (for N-to-1 reads) we have all IOzone processes test the read/re-read performance on that particular file.

Out of the variety of parameters IOzone supports, we only used the following:

- cache size
- cache line size
- record size
- mode operation (read)
- create/do not create files
- test file size

3.5.2 mdtest

While we use IOzone for data-intensive operations (such as reads and writes), our solution uses mdtest for metadata measurements, like file creation and stat. Because mdtest already supports running on multiple nodes via MPI, we did not have to use our own MPI coordination layer. The only additions were for parsing and plotting, which will be discussed later on.

Out of the variety of parameters mdtest supports, we only used the following:

- number of iterations
- number of directories and file per process
- shared filesystem mode

3.5.3 Producing structured results

By default, both IOzone and mdtest produce unstructured, human-readable output. In order to compare performance across different numbers of nodes and plot the numbers automatically, there needs to be a conversion of the output to machine readable (or structured) format.

With the way our distributed benchmark is set up, every node produces its own output and then sends it back to the master node. Thus, it is natural for the conversion of the output to happen on the master node as well.

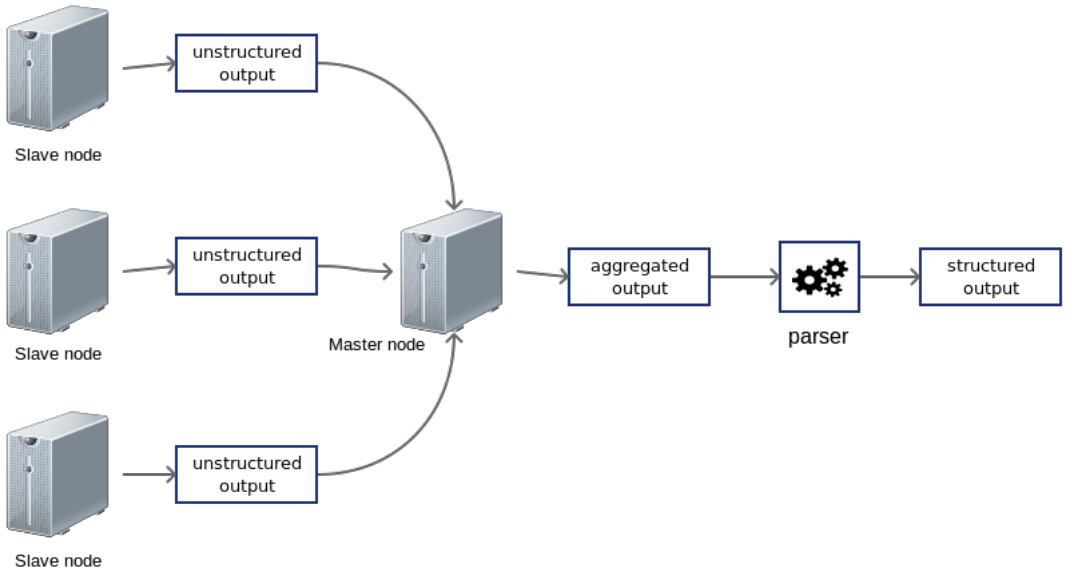

FIGURE 3.6: Flow of output through the system

As such, the master node runs a parser (which one depends on what benchmark was run - IOzone or mdtest) to process the test results for every individual node. It then aggregates them, so it produces numbers such as the total throughput of the filesystem. After that, it groups the results by number of nodes in the test run. Finally, the master node saves the processed results in a JSON file.

The current implementation contains two parsers - one for IOzone and one for mdtest. Our project can be extended by adding a new benchmark to it. Doing this will only require implementing a new parser, along with the code for plotting the graphs (more about this in the next section).

3.5.4 Plotting the results

Having the test results in structured form, one can process them in multiple ways. One of the more common and visual ways that are used to process benchmark results is plotting. Making plots is useful because they are easier to reason about than plain, raw data, and differences in performance can be spotted immediately.

The plotting logic is decoupled from the main benchmarking code, and it resides in its own Python script. This way, one can make changes to the plotting code in order to obtain different graphics without having to tackle the benchmarking code. Furthermore, the plots can be generated on a different (maybe personal) computer, without having to use the non-graphical SSH connection.

The current implementation contains plotting code both for IOzone and mdtest. It generates a bar plot for each of the metric that is benchmarked, with the results being a function of the number of nodes. We will show a set of plots in Chapter 4.

3.6 Specifying complex test cases

Most metrics from the MTC Envelope can be benchmarked with just one IOzone or mdtest command. For example, to measure 1-to-1 read performance, one can run:

```
./dfs_bench.py './iozone -f /var/scratch/mdr222/${NODENAME}_test -S 12000 -L 64 -c -e
    -s 1M -i0 -i1 -r 128 -R'
```

Listing 3.1: Sample benchmark command

However, simulating an N-to-1 access pattern requires two commands - one to create the file that every node will read, and one to actually run the benchmark. More complicated test cases might require even more steps. Thus, a proper benchmarking tool should have support for specifying complex test cases like this:

1. Create a test file of size 100MB (single node)
2. Move it to the mounted shared directory (single node)
3. Run IOzone (all nodes, yields results)
4. Remove created test files (single node)

This raises multiple challenges. First, there should be a way to run some commands only on one node (and it should deterministically be the same for all steps). Then,

by running multiple commands, it becomes ambiguous which of the commands' output should be captured and processed.

Our project comes with a solution for this problem. It supports using a JSON file that describes a test case. The file contains an array of commands, and every command can have attributes describing if it should be run on a single or all nodes, as well as if its output should be processed or not. We will exemplify such a test file in Chapter 4.

3.7 Putting it all together

Now that we have examined the main components of our project, it is a good moment to look at them in ensemble.

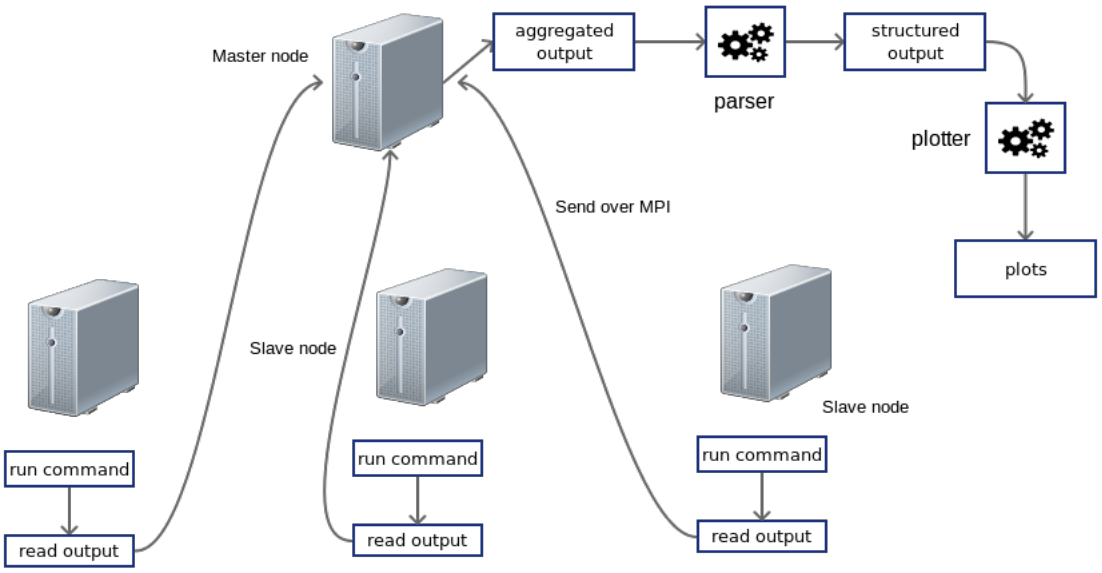

FIGURE 3.7: General architecture

At the core, there is an MPI program (called `run_mpi`) that runs a given command across multiple nodes and then outputs to *stdout* all the nodes' individual outputs. On top of that, there is a Python wrapper (called `dfs_bench.py`) that runs the whole benchmark. It uses two extra components - the node reservation script (`reserve.sh`) and the output parsers. Having all the components together, it runs the following steps in a loop, by the number of nodes in a test:

1. run the command(s)
   (a) if using a JSON file to specify, run each command
   (b) if it is a "single node" command, run it on the first node
   (c) otherwise, run it on all nodes through run_mpi
2. get the output
3. parse the output
4. store the output in the results set, under the "node_count" index

After these steps, the results set is stored in JSON format in a file. That file can then be passed to the *make_bar_plots.py* file in order to create the corresponding plots.

Chapter 4

Evaluation

To evaluate our solution, we follow two directions. First, we test the fact that our benchmark supports MTC filesystems and that it can measure the metrics described by the MTC Envelope. Second, we observe whether our project is easier to use than the current benchmarking solutions for MTC filesystems.

4.1 Functionality

To assess the first part, we ran the following benchmarks on the two discussed MTC filesystems - MemFS and AMFS.

- 1-to-1 read
- N-to-1 read
- mdtest suite

All the tests were run on the following numbers of nodes: `[1, 2, 4, 8, 16, 32]`. The benchmarks were run on the DAS-4 cluster, using OpenMPI.

The N-to-1 access pattern was simulated using a test file such as the following:

```
{
    "commands": [
        {
            "type": "single",
            "command": "dd if=/dev/zero of=/local/aua400/kvstore/fisier_tst bs=1M
    count=256"
```

```
        },
        {
            "type": "all",
            "command": "./iozone -+E -f /local/aua400/kvstore/fisier_tst -S 12000 -L
    64 -c -e -s 256M -i1 -r 128 -R",
            "parse": true
        }
    ]
}
```

LISTING 4.1: Sample test file

Having this file stored as *test.json*, we can run the benchmark with:

```
./dfs_bench.py --file test.json
```

The full set of results can be found in the Appendix. However, we include here a sample results file and a plot to show the outputs of our benchmark.

The following listing represents a part of a results file of an IOzone-based benchmark, testing N-to-1 read. It depicts the processed result set for the test run with 2 nodes.

```
{
  "2": {
    "total": {
      "re-reader": 1433371,
      "reader": 1345494
    },
    "individual": [
      {
        "re-reader": 728699,
        "reader": 721448
      },
      {
        "re-reader": 704672,
        "reader": 624046
      }
    ]
  }
}
```

Based on a processed set of results, the *make_bar_plots.py* script creates a plot like the following.

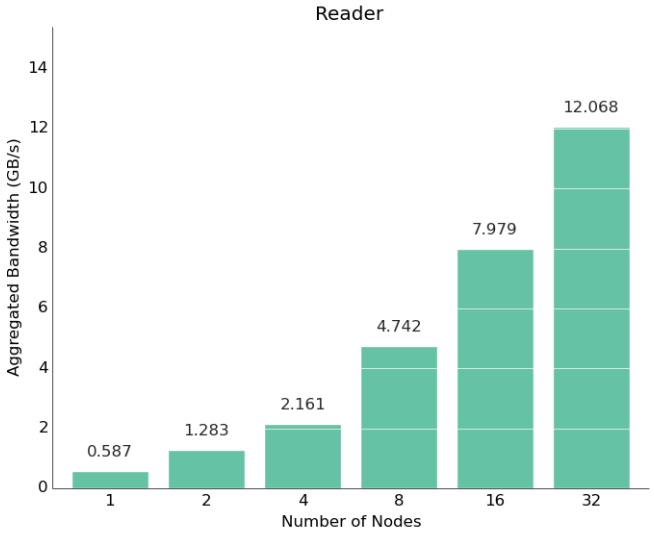

FIGURE 4.1: Generated sample plot

4.2 Maintainability and ease of use

The current way of benchmarking an MTC filesystem is to write a number of scripts, then use SSH to distribute them across the nodes. Finally, after the scripts are in place, there needs to be another script that iterates through all the nodes and triggers the benchmarks as close to simultaneously as possible over SSH. This requires a considerate amount of work and there is no true coordination in place, as there is no synchronization method (like the barrier in our solution) to enforce the same start time for the benchmark across all nodes. If at some later time a new test case has to be run, all scripts have to be modified to account for the change.

In contrast, using our project to run the same set of benchmarks on varying numbers of total nodes, there is only one command to execute, as we have shown. Furthermore, coordination is built-in, thanks to the MPI runner, so we can be certain that all nodes start at the same time. If at some later time a new test case has to be run, it is only a matter of specifying it using our JSON format and the benchmark will keep working. Aside from just running the tests, our solution also includes processing the outputs into machine-readable form, as well as plotting the results.

Chapter 5

Conclusions and Future Work

We have seen how today's scientific computations fit the Many-Task Computing paradigm. Researchers in various fields, from astronomy to neuroscience, are using MTC applications to do their work more efficiently. Providing better support for MTC, we enable those researchers to work with better performing tools, and that translates to more accomplishments in research.

To achieve this, we need to look at the core of MTC. There are two main subsystems that it relies on. One is the task scheduler, the other is the filesystem used to share task results. By improving them, we can improve MTC performance no matter what field it is used in.

Our project targets the filesystem layer. While there are numerous ongoing projects to implement MTC specific filesystems, such as MemFS[4] and AMFS[3], there is no benchmarking tool tailored to the specific needs of these filesystems.

To properly support an MTC filesystem, a benchmark should support running on multiple nodes at the same time. These nodes should be synchronized such that all operations can be run at the same time. It should also scale easily - testing a filesystem on 16 nodes should not be different than testing a filesystem on 256 nodes. Furthermore, the benchmark should provide a tight feedback loop to the user, allowing them to easily observe the differences in performance between successive runs. This means that the user should not have to worry about aggregating the results and plotting them. All these features, and possibly others, should be built in to the MTC filesystem benchmark.

We have shown that our solution fits all the above mentioned requirements. The test commands are executed through MPI, which handles coordination and scaling in a standard, documented way. Our Python-based test runner handles processing the output as well as plotting it. Besides, it also supports specifying complex test cases. By writing a

list of steps to be executed in a file, the user can outline the behavior of the nodes during a benchmark, having control both on the collective level, as well as on the individual level.

By using IOzone and mdtest, our project measures all the performance metrics described in the MTC Envelope[9]. We use IOzone to measure data-intensive operations like *read*, while mdtest is used to measure file metadata operations such as *stat* and *create*.

Although our project has all the mentioned features, it is built with the goal of making people working on MTC filesystems more efficient. That means that it should not get in the way and that it should be easy to run. To achieve this, we have made it so that it only needs one command to run a given benchmark, and then just one command to generate the corresponding plots. This saves a considerable amount of time, compared to the current state of the art in distributed filesystem benchmarking, which involves writing one-off scripts, copying them to the nodes over SSH and then finding a way to run them simultaneously. This is not even including the aggregation and plotting parts.

We consider that our solution is ready to be used by MTC filesystem developers, and that it should start improving their productivity right away. However, there is always room for improvement. As such, we see two directions for future work. One is adding support for benchmarking other metrics. Maybe a research group has developed their own metric that is relevant for MTC filesystems and it is not included in the MTC Envelope. We designed our project to be modular, so including other benchmarks is definitely an options. The other direction involves setting up a continuous integration workflow, where one can have a benchmark server that runs a set of benchmarks every day, and keeps historic records of the performance evolution. This data could be presented in an interactive set of plots online, accessible to all members of a team. This way, using the details from the version control system and the data from the benchmarks, research groups could compare the impact of different design decisions on the filesystem's performance, leading to better implementations.

Appendix A

Example Test Files and Commands

```
{
    "commands": [
        {
            "type": "all",
            "command": "./iozone -f /var/scratch/mdr222/${NODENAME}_test -S 12000 -L
    64 -c -e -s 1M -i0 -i1 -r 128 -R",
            "parse": true
        }
    ]
}
```

Listing A.1: Test file for 1 to 1 read

```
{
    "commands": [
        {
            "type": "single",
            "command": "dd if=/dev/zero of=/local/aua400/kvstore/fisier_tst bs=1M
    count=256"
        },
        {
            "type": "all",
            "command": "./iozone -+E -f /local/aua400/kvstore/fisier_tst -S 12000 -L
    64 -c -e -s 256M -i1 -r 128 -R",
            "parse": true
        }
    ]
}
```

Listing A.2: Test file for N to 1 read

```
./dfs_bench.py --file test_example.json
```

Listing A.3: Running benchmark from a file

```
./dfs_bench.py './iozone -f /var/scratch/mdr222/${NODENAME}_test -S 12000 -L 64 -c -e
    -s 1M -i0 -i1 -r 128 -R'
```

Listing A.4: Running 1 to 1 read benchmark directly

```
./run_mdtest.py 'mdtest -C -i 10 -n 100 -S -d /local/aua400/kvstore'
```

Listing A.5: Running mdtest

Appendix B

Example Plots

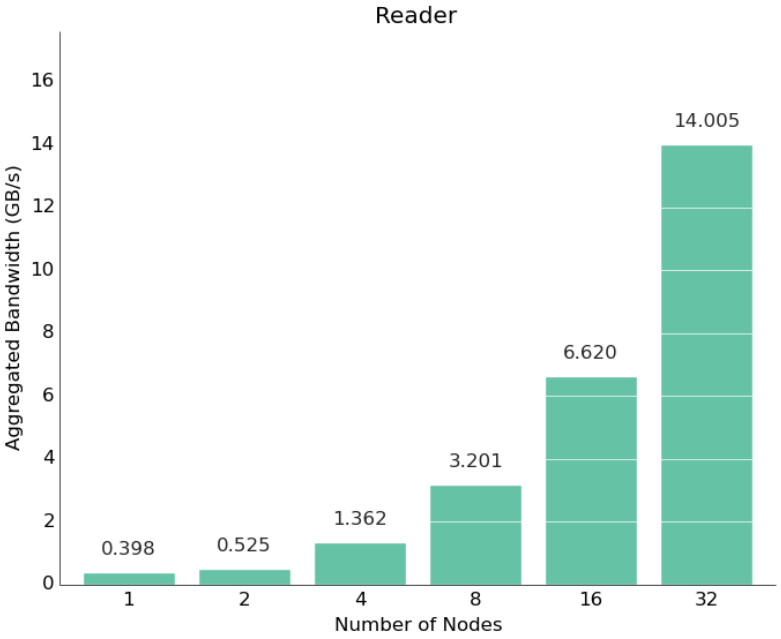

FIGURE B.1: AMFS, N-to-1 read

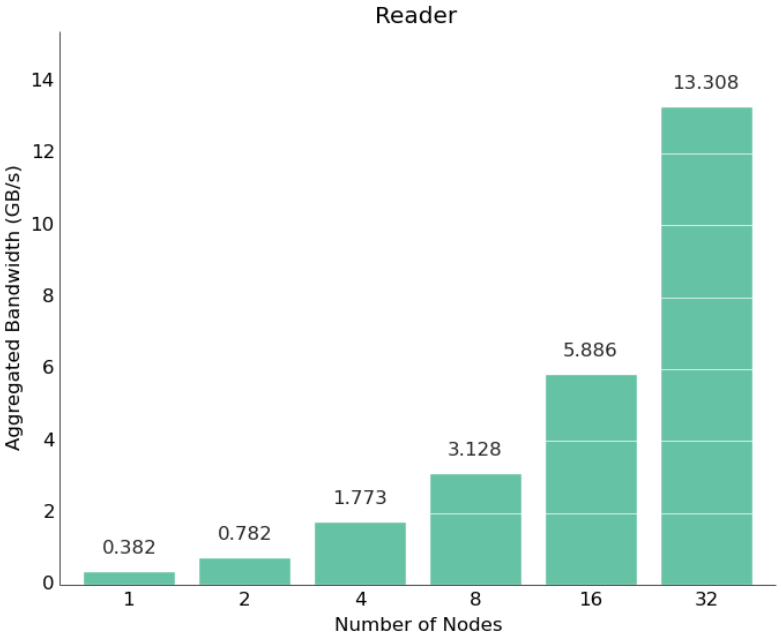

FIGURE B.2: AMFS, 1-to-1 read

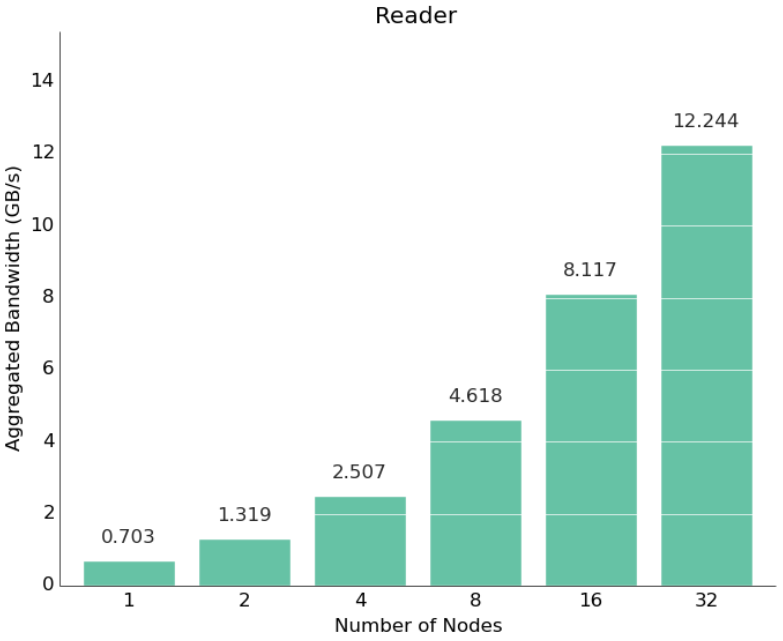

FIGURE B.3: MemFS, N-to-1 read

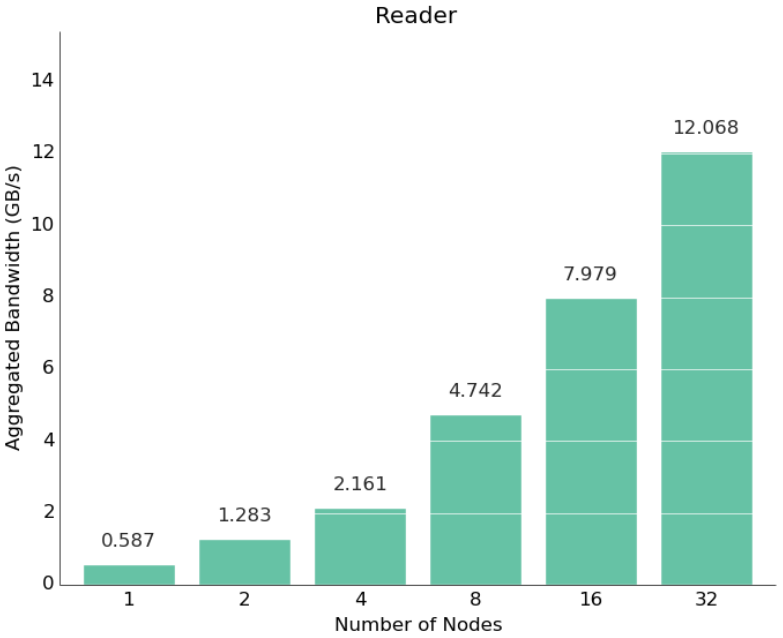

FIGURE B.4: MemFS, 1-to-1 read

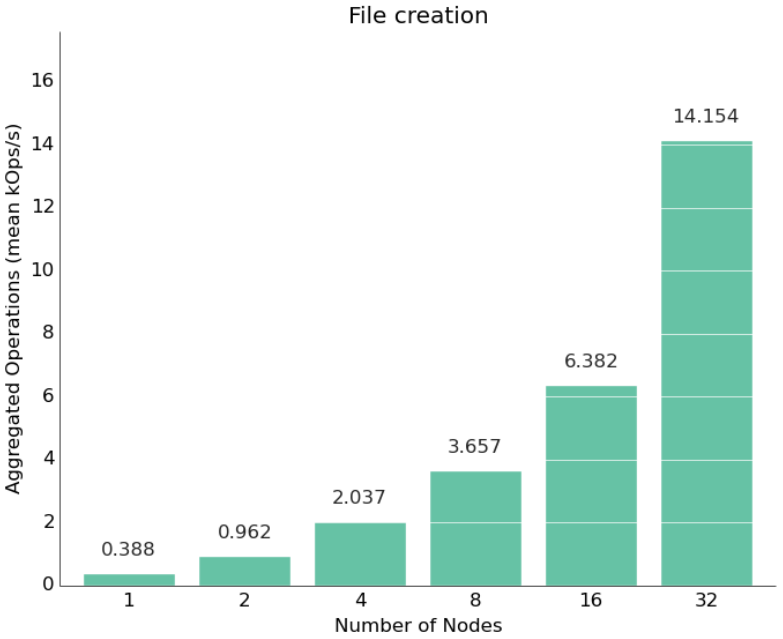

FIGURE B.5: MemFS, mdtest, file creation

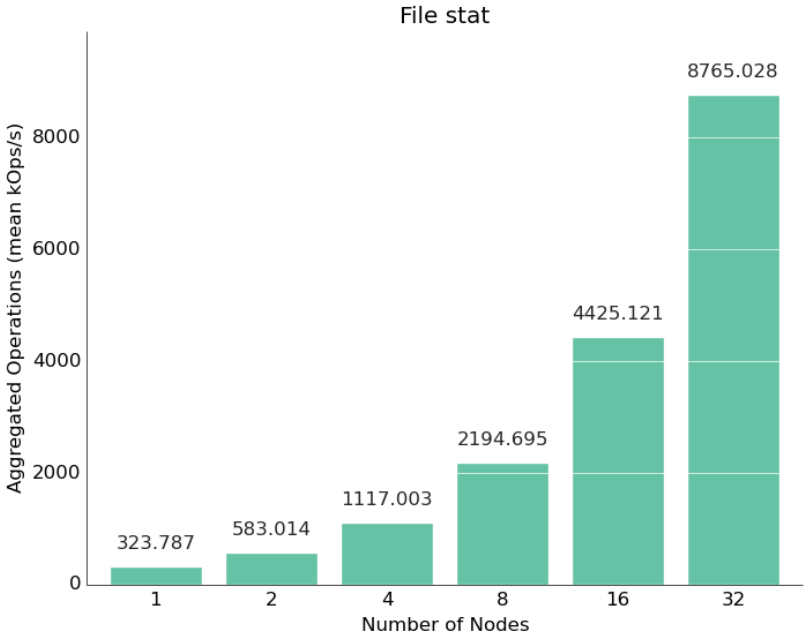

FIGURE B.6: MemFS, mdtest, file stat

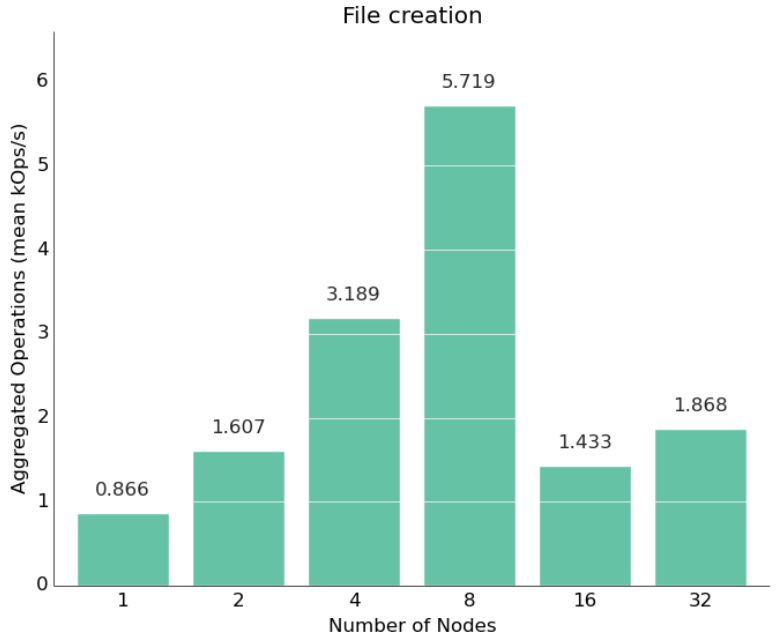

FIGURE B.7: AMFS, mdtest, file creation

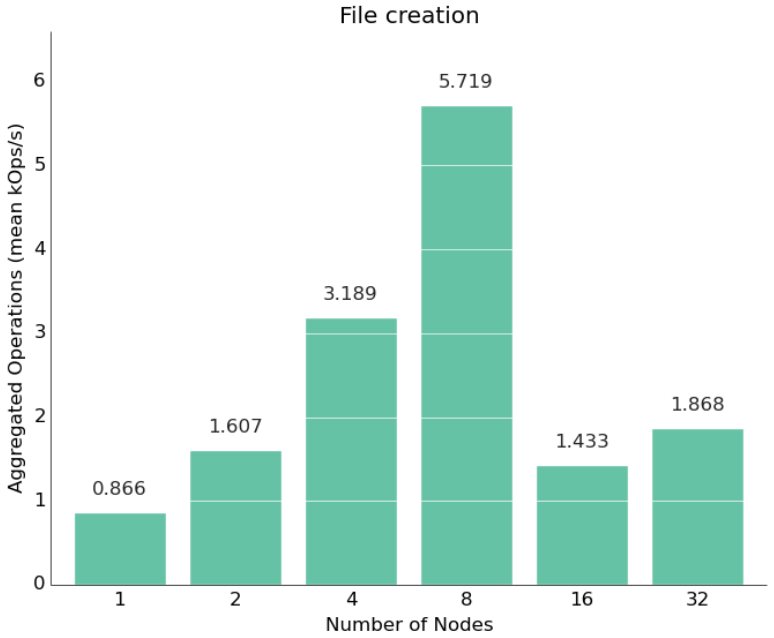

FIGURE B.8: AMFS, mdtest, file stat

Bibliography

[1] Ioan Raicu, Ian T Foster, and Yong Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11. IEEE, 2008.

[2] Joseph C Jacob, Daniel S Katz, G Bruce Berriman, John C Good, Anastasia Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas Prince, et al. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2): 73–87, 2009.

[3] Zhao Zhang, Daniel S Katz, Timothy G Armstrong, Justin M Wozniak, and Ian Foster. Parallelizing the execution of sequential scripts. In *Proc. of the International Conf. on High Performance Computing, Networking, Storage and Analysis (SC13)*, 2013.

[4] Alexandru Uta, Andreea Sandu, and Thilo Kielmann. Overcoming data locality: an in-memory runtime file system with symmetrical data distribution. submitted for publication. 2014.

[5] Memcached website, June 2014. URL `http://memcached.org/`.

[6] libmemcached website, June 2014. URL `http://libmemcached.org/`.

[7] Fuse website, June 2014. URL `http://fuse.sourceforge.net/`.

[8] *The JSON Data Interchange Format*. ECMA International, first edition edition, October 2013. URL `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[9] Zhao Zhang, Daniel S Katz, Michael Wilde, Justin M Wozniak, and Ian Foster. Mtc envelope: Defining the capability of large scale computers in the context of parallel scripting applications. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 37–48. ACM, 2013.

[10] Fsbench benchmark portal, June 2014. URL http://fsbench.filesystems.org/.

[11] Iozone benchmark website, June 2014. URL http://www.iozone.org/.

[12] Mdtest benchmark website, June 2014. URL http://mdtest.sourceforge.net/.

[13] Openmpi website, June 2014. URL http://www.open-mpi.org/.

[14] A list of mpi implementations on wikipedia, June 2014. URL http://en.wikipedia.org/wiki/Message_Passing_Interface#Implementations.

[15] The das4 cluster website, June 2014. URL http://www.cs.vu.nl/das4/.

[16] Linux man page for popen, June 2014. URL http://linux.die.net/man/3/popen.

[17] Linux man page for system, June 2014. URL http://linux.die.net/man/3/system.